Vaasan yliopisto
UNIVERSITY OF VAASA

Waltteri Kauppila

Developing an architecture for a test sequencer

School of Technology and Innovations
Master’s thesis in Automation and Computer Science
Master's Programme in Energy and Information Technology

Vaasa 2023

UNIVERSITY OF VAASA
School of Technology and Innovations

Author:	Waltteri Kauppila		
Title of thesis:	Developing an architecture for a test sequencer		
Degree:	Master of Science in Technology		
Programme:	Automation and Computer Science		
Supervisor:	Teemu Mäenpää		
Year of graduation:	2023	**Number of pages:**	80

ABSTRACT:
Testing is an important aspect of developing systems and products. It ensures quality of the product and that it works as it has been specified to work. Testing is a very expensive and time-consuming process. It can be made more effective by automating it using test automation tools. In this work a test automation tool called DriveTest2 is developed. DriveTest2 is a test sequencer that can be used to test different kinds of devices. A test sequencer is a program that controls a test setup by creating command sequences to device under test and other devices in the test setup. DriveTest2 will be capable of data acquisition during tests. This data is used for monitoring the sequence and for test reports. DriveTest2 can be used for different kinds of testing such as verification, reliability, accelerated lifetime and stress testing. DriveTest2 will replace an existing software called DriveTest. DriveTest is replaced because it is hard to maintain and update, and there is no documentation. The objective of this work is to find out what kind of software architecture a test sequencer should have.

DriveTest2 and its architecture is developed using an iterative development process. Development starts from gathering and analyzing requirements for the software which was done in previous work. The first iteration of the architecture is created from these requirements. Then the first design and implementation of DriveTest2 is created and given to users to get feedback. According to the feedback and requirements, refinements to the architecture and implementation are made. Then this process is repeated until we have software that we are satisfied with.

DriveTest2 is developed using C# programming language. DriveTest2 uses Model-View-View-Model architectural pattern to separate user interface and business logic. The business logic of DriveTest2 is designed to be object-oriented and is modelled using UML class diagrams. Throughout the iterations the architecture and design are refined with changes to the initial architecture and new functionalities like new interfaces to handle device specific functionalities.

As a result of this work, we have an iteration of DriveTest2 that is used in test laboratories with success. The software architecture of DriveTest2 enables DriveTest2 to satisfies its critical functional and non-functional requirements. We can conclude that a test sequencer software architecture should be able to enable the software to satisfy its requirements, have decoupled components with clear responsibilities, be flexible enough to accommodate changes to the architecture during the development and be extensible so that new functionalities can be added and make use of abstraction to abstract lower-level components and the actual hardware.

KEYWORDS: automation, software architecture, software design, software development, software engineering, testing

VAASAN YLIOPISTO
Tekniikan ja innovaatiojohtamisen akateeminen yksikkö

Tekijä:	Waltteri Kauppila		
Tutkielman nimi:	Developing an architecture for a test sequencer		
Tutkinto:	Diplomi-insinööri		
Oppiaine:	Automaatio ja tietotekniikka		
Työn ohjaaja:	Teemu Mäenpää		
Valmistumisvuosi:	2023	**Sivumäärä:**	80

TIIVISTELMÄ:

Testaaminen on tärkeä osa tuotteiden ja systeemien kehityksessä. Sillä varmistetaan, että tuote pystyy tekemään sille määritetyt asiat ja, että se on laadukas. Testaaminen on hyvin kallis ja aikaa vievä prosessi. Sitä voidaan tehostaa automatisoimalla käyttäen testiautomaatiotyökaluja. Tässä työssä kehitetään testiautomaatiotyökalu nimeltä DriveTest2. DriveTest2 on testisekvensseri, jota voidaan käyttää erilaisten laitteiden testaamiseen. Testisekvenssi on ohjelma, jolla voidaan ohjata testiympäristön laitteita luomalla sekvenssejä, jotka muodostuvat komennoista. DriveTest2 pystyy keräämään dataa sekvenssin ajon aikana. Tätä dataa käytetään sekvenssin ajon monitorointiin ja testiraportissa. DriveTest2:sta voidaan käyttää erilaisissa testeissä kuten verifikaatio-, luotettavuus-, elinikä- ja stressitesteissä. DriveTest2 korvaa olemassa olevan testisekvensserin nimeltä DriveTest. DriveTest korvataan, koska sitä on vaikea ylläpitää ja päivittää ja siihen ei ole olemassa dokumentaatiota. Tämän työn tavoite on selvittää millainen ohjelmistoarkkitehtuuri testisekvensseri pitää olla.

DriveTest2 ja sen ohjelmistoarkkitehtuuria kehitetään iteratiivisella kehitysprosessilla. Kehitys alkaa vaatimusten keräämisellä ja analysoinnilla, joka tehtiin aikaisemmassa työssä. Näiden vaatimusten perusteella luodaan ensimmäinen versio ohjelmistoarkkitehtuurista. Tämän ohjelmistoarkkitehtuurin perusteella luodaan ensimmäinen iteraatio DriveTest2:sen toteutuksesta ja se annetaan käyttäjille saadaksemme palautetta. Palautteen ja vaatimusten perusteella arkkitehtuuria ja toteutusta hiotaan paremmaksi. Tätä prosessia sitten toistetaan, kunnes todetaan, että ohjelmisto on riittävän hyvä.

DriveTest2 kehitetään C# ohjelmointikielellä. DriveTest2 käyttää Model-View-ViewModel arkkitehtuurimallia erottaakseen käyttöliittymän muusta logiikasta. DriveTest2:sen logiikka on tehty olio-ohjelmoinnilla ja se on mallinnettu UML luokkakaavioilla. Iteraatioiden aikana DriveTest2:sen ohjelmistoarkkitehtuuria hiottiin paremmaksi muuttamalla aikaisempaa arkkitehtuuria ja lisäämällä uusia toiminnallisuuksia kuten uusia rajapintoja hoitamaan laitekohtaisia toiminnallisuuksia.

Työn tuloksena saatiin luotua iteraatio DriveTest2:sta, jota pystytään käyttämään testilaboratorioissa hyvällä menestyksellä. DriveTest2:sen ohjelmistoarkkitehtuuri pystyy mahdollistamaan, että DriveTest2 pystyy toteuttamaan sille määritetyt toiminnalliset ja ei-toiminnalliset vaatimukset. Voidaan päätellä, että testisekvensserin ohjelmistoarkkitehtuurin on pystyttävä mahdollistamaan sille määritettyjen vaatimusten täyttäminen. Sillä on oltava komponentejä, joilla on selvät vastuualueet, ja sen on hyödynnettävä abstraktointia varsinkin laitetasolla. Arkkitehtuurin on oltava joustava, jotta sitä voidaan muokata ja kehittää kehityksen aikana sekä lisäämään uusia toiminnallisuuksia.

Avainsanat: automation, software architecture, software design, software development, software engineering, testing

Contents

Figures

Tables

Abbreviations

ACM	Association for Computing Machinery
DUT	Device Under Test
IEEE	The Institute of Electrical and Electronics Engineers
MVC	Model-View-Controller
MVVM	Model-View-ViewModel
PID	Proportional Integral Derivative
SDLC	Software development life cycle
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UI	User Interface
WPF	Windows Presentation Foundation
XAML	Extensible Application Markup Language

1 Introduction

Testing is an important aspect of developing systems and products. It ensures quality of the product and that it works as it has been specified to work (Harrold, 2000, p. 63). Testing, however, is a very time-consuming and expensive process. According to Myers et al., (2012, p. ix) it takes about 50% of a projects time and cost. There are different ways to speed up testing and/or make it cheaper. One of those ways is test automation. Test automation is the process of automating testing of product using test automation tools (Garousi & Mäntylä, 2016, p. 93). It cannot and should not be applied to all kinds of testing but where it can be applied it is a great help. It speeds up testing, helps ensure the quality of products and reduces human errors. Test automation can be achieved using some kind of test automation tool and there are many different kind of tools that can be used to automate testing (Garousi & Elberzhager, 2017, p. 93).

In this study a test automation tool called DriveTest2 is developed. DriveTest2 is a test sequencer that is being developed to replace an existing test sequencer called DriveTest. DriveTest is currently used to automate tests like endurance tests, highly accelerated life testing, cyclic testing, and load testing. It is being replaced because in its current state it is hard to maintain and update and there is no documentation. Test sequencer is a software that is used to automate tests by controlling devices in a test setup including the device under test (DUT). It is used to create test sequences which are sequences of commands to the DUT and test setup devices. Sequences can be executed and monitored. Monitoring is done by selecting parameters and channels that are measured and logged into a data log file during the sequence. The data log file is then used as the test report of the sequence run. The test report contains information about the sequence, test system and the logged values.

DriveTest2 is aimed to be used by users that do not have programming background. It is also not meant to be a fully automated in a sense that it needs to be configured and started by a user. There are many similar test automation tools that have more functionalities and can handle more complex test cases, but they can be hard to use and can be excessive for simpler test cases. Most of these solutions use test scripts while DriveTest2 is meant to use graphical representation for test that are used command devices. Also, most of these solutions do not have real-time data monitoring during test execution. Tests are usually executed, and the results analysed after. In DriveTest2 use cases there is a need for the users to be able to monitor devices during the tests.

In this study the focus will be on the software architecture of DriveTest2. The objective of this research is to find out what kind of software architecture a test sequencer should have, what kind of properties software architecture should have and what are key points to focus on when developing a software architecture for test sequencer.

Research question:

- What kind of architecture a test sequencer should have?

This study uses constructive research method. Constructive research method is a method that aims to solve real life problems by creating constructions (Lukka, 2014). Real life problem in this case being how to automate testing and the innovative construction to solve the problem is DriveTest2. Development of the construction will be done using an iterative process where each iteration builds on the previous iterations. DriveTest2 iterations are given to the users to get feedback. Next iteration is then refined from the previous iteration. This process is repeated until the construction is good enough.

Expected outcome of this work is to have iteration for DriveTest2 that can be used to replace DriveTest. This requires an architecture that can enable DriveTest2 to satisfy its requirements. DriveTest2 is meant to be done with object-oriented programming, so the architecture is expected to use classes for modelling.

This work will have the following structure. First basic theory regarding software engineering and test automation will be explained. Then we will explain the constructive research method in more detail and how it will be applied in this work. After the theory parts of the work, we will look at the iterative development process and background of DriveTest2 in detail. Then the initial architecture and design of DriveTest2 will be looked and how they are refined through iterations. Then we will look at the finished architecture to analyse and evaluate it. Lastly, we will have the conclusions of the work.

2 Software engineering

Nowadays software can be found everywhere from mobile phones, personal computers, Internet to medical life-support systems, space programs and national utilities and infrastructure (McConnell, 2004, p. 6; Sommerville, 2016, p. 18). Software being part of almost all parts of our lives makes software engineering very important as it makes it possible to build high quality and reliable software in a timely fashion (Pressman, 2010, p. 1). Software engineering can be defines as a systematic, disciplined and quantifiable approach to software development (Bourque et al., 2014, p. 1; ISO/IEC/IEEE, 2017, p. 421). Booch (2018) calls software engineering the art of the practical. Software engineering contains processes, methods, and tools to develop software (Pressman, 2010, p. 25).

2.1 Software development process

Software development process is a set of tasks that translate user needs into a software (ISO/IEC/IEEE, 2017, p. 421). There is no single software development process that fits in every project. It can and should be modified to fit the needs of the project. It, however, usually contains few fundamental tasks. These tasks are analysis, design, implementation, testing and maintenance (Leach, 2020, p. 10). There can be other tasks and the fundamental tasks can contain subtasks (Sommerville, 2016, p. 44). Tasks can be done in different manners ranging from being done sequentially, overlappingly, or iteratively.

Analysis task involves problem and requirements analysis. It is the process of understanding and defining what services are required from the system and identifying the constraints on the system's operation and development. In other words, this is the part where the requirements are done in order to decide what to build in order to satisfy the customer's needs (Sommerville, 2016, p. 54). Developing requirements are believed by many to be the most important part of software development (Leach, 2020, p. 89). The result of this task is a requirements specification that contains refined requirements of the software which define what the software should do and what kind of constraints it has.

Once the problem and requirements are understood can the design begin (Tsui et al., 2014, p. 130). Software design describes the architecture, interfaces, data models and structures, and components used in the software (Pressman, 2010, p. 215). Design is developed iteratively (Leach, 2020, p. 157; Sommerville, 2016, p. 56). It consists of two phases: architectural design and detailed design. Architectural design is the top-level overview and structure of the software while detailed design specifies components in more detailed fashion (Bourque et al., 2014, pp. 2–1; Tsui et al., 2014, p. 130). Output of design task can be various models, diagrams, documentations, and descriptions that describe the software and its design.

Goal of software projects is to create a working software and transforming designs into a working program is what implementation is about (Tsui et al., 2014, p. 188). Implementation includes coding, debugging and unit testing among other activities (McConnell, 2004, pp. 5–6). Design, implementation and testing go hand in hand, and their boundaries depends on software process of the project itself (Bourque et al., 2014, pp. 3–1). Some group up design and implementation tasks as one larger task and implementation should be taken into account when developing designs (Sommerville, 2016, p. 197).

Testing is done to achieve, assess and improve quality of the implemented software (Naik & Tripathy, 2011, p. 7). Goal is to find any defects that prevent the software to work as it is supposed to do (Leach, 2020, p. 268). Testing is also done to make sure that the implemented software satisfies its requirements. This is called software verification. Software validation is. Testing usually has three stages: Development testing, release testing and user testing. Development testing is testing that is done during the implementation process. Release testing is done to test complete software versions. User testing is done by testing the software on user’s own environment (Sommerville, 2016, p. 231). There are various ways to test software. They differ by their scale starting from individual modules to combined unit and eventually the complete system (Leach, 2020, pp. 269–270).

Software, especially larger ones, have long lifetimes and need to be maintained and supported during its lifecycle (Sommerville, 2016, p. 256). Maintenance consists of correcting defects, adapting the software to meet environmental changes and adding new features to support changed or new requirements (Pressman, 2010, p. 761; Sommerville, 2016, p. 271). Part of maintenance is customer support which consists of program defect related support and non-defect related support and services such as training and consulting (Tsui et al., 2014, p. 250). Maintenance task is longest lasting task and it ends when it has been assessed that the software is no longer needed, does not provide value or is to be replaced by a new software (Sommerville, 2016, p. 266).

2.2 Software development lifecycle models

Software development life cycle (SDLC) models are descriptive models that tell what to do not how to do it (Ruparelia, 2010, p. 8). They work as a guideline or a framework and their purpose is to provide guidance and structure on how to perform software development tasks (Scacchi, 2002, p. 3; Tsui et al., 2014, p. 58). SDLC models usually define what development tasks should be done, flow and sequence of the tasks and what kind of inputs and outputs each task has (Tsui et al., 2014, p. 58). Because there are different types of software, there is no one size fit all software development lifecycle model (Sommerville, 2016, p. 18).

2.2.1 Waterfall model

Waterfall model is a SDLC model by Royce (1970). Regarded as the classic SDLC model (Scacchi, 2002, p. 5) that created a foundation for requirements to be defined and analysed prior design (Ruparelia, 2010, p. 8). As seen in Figure 1 the waterfall model contains the following tasks: requirements, analysis, design, coding, testing and maintenance (Royce, 1970).

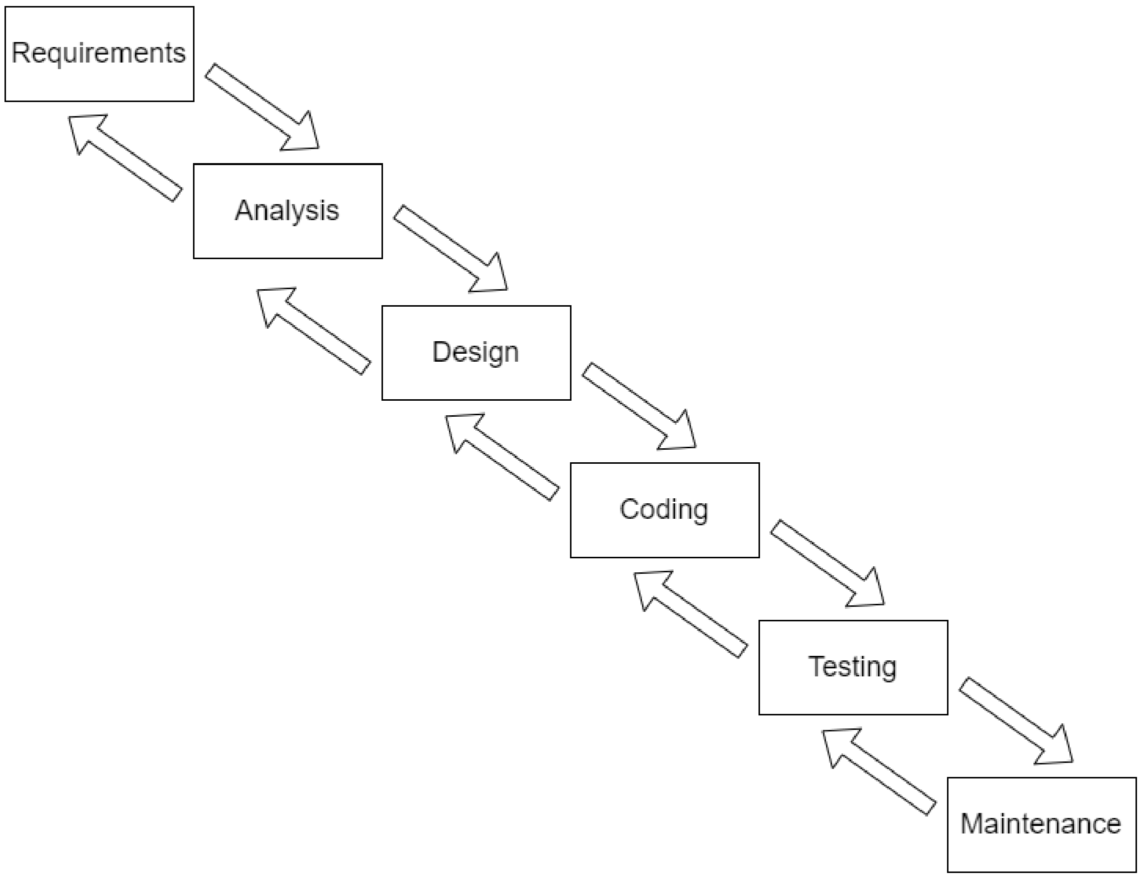


Figure 1. Waterfall model by Royce (1970)

It is a sequential model where next task is started after the previous one is completed but the previous stage could be revisited if necessary (Royce, 1970). Waterfall model is rigid model that cannot react swiftly to requirements changes or handle new information in later stages because everything has been planned (Sommerville, 2016, pp. 48–49).

2.2.2 Incremental model

Incremental along with iterative development models are nowadays popularized by agile development but they have been used since 1950s (Larman & Basili, 2003, p. 47). Unlike the waterfall model, incremental development is an iterative model where development is done in multiple iterations. These iterations are done in in increments where each increment implements a new functionality (Pressman, 2010, p. 41). First increments usually focus on core functionalities (Tsui et al., 2014, p. 61). Incremental development is sometimes referred as iterative waterfall because each increment is similar to waterfall model (Ruparelia, 2010, p. 9). This is also evident in Figure 2 which depict a typical incremental development and how each release contains a waterfall model like process.

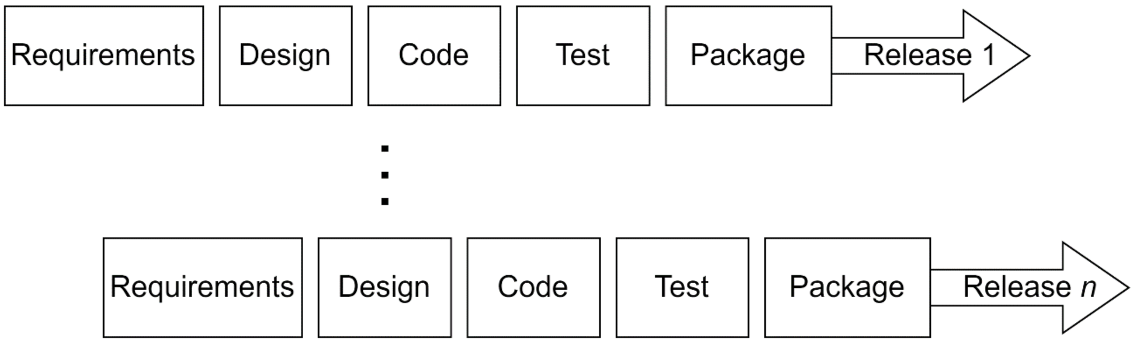


Figure 2. Incremental model (Tsui et al., 2014, p. 62).

Incremental development allows early feedback from each release and allows stakeholders to be more involved in each iteration (Ruparelia, 2010, p. 10; Sommerville, 2016, p. 50). Another good side of incremental development is that it makes it possible to adapt to requirements changes which is something that the waterfall model can't do.

2.2.3 Iterative model

Iterative development model also known as evolutionary is a model where each iteration of the software builds on the previous software (Pressman, 2010, p. 42). Iterative development should start from an initial implementation that is then enhanced in later iterations (Basil & Turner, 1975). Iterative development differs from incremental development because in incremental each increment fully implements a new functionality while in iterative these functionalities can be implemented across multiple iterations. Like incremental development, iterative also allows early feedback from each iteration and can react to new requirements or requirements changes.

2.3 Software architecture

There is no universal definition for software architecture as there are more than 150 definitions for it (Clements et al., 2014, p. 3) however the ISO/IEC/IEEE defines software architecture as the following: “Fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in principles of its design and evolution” (ISO/IEC/IEEE Systems and Software Engineering -- Architecture Description, 2011, p. 2). Simply put software architecture is composed of elements, their connections and how they are organized.

Software architecture is the foundation of a software system (Clements et al., 2002, p. xvii; Ingeno, 2018, p. 11). It is a set of early design decisions which works as the blueprint for the system (Clements et al., 2002, p. 2; Hofmeister et al., 2000, p. 4; Jansen & Bosch, 2005). These design decisions come from both functional and non-functional requirements of the system. Architecture is about defining a solution that will satisfy the these requirements (Ingeno, 2018, p. 11). Although software architecture is the foundation and early decisions, it does not mean that it is set in stone. Architecture should be analysed and evaluated during the development life cycle to see if it still is capable of meeting all requirements (Bass et al., 2013, p. 45). Architecture can and should be honed down if possible.

Software architecture is an abstraction of a system (Bass et al., 2013, p. 5; Clements et al., 2002, p. 2; Hofmeister et al., 2000, p. 5). It is only concerned with public aspects of elements and their connections (Bass et al., 2013, p. 6). Implementation details are thought of as non-architectural and are not part of software architecture (Clements et al., 2014, p. 6; Ingeno, 2018, p. 10). Abstracting details helps deal with the complexity of the system (Bass et al., 2013, p. 6; Hofmeister et al., 2000, p. 5).

2.3.1 Software architecture views

Software architecture views are a way of documenting software architecture. Software architecture can be very complex, so it needs to be divided into different views that together can better describe the whole architecture (Bass et al., 2013, p. 9; Perry & Wolf, 1992, p. 42). Views can be described as representations of sets of related elements and relations associated with them (Clements et al., 2014, p. 22). Kruchten (1995) organizes architecture into 5 different views using what is called the 4 + 1 view model. These views are used to describe the architecture from different viewpoints. 4 + 1 view model can be seen in Figure 3. Each view is described below.

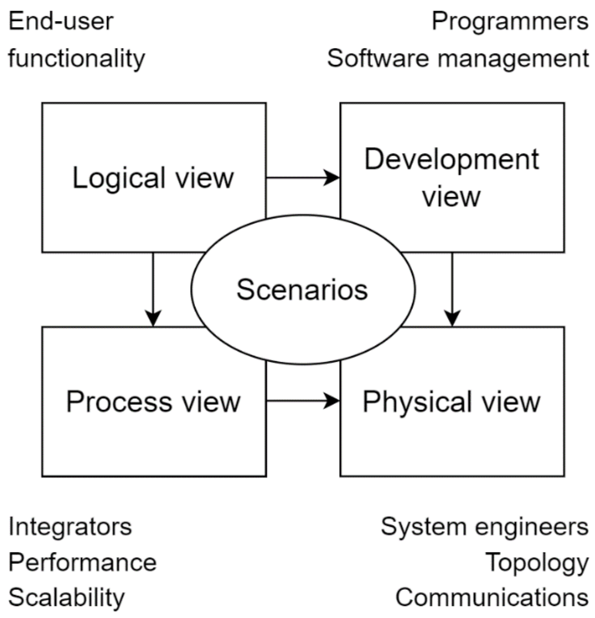


Figure 3. The “4+1” view model (Kruchten, 1995, p. 43).

The first view is called the logical view. Logical view describes how the architecture meets functional requirements of the system and how the system is divided into abstractions (Kruchten, 1995, p. 43). It can be represented with UML diagrams such as class diagrams that models the static structure of the system (Seidl et al., 2015, p. 49) and state machine diagrams that describes the behaviour of the system (Seidl et al., 2015, p. 85)

The second view is the process view. According to Kruchten (1995, p. 44) the process view is concerned of non-functional requirements such as performance and system availability. It addresses things like processes, threads, and concurrency. UML diagrams that can be used to represent this view are activity diagram, communication diagram sequence and sequence diagram. Activity diagram is about modelling procedural processing of the system (Seidl et al., 2015, p. 141) while sequence and communication diagrams focus more on interactions and communications between objects (Seidl et al., 2015, pp. 107, 137).

Next is the development view. Development view addresses the organization of the software modules into subsystems which are then organized in a hierarchy of layers (Kruchten, 1995, p. 45). Development view can be represented by using UML diagrams like package diagram and component diagram. Package diagram groups model elements according to common properties and are often part of other diagrams while component diagram describes components and their relationships (Seidl et al., 2015, p. 18).

Fourth is the physical view. Kruchten (1995) say that the physical view is about mapping the software to the hardware. It documents physical nodes for different configurations such as development, testing and deployment (Clements et al., 2014, p. 406; Kruchten, 1995, pp. 46–47). Fitting UML diagram for representing the physical view would be the deployment diagram that represents hardware topology in form of nodes and relationships between the nodes (Seidl et al., 2015, p. 18).

Last one is scenarios or use-case view. In this view all four previous views come together. The architecture is described using scenarios or use-cases that are significant to the architecture. These scenarios and use-cases are abstractions of the most important requirements (Kruchten, 1995, p. 47). Kruchten (1995) calls this view as plus-one view because it is redundant to the other views but it can be used during architecture development to discover element and to validate architecture design. UML use-case diagrams can be used to represent scenarios and use-cases (Seidl et al., 2015, p. 19).

2.3.2 Software design principles

There are many software design principles that have been either created for software or loaned from other disciplines that are used to create higher quality software. These principles help design software that is of higher quality, easier to maintain. Good example of software design principles are the SOLID principles found in Table 1. They are a collection of principles that help create flexible, maintainable, reusable, testable and understandable code (Ingeno, 2018, p. 184).

Table 1. SOLID principles with descriptions

Principle	**Description**
Single Responsibility Principle (SRP)	A module should only have one reason to change (Martin et al., 2018).
Open/Closed Principle (OCP)	A software artifact must be design so that they can be extended without having to modify it (Martin et al., 2018).
Liskov Substitution Principle (LSP)	If a base class has subclasses these subclasses should be substitutable for the base class (Martin et al., 2018).
Interface Segregation Principle (ISP)	Clients should not be dependable of properties and methods that they do not use (Martin et al., 2018).
Dependency Inversion Principle (DIP)	For code to be loosely coupled, high-level code should not depend on low-level detail implementations (Martin et al., 2018).

Keep it simple, stupid (KISS) is not a software specific design principle but it works very well in software design. Avoiding complexity is important because software that is more complex than it needs to be is usually of lower quality and harder to maintain (Ingeno, 2018, p. 177). There are principles such as "don't repeat yourself" (DRY) and "you aren't going to need it" (YAGNI) that have been created to keep software design simpler.

DRY principle tackles the problem of having duplicated code which will make it harder to maintain the code because having code duplications means that if there is a need to make a change you must make the same change multiple times (Hunt & Thomas, 2000, p. 27; Ingeno, 2018, p. 178). Duplication can be avoided e.g., with the use of constants and functions.

YAGNI is about just implementing what you currently need instead of trying to implement something what you might need in the future. (Ingeno, 2018, p. 182). YAGNI also means that you don't have to have a complete architecture and design of the program before you start writing it, instead you can have initial design and hone that design while coding (Fowler, 2019).

2.3.3 Software architecture patterns

Patterns are existing solutions to specific, recurring problems (Buschmann, 1996, p. 1; Gamma, 1995, p. 2). In software there are architectural and design patterns. Architectural patterns are bigger in scale than design patterns. Design patterns solve general design problems and deal with structuring of functionality instead of implementing it (Buschmann, 1996, p. 223).

Software architecture patterns are generic software architectures (Clements et al., 2002, p. xvii). They are reusable sets of design decisions with known properties (Bass et al., 2013, p. 203) and they are used to share and reuse software architecture knowledge (Sommerville, 2016, p. 175). Table 2 contains some common architecture patterns. These patterns have been used in many different software systems and they solve specific problems with some trade-offs.

Table 2. Table of common architecture patterns with short descriptions

Principle	**Description**
Layered architecture	Software is divided into hierarchical layers with specific roles like presentation layer and business layer (Bass et al., 2013, pp. 206–207; Buschmann, 1996, pp. 31–32; Richards, 2015, pp. 1–2).
Event-driven system	A distributed asynchronous pattern made of decoupled, single-proposed event processing components that receive and process events (Richards, 2015, p. 11)
Microkernel architecture	Separates the software into a core system and plug-in modules that can extend the core system (Buschmann, 1996, p. 171; Richards, 2015, p. 21).
Microservices architecture	Software system is composed of separately deployed units that work as service components (Richards, 2015, p. 27).
Model-View-Controller (MVC)	Decouples user interface from business logic by separating them into their own components and adding a controller component that mediates between them, however the view gets data straight from the model component (Bass et al., 2013, p. 213; Buschmann, 1996, p. 127).
Model-View-ViewModel (MVVM)	A variation of the Model-View-Controller pattern where the view and model do not have any knowledge about each other (Smith, 2009).

In this work the MVVM pattern is used. MVVM is a variation of MVC pattern (Smith, 2009). It is designed to use with Windows Presentation Foundation (WPF) UI framework but can be used with other UI frameworks (Garofalo, 2011, p. 43). MVVM aims to separate UI from the presentation logic and business logic and to make the UI testable (Garofalo, 2011, p. 1). This is done by dividing the software into three components: Model, ViewModel and View. The model encapsulates business logic and data. The model does not know anything about the view or ViewModel (Garofalo, 2011, p. 155). View contains UI while ViewModel contains presentation logic and works as a mediator between the view and the model (Brumfield, 2011, p. 70). The view does not have any knowledge about the model instead it only works with the ViewModel and this allows fully decoupling of UI and business logic (Smith, 2009). Figure 4 contains all MVVM components and shows how they interact with each other.

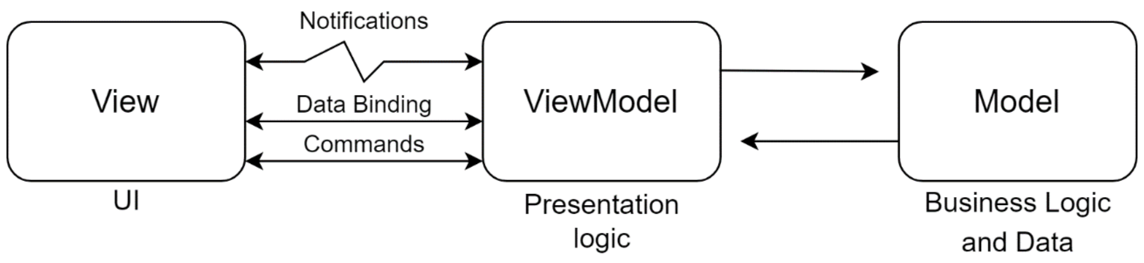


Figure 4. MVVM components and their interactions (Brumfield, 2011, p. 70).

View and ViewModel work together using data binding, commands, and notifications. Using these three the ViewModel can provide data and commands to the view (Brumfield, 2011, p. 161). Data binding allows binding ViewModel properties to the view (Smith, 2009). With commands the view can bind methods from the ViewModel to UI events like clicking a button. Using notifications, the ViewModel can notify the view that a data binding has its value changed and then the view can update itself to show the new value. With data binding and notifications, the ViewModel does not need to handle UI updating.

3 Test automation

Testing is an expensive and time-consuming process. It takes about 50 % of a projects time and cost (Myers et al., 2012, p. ix). Different tools and techniques have been developed to help with the testing process. One of these techniques is test automation or automated testing which can be done using testing tools.

Testing can be divided into manual and automated testing (Garousi & Mäntylä, 2016, p. 92; Taipale et al., 2011, p. 114). Test automation is the process of automating testing with the use of software testing tools (Garousi & Mäntylä, 2016, p. 93). Test automation takes longer and it is more expensive to develop than manual testing but it can minimize human errors, allow repeating tests frequently and reduces test duration thus increasing test efficiency and possibly reducing overall cost of testing (Avcioglu & Demirer, 2015; Polo et al., 2013, p. 84). Some test types like smoke tests, component and integration tests are often repeated and benefit greatly from automation (Berner et al., 2005, p. 574). It is important to properly implement test automation or it might cost more than what it is worth (Garousi & Elberzhager, 2017, p. 90).

3.1 Test automation techniques

There are different degrees of automation test automation ranging from partial automation to full automation. Depending on the tests partial automation can be better than full automation (Hoffman, 1999, p. 3). There are different test automation tools and each one has their pros and cons. They offer varying features that fit different kind of tests (Ateşoğulları & Mishra, 2020, p. 70). Some tools focus on automating unit tests while others focus on automating more complex tests such as verification tests. Selecting a correct test automation tool is difficult process and depends on what kind of context it will be used in (Raulamo-Jurvanen et al., 2017). We will look at two types of test automation techniques and tools: Automated test systems and test automation frameworks.

3.1.1 Automated test systems

According to United States Department of Defense (DoD) automated test system (ATS) is a computer-controlled suite of electronic test equipment and instrumentation hardware and software that are used to test DUTs (2016, p. 1). ATS is composed of a test executive, test program set (TPS) and automated test equipment. Automated test system software architecture can differ from each automated test system but generally it looks something like the architecture seen in Figure 5.

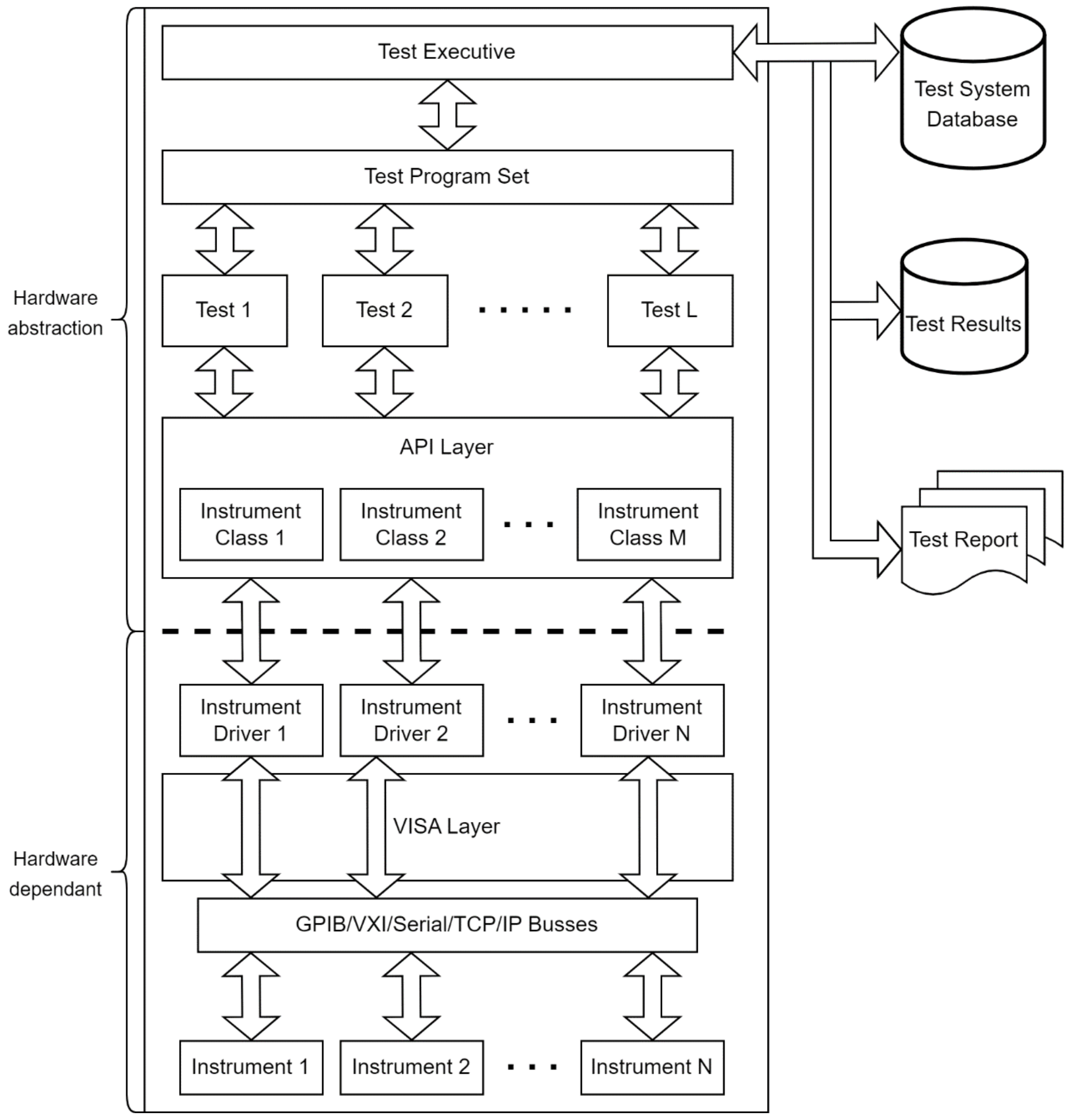


Figure 5. ATS software architecture. Adapted from Carey (2019).

ATS software architecture is divided into multiple layers. These layers are then divided into hardware abstraction and hardware dependant layers. Top layers do not know what kind of hardware they are dealing with and such they use hardware abstraction. This allows easily to switch the hardware that is used in tests. Bottom layers are hardware dependent as these layers interact with the actual hardware. The first layer is the test executive that executes the test program set and creates test results and reports. Test program set consists of tests that are usually done using a programming language. These tests then use hardware abstraction API layers to communicate with the real hardware. API uses drivers that use different communication protocols to communicate with the hardware.

3.1.2 Test automation frameworks

Test automation frameworks are set of test tools, equipment, test scripts, procedures used together to automate testing (Naik & Tripathy, 2011, p. 400). Test automation framework provides core functionalities such as automatic test execution, logging and reporting (Amaricai & Constantinescu, 2014, p. 152).

Test automation frameworks can be used to test software, system, and hardware while automated test systems are designed for testing hardware. Test automation frameworks are a bit bigger in scale compared to an automated test system as it can contains different tools, equipment, scripts, and procedures. Automated test systems usually use specific equipment to test DUTs or are designed to test specific kinds of DUTs while test automation frameworks are designed for more generic use.

There are different types of test automation frameworks such as data-driven, keyword-driven and hybrid (Amaricai & Constantinescu, 2014, p. 153):

- Data-driven frameworks separates test data from test scripts. Test data is loaded during test execution. Data-driven allows use of multiple sets of test data for single test script (Albarka & Zhanfang, 2019, p. 220; Soe et al., 2022, p. 201).

- Keyword-driven framework stores test data separately but test scripts are built using keywords that are also stored separately (Albarka & Zhanfang, 2019, p. 220; Soe et al., 2022, p. 201).
- Hybrid frameworks are combination of multiple frameworks aiming to create a flexible framework. Combining frameworks can have their strengths while possibly mitigating some of the weaknesses (Albarka & Zhanfang, 2019, p. 220; Soe et al., 2022, p. 201).

3.2 Existing practical solutions

There are multiple commercial test automation tools in the market. Among them TestStand and Robot Framework are popular ones. They are used in the company that this study is done in. They can both be used as test sequencers and contain similar functionalities that DriveTest2 is required to have but both require either programming or use of code modules.

3.2.1 TestStand

TestStand is a test executive software by National Instruments (n.d.-b) that can be used to develop and deploy tests. With the use of adapters TestStand can execute code developed in LabView or other programming languages (Arpaia et al., 2015, p. 13). TestStand works by creating sequences with the TestStand Sequence Editor (National Instruments, 2014). Sequences can contain calls to code modules or other sequences. Using TestStand and creating sequences does not require programming background but the code modules need programming. TestStand can execute sequences and generates a test report of the execution. Test report contains information about whether a test step has passed or failed.

TestStand is a very comprehensive and versatile test management software, but it can be bit too excessive to use for smaller scale of tests. It also does not have the option of real-time monitoring of measurements during test execution. TestStand requires tests to be written in a programming language such as LabView or Python making it difficult to learn for people without programming or scripting background. TestStand also requires a license for running making it possible expensive to operate.

3.2.2 Robot Framework

Robot Framework is a generic open source automation framework used for test automation (Robot Framework, n.d.-a). Robot Framework is keyword-driven automation framework that is highly extensible with libraries written in Python, Java or other programming languages (Robot Framework, n.d.-b). Robot Framework has layered architecture seen in Figure 6.

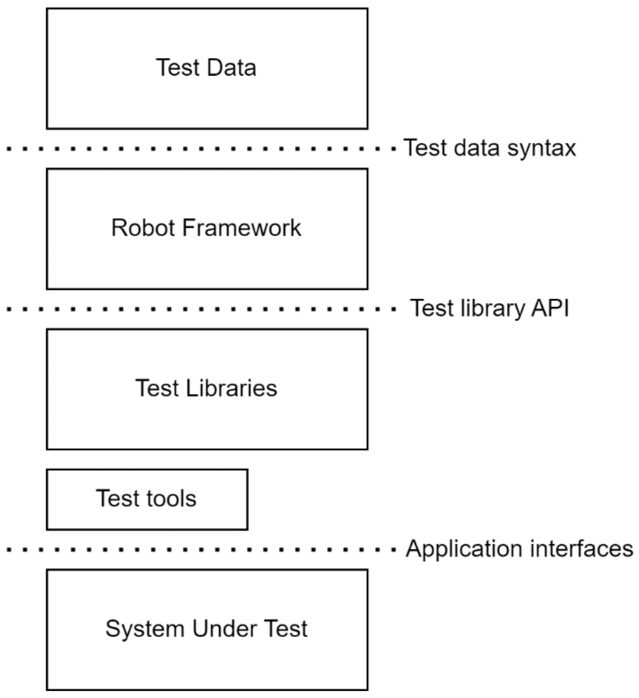


Figure 6. Robot Framework architecture (Robot Framework, n.d.-b).

Test data contains test cases which are created using table-like syntax and keywords that are meant to be easy to read. Instead of having a sequence editor like TestStand, Robot Framework test cases are written using a text editor. Robot Framework is the engine that processes test data, executes tests and generates logs and reports (Robot Framework, n.d.-b). Robot Framework is meant to be application and technology independent framework meaning that the Robot Framework does not know about the system under test, but instead the different test libraries handle all interactions with the system under test as seen (Robot Framework, n.d.-b).

While Robot Framework is designed to be easy to use it still requires programming background to create test cases. Test cases can easily require loops or if-else clauses that can complicate test cases. Not having a simple editor for creating test cases can make learning Robot Framework difficult without programming background. There is also the need for test libraries and there is a possibility that you need to create your own libraries if there are no fitting third-party libraries. For simple tests using a framework can be excessive and take time to get running.

3.3 Related works

In this chapter few related works about automated test systems and frameworks are looked at. When researching prior literature, five related works were found that talk about test automation tools and their architecture. They were found from The Institute of Electrical and Electronics Engineers (IEEE) and Association for Computing Machinery (ACM) digital library. Their titles, descriptions and shortcoming can be found in Table 3. Shortcomings in this case means the reasons why these works would not work as solution for this works problem.

Table 3. Related works with descriptions and shortcomings.

Title	Description	Shortcomings
COM-based test foundation framework (Thompson, 1999)	COM based framework for test development, test execution and test data usage.	Uses outdated technologies and test scripts
An object-oriented framework for automatic test systems (Xu et al., 2003)	Object-oriented framework for ATS with layered architecture.	Objects do not match DriveTest2 setups
Server power supply test automation approach (Apostolaki-Iosifidou et al., 2018)	Python and open-source library based custom ATS to replace expensive commercial solutions	Made specifically for their test setup
A system for automated testing in development of measuring devices for industrial process instrumentation (Keimling et al., 2013)	Custom automated testing system with main component that handles test execution and child components that handle device interaction	Uses test scripts and does not support real time data monitoring
Next Generation of Test Automation and Systems (Faraj et al., 2019)	ATS that uses a plug-in architecture where core component is open-source sequence engine.	Uses third party sequence engine and plug-in architecture can be very complex and hard to maintain

Thompson (1999) introduces a testing framework called the Test Foundation Framework (TFF). This framework is based on Microsoft Common Object Model (COM). TFF architecture is divided into components which are accessed through their interfaces. Test environment of TFF is divided into three major parts: Test development, Test execution and Test data usage. Of these three, test development and test execution are looked at in more detail. Test development component creates test program components that are initiated by the test executive component. Then the test program is executed by runtime software that interacts with the test hardware using instrument drivers. Nowadays Microsoft COM is outdated technology, and the test are created using programming languages.

To quickly create complex ATSs, Xu et al., (2003) created an object-oriented framework for automatic test systems. This framework has its architecture divided into four layers: Application, instrument drivers, I/O library, and instrument. Framework is modelled using class diagrams to show how classes interact with each other. Using object-oriented approach allows more stable, reusable, and maintainable system compared to function-oriented models. Classes used in here do not match the DriveTest2 use cases but in a similar way object-oriented programming is used.

To replace an expensive commercial solutions in an automated test system, Apostolaki-Iosifidou et al., (2018) used Python and open-source libraries to create a more flexible and simpler custom automated test system to test power supply units (PSU). Tests are divided into configuration and execution phases. During execution the python program record data from test PSU and an oscilloscope. After test execution recorded data is shown in different figures and tables that are then analysed by the tester. This work was done specifically for their test setup and does not contain many details about the architecture of the software system.

Keimling et al., (2013) present their custom test system for automated testing called KROHNE TestCenter (KTC). KTC focuses on automating functional tests, regression tests and long-term tests. KTC is divided into a main component TestCenterMain (TCM) and child components TestCenterApps (TCA). TCM handles executing test cases, controls TCAs required by the test cases and creates test reports. Test cases in KTC are done using ECMAScript language and each script contains list of required TCAs and three main functions: setup(), run() and cleanup(). TCAs receive commands from the TCM, communicate with devices using different interfaces and collect data to CSV files. KTC has automated test data analysis that supports use of different software tools like MATLAB, GNU Octave or Python. Their architecture is modular, flexible, and extensible with more devices in future, but their tests are made with scripting language which makes it harder to create test by users without programming background. There does not seem to be a support for real time data monitoring during test execution.

To have effective test automation development, Faraj et al., (Faraj et al., 2019) present a high-level software architecture for test automation. Faraj et al., have created a plug-in architecture where test sequencer engine is the core system. They use following plug-in types: DUT drivers, test steps, user interfaces, instrument drivers, result listeners and tools. Faraj et al., demonstrate their architecture by creating an automated test setup where they use an open-source test sequence engine as the core and the create plugins for it. Using plug-in architecture allows to create different kind of plug-ins and makes system modular and flexible but can make it difficult to implement. Other weaknesses are complexity, the need to maintain all plugins or make the core system be backwards compatible with existing plugins and it can be hard to test how all plugins work with other plugins.

4 Research method

Constructive approach is a methodology that produces innovative constructions that aim to solve real life problems (Lukka, 2014). These constructions are linked to previous research and theory and are normative in their nature (Virtanen, 2006). To create a construction existing theoretical knowledge and new empirical knowledge are needed (Ojasalo et al., 2009). Ideal result of constructive approach is a construction that manages to solve a real life problem and through the problem solving process it is capable of producing theoretical and practical contribution (Lukka, 2014). Constructive approach was originally developed for business economics, but it fits for technology related research. Constructive research is a great fit when the problem requires concrete results such as a new product, system or model (Ojasalo et al., 2009, p. 66). Core elements of constructive approach can be seen in Figure 7.

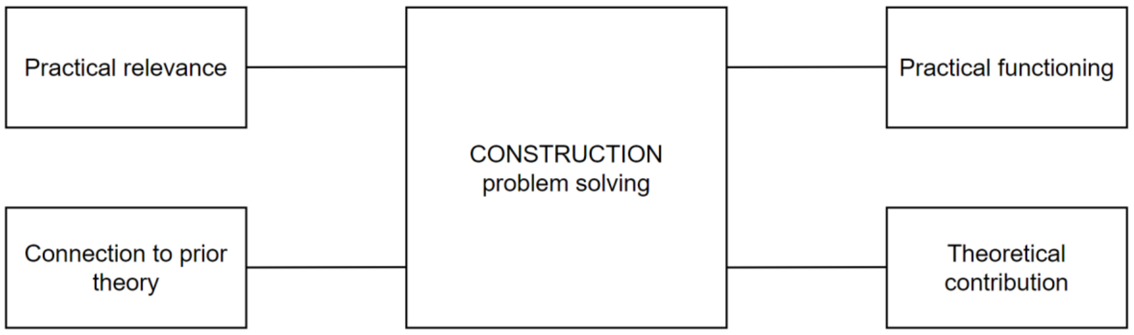


Figure 7. Core elements of constructive approach (Kasanen et al., 1993).

Core elements focus on the construction being practical solution to a problem and linked to prior theory. Practical relevance means that the constructive approach should focus on practical problems and result in a construction that aims to solve one (Lukka, 2014). Practical functioning means that the construction should be tested how well it works. The construction should be connected to prior theory and the findings of the research should be reflected back to prior theory (Lukka, 2014). Theoretical contribution is about the analyzation and reflection of the findings of the work.

Constructive approach can be understood as a methodological research approach (Lukka, 2014). Constructive approach is both empirical and descriptive as seen in Figure 8. It is empirical because it contains practical implementation, and it is based on experience. Normative is about measuring or assessing how good or functional something is instead of just describing how something is. In constructive research it can be seen when the implemented construction is tested how well it solves the problem.

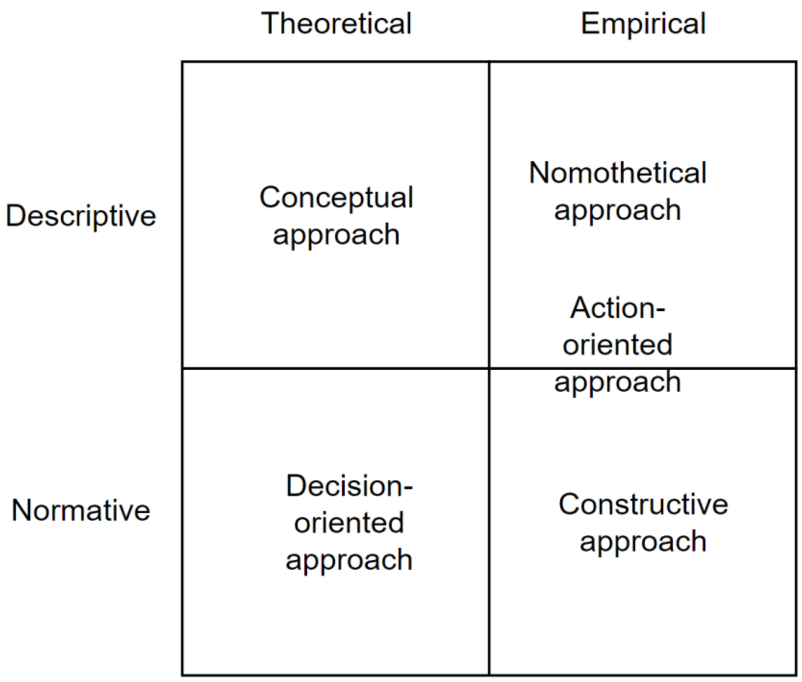


Figure 8. Constructive approach as a methodology (Kasanen et al., 1993).

Constructive research process is divided into seven steps as seen in Figure 9. The process flows from one step to another, and these steps are usually done in a sequence. Each step is described more in detail below.

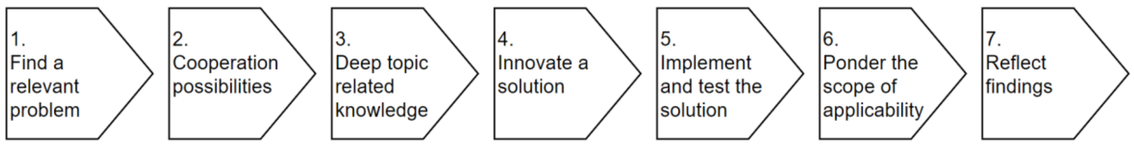

1. Find a relevant problem
2. Cooperation possibilities
3. Deep topic related knowledge
4. Innovate a solution
5. Implement and test the solution
6. Ponder the scope of applicability
7. Reflect findings

Figure 9. Constructive research process (Lukka, 2014).

Constructive research process starts as any research process by finding a research problem. Identifying and formulating a research process is regarded as the most important/difficult part of research process. Ideal problem would be one that has practical meaning while being paradoxical or insufficiently analysed in prior works (Lukka, 2014). Problem can be found by discussing with organizations (Virtanen, 2006).

Second part of the process is to examine the potential for long-term cooperation with the target organization. According to Lukka (2000) the researcher should become part of a team devoted to the project handling the problem in hand and the researcher should have the possibility of publishing the findings of the work or the will not be realized academic contribution from the work. If the researcher works alone there is a high probability that the project will fail because it won't lead to a true implementation phase (Lukka, 2000, p. 5).

In order for the researcher to be able to reflect his finding and to analyse works theoretical contribution, proper knowledge on the topic, both theoretical and practical, should be acquired (Lukka, 2014). This could for example be done by studying topic related literature and getting to know the target organization.

According to Lukka (Lukka, 2000) innovation step is in its nature creative and heuristic, thus there is little generic advice to be given. This step should be cooperative work with the target organization. Just applying previous solutions to a new environment can't be regarded as application of constructive approach (Lukka, 2014). Innovating a solution idea is a critical part of the process as if no construction can be designed the work can't continue (Lukka, 2000, p. 5).

After innovating and designing a solution, the next step is to implement and test it. This step tests one of the core elements of constructive approach which is the practical functionality of the construction. The construction is not only tested technically but also how well the research process worked as a whole (Lukka, 2014). For the construction to be implemented and used in the target organization, the researcher and project team needs to be able to sell the construction for the organization or the implementation will probably fail (Lukka, 2014).

When the construction has been tested, the researcher needs to analyse results of the process and its preconditions (Lukka, 2000, p. 7). In this step the researcher should ponder if the construction could be implemented in other organizations, what kind of changes the construction would need for that to happen and if the construction could be used to solve similar problems (Lukka, 2014). In this step and the last one, the researcher needs to be able to distance himself or herself from the empirical work and think from an academical point of view (Lukka, 2000, p. 7).

Final step of the process is to reflect the findings of the project to prior literature. According to Lukka (2000) This part is inevitable from an academic point of view because the researcher has to identify the theoretical contribution of the work. There are two types of theoretical contribution: The new construction itself and the fact that the construction truly works or does not work (Lukka, 2000, p. 7).

How the constructive approach process is done and implemented is up to the research. Below in Table 4 is a recap of how the constructive approach process was implemented in this work. Each step will be looked at in more detail below.

Table 4. How Lukka's seven process steps were implemented in this work.

Lukka's step	**Implementation**
1. Find relevant problem	Industrial need to replace prior test automation tool
2. Long-term co-operation	Employed by the target organization
3. Practical and theoretical understanding of topic	Study topic related literature, have meetings, and interviews about the problem
4. Innovate solution, develop construction	Design initial architecture and model based on requirements
5. Implement and test the solution	Create multiple iterations, test them, and get feedback from users
6. Applicability of the solution	DriveTest2 is generic sequencer that could be used for different kind of tests
7. Reflect findings	Findings compared to similar works and topic literature

Problem of this research is to create a test automation tool that to help the process of product testing especially reliability and endurance testing. The target organisation already has a tool that is used to solve this problem, but it has been found to be inadequate thus it is being replaced by DriveTest2. Manual testing can be very laborious work especially when a test setup can have many kinds of devices that all need to be used and controlled. Creating a tool to control these devices and gather data from them is makes it easier but creating a tool that makes the test process repeatable makes it even easier.

Co-operation with the target organisation was done by being fully employed by the target organisation. Main task of the employment being developing and maintaining DriveTest2. The organization has a big need to solve their test automation problems thus they have employed the researcher as an employee to make sure that the project will be done and that they can offer as many resources to it as needed. This has made it so that everyone was committed to the project. Getting feedback and resources from the organization has helped greatly on this project.

Theoretical understanding was acquired by studying the topic. There was prior education and knowledge on the topic acquired from courses such as programming and modelling courses. More studying on the topic was done by reading books and research papers and watching educational videos related to the topic. Practical understanding came from trying out what was learned from books and videos to better understand. Meetings and interviews were conducted during the prior requirements analysis to understand purpose of DriveTest2, how and where it would be used, what kind of devices and technologies it would interact with. There was also some trial-and-error style of practical learning that was beneficial in learning the programming language used in implementing DriveTest2.

Innovation, development, and testing of DriveTest2 was done by first designing the initial architecture and model of the system from the requirements that were gathered previously. Then the first iteration of DriveTest2 was created. From here started an iterative loop where feedback was acquired by DriveTest2 users. Each iteration was tested by the developer and by the DriveTest2 users. From the feedback and requirements, the next iterations were honed and DriveTest2 was gradually being developed.

Scope of DriveTest2's applicability is pondered by how well it could be used for different kind of testing. DriveTest2 was first designed to handle reliability testing but since its functionalities are generic it can possible be used for other kinds of tests. Having the capability to control the DUT and other devices while collecting data from them makes it possible to use DriveTest2 e.g., functional, non-functional, validation and verification testing among others. One good test type that DriveTest2 could be used for is regression testing. After updating the DUT, DriveTest2 could be used to run the same test sequence and the test data could be compared to previous runs.

Findings of this work will be reflected to prior literature. The prior literature will consist of test automation and test automation architecture related research papers. Findings will be compared to prior works, how things were possibly done differently and why.

5 Test sequencer development

In this chapter we will look at how DriveTest2 will be developed. We will look at the development process and what kind of requirements DriveTest2 has. Then we will talk about the backgrounds of DriveTest2 in more detail. After this we will look at the initial architecture and design of DriveTest2. Lastly, we will look at how DriveTest2, the architecture and design has evolved from iteration to iteration.

Architecture of DriveTest2 was developed iteratively by first creating the initial architecture and design and then evolving them iteratively during the development process. DriveTest2 has been developed using an iterative development model. Figure 10 shows the development process of DriveTest2. It starts with gathering and analyzing requirements for DriveTest2. This was done in a previous work. From the initial requirements the first architecture and design are created. Then the first iteration of the software is implemented, tested, and given to users to get feedback. With feedback, the iteration is reviewed and then refined to create the next iteration. Then the development cycle just keeps repeating itself until it is deemed that the product is completed. First iteration does not implement the whole architecture, merely a part of it and then through new iterations the implementation gets enhanced so that it will better implement the whole architecture.

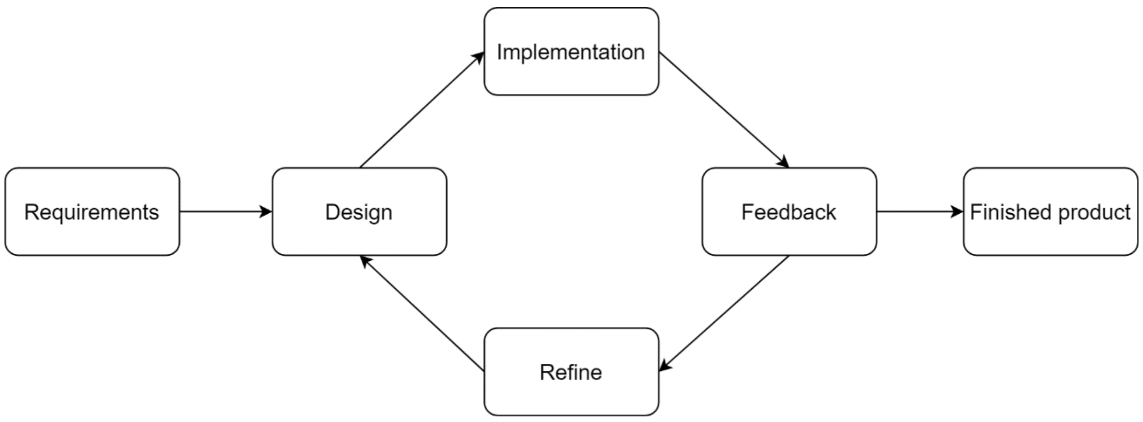


Figure 10. Development process of DriveTest2.

As previously mentioned, the requirements for DriveTest2 were gathered in a previous work. Both functional and non-functional requirements are what guide the architecture and design of DriveTest2. Requirements are what DriveTest2 should be able to do and how well it should do them. DriveTest2 is designed in a way that it would be able to satisfy all requirements that it has. First iterations of DriveTest2 will only satisfy portion of the requirements but the architecture and design should be designed in a way that allows it to satisfy most if not all requirements, and it should be easily extensible in case of new requirements. Each iteration tries to satisfy more and more requirements. Requirements were gathered mainly by conducting interviews with users of DriveTest, having meetings about where and how DriveTest2 will be used and analyzing the functionalities of DriveTest. Requirements of DriveTest2 are divided into functional and non-functional requirements.

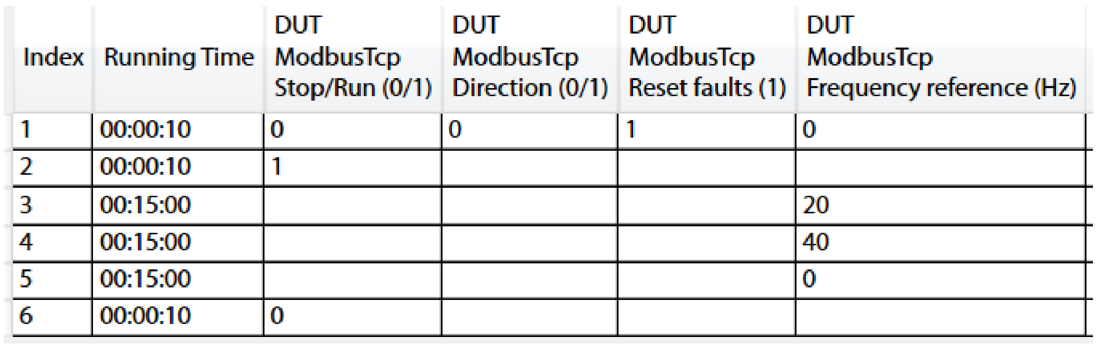
Functional requirements tell what the software should be able to do. Core functional requirements of DriveTest2 are the following: Defining what devices will be used in the sequence, creating, and editing a test sequence, executing the sequence, monitoring sequence execution, measuring parameters, reading data from channels during the sequence, logging measured data to a report file and lastly looking at the report. There are varying levels of functional requirements ranging from core functional requirements to smaller functional requirements that could be UI related functionalities like the functionality of being able to rename a test.

Non-functional requirements tell how well the software should be able to do and what kind of constraints it might have. Since DriveTest2 will be used for performance and lifetime testing it will need to be running for very long periods of time. This means that DriveTest2 needs to be stable and reliable enough to run for long periods of time. DriveTest2 must be able to execute the sequence and measure data from devices at the same time meaning that there must be concurrency. Other non-functional requirements are usability, maintainability, and extensibility.

5.1 Background of DriveTest2

DriveTest2 is set to replace an existing test sequencer called DriveTest. DriveTest is being replaced due to many reasons such as lack of documentation, unstableness, feature bloat and the original developer not being part of the company for years. All of these have made maintaining and updating DriveTest difficult. Thus, it was decided to replace DriveTest with DriveTest2.

DriveTest is a test sequencer used to test frequency converters. Test sequences are created using DriveTest. These sequences are divided into subsequences that are called tests. Each test contains test vectors. A test vector contains a time value and a set of commands. When a test vector is executed, it will execute all its commands and then wait its time value before going to the next test vector. With these test vectors you can create easily create a test where same parameters or commands are done multiple with same or different values. Figure 11 shows an example test with test vectors.



Index	Running Time	DUT ModbusTcp Stop/Run (0/1)	DUT ModbusTcp Direction (0/1)	DUT ModbusTcp Reset faults (1)	DUT ModbusTcp Frequency reference (Hz)
1	00:00:10	0	0	1	0
2	00:00:10	1			
3	00:15:00				20
4	00:15:00				40
5	00:15:00				0
6	00:00:10	0			

Figure 11. Example test of DriveTest2 with test vectors and test commands.

Test and its test vectors are represented by a data table. Each row in the table is a test vector. Columns after “Running Time” are the test commands of test vectors. Every test vector shares the same test commands, but they can have different values. This allows quick and simple way to create tests if there are not many test commands. Running times allows creation of longer tests which are suitable for endurance tests. Having tests in a table like structures with commands to devices makes it possible to create tests without having any experience with programming.

DriveTest2 will be used for automating the testing of frequency converter drives. DriveTest2 will be used for reliability, lifetime, and load testing among other possible tests. An example test setup for endurance testing where DriveTest2 will be used would contain the following devices: DUT, temperature controller and recorder. DriveTest2 will be used to control all these devices and measure and read parameters from them. A test sequence will be created with DriveTest2 where the temperature of the oven will be controlled by commanding the temperature controller. DUT could for example be running in a run state while the recorder is used to measure temperatures from different components of DUT. This sequence is then repeated until the DUT breaks, or some issue occurs. Figure 12 shows an example test setup of DriveTest2.

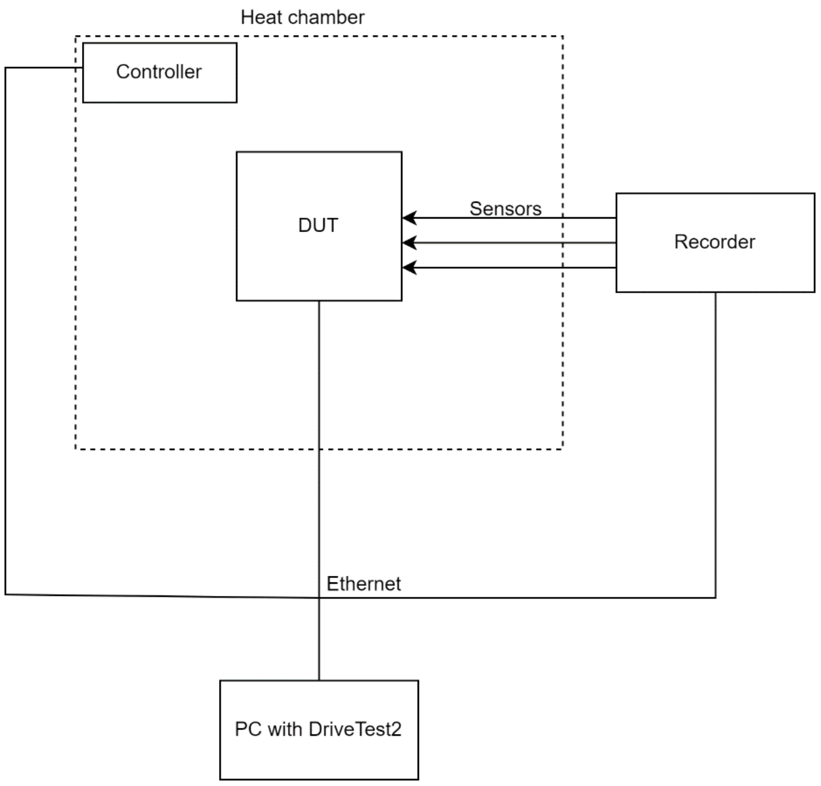


Figure 12. Example test setup of DriveTest2 with DUT, controller and recorder connected to a PC with DriveTest2.

The example test setup has the following device: Heat chamber controller, DUT, recorder and a PC. The PC has DriveTest2 in it and the PC is connected to the devices through ethernet. Heat chamber controller is used to control the heat chamber where the DUT is in. Recorder is used to measure DUT and its components using sensors. DUT is tested by controlling the heat chamber and by controlling the DUT itself e.g., running it in different settings.

Being a replacement for DriveTest, most of DriveTest2 functional requirements come straight from DriveTest functionalities. These mainly being the ability to configure test setup, create test sequences, execute them, and monitor the execution. There are smaller functionalities not listed here that carry on from DriveTest to DriveTest2 but those listed are the core functionalities. Since DriveTest2 is about replication DriveTest's functionalities, DriveTest will have an influence on how these functionalities will be developed. In addition to DriveTest's functionalities, DriveTest2 will be used to open test reports and to analyse test data which previously was done with a separate program called TdmsAnalysis. DriveTest2 will aim to replicate DriveTest's functionalities while also being easier to use, maintain and update. DriveTest2 does not automatically analyse test reports but instead shows them in a way that the user can analyse the data easier.

5.2 Initial architecture and design

One of the earliest design choices for a software system is to decide what programming language or languages will be used to develop it. For DriveTest2 it was decided to use C# and .NET Framework. Two main reasons are being able to reuse code from DriveTest like device communication related codes and the use of Measurement Studio (National Instruments, n.d.-a). Measurement studio comes with libraries that can be used for device communication, and it has built in graph components that can be used to display data during sequence execution and when looking at logged data. Measurement Studio also contains TDMS file format that can be used to log measurement data and libraries for analyzing measured data like calculating RMS values. Measurement Studio is an extension for .NET Framework that contains UI components such as graphs and libraries for data acquisition and analyzation. Measurement Studio is also used in DriveTest and has been proven as a working solution.

One of the problems of DriveTest was that the UI and business logic were tightly coupled. This caused problems that we want to avoid in DriveTest2 and will increase testability. Solution for this was to decouple UI from presentation and business logic using the MVVM pattern. Since the MVVM pattern was created for WPF and .NET it is a great fit for DriveTest2. With the MVVM DriveTest2 is divided into three components as seen in Figure 13: DriveTest2UI.View, DriveTest2UI.ViewModel and DriveTest2.Common. These three components are divided into two different projects. DriveTest2UI.Views and DriveTest2UI.ViewModels are part of DriveTest2UI and DriveTest2.Common is part of DriveTest2.

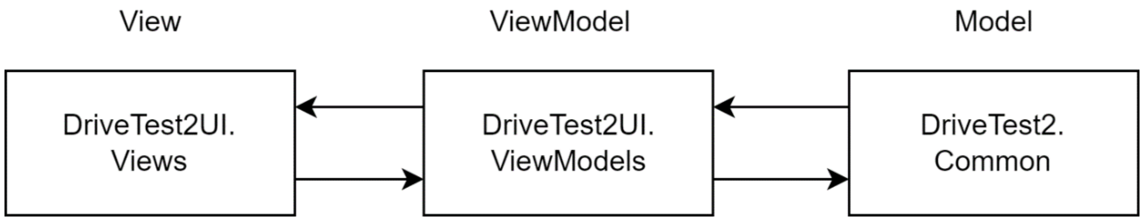


Figure 13. DriveTest2 MVVM

For implementing the MVVM pattern a third-party framework called Caliburn.Micro (Caliburn.Micro, n.d.) is used. A third-party framework is used because it allows faster development, and we don’t have to reinvent the wheel. Caliburn.Micro is a framework designed to build applications using the MVVM pattern. Caliburn.Micro is powerful and configurable framework. Many features of Caliburn.Micro such as the conventions increase development speed and allow to create cleaner looking code. Other popular and similar frameworks include Prism, MVVM Light and ReactiveUI. These frameworks have lots of similarities with each other as they all have been developed to implement the MVVM pattern.

DriveTest2UI.View is the view component of DriveTest2, and it contains UI elements of DriveTest2. UI in DriveTest2 is divided into different views as seen in Figure 14. Every separate window or tab in DriveTest2 is its own view. Views are done using WPF, so they have a XAML file and code-behind file. Because DriveTest2 uses MVVM, code-behind files should not contain any business or presentation logic as they should be in the model and ViewModel.

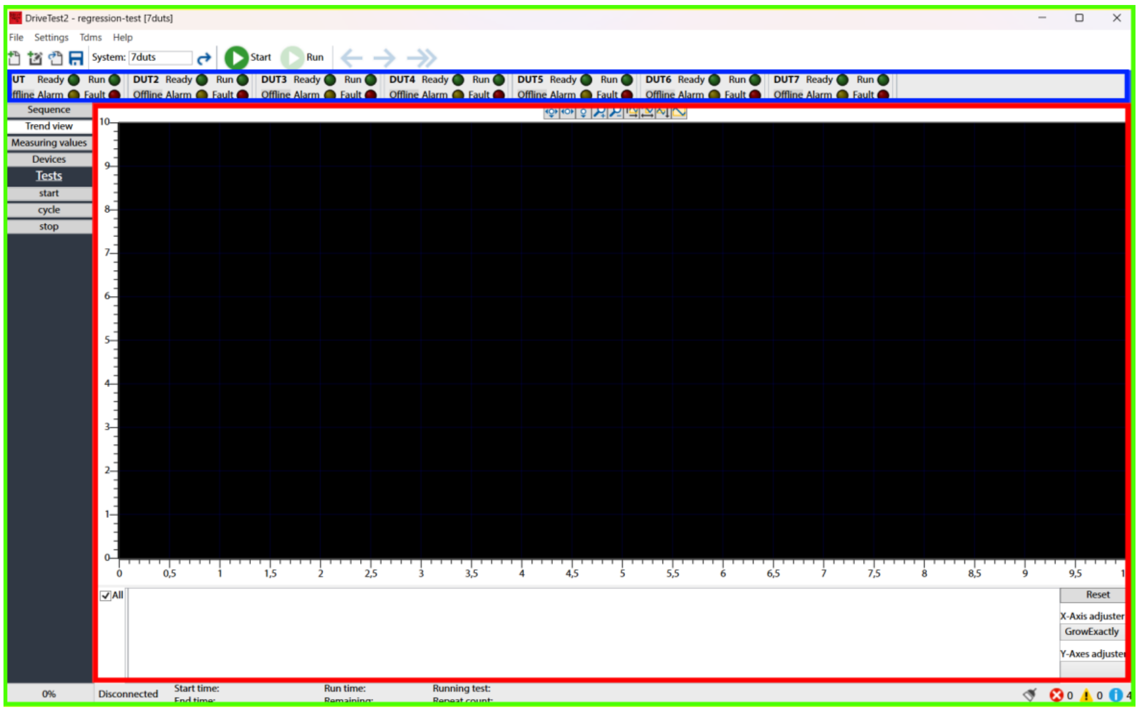


Figure 14. Example of how DriveTest2 UI is divided into different views. Green box is ShellView and inside the ShellView there is StatusBarView indicated by the blue box and TrendView indicated by the red box.

For every view in DriveTest2 there is corresponding ViewModel e.g., for ShellView there is a ShellViewModel. ViewModels contains properties that are databinded to UI elements in the views. Properties are databinded to display the value of the property in the UI e.g., showing the name of the sequence in the title or the name of a device in the status bar. When a property is databinded to UI element, the user can change value of the property in the UI. Functions in ViewModels can be called from the UI using commands e.g., Start- button calls a function in the ViewModel.

DriveTest2.Common is the model component of DriveTest2. It contains the business logic of DriveTest2. It is designed to be object-oriented. DriveTest2 is modelled into different classes. Figure 15 shows how classes in DriveTest2.Common are structured and what kind of relations they have. DriveTest2 uses interfaces to make the code loosely coupled.

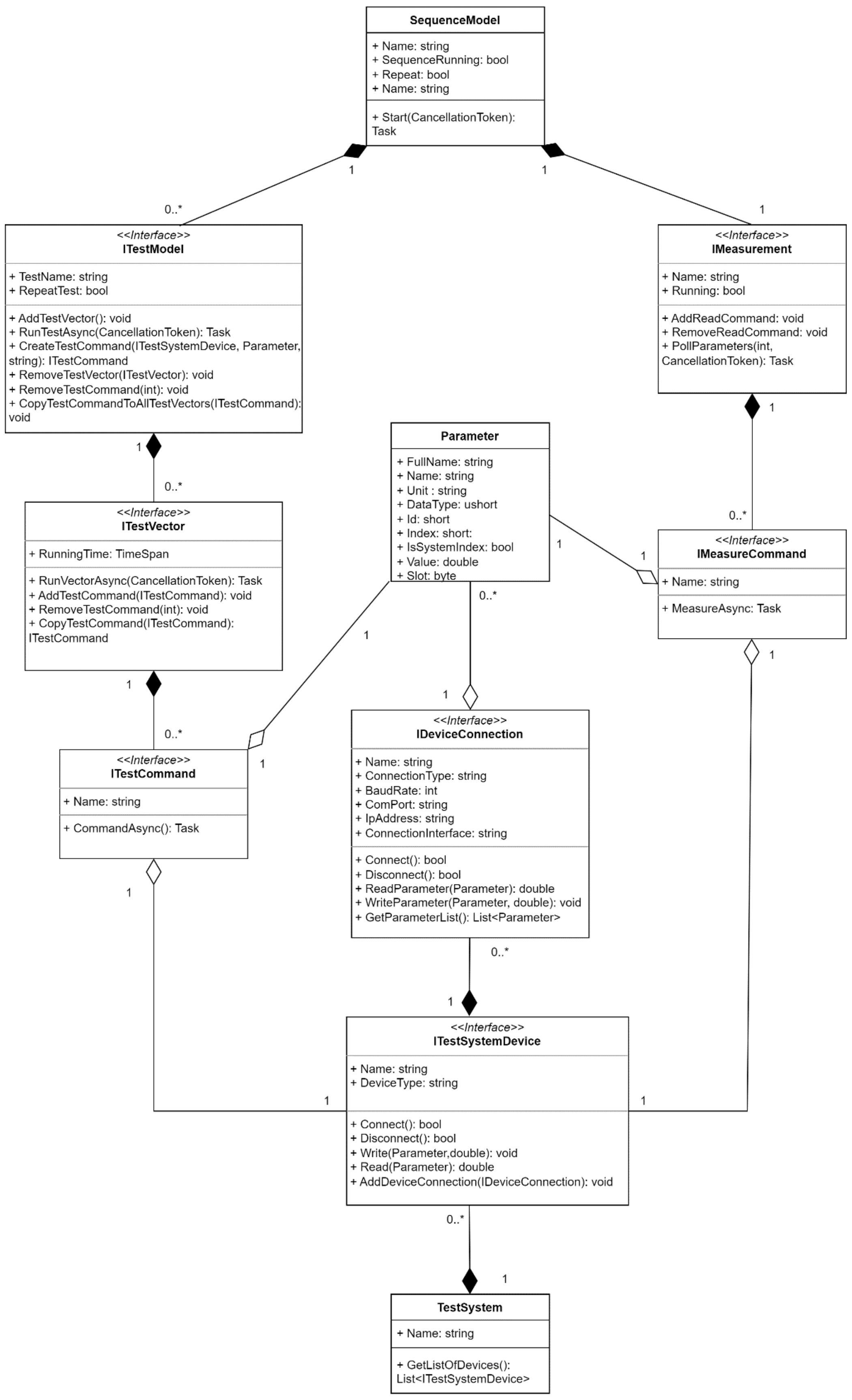


Figure 15. Class diagram of DriveTest2 initial architecture.

Highest level class of DriveTest2 is SequenceModel-class. SequenceModel as its name indicates models the sequences of DriveTest2. SequenceModel contains multiple objects that implement ITestModel-interface. DriveTest2 sequences are divided into tests that work as subsequences. Each test is then divided into test vectors. Test vectors are modelled by the ITestVector-interface. This design choice comes from DriveTest and allows to create matrix-like sequences with multiple test vectors with different running times but with same set of test commands. Running time in this case means the length of time waited before going to the next test vector.

SequenceModel has an IMeasurement- implementation that is used to handle measurement and logging during the sequence execution. IMeasureCommands are used to interact with ITestSystemDevice device to read parameters or get measurements from the device. Figure 16 shows a flowchart of how the measurement is done. It is a simple loop where after measuring values or reading parameters from devices, those values then get saved to graph data so that data can be shown in a graph and into a logging buffer. If the buffer is not full yet, the loop starts from the beginning. When the buffer is full its contents are then logged into the TDMS file, buffer is emptied and then the loop starts again after waiting *n* seconds which the user can specify.

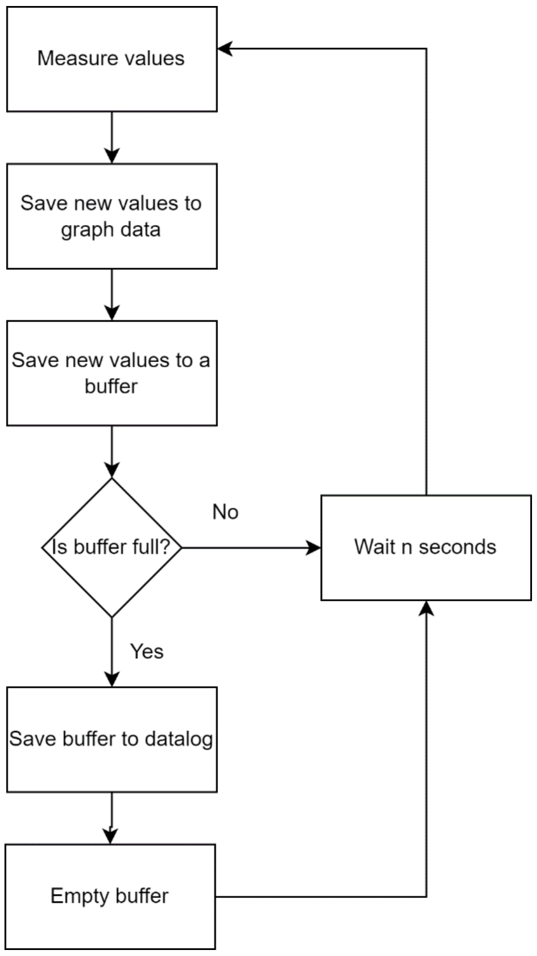


Figure 16. Flowchart of the measurement and logging loop

Sequence executes tests and measures concurrently. This can be seen in Figure 17. When sequence is started it will start two tasks: Run tests and measurement and logging. Measurements will be made and logged while tests are running. After all tests have been finished or the user has stopped the execution, then measurements will be stopped. To make sure that these two tasks do not freeze UI or affect each other, they are run asynchronously using C# async/await which queues these two tasks to the thread pool.

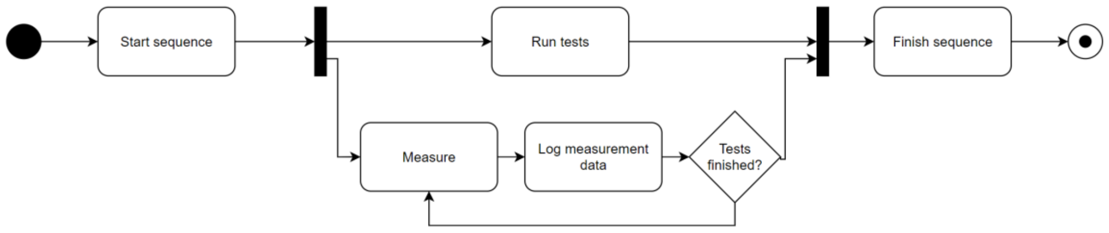


Figure 17. Activity diagram of sequence execution

Since test execution and measurements can both try to use same device connection at the same time there is a need to synchronize resources that cannot handle multiple threads accessing them at the same time. These resources, device connections in our case, are synchronized by locking them when someone is using them. This makes it so that the resource is blocked for all others when it is being used. After the resource is released whoever tries first can now access the resource.

DriveTest2 has a TestSystem- class that contains all devices of the test setup. Device classes implement ITestSystemDevice- interface that can contain one or more IDeviceConnections. IDeviceConnection is an interface for communicating with the actual device. Different IDeviceConnection implementations use different drivers and communication methods like serial and ethernet communication. The reason why a device can have multiple connections is simply because some devices support having different kinds of connections to it at the same time and this functionality is a requirement for DriveTest2. With ITestSystemDevice and IDeviceConnection- interfaces different devices and connections can be supported while having loosely coupled code.

5.3 Refinements through iterations

In this chapter we will look at what kind of refinements were made to DriveTest2 architecture and design throughout iterations. We will be looking at what kind of feedback we got from iterations and how the feedback affected future iterations.

Feedback for the iterations has come mainly from the users of DriveTest2. Each iteration of DriveTest2 is given to users of DriveTest2 and they test it out and give feedback. Feedback was often about UI, bugs, missing functionalities, new functionalities, or current implementation of functionalities. There was not always feedback on each iteration if the users did not have time to test the iteration. In cases like those the next iteration could have implementations of missing functionalities, improvements to previous implementations or if previous feedback was still valid then they were taken into consideration for the next iteration. According to the feedback the architecture and designs are refined if needed and a new iteration is developed. Iterations range from being a bunch of bug fixes or UI changes to changes to an existing functionality or a new functionality altogether.

5.3.1 Architectural and design changes

One design that was changed the most during iterations is how the measurement and logging was done. At first it was simple loop where after getting measurements from the devices, the measurement data was given to the graph and data log buffer. This was explained earlier and shown in Figure 16. This way measured devices consecutively and did not contain a way to stop the measurement loop.

First refinement to the measurement and logging was to get measurement data from devices concurrently. This was done creating asynchronous tasks for each device where measurement data was acquired. After each data acquisition task is completed, measurement data are combined and handed out to the graph and data log buffer. Flowchart of this can be seen in Figure 18. Another addition to the measurement and logging flowchart is the check for if measurement should continue or not. Measurement should be stopped when tests have been finished or if sequence has been stopped manually.

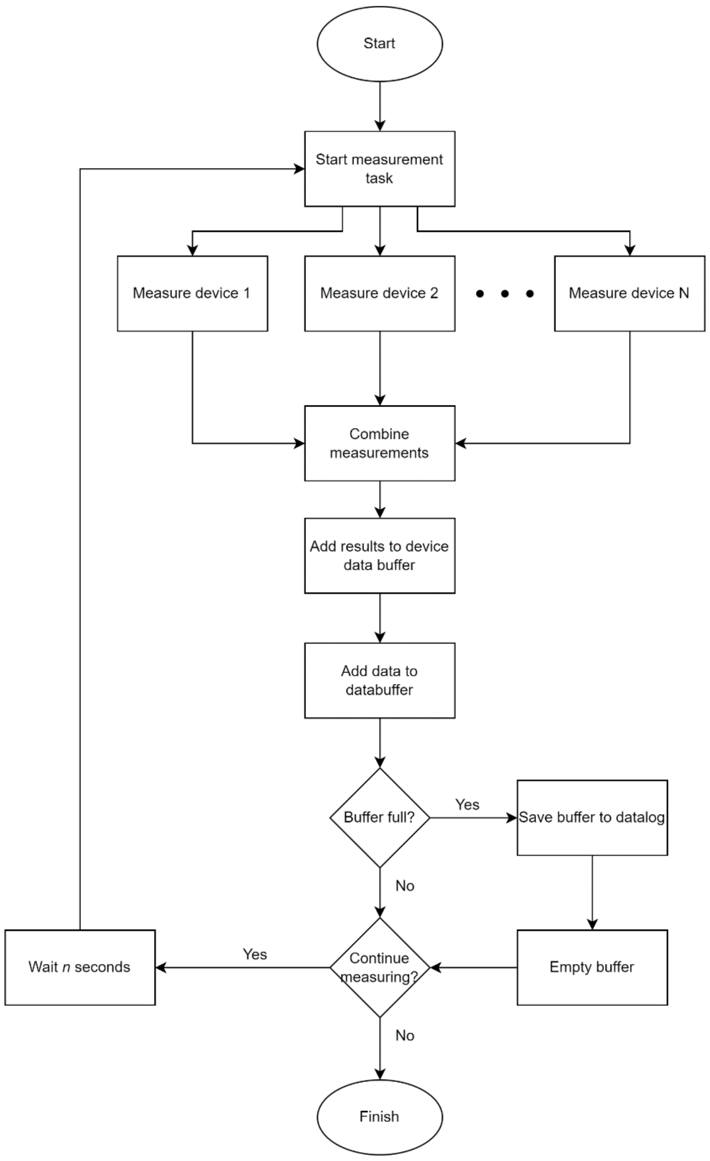


Figure 18. Improved measurement and logging with concurrent data acquisition and check for if the loop should continue.

This was a working design until there was some feedback, and couple requirements were created. There was need for device specific wait times between measurements meaning that device A could wait for 1 second before measuring again while device B could wait for 5 seconds. This was due to some measurements changing values slowly meaning that there was simply no need to measure often. The other requirement was that there was no need to log measurement data each time that measurement data was acquired.

There was similar reasoning as to the device specific wait times. Some measurement where very static so there was no need to log every time they were measured.

This led to each device having their own measurement and logging loops. This change can be seen in Figure 19. Instead of concurrently acquiring measurement data from each device and then combining them seen in the left side of the figure, now there are measurement and logging loops running for each device in parallel seen on the right side of the figure. This allows each loop to have different waiting times before the next loop iteration.

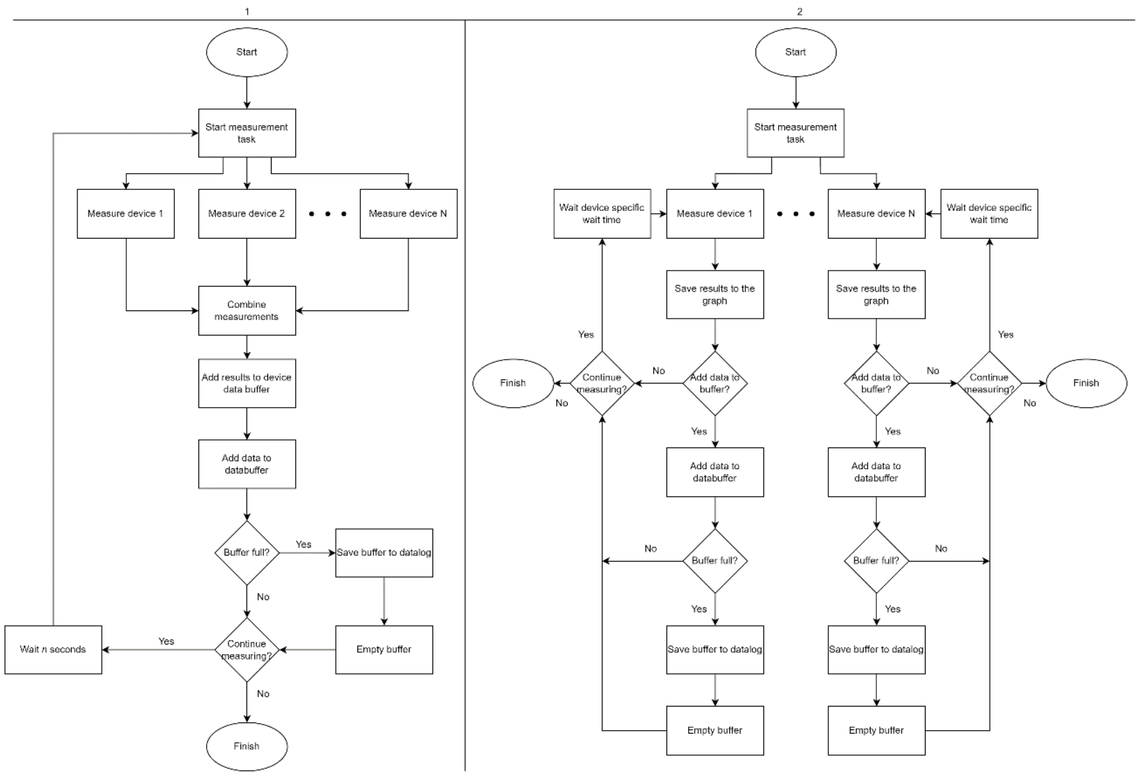


Figure 19. Change from to single loop with concurrent device measurements (1) to parallel device measurement and logging loops (2).

This also makes it that each device has their own data log buffer. Previously measurement data was logged to one place but now measurement data is divided by each device. This makes it possible to log measurement data in different intervals and measurement data from other devices do not affect each other.

In each loop operation there is a check whether the measurement data is to be added to the buffer. This implements the requirement of not always needing to log the data. This way we can log the data once every *K* iterations e.g., if the measurement and logging iterations are executed once per second but logging is done once every five iterations, it will make the logging happen once per five seconds.

Another measurement related change was the addition of test specific measurements. Initially measurements were defined for the whole sequence, but tests differ from each other and might not need the same measurements. This led to each test having their own measurements and the sequence having its own measurements. Seen on Figure 20 the ITestModel- interface now contains one IMeasurement implementation and a new boolean property which is used to indicate if the test should use its own measurements or sequence measurements.

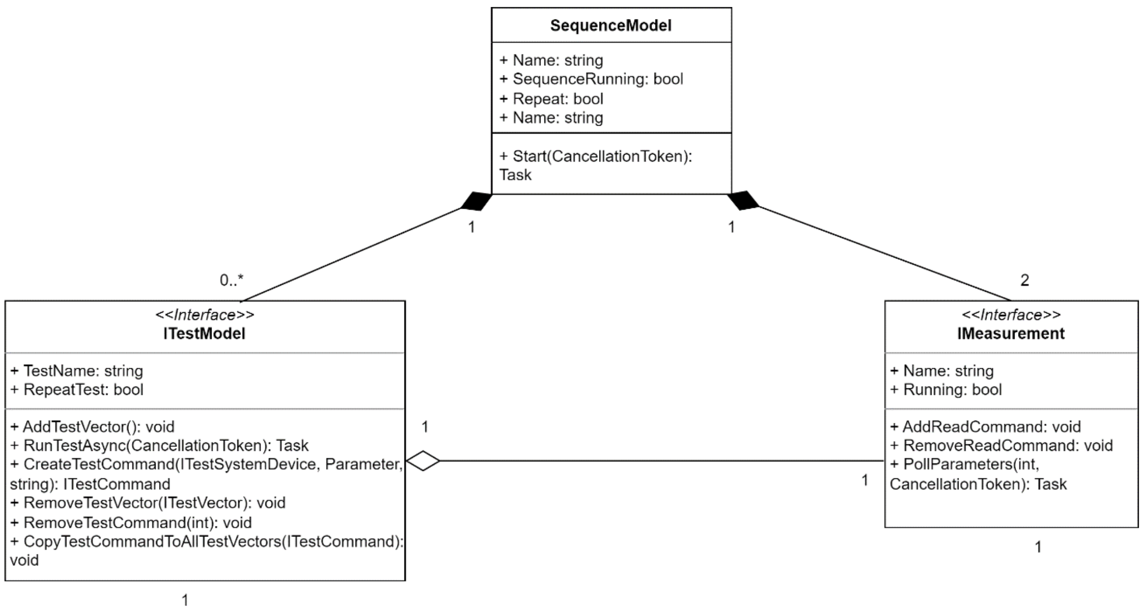

SequenceModel
+ Name: string
+ SequenceRunning: bool
+ Repeat: bool
+ Name: string
+ Start(CancellationToken): Task
1
1
0..*
2
<<Interface>>
ITestModel
+ TestName: string
+ RepeatTest: bool
+ AddTestVector(): void
+ RunTestAsync(CancellationToken): Task
+ CreateTestCommand(ITestSystemDevice, Parameter, string): ITestCommand
+ RemoveTestVector(ITestVector): void
+ RemoveTestCommand(int): void
+ CopyTestCommandToAllTestVectors(ITestCommand): void
1
1
1
<<Interface>>
IMeasurement
+ Name: string
+ Running: bool
+ AddReadCommand: void
+ RemoveReadCommand: void
+ PollParameters(int, CancellationToken): Task
1

Figure 20. Addition of IMeasurement-object to ITestModel

Not only does having test specific measurements allow unique measurements for each test but this allows tests to have their own limit actions for measurements. Limit actions are a functionality of having upper and lower limits for measurements which when crossed will trigger a limit action to happen. Limit actions range from moving to a next vector or to next test to stopping the sequence. Limits are checked for every time measurements are done. Limits and limit actions are added straight to the DeviceParameter-class seen in Figure 21.

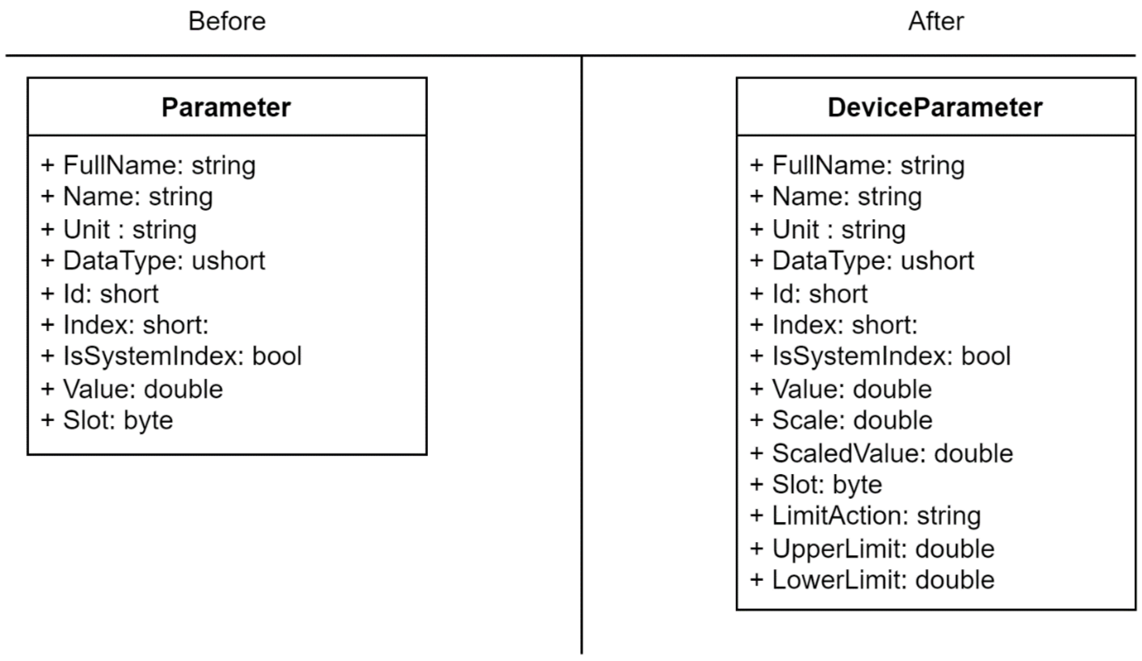


Figure 21. Before and after of DeviceParameter- class

Previously DeviceParameter was called Parameter, but this was changed to make it little bit more specific. Not only did DeviceParameter get LimitAction, UpperLimit and LowerLimit properties it also got Scale and ScaledValue properties. This is due to some devices returning values that do not contain decimals e.g., a device can return value of 1000, but it is in fact 10,00. This requires the scaling of the raw values returned by devices if needed.

5.3.2 New functionalities

Using the Interface Segregation Principle, some of the ITestSystemDevice- implementations were extended with new functionalities. This was done by creating new interfaces such as IStatusWord which contains status related properties that only frequency converters support. Because not all devices support this feature, an interface was created for it and only the devices that support this feature implement it. There are other similar interfaces that contain functionalities that only specific devices support so only those device classes then implement these functionalities. In Figure 22 we can see how DAQ- class implements IDaqChannels and ITestSystemDevice- interfaces but does not implement IStatusWord while Amn- class implements IStatusWord but not IDaqChannels. Using interfaces like this we can extend classes while following Interface Segregation Principle meaning that classes do not need to depend on properties and methods that they do not use or support in this case.

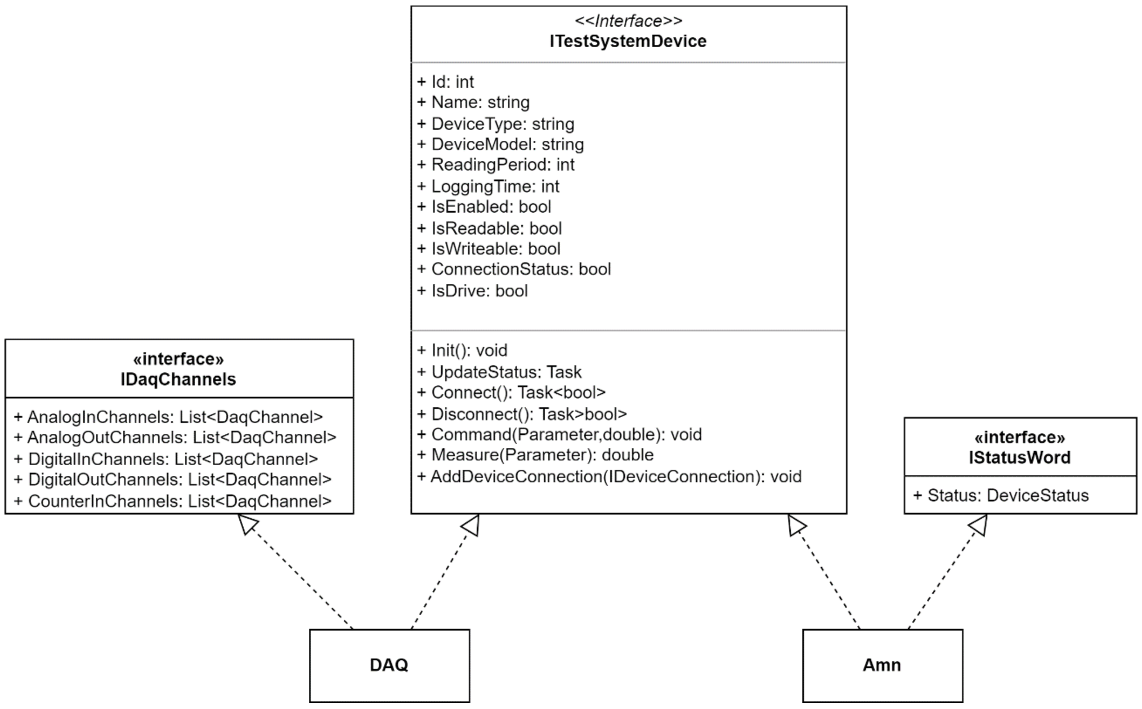


Figure 22. Example on how ITestSystemDevice implementations inherit different interfaces.

During the development of DriveTest2 new functionalities were requested by the users to implement tests where load of the load motor is controlled using a PID controller. There are test setups where there are two motors connected to each other and are controlled by different frequency converters. This can be seen in Figure 23.

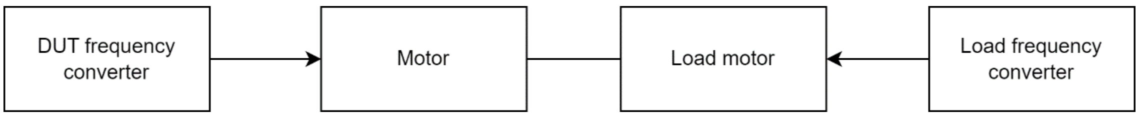


Figure 23. Two connected motors with their own frequency converters.

One of the motors is a load motor that is used to generate load for the other motor so that. This motor is controlled by the load frequency converter which can be controlled with DriveTest2. To control the load of the load motor, a control system that used a PID controller was added to DriveTest2. This control system can control the load by using a parameter from the DUT or measurements from recorder as a process value. The user can give a setpoint for the process value and DriveTest2 controls the load so that the measured process value will reach the setpoint and stay on the setpoint. Control systems are added to tests and each test can have one.

Figure 24 shows the changes to ITestModel class diagram. ITestModel interface is extended with a Derating- object and an object that implements IControlSystem. IControlSystem uses a PIDController- object to get an output that is used calculate a new value for the load frequency converter. IControlSystem and Derating both contain asynchronous loops which are called when the test is started and are stopped when the test has been executed.

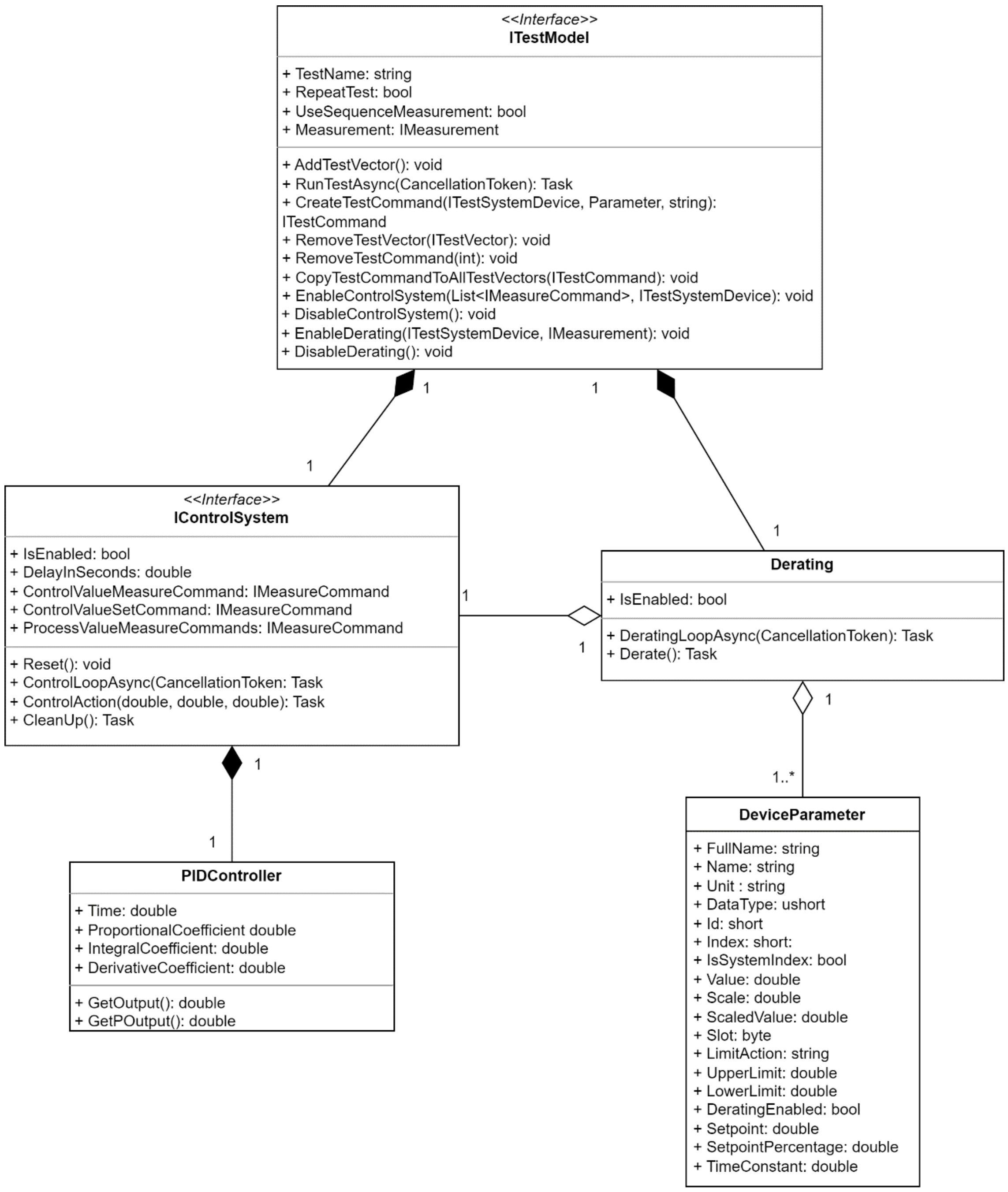

<<Interface>>
ITestModel
+ TestName: string
+ RepeatTest: bool
+ UseSequenceMeasurement: bool
+ Measurement: IMeasurement
+ AddTestVector(): void
+ RunTestAsync(CancellationToken): Task
+ CreateTestCommand(ITestSystemDevice, Parameter, string): ITestCommand
+ RemoveTestVector(ITestVector): void
+ RemoveTestCommand(int): void
+ CopyTestCommandToAllTestVectors(ITestCommand): void
+ EnableControlSystem(List<IMeasureCommand>, ITestSystemDevice): void
+ DisableControlSystem(): void
+ EnableDerating(ITestSystemDevice, IMeasurement): void
+ DisableDerating(): void
1
1
1
1
<<Interface>>
IControlSystem
+ IsEnabled: bool
+ DelayInSeconds: double
+ ControlValueMeasureCommand: IMeasureCommand
+ ControlValueSetCommand: IMeasureCommand
+ ProcessValueMeasureCommands: IMeasureCommand
+ Reset(): void
+ ControlLoopAsync(CancellationToken: Task
+ ControlAction(double, double, double): Task
+ CleanUp(): Task
1
1
Derating
+ IsEnabled: bool
+ DeratingLoopAsync(CancellationToken): Task
+ Derate(): Task
1
1
1..*
1
PIDController
+ Time: double
+ ProportionalCoefficient double
+ IntegralCoefficient: double
+ DerivativeCoefficient: double
+ GetOutput(): double
+ GetPOutput(): double
DeviceParameter
+ FullName: string
+ Name: string
+ Unit : string
+ DataType: ushort
+ Id: short
+ Index: short:
+ IsSystemIndex: bool
+ Value: double
+ Scale: double
+ ScaledValue: double
+ Slot: byte
+ LimitAction: string
+ UpperLimit: double
+ LowerLimit: double
+ DeratingEnabled: bool
+ Setpoint: double
+ SetpointPercentage: double
+ TimeConstant: double

Figure 24. Class diagram of ITestModel with IControlSystem and Derating.

Derating also controls the load of the load motor, but it differs slightly to the IControl-System. Derating is about controlling the load of the load motor according setpoints given to multiple measurements. This way we can lower the load if measurements gets close to its given setpoint. The load is controlled according to the measurement that is closest to its setpoint percentually. DeviceParameter is extended with four new properties: DeratingEnabled, SetPoint, SetpointPercentage and TimeConstant. In each derating loop iteration DeviceParameters are checked for which one has the highest SetpointPercentage which is the percentage of ScaledValue from Setpoint. Then this DeviceParameter is used in the PIDController and IControlSystem to control the load motor load. TimeConstant of DeviceParameter tells how often a control should happen if that DeviceParameter has the highest SetpointPercentage. E.g., if measuring component temperatures and some components warm up slowly, there is no need to control the load so often if such component is the one that is used to control the load.

6 Architecture of a test sequencer

In this chapter we will look at the architecture of DriveTest2 to analyse and evaluate it. Architectural decisions will be analysed by looking at why that decision was chosen, what kind of pros and cons it has and compare it to prior literature. The architecture will be evaluated based on how it manages to enable DriveTest2 to satisfy its critical functional requirements and non-functional requirements. Lastly, we will analyse how DriveTest2 works as a solution for the testing needs and if it can replace DriveTest.

Critical functional requirements are the following:

- Sequence creation
- Sequence execution
- Sequence monitoring
- Sequence result handling

Critical non-functional requirements are the following:

- Stability
- Reliability
- Maintainability
- Usability
- Concurrency
- Extensibility

The first architectural decision to look at is the decision to use the MVVM pattern. MVVM divides the system into three components that can be seen in Figure 25: View, View-Model and Model. This separates the UI from the business logic. Having the ViewModel to mediate between the model and view allows the ViewModel to convert the data of the model in different ways. This makes it possible to use the same data to make different views. Separating the UI from the business logic also allows developing them on their own. Faraj et al.,(2019) also have their UI separated from as it is a separate plugin in their plugin architecture.

MVVM tackles a couple of the problems that DriveTest had which are tightly coupled UI and business logic, testability, and maintainability. Maintainability is one of the critical non-functional requirements of DriveTest2. Dividing DriveTest2 into components that have clear responsibilities makes it easier to maintain the code. MVVM makes the system more testable because components can be tested separately. With ViewModels the presentation logic can also be tested, which in most cases is not possible or is hard to do. With better testability and clear separation of UI and business logic, maintaining the code becomes easier.

MVVM pattern also makes it easier to make the UI responsive and easy to use for user because we can abstract the data in different ways in ViewModels so that the data is shown in an intuitive way in UI. Making it possible to also change the UI without having to worry about it affecting the business logic makes it easier to develop the UI.

MVVM can make the code more complex and there will be more classes and components due to having a ViewModel for each view. Debugging can also be bit harder when using databinds and notifications but the testability and maintainability that the MVVM offers outweighs these cons.

MVVM components of DriveTest2 are divided into two separate projects: DriveTest2UI and DriveTest2.Common. This is seen in Figure 25. DriveTest2UI contains view and View-Model components while DriveTest2.Common contains the model component. Dividing them into two projects like this further decouples the model from the view. Having a separate project for the model means that the model does not need to know anything about the view. If they were in the same project, they would be contained in the same assembly thus would be coupled in a way.

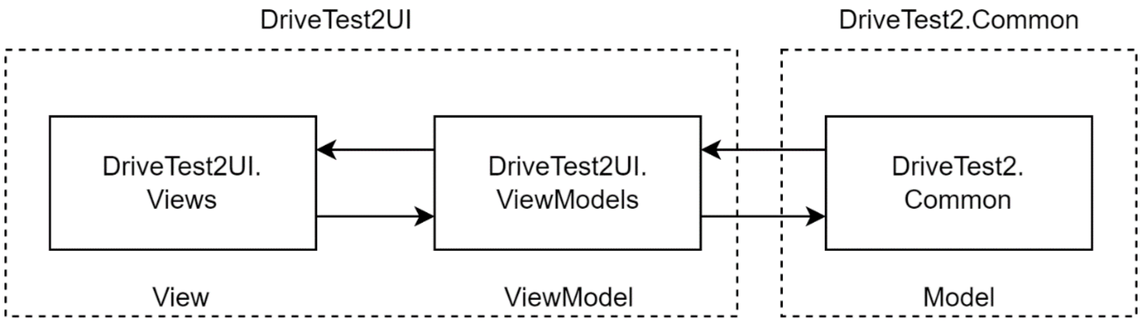


Figure 25. DriveTest2 MVVM components divided into two projects.

DriveTest2 is divided into DriveTest2UI, DriveTest2.Common and multiple projects that are under DriveTest2.DeviceConnections. As seen on the Figure 26 DriveTest2.Common does not reference other projects at all instead other projects references the DriveTest2.Common. This way DriveTest2.Common does not have to know about the implementations of IDeviceConnection which are in their own projects in DriveTest2.DeviceConnections. DriveTest2UI contains the UI of the DriveTest2 and the executable. Because of this DriveTest2UI needs to reference DriveTest2.DeviceConnections to get the IDeviceConnection implementations.

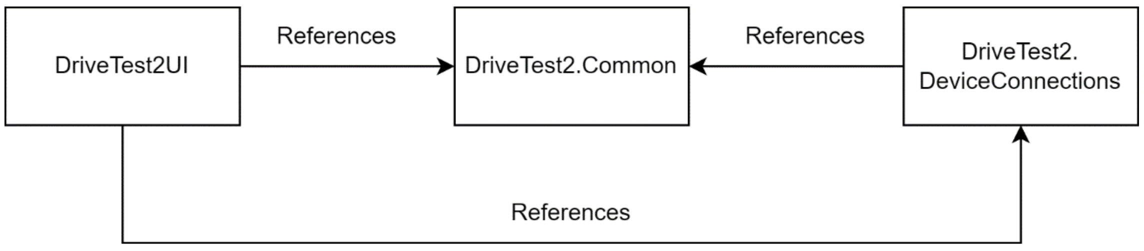


Figure 26. DriveTest2 projects and how they reference other projects.

Next, we will look at the model component of DriveTest2 which consists of classes and interfaces. Classes and interfaces will be looked at as their own class diagrams instead of looking at the full class diagram that contains everything. The full class diagram can be seen in appendix 1. Main classes and interfaces and their descriptions can be seen in Table 5.

Table 5. Classes and interfaces of DriveTest2 model with descriptions

Class or interface	**Desciption**
SequenceModel	Top class of the model that contains list of ITestModels and IMeasurement. Contains methods to start and stop sequence
ITestModel	A subsequence that consists of ITestVectors, IControlSystem, Derating and IMeasurement
IControlSystem	Controls load of a load motor using a PID controller
PIDController	Calculates outputs for IControlSystem
Derating	Uses a IControlSystem to perform a derating run
ITestVector	Part of ITestModel, contains a running time and list of ITestCommands
ITestCommand	Commands a device
IMeasurement	Handles measuring data from devices using IMeasureCommands
IMeasureCommand	Measures a parameter from a ITestSystemDevice
DeviceParameter	Contains information about a device parameter and is used for commanding and measuring from the device
ITestSystem	Collection of all devices in the test setup
ITestSystemDevice	Abstraction of a device and uses IDeviceConnections to communicate with the device
IDeviceConnection	Handles communication with the actual device

DriveTest2 model contains a lot of interfaces. This is so that we can abstract the lower-level components so that the higher-level code does not need to know about the implementation and is only interested in what the interfaces offer. This helps decoupling the code and allows to create multiple implementations per interface. Currently in most cases there is only one implementation but having interfaces allows there to easily be new implementations in the future and makes future refactoring easier. Other pros of interfaces include easier unit testing since we can just use mock implementations for interfaces when testing and having loosely coupled code makes it easier to maintain.

Let’s look at the top class of DriveTest2 model which is the SequenceModel class. Class diagram of SequenceModel is seen in Figure 27 Among the critical functional requirements of DriveTest2 were creating and editing sequences, executing sequences and monitoring them. These three can be satisfied with the SequnceModel. Creating and editing a sequence is handled in the UI of DriveTest2. Users can create a SequenceModel object and modify it with databinds or method calls. User can save a sequence by serializing it into a file like JSON-file and then that same file can be used to load the sequence.

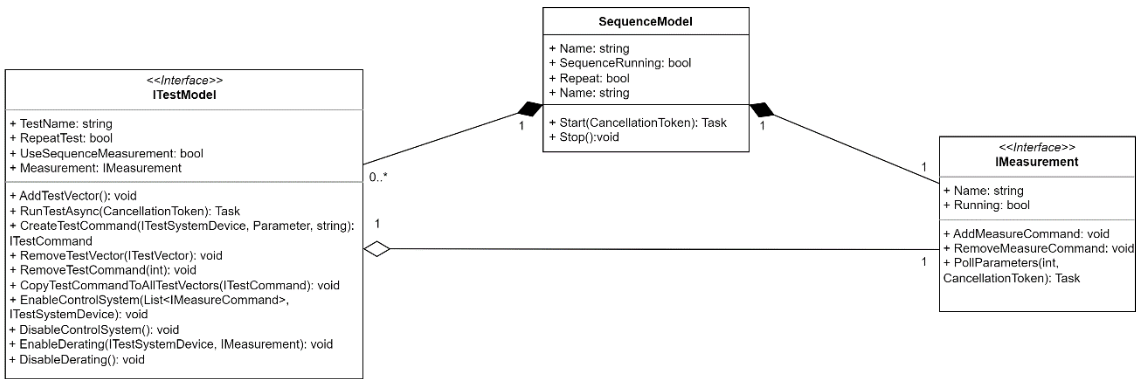

SequenceModel
+ Name: string
+ SequenceRunning: bool
+ Repeat: bool
+ Name: string
+ Start(CancellationToken): Task
+ Stop():void
<<Interface>>
ITestModel
+ TestName: string
+ RepeatTest: bool
+ UseSequenceMeasurement: bool
+ Measurement: IMeasurement
+ AddTestVector(): void
+ RunTestAsync(CancellationToken): Task
+ CreateTestCommand(ITestSystemDevice, Parameter, string): ITestCommand
+ RemoveTestVector(ITestVector): void
+ RemoveTestCommand(int): void
+ CopyTestCommandToAllTestVectors(ITestCommand): void
+ EnableControlSystem(List<IMeasureCommand>, ITestSystemDevice): void
+ DisableControlSystem(): void
+ EnableDerating(ITestSystemDevice, IMeasurement): void
+ DisableDerating(): void
<<Interface>>
IMeasurement
+ Name: string
+ Running: bool
+ AddMeasureCommand: void
+ RemoveMeasureCommand: void
+ PollParameters(int, CancellationToken): Task
1
0..*
1
1
1
1

Figure 27. SequenceModel class diagram

SequenceModel contains multiple ITestModel objects which are subsequences that the sequence consists of. Then there is an IMeasurement object that handles measurement and monitoring during the sequence execution. ITestModels contain an IMeasurement object so that there can be test specific measurements if needed. When a sequence is being executed ITestModels and IMeasurement are running concurrently. This makes it possible to have real time monitoring with measurements and being able to execute the tests at the same time. Separating tests and measurement into separate interfaces allows each interface to be responsible for one this.

ITestModels are what the sequences are made of. Its class diagram is seen in Figure 28. ITestModel can be divided into three parts. ITestVectors, IControlSystem and Derating. ITestVectors are building blocks that create test is made of while IControlSystem and Derating are an additional functionality for a test. A test usually executes the ITestVectors that it has but it can also use an IControlSystem or Derating- objects to control the load of a load motor during the test. They all use ITestCommands to command ITestSystemDevices. ITestCommand contains the ITestSystemDevice that it wants to command and a DeviceParameter that specifies the command or the parameter that it wants to write. IControlSystem also uses IMeasureCommands to get measurement data from the ITestSystemDevice.

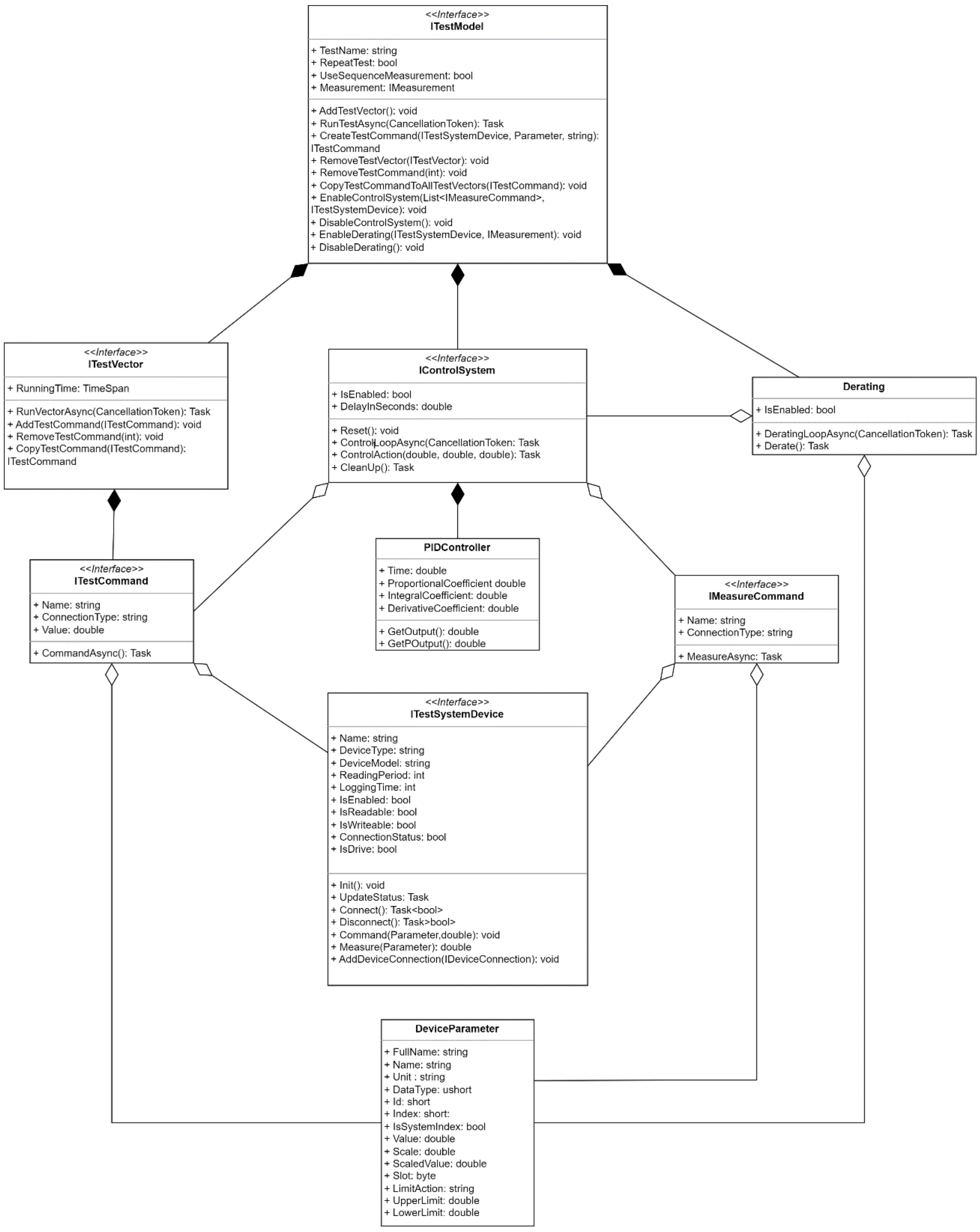


Figure 28. Class diagram of ITestModel

ITestModel, ITestVector and ITestCommands are the elements that together create a sequence. Users can create and manipulate these elements in the UI to create tests with the help of databinding. Instead of having test scripts like how Keimiling et al., (2013) have in KROHNE TestCenter, ITestVectors are more like test steps that Faraj et al., (Faraj et al., 2019) use in their test system. Without the need to use a script language to create tests, users can create tests in the UI which is easier and simpler with the downside of not being able to create complex tests. This approach was chosen because DriveTest2 is meant to be easy to use without the need for any scripting or programming experience.

Two important features of DriveTest2 are the ability to monitor the sequence execution by polling parameters and measuring data from devices during the sequence and then being able to see this same data after the sequence has been executed. These two functionalities are handled by IMeasurement. IMeasurement is the interface that is responsible for acquiring data from devices, holding the data, and logging them to a data log file. Users can select parameters that they want to be polled and logged during the sequence execution. Polled parameter values can be seen in a graph during sequence execution. After the sequence has been finished the measurement data is in the data log file and can be analysed. The data log works as a test report as it contains information about the executed sequence, test setup and data gathered during the sequence execution.

IMeasurement uses IMeasureCommands to read data from the devices. This is seen in the IMeasurement class diagram in Figure 29. Like ITestCommands, IMeasureCommands contain the device that they want to interact with and DeviceParameters that they want to get values for. If ITestCommand is used for commanding the device and writing parameters, IMeasureCommand if used for reading parameters from the device. When an IMeasureCommand gets data from the ITestSystemDevice, the data is saved in the DeviceParameters. There are one or multiple IMeasureCommands per device depending on the fact whether the device supports reading multiple parameters at once or not.

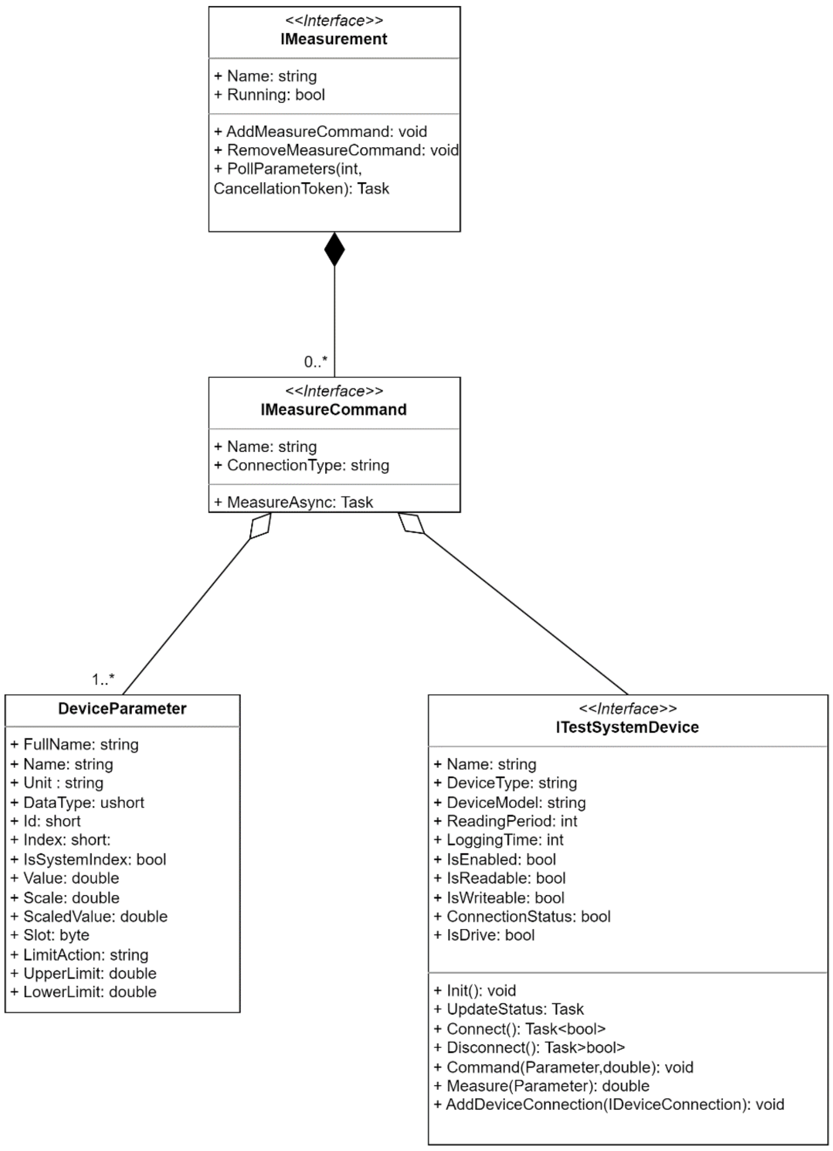


Figure 29. Class diagram of IMeasurement.

TestSystem in DriveTest2 represents the devices of the test setup that are connected to with DriveTest2. Its class diagram is seen in Figure 30. ITestSystemDevice interface is the abstraction of a device controlled by DriveTest2. This interface is used by ITestCommand and IMeasureCommand to interact with the device. ITestSystemDevice uses one or more IDeviceConnections to communicate with the device. IDeviceConnections use the communication protocols that devices support like serial, Transmission Control Protocol (TCP) or User Datagram Protocol (UDP) and they implement functions like Connect(), Disconnect(), Command(), and ReadParameter() that the ITestSystemDevice can call.

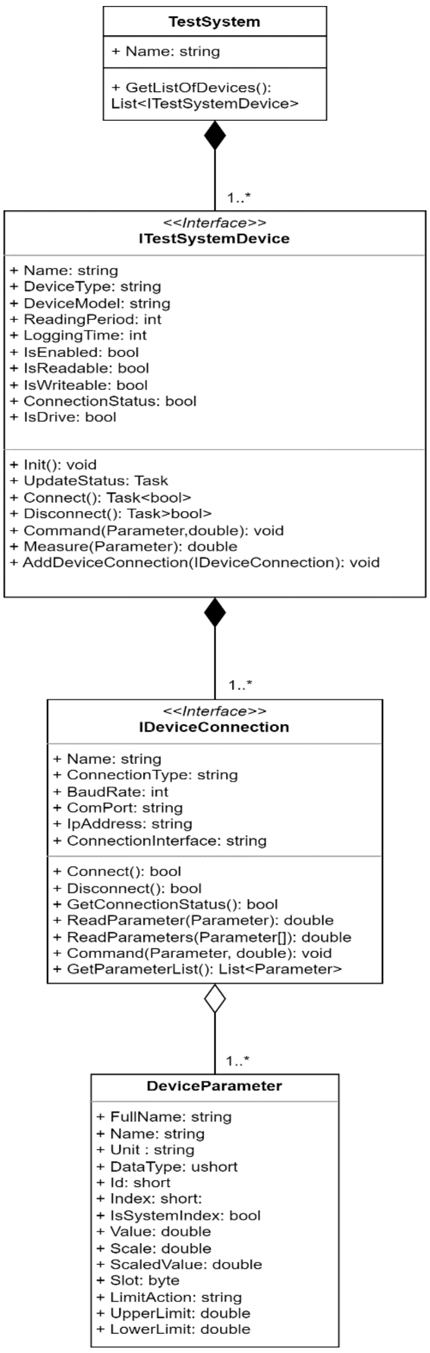


Figure 30. Class diagram of TestSystem.

Then there is the DeviceParameter- class that represents a parameter, channel or a command that is used to specify which parameter is to be read from the device or what command is to be given to the device. IDeviceConnection returns all possible DeviceParameter that the device has, and these are then used by ITestCommands and

IMeasureCommands. DeviceParameter contains upper and lower limits that are checked for when it gets a new value. If either of these limits are crossed the DeviceParameter triggers a LimitAction. Limits and the LimitAction are set by the users and can be used e.g., skipping to the next test, or stopping the sequence altogether.

If we compare DriveTest2 classes to ATS software architecture layers, we can see some similarities. Like ATS layers, DriveTest2 classes and interfaces can be divided into hardware abstraction and hardware dependent elements. This division can be seen in Table 6.

Table 6. ATS layers compared to DriveTest2 classes and interfaces.

	ATS	**DriveTest2**
Hardware abstraction	Test Executive	
	Test Program Set	SequenceModel
	Tests	ITestModel
	Instrument Classes	ITestSystemDevice
Hardware dependant	Instrument Drivers	IDeviceConnection
	VISA Layer	
	Communication protocols	
	Instruments	Device

Elements like SequenceModel in DriveTest2 or Test Program Set in ATS use abstractions of the hardware. They do not need to know about the actual hardware, and this decouples them from the implementation and the hardware. This makes it possible to create a sequence or a test using device A and then use the same sequence or test with device B. Abstraction makes code more maintainable because it is loosely coupled and does not depend on so many things. If the code is tightly coupled changing things will break code that depends on the code that gets changed.

Non-functional requirements are very important and software architecture affects greatly how well they are satisfied. Software architecture must be developed with them on the mind. Table 7 contains small recap of non-functional requirements of DriveTest2 and how they are achieved.

Table 7. Table of non-functional requirements and architecture elements to achieve them.

Non-functional requirement	**Architecture**
Stability	Loosely coupled code, interfaces, and single responsible components
Reliability	Loosely coupled code, interfaces, and single responsible components
Maintainability	Abstraction, MVVM pattern and interfaces
Usability	ViewModels can abstract the model which allows more flexible UI development
Concurrency	Test execution and measurements are executed concurrently with the use of asynchronous programming
Extensibility	Classes and interfaces allow extensions

Most of the non-functional requirements are achieved with interfaces, single responsible components and loosely coupled code. These are same things that the SOLID principles try to convey. DriveTest2 software architecture tries to uphold SOLID principles to be maintainable, stabile, testable and extensible. Then there is the MVVM pattern that also makes code more maintainable due to it helping with decoupling components and it also helps UI development which helps with usability.

Software architecture and requirements go together. Requirements define what the software should be capable of while architecture defines the software in a way that it can satisfy those requirements. DriveTest2 software architecture fulfils those requirements. Software architecture must be designed and developed well so that it can enable the software to satisfy its requirements. If the software can’t implement functionalities due to lacking architecture, the architecture must be changed or developed more.

DriveTest2 has been tested and is currently used in cyclic and load testing. Although not all functionalities have been implemented to DriveTest2, it is still usable because the core functionalities have been implemented. In the future DriveTest2 is to be extended so that it can possibly be used for even more tests and to support more devices. With the use of interfaces supporting more devices is made simple.

Compared to DriveTest, DriveTest2 is more maintainable and testable. Using interfaces to make the code loosely coupled makes it more maintainable because components do not depend on other components if there is no need. With interfaces the high-level components do not have to know about low-level implementation when they can just use methods that the interfaces offer.

7 Conclusions

In this work a test sequencer called DriveTest2, and its software architecture were developed. The development was done using an iterative process where each iteration builds upon previous iterations. The process started from requirements analysis which was done in a previous work. From the gathered requirements an initial architecture and design for DriveTest2 were created. Then the first iteration was created based on the initial architecture and design. Iterations were given to users to get feedback on them and according to the feedback and requirements changes the architecture was refined to create the next iteration. The iterative development process allowed to swiftly react to feedback and any requirement changes.

The aim of this study was to find out what kind of software architecture a test sequencer should have. A test sequencer is a test automation tool that is used to create test sequences to test devices. A test sequencer needs to be able to control different devices in test setups to execute created test sequences. It can be used to monitor the sequence execution and to handle test data after the execution. These are the core functionalities that a test sequencer has.

When developing an architecture, you need to take both functional and non-functional requirements into account. The architecture must be designed and developed in a way that it can enable the implementations to satisfy their required functionalities while satisfying the non-functional requirements. As was expected, a test sequencer should have a software architecture that can enable it to implement required functionalities in a way that is specified in the non-functional requirements as well as possible. Since requirements can change during development, the architecture needs to evolvable. The architecture must be flexible enough to accommodate changes to or easily extensible so that new or changed requirements can be satisfied.

Another thing that the software architecture should consider is the separation of responsibilities. Architecture should be designed in a way that separates components based on their responsibilities. E.g., UI should be separated from business logic. These components should be decoupled from each other to make them more maintainable and testable.

Then there is the use of abstraction. Abstraction can be achieved e.g., with use of interfaces. Abstraction is important in a test sequencer especially when abstracting the hardware. Higher-level components like the test sequence should not have to depend on the implementation of hardware dependent components like drivers or communication protocols. Abstracting the hardware with an interface makes this possible and allows having many different hardware dependent components that the higher-level components can use without needing to know how they are implemented. Hardware abstraction makes the code loosely coupled thus making it more maintainable and reliable.

DriveTest2 architecture uses object-oriented programming and the model of DriveTest2 has similarities to the model of ATS framework developed by Xu et al., (2003). DriveTest2 has many similarities to ATS like abstracting hardware and having test sequences. ATS and other automatic test setups (Keimling et al., 2013) use tests created in programming languages and specific test equipment while DriveTest2 tests are created graphically and are simple commands to test setup devices. DriveTest2 allows easier creation of sequences but compared to other solutions it does not support creating complex test cases. This is due to DriveTest2 being designed to be used by users without programming background. DriveTest2 supports real-time monitoring of test execution that was not part of the related works architecture. This makes the DriveTest2 architecture differ a bit from the other solutions as it needs to take concurrent execution of tests and measurement into consideration.

DriveTest2 is being used in couple of test laboratories with success. It is being used for cyclic and load testing. Using a test sequencer like DriveTest2 makes it easier to test devices and saves lots of time. It reduces lots of manual and repeatable work and reduces human errors by automating the testing process with sequences. Data gathering during the sequence execution allows monitoring status of devices. The architecture developed for DriveTest2 has made implementing required core functionalities possible. According to the feedback DriveTest2 is responsive and easier to use than DriveTest. In the future DriveTest2 will support more devices and have functionalities to perform different tests.

Among the future research topics, the direct continuation to this work would be the complete development of the DriveTest2 from start to finish. This would include requirements, architecture, implementation, UI, and testing. We could look at how the architectural decisions affect the implementation. Other research topics could be about software architecture and how it continues to evolve during the DriveTest2 lifecycle and what kind of new requirements and requirement changes there are.

References

Albarka, U. M., & Zhanfang, C. (2019). *A Study of Automated Software Testing: Automation Tools and Frameworks*. https://doi.org/10.5281/ZENODO.3924795

Amaricai, S., & Constantinescu, R. (2014). Designing a Software Test Automation Framework. *Informatica Economica*, *18*(1/2014), 152–161. https://doi.org/10.12948/issn14531305/18.1.2014.14

Apostolaki-Iosifidou, E., Ramachandran, N., & Dong, L. (2018). Server power supply test automation approach. *2018 IEEE International Instrumentation and Measurement Technology Conference (I2MTC)*, 1–6. https://doi.org/10.1109/I2MTC.2018.8409548

Arpaia, P., De Matteis, E., & Inglese, V. (2015). Software for measurement automation: A review of the state of the art. *Measurement*, *66*, 10–25. https://doi.org/10.1016/j.measurement.2015.01.020

Ateşoğulları, D., & Mishra, A. (2020). AUTOMATION TESTING TOOLS: A COMPARATIVE VIEW. *International Journal on Information Technologies & Security*, *12*, 63–76.

Avcioglu, A., & Demirer, M. (2015). Implementation of system testing automatization on computer aided systems for hardware and software. *2015 IEEE AUTOTESTCON*, 127–133. https://doi.org/10.1109/AUTEST.2015.7356478

Basil, V. R., & Turner, A. J. (1975). Iterative enhancement: A practical technique for software development. *IEEE Transactions on Software Engineering*, *SE-1*(4), 390–396. https://doi.org/10.1109/TSE.1975.6312870

Bass, L., Clements, P., & Kazman, R. (2013). *Software architecture in practice* (3rd ed). Addison-Wesley.

Berner, S., Weber, R., & Keller, R. K. (2005). Observations and lessons learned from automated testing. *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 571–579. https://doi.org/10.1109/ICSE.2005.1553603

Booch, G. (2018). The History of Software Engineering. *IEEE Software*, *35*(5), 108–114. https://doi.org/10.1109/MS.2018.3571234

Bourque, P., Fairley, R. E., & IEEE Computer Society. (2014). *Guide to the software engineering body of knowledge*.

Brumfield, B. (Ed.). (2011). *Developer's guide to Microsoft Prism 4: Building modular MVVM applications using Windows Presentation Foundation and Microsoft Silverlight*. Microsoft.

Buschmann, F. (Ed.). (1996). *Pattern-oriented software architecture: A system of patterns*. Wiley.

Caliburn.Micro. (n.d.). *Caliburn.Micro · 'Xaml made easy' · Caliburn.Micro*. Retrieved 28 March 2023, from https://caliburnmicro.com/

Carey, D. R. (2019). Introduction to Automated Test Systems—Back to Basics. *2019 IEEE AUTOTESTCON*, 1–7. https://doi.org/10.1109/AUTOTESTCON43700.2019.8961061

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Merson, P., Nord, R., & Stafford, J. (Eds.). (2014). *Documenting software architectures: Views and beyond* (2. ed., 4. print). Addison-Wesley.

Clements, P., Kazman, R., & Klein, M. (2002). *Evaluating software architectures: Methods and case studies*. Addison-Wesley.

Faraj, J., Sanderson, J., & Carolus, K. (2019). Next Generation of Test Automation and Systems. *2019 IEEE AUTOTESTCON*, 1–4. https://doi.org/10.1109/AUTEST.2019.8878475

Fowler, M. (2019). *Refactoring: Improving the design of existing code* (Second edition). Addison-Wesley.

Gamma, E. (Ed.). (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley.

Garofalo, R. (2011). *Building enterprise applications with Windows Presentation Foundation and the Model View ViewModel pattern*. Microsoft.

Garousi, V., & Elberzhager, F. (2017). Test Automation: Not Just for Test Execution. *IEEE Software*, *34*(2), 90–96. https://doi.org/10.1109/MS.2017.34

Garousi, V., & Mäntylä, M. V. (2016). When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology*, *76*, 92–117. https://doi.org/10.1016/j.infsof.2016.04.015

Harrold, M. J. (2000). Testing: A roadmap. *Proceedings of the Conference on The Future of Software Engineering*, 61–72. https://doi.org/10.1145/336512.336532

Hoffman, D. (1999). Cost Benefits Analysis of Test Automation. *STAR West*, *99*. https://www.agileconnection.com/sites/default/files/article/file/2014/Cost-Benefit%20Analysis%20of%20Test%20Automation.pdf

Hofmeister, C., Nord, R. L., & Soni, D. (2000). *Applied software architecture*. Addison-Wesley.

Hunt, A., & Thomas, D. (2000). *The pragmatic programmer: From journeyman to master*. Addison-Wesley.

Ingeno, J. (2018). *Software architect's handbook: Become a successful software architect by implementing effective architecture concepts*. Packt Publishing.

ISO/IEC/IEEE. (2017). *ISO/IEC/IEEE International Standard—Systems and software engineering—Vocabulary*. IEEE. https://doi.org/10.1109/IEEESTD.2017.8016712

ISO/IEC/IEEE Systems and software engineering—Architecture description. (2011). IEEE. https://doi.org/10.1109/IEEESTD.2011.6129467

Jansen, A., & Bosch, J. (2005). Software Architecture as a Set of Architectural Design Decisions. *5th Working IEEE/IFIP Conference on Software Architecture (WICSA'05)*, 109–120. https://doi.org/10.1109/WICSA.2005.61

Kasanen, E., Lukka, K., & Siitonen, A. (1993). *The Constructive Approach in Management Accounting Research*.

Keimling, R., Hansen, C., & Bilgic, A. (2013). A system for automated testing in development of measuring devices for industrial process instrumentation. *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation*, 65–70. https://doi.org/10.1145/2489280.2489289

Kruchten, P. B. (1995). The 4+1 View Model of architecture. *IEEE Software*, *12*(6), 42–50. https://doi.org/10.1109/52.469759

Larman, C., & Basili, V. R. (2003). Iterative and incremental developments. A brief history. *Computer*, *36*(6), 47–56. https://doi.org/10.1109/MC.2003.1204375

Leach, R. J. (2020). *Introduction to software engineering*.

Lukka, K. (2000). *The Key Issues of Applying the Constructive Approach to Field Research*. *Reponen, T., Ed., Management Expertise in the New Millennium: In Commemoration of the 50th Anniversary of Turku School of Economics and Business Administration*(Series A-1:2000), 113–128.

Lukka, K. (2014). *Konstruktiivinen tutkimusote*. Metodix. https://metodix.fi/2014/05/19/lukka-konstruktiivinen-tutkimusote/

Martin, R. C., Grenning, J., Brown, S., Henney, K., & Gorman, J. (2018). *Clean Architecture: A craftsman's guide to software structure and design*. Prentice Hall.

McConnell, S. (2004). *Code complete: A practical handbook of software construction* (2. ed). Microsoft Press.

Myers, G. J., Sandler, C., & Badgett, T. (2012). *The art of software testing* (3rd ed). John Wiley & Sons.

Naik, K., & Tripathy, P. (2011). *Software Testing and Quality Assurance: Theory and Practice*. Wiley.

National Instruments. (n.d.-a). *What Is Measurement Studio?* Retrieved 28 March 2023, from https://www.ni.com/fi-fi/shop/electronic-test-instrumentation/application-software-for-electronic-test-and-instrumentation-category/what-is-measurement-studio.html

National Instruments. (n.d.-b). *What is TestStand?* Retrieved 12 April 2023, from https://www.ni.com/fi-fi/shop/electronic-test-instrumentation/application-software-for-electronic-test-and-instrumentation-category/what-is-teststand.html

National Instruments. (2014). *TestStand System and Architecture Overview—NI*. https://www.ni.com/docs/en-US/bundle/teststand-system-and-architecture-overview/resource/373457f.pdf

Ojasalo, K., Moilanen, T., & Ritalahti, J. (2009). *Kehittämistyön menetelmät: Uudenlaista osaamista liiketoimintaan*. WSOYpro.

Perry, D. E., & Wolf, A. L. (1992). Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, *17*(4), 40–52. https://doi.org/10.1145/141874.141884

Polo, M., Reales, P., Piattini, M., & Ebert, C. (2013). Test Automation. *IEEE Software*, *30*(1), 84–89. https://doi.org/10.1109/MS.2013.15

Pressman, R. S. (2010). *Software engineering: A practitioner's approach* (7th ed). McGraw-Hill.

Raulamo-Jurvanen, P., Mäntylä, M., & Garousi, V. (2017). Choosing the Right Test Automation Tool: A Grey Literature Review of Practitioner Sources. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*, 21–30. https://doi.org/10.1145/3084226.3084252

Richards, M. (2015). *Software architecture patterns*. O'Reilly Media.

Robot Framework. (n.d.-a). *Robot Framework*. Retrieved 12 April 2023, from https://robotframework.org/

Robot Framework. (n.d.-b). *Robot Framework User Guide*. Retrieved 25 April 2023, from https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html

Royce, W. W. (1970). Managing the development of large software systems: Concepts and techniques. *Proceedings of IEEE WESCON*, 1–9.

Ruparelia, N. B. (2010). Software development lifecycle models. *ACM SIGSOFT Software Engineering Notes*, *35*(3), 8–13. https://doi.org/10.1145/1764810.1764814

Scacchi, W. (2002). Process Models in Software Engineering. In J. J. Marciniak (Ed.), *Encyclopedia of Software Engineering* (p. sof250). John Wiley & Sons, Inc. https://doi.org/10.1002/0471028959.sof250

Seidl, M., Scholz, M., Huemer, C., & Kappel, G. (2015). *UML @ Classroom: An Introduction to Object-Oriented Modeling*. Springer International Publishing. https://doi.org/10.1007/978-3-319-12742-2

Smith, J. (2009, February). Patterns—WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine Issues*, *24*(2). https://learn.microsoft.com/en-us/archive/msdn-magazine/2009/february/patterns-wpf-apps-with-the-model-view-viewmodel-design-pattern

Soe, N. T. T., Wild, N., Tanachutiwat, S., & Lichter, H. (2022). Design and Implementation of a Test Automation Framework for Configurable Devices. *2022 4th Asia Pacific*

Information Technology Conference, 200–207. https://doi.org/10.1145/3512353.3512383

Sommerville, I. (2016). *Software engineering* (10. ed., global ed). Pearson.

Taipale, O., Kasurinen, J., Karhu, K., & Smolander, K. (2011). Trade-off between automated and manual software testing. *International Journal of System Assurance Engineering and Management*, *2*(2), 114–125. https://doi.org/10.1007/s13198-011-0065-6

Thompson, K. D. (1999). COM-based test foundation framework. *1999 IEEE AUTOTESTCON Proceedings (Cat. No.99CH36323)*, 19–25. https://doi.org/10.1109/AUTEST.1999.800352

Tsui, F. F., Karam, O., & Bernal, B. (2014). *Essentials of software engineering* (Third edition). Jones & Bartlett Learning.

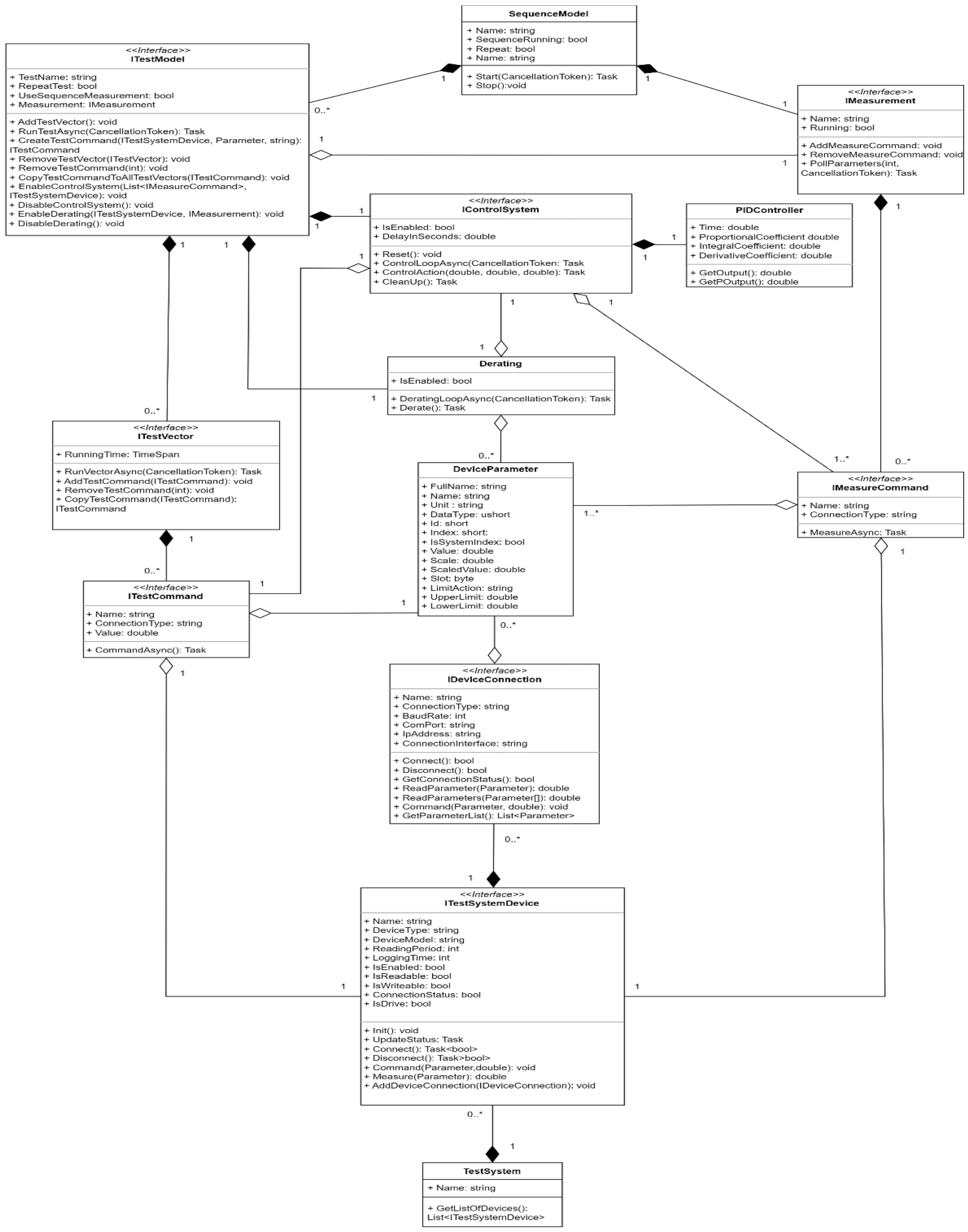
U.S. Department of Defense. (2016). *DoD ATS Selection Process 2016 Revision A*. https://www.acq.osd.mil/log/mr/ATS/.ats_library.html/DoD_ATS_Selection_Process_RevA_092017.pdf

Virtanen, A. (2006). Konstruktiivinen tutkimusote Miten koulutus ja elinkeinoelämän odotukset kohtaavat ammattikorkeakoulun opinnäytetöissä. *Ammattikasvatuksen Aikakauskirja*, *8(1*, 46–52.

Xu, X., Wang, L., & Zhou, H. (2003). An object-oriented framework for automatic test systems. *Proceedings AUTOTESTCON 2003. IEEE Systems Readiness Technology Conference.*, 407–410. https://doi.org/10.1109/AUTEST.2003.1243605

Appendices

Appendix 1. Full class diagram of DriveTest2 model


SequenceModel
+ Name: string
+ SequenceRunning: bool
+ Repeat: bool
+ Name: string
+ Start(CancellationToken): Task
+ Stop():void
<<Interface>>
ITestModel
+ TestName: string
+ RepeatTest: bool
+ UseSequenceMeasurement: bool
+ Measurement: IMeasurement
+ AddTestVector(): void
+ RunTestAsync(CancellationToken): Task
+ CreateTestCommand(ITestSystemDevice, Parameter, string): ITestCommand
+ RemoveTestVector(ITestVector): void
+ RemoveTestCommand(int): void
+ CopyTestCommandToAllTestVectors(ITestCommand): void
+ EnableControlSystem(List<IMeasureCommand>, ITestSystemDevice): void
+ DisableControlSystem(): void
+ EnableDerating(ITestSystemDevice, IMeasurement): void
+ DisableDerating(): void
<<Interface>>
IMeasurement
+ Name: string
+ Running: bool
+ AddMeasureCommand: void
+ RemoveMeasureCommand: void
+ PollParameters(int, CancellationToken): Task
<<Interface>>
IControlSystem
+ IsEnabled: bool
+ DelayInSeconds: double
+ Reset(): void
+ ControlLoopAsync(CancellationToken: Task
+ ControlAction(double, double, double): Task
+ CleanUp(): Task
PIDController
+ Time: double
+ ProportionalCoefficient double
+ IntegralCoefficient: double
+ DerivativeCoefficient: double
+ GetOutput(): double
+ GetPOutput(): double
Derating
+ IsEnabled: bool
+ DeratingLoopAsync(CancellationToken): Task
+ Derate(): Task
<<Interface>>
ITestVector
+ RunningTime: TimeSpan
+ RunVectorAsync(CancellationToken): Task
+ AddTestCommand(ITestCommand): void
+ RemoveTestCommand(int): void
+ CopyTestCommand(ITestCommand): ITestCommand
DeviceParameter
+ FullName: string
+ Name: string
+ Unit : string
+ DataType: ushort
+ Id: short
+ Index: short:
+ IsSystemIndex: bool
+ Value: double
+ Scale: double
+ ScaledValue: double
+ Slot: byte
+ LimitAction: string
+ UpperLimit: double
+ LowerLimit: double
<<Interface>>
IMeasureCommand
+ Name: string
+ ConnectionType: string
+ MeasureAsync: Task
<<Interface>>
ITestCommand
+ Name: string
+ ConnectionType: string
+ Value: double
+ CommandAsync(): Task
<<Interface>>
IDeviceConnection
+ Name: string
+ ConnectionType: string
+ BaudRate: int
+ ComPort: string
+ IpAddress: string
+ ConnectionInterface: string
+ Connect(): bool
+ Disconnect(): bool
+ GetConnectionStatus(): bool
+ ReadParameter(Parameter): double
+ ReadParameters(Parameter[]): double
+ Command(Parameter, double): void
+ GetParameterList(): List<Parameter>
<<Interface>>
ITestSystemDevice
+ Name: string
+ DeviceType: string
+ DeviceModel: string
+ ReadingPeriod: int
+ LoggingTime: int
+ IsEnabled: bool
+ IsReadable: bool
+ IsWriteable: bool
+ ConnectionStatus: bool
+ IsDrive: bool
+ Init(): void
+ UpdateStatus: Task
+ Connect(): Task<bool>
+ Disconnect(): Task>bool>
+ Command(Parameter,double): void
+ Measure(Parameter): double
+ AddDeviceConnection(IDeviceConnection): void
TestSystem
+ Name: string
+ GetListOfDevices(): List<ITestSystemDevice>
1
0..*
1..*