Behave and PyUnit
A Testers Perspective

Johan Borgenstierna

Supervisor: Anders Fröberg
Examiner: Rita Kovardanyi
Upphovsrätt
Detta dokument hålls tillgängligt på Internet – eller dess framtidiga ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/.

Copyright
The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/her own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/.

© Johan Borgenstierna
Behave and PyUnit – A Testers Perspective

Johan Borgenstierna
Innovative Programming,
Linköping University
Linköping, Sweden
johan.borgenstierna@outlook.com

1. ABSTRACT
A comparison between two different testing frameworks Behave and PyUnit is demonstrated. PyUnit is TDD driven and Behave is BDD driven. The method SBTS shows that Behave enforces better quality of software in the maintainability branch than PyUnit. The Gherkin language used in Behave is easy to read and widens the scope of protentional testers. Although Behave is not as fine grained with the cover of the tests than PyUnit since Behave is limited to the behaviour of the system.

1.1 Author Keywords
BDD; TDD; Behave; PyUnit; unittest; behaviour; testing; yubikey; nfc; maintainability; readability; Gherkin;

2. INTRODUCTION
“BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology. It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software that matters.” – Dan North the creator of BDD (Behaviour Driven Development), speech at the annual Agile Specification, BDD and Testing eXchange in London, November 2009 [7]. Test frameworks are a given in most companies but most of them are limited to TDD tests. This article will demonstrate that with BDD one can widen the scope of possible quality improvements in the code.

2.1 Motivation
A big part of developing quality software is maintainability which includes testability, understandability and modifiability [4]. Dan North developed BDD as a response for what TDD was lacking. BDD includes specification combined to the tests and executable clear text scenarios. BDD goes very well hand in hand with agile development since it is making it easy for business people to request new features and for developers to demonstrate that new features indeed are implemented and working.

The company Yubico want their testing framework for the Yubikey to be more accessible, readable, reusable and easy to use to all their programming and “semi-programming” employees. Their reasoning to improve above standing categories are: Accessible – unit-tests should not be spread out in multiple projects. Readable – tests should be easy to follow and understand. Reusable – small modular tests that build larger tests e.g. one test should not have to be completely rewritten when only a small change is needed. Easy to use – A user that are not deeply involved in the code behind the software should still be able to modify, read and write new tests/test-requests.

2.2 Purpose
A comparison between TDD and BDD is looked upon from a testers perspective. The BDD part consist of the python framework Behave which uses Gherkin DSL as its ubiquitous language. The TDD part consist of the python unit-test framework PyUnit. The observations in the comparison includes the learning curve that BDD and TDD possesses. The comparison result will act as further incitement whether to use Behave or PyUnit. It will show if BDD lives up to the Yubico request of features in their testing framework.

To demonstrate the power of abstraction a test can have, the Yubikey Neo NFC communication features and technical specifications will be explained and used in the given examples.

2.3 Research Questions

2.3.1 What is the difference in Gherkin readability compared to PyUnit from a testers perspective?

2.3.2 How can a tester modify existing tests in Behave compared to PyUnit?

2.3.3 How can a tester write new tests in Behave compared to PyUnit?

3. THEORY
This chapter will cover basic concepts in BDD, TDD, Yubikey and NFC. Yubico lacked basic behaviour-NFC-tests for their Yubikey Neo, and since they have already started moving other tests over to the Behave framework, they saw it as a good opportunity to write new tests in Python code and Gherkin DSL. Before writing tests, questions were asked such as; How can I as a developer communicate with the Yubikey over NFC? How can one command it to execute a specific command e.g. generate a new crypto-key? What makes the communication over NFC different than from USB? In this article, this deep knowledge can serve as an example of how a simple behaviour can have a deeper understanding and knowledge underneath it. Ultimately, this looked upon simple behaviour can be written in Gherkin, abstracting much of the complicated parts.

3.1 BDD with Gherkin and Behave
Gherkin is designed not to be hard to understand, in fact, by explaining what Gherkin is and how it is used it can be easier
to understand what BDD is. Gherkin is a naturally readable which makes it possible for non-programmers to understand and write in. Even business people working together with a developer can write new features in the planning stage [13]. BDD is the mean to develop a product by analysing its expected behaviour in certain scenarios. An expected behaviour could be for example that a car is expected to drive forward. A car going forward could serve as a very practical feature for a car. The behaviour that the car is going forward could for example happen in the scenario that the driver is accelerating. So, let’s break that feature up into scenarios with steps seen in Figure 1.

```
Feature: Car going forward
Scenario: The car should be accelerating
  Given there is a driver.
  When the driver trample the gaspedal.
  Then the cars speed increased.
```

**Figure 1. Feature example of a car written with gherkin DSL in the file car_forward.feature**

I personally think that Gherkin makes it easy to understand what the feature is and how a scenario is expected to behave. The different types of steps are: Given, When and Then. The statement on each line with a step is to be confirmed and accepted with e.g. a BDD framework such as Behave. To confirm that the scenario is behaving as expected, each step is mapped to an acceptance test written in code. Figure 2 demonstrates how the python code mapped to each step line in behave would look like.

```python
def step_impl(context):
    assert context.car.driver is not None

def step_impl(context):
    context.old_speed = context.car.speed
    assert context.car.driver.gas() is True

def step_impl(context):
    assert context.car.speed > context.old_speed
```

**Figure 2. Tests for the car_forward.feature written in python with behave in the file car_forward.py**

The decorators “@given”, “@when” and “@then”, are mapping the cleartext step lines in the car_forward.feature file to the corresponding function that will run a test. The context parameter is used to pass variables between functions, and the `assert statement` will raise a runtime-error if the `condition` in it results in False. The next step in the developing process is to implement e.g. the car class.

Now when the basics of Gherkin and Behave is out of the way, lets step over to BDD. According to Dan North the flaw in TDD was among others that it does not encourage collaboration between developers, QA and non-technical, or business participants. As a response, Dan North created BDD [6], but as more and more frameworks developed their own implementation of BDD, the BDD approach became less clearly defined. To make it more defined and clear, Carlos and Xiaofeng analysed the literatures and BDD toolkits and released the article “A study of the characteristics of behaviour driven development. [16]” where they identified six main characteristics:

A. **Ubiquitous Language**
   A language that comes from a domain model, this language is seen as the core of BDD. The ubiquitous language used in Behave is Gherkin.

B. **Iterative Decomposition Process**
   The decomposition process is derived from what a business intends to produce. In BDD the expected behaviour of the system is analysed for a more concrete and easy perspective. Features are derived from business outcomes, scenarios are derived from features and steps are derived from scenarios.

C. **Plain Test Description with User Story and Scenario Templates**
   According to Dan North, BDD should follow a specified User Story Template and a Scenario Template [6]:

User Story Template:

**Title for the User Story** (small description)
- As a [User type, Business Type, Persona]
- I want [Objectives, Actions, Tasks]
- So that... [Benefit, Result, Value]

Scenario Template:

**Title for the Scenario** (small description)
- Given [Context]
- And [More context, is optional]
- When [Event]
- Then [Outcome]
- And [More outcome, is optional]

The User Story Template and Scenario Template should both be writeable in the projects chosen ubiquitous language.

D. **Automated Accepting Testing with Mapping Rules**
   Every step in a scenario written in the ubiquitous language should have a mapping from the clear-text-step-sentence to one test method. The test method responds with success or failure and is given to be refactored at some point, just like a TDD test [1].
should be automatic and therefore resulting in having executable plain text. There is no specific technique for how the mapping rules should function, e.g. could regex be used.

**E. Readable Behaviour Oriented Specification Code**

Implemented code should be readable and describe the behaviour of objects. Implemented names of classes and methods should follow what is written in the scenarios, which in its turn makes the ubiquitous language help developers to produce code that becomes behaviour-oriented. When using the same structure in the code implementation such as it follows the ubiquitous language, results in that the ubiquitous language can serve as a specification to the code itself.

**F. Behaviour Driven at Different Phases**

The first phase is planning the business outcomes, the second one is the analysis of what features needed to achieve the business outcomes. Third and last is the implementation phase, which maps the ubiquitous language to acceptance tests.

**More on Gherkin and Behave in Python**

Those are all the six characteristics briefly explained and from what I can see, Behave in Python complies with all six characteristics except some parts of A, B and F:

- **A.** Behave does not support modifying the used ubiquitous language, instead they use a modified version of Gherkin that supports lowercase step keywords [2], as the ubiquitous language.
- **B.** Behave does not have support for iterative decomposition from a business outcome perspective.
- **F.** Behave does not have any tools for a planning part that are derived from what the business outcomes intends to produce.

Behave also has its own set of extra tools such as fixtures – startup and cleanup function, just like in TDD. **Before-and-after hooks** – Functions that can run before and after Steps, Scenarios, Features, Tags and the whole test run. Tags – special marks in Gherkin which can supports before-and-after hooks. On top of the features described above, The Gherkin Language also support **Background** – a Gherkin described before hook / startup that runs before each scenario. **Scenario Outlines** – makes it possible to run the same scenario with different values defined in an example table.

**3.2 TDD – Test Driven Development**

TDD stand for Test Driven Development and just like BDD the test-cases can be written before new functionality is implemented. After writing test-cases the tests are executed and the will often contain errors before implemented code is written. Once errors occur the programmer implements the missing code, execute the tests again, check for success, and so on. When the tests are successful the developer writes new test-cases and follow the same cycle. The test-cases are bound to be refactored at multiple occurrences during the project, commonly referred to as a micro-cycle: RGR – Red-Green-Refactor [1]. RGR is based on the philosophy that our minds have a hard time both implementing the correct behaviour and the correct structure at the same time. So RGR focuses on making the code work correctly and after that refactor the code, giving it a more durable and longer life-span. The name comes from that the test-case produces red error when failing, green success message when succeeding and once succeeded more work is needed to refactor the code. According to [12], there are some professional benefits to TDD:

- When writing tests before coding a more open dialog and planning is bound to occur between the ones implementing and the ones designing the software, reducing more implementing faults that occurs from miscommunication.
- TDD forces programmers to write code that can be automatically testable, such as having methods returning values.
- If a new implementation breaks an older functionality, the automated tests in TDD can easily detect where the issue stands, resulting in a smoother integration for newer implementations.

Some other benefits are:

- Engineers concerned with the “Cost-of-Change” can rest assured that TDD will help them find faults earlier [3].
- Benefits with **automatic testing** [8]
  - Production of a reliable system.
  - Improvement of the quality in the test effort.
  - Reduction of the test effort.
  - Minimization of the schedule.

**3.3 TDD with PyUnit**

PyUnit is the Python unit testing framework it “... supports test automation, sharing of setup and shutdown code for tests, aggregation of tests into collections, and independence of the tests from the reporting framework.” [14]. To achieve this support, PyUnit follows these concepts:

**3.3.1 Test Fixtures**

Before a test is run, some resources might need to be accessed (startup) and then gracefully shut down (cleanup), such as connection to a database or a virtual web browser. Fixtures makes it possible to perform tests associated with startup and cleanup actions.

**3.3.2 Test-Case**

The smallest unit of testing is a test-case. A test-case checks for expected responses for specific inputs. The response could e.g. be the return value of a function or condition. The input could e.g. be the parameters for a function. In PyUnit, new test-cases inherits from a provided test-case base class.

**3.3.3 Test-Suite**

A test-suite is a collection of test-cases, test-suites or both. It binds together multiple test-cases with each other. It can also
separate functions from a test-case into new test-cases. The test-suite is for organizing and reusing test-case code.

3.3.4 Test Runner
The representation and director of the test executions. It tells the tests to execute and how to represent the tests running and the test result. The Test Runner can be represented and controlled from interfaces such as a textual interface like a command line interface, or a graphical interface such as a web application.

3.4 The Yubikey and NFC
The Yubikey is a USB hardware device used to login to various systems securely over a network. In this article the Yubikey Neo is BDD tested over NFC with Behave, using Gherkin as the Business Readable DSL. The Yubikey is a hardware USB device, that among other features can generate its own private keys and public keys for cryptographic communication. The crux of the Yubikey is that the private key can be stored in a writable only memory which the Yubikey only have access to, making it impossible for humans to access or read. The Yubikey also have support for several cryptographical standards over Personal Identification Verification (PIV), used in smartcards. By utilizing existing operating system drivers, the Yubikey does not need any driver installation before usage, resulting in a seamless user experience where the Yubikey works straight out of the box. The company Yubico releases almost all their software as open source, making it easier for business to implement their own security solutions that utilizes the strong security that the Yubikey provides.

![Yubikey Neo](image)

Figure 3. Yubikey Neo – A USB security key with NFC capabilities
For this article the scope of the Yubikey capabilities will be stripped down to only the Yubikey Neo’s NFC communication protocol.

A smartcard can for example be a credit card and a smartcard reader can for example be a cash register. Just like a credit card is inserted into cash register – The Yubikey is recognized as both a smartcard and a smartcard-reader when inserted via USB. Although the Yubikey is only recognized as a smartcard-reader when inserted via USB. For NFC communication, a separate smartcard-reader is required, e.g. the NFC smartcard-reader device SCL3711. For the sake of simplicity, it is easier to explain the communication flow when seeing the Yubikey as a combined smart card and smartcard reader.

3.4.1 General smartcard communication flow
Smartcard ↔ insert mechanic (e.g. card reader in cash-register) ↔ Smartcard-reader ↔ USB ↔ OS ↔ Controlling software

3.4.2 Yubikey smartcard communication flow over USB
Yubikey as smartcard and smartcard-reader ↔ USB ↔ OS ↔ Controlling software

3.4.3 Yubikey smart card communication flow over NFC
Yubikey as a smartcard with NFC ↔ NFC ↔ NFC Reader as a smartcard-reader ↔ USB ↔ OS ↔ Controlling software

3.4.4 Smartcard-reader to smartcard flow via APDU
The software writes a request to the smart card via APDU (Application Protocol Data Unit) [11] that a connected smartcard-reader sent out to the smartcard via e.g. NFC. The smartcard in its turn return with the response data to the smartcard-reader, the response is then read and interpreted by the software.

Controlling software ↔ APDU (response or request) ↔ Smartcard-reader ↔ APDU (response or request) ↔ Smartcard

3.4.5 A deeper dive into communication protocols
The protocol for NFC communication used in the Yubikey is ISO/IEC 14443-4. The -4 indicates that the NFC protocol supports the smart card protocol ISO/IEC 7816-4. The protocol in its turn use the communication unit APDU which is sent between smart card and smart card reader. APDU header is filled with what type of command to send (CLA), whether a command should write data (INS), two parameters for e.g. offset on the file at which to write data (P1, P2), a length of the command data (L) and lastly the data. The data in its turn formatted in the Type Length Value (TLV) format. The TLV is represented as hexadecimal numbers which in its turn are represented by one byte each. This low-level protocol ensures fast communication with low overhead. The response from the smart card contains response data in the TLV format, followed by 2 hexadecimals, representing the commands processing status, e.g. 9000 is success.

4. RELATED WORK
BDD can be applied in many ways, for example, in “Behavior-Driven Development for Computer-Interpretable Clinical Guidelines [10]” the authors propose the idea of applying behaviour-driven development to Computer-Interpretable Clinical Guidelines (CIGs). In basic terms, this proposal was to interpret a medical description in a more natural language such as Gherkin, then map that language to test-cases based on the CIGs.

In “Beast methodology: An agile testing methodology for multi-agent systems based on behaviour driven development [5]”, BDD is used to make it easier for developers and stakeholders to understand one another. They also went one step
further and implemented BDD in their own test-suite for Multi-Agent Systems with autogenerated mock-objects.

"Executable acceptance tests for communicating business requirements: customer perspective [13]" is a study to conclude results from four hypothesis, one of which is weather customers can specify functional business requirements in the form of executable acceptance tests. They found that learnability and ease of use analysis indicated that average customers can have difficulties learning the technique.

5. METHOD
In this section a pilot study is presented which leads to the proposed improvised method; Simple Battle Between Testing Frameworks Effects on Quality Software (SBTS). The environment the project takes place in is defined and an investigation is performed on the research questions.

When comparing testing frameworks, one can compare what the testing frameworks intends to achieve. In the end, a business wants their code to be of good quality [4] and testing is one way to achieve such a goal. In this method the Yubikey will serve as the testing example for both PyUnit and Behave.

5.1 Pilot Study
Before setting my teeth in the world of testing I had to dig deep into what was to become one of the most complicated devices I have ever researched, the Yubikey. The Yubikey is over NFC and USB, two areas I had no knowledge. The Yubikey is a hardware security device – security on an advanced level which I also never heard of. The area of PIV – Personal Identification Verification – Smartcards – Smartcard readers – never heard of. The hardware world came as a crash from above, a world of zeroes, bytecode, hexadecimals, low protocols with APDU, TLV, standards such as ISO/IEE 14443-4, etc – all new knowledge. Working my way up from the hardware connection of things up to the more abstract level of Python code was the biggest challenge. But once connecting the dots the picture became clearer, so clear it seemed like this deep knowledge was something others might have already abstracted for me. That’s when I found the Python libraries. All knowledge up to this point had not gone to waste, now instead I had a deeper knowledge behind the packages which I was working with. Although a lot of time had gone wasted in theory on NFC which wasn’t even in my research questions.

The first approach was to test the Yubikey over BDD, but since the Yubikey is already extremely advanced the required knowledge in the area was too great for me. Instead more focus was spent on the BDD part of things, the Gherkin code, the behaviour thinking, steps, scenarios, hooks, fixtures, etc. Before digging into Behave I thought I knew about tests but there was a lot of missing pieces to take in. BDD had thought of everything, it was the missing piece in the agile-development course I had just a year ago. TDD felt so old and outdated, why isn’t everyone using BDD? Then it struck me, BDD is nice and all but there is so much abstraction, easy to read code, a language for business people, but for programmers who knows how code works, it seemed almost like a child’s play. At least on first glance, it wasn’t until I tried writing the same tests in TDD with PyUnit when I noticed that the refactoring to make it behave as Behave was overwhelming. Suddenly it was easier to work with BDD and although the Gherkin language did set me back in time, it also force me to understand how the system was working and how I wanted my own code to work. And once the planning stage were done the object oriented easy-to-read-code came natural for me. I had already figured out what classes I needed and how they communicated with each other, all stood there, in Gherkin code, ready to be tested once my implementation had been completed. An idea was slowly setting place, I wanted to show why BDD was so great and why TDD was outdated. The article changed name from Testing a Yubikey Neo over NFC with BDD to simply Behave and PyUnit – A testers perspective.

After carefully collecting intel and abstracting learned theory into the theory chapter of this article, I’ve came upon the next crisis. How on earth was I supposed to compare two testing frameworks? After searching articles on Google Scholar like a fanatic I could still not find any evaluation methods for testing frameworks. The only articles I could find was empiric studies, in which they divided people into groups and made them use different testing frameworks. They measured the errors the code produced and how precise the completed task was in course with the given business goals. Code-coverage I thought, errors and what not, but that wasn’t what I wanted to show to others. In my search I found an article about the characteristics on quality of code [4]. Many of the characteristics was closely connected to what one achieves with BDD. Why can’t I just make my own method I thought. A method that scanned the characteristics of quality software and compared it to what could be achieved with a testing framework – so that is what I improvised.

5.2 SBTS – An Improvised Method
In the lack of comparison of testing frameworks methodology, I propose a new methodology for comparing testing frameworks. This method uses the Software Quality Characteristic Trees [4] maintainability branch, see Figure 4.

The SBTS method will look at each characteristic and make a connection to what the testing framework intends to achieve. To minimize the scope of possibilities a testing framework can produce, more focus will be on what a testing framework intends and enforces for programmers and what effect it can have on the quality of software. In 9 rounds, one for every unique leaf of characteristics, each testing framework will either lose, win or be even according to the impact the testing framework effect has on the characteristic of quality software. The determination of the winning framework is derived from logical arguments. This method should be referred to as: Simple Battle Between Testing Frameworks Effects on Quality Software (SBTS).
part are much of what the Gherkin steps Given, When and Then enforces one to follow. As a side-effect the inputs and outputs will be easy to explain with the help of the human readable specification. Behave wins this round.

5.3.5 Conciseness
Imply that code shouldn’t be overflown with excessive information due to not being fragmented in modules, overlays, functions, sub-routines, or that code is repeated in multiple places. Since Behave is better at reusing old tests instead of writing new test-cases, that effect could lead to that the specification in Gherkin can give a hint to when code is about to be repeated. In PyUnit one can more easily try to make a new test-case which tests the same thing as an older test-case. Due to not having a specification, I argue that repeating code could happen more often and can be harder to detect. The fragmentation of things is also supported with the concept of object-oriented programming. Behave enforces object-oriented programming better than PyUnit, as mentioned in communicativeness. Behave is the winner in conciseness.

5.3.6 Consistency
Code, comments, terminology, symbology, notation, etc should all stay consistent throughout the software. Neither PyUnit or Behave have the intention or enforcement to follow consistency characteristics, except for natural occurrences such as class names and function names staying the same. Therefore, I call it a draw.

5.3.7 Legibility
The codes function is explained by reading the code, e.g. the name of the functions corresponds to its intended behaviour. PyUnit does not enforce any legibility whilst Behave derives its name from Gherkins cleartext which also explains the expected behaviour / function in the code. Behave wins this round.

5.3.8 Self-Descriptiveness
Code which contains enough information to determine or verify its objectives, assumptions, constraints, inputs, outputs, components, and revision status. PyUnit does not enforce any Self-Descriptiveness whilst Behave not only have better legibility, Gherkin also works as documentation on the behaviour of the code. Although, the code itself is not entirely enforced into being Self-Descriptiveness, I argue that one has a higher chance of achieving such a quality with Behave. Even though Behave and PyUnit lack in this area – Behave still lays ahead of PyUnit. Behave wins this round.

5.3.9 Structuredness
Code that possess a clear pattern of organization of its connected parts and that the evolution of the program design has proceeded in an orderly and systematic manner – not to be confused with robustness. With Behave one can structure and organize the behaviour of the system before implementation of code. That is of course also a possibility when coding in PyUnit, but what tends to happen when coding tests before implementation is that the tests needs refactoring before they can execute the implemented code.
In Gherkin the test scenarios usually stay the same while the test code behind it is refactored. Therefore, I would argue that because Gherkin does not need the same refactoring as PyUnit, the evolution of the programs design proceeds better in Behave. When it comes to a clear pattern of organizations connected parts, I would say that neither PyUnit or Behave intends to achieve such an effect. Behave wins this round.

5.4 Environment
The device running this project is a Raspberry Pi 3 model B (rpi3), running the operating system Rasbian Stretch Lite, released April 2018 (kernel version 4.14). The chosen programming language is Python 2.7.15 with the packet manager pip (version 10.0.1). Python is installed with Python\'s Virtualenv (version 15.2.0) which makes the installation of Python and its packages encapsulated from example the systems installation of Python. Pip makes the setup of the required python packages for this project easy as pie. PyUnit is referred to as unittest in Python and is included in the standard libraries. After that, BDD is installed with the pip packet behave (version 1.2.6). NFC communication in python is done with the pip packet nfcpy (version 0.13.4), referred to as nfc when imported. A support library called cryptography (version 2.2.2) is used for cryptographic functions e.g. the authentication in the Yubikey. The package pyusb (version 1.0.2) is used to find devices connected with USB. Installing these packages is as simple as executing: $ pip install behave nfcpy cryptography pyusb. Though, underneath these python packages a lot of other dependencies are used, but under the projects used operating system most of these packages are preinstalled. The only problem I had with the dependencies was the cryptography packet which also required the Linux package libffi-dev, installed with: $ sudo apt install libffi-dev.

The NFC reader used in this project is called SCL3711, produced by the company Identiv (Part No: 905169). The reader supports ISO14443-4.

The Yubikey used is the Yubikey Neo (released 2014), produced by the company Yubico.

Other developing tools are:
- Visual Studio Code.
- Remote VSCode (rmate) plugin (version 1.1.0) to Visual Studio Code (VS Code, version 1.22.2) in Windows 10 (version 16299.371) for rpi3 files over SSH.
- rmate for linux used in the rpi3 (aurora\’s rmate version 1.0.1).
- Cucumber full language support plugin for VS Code (version 2.10.0).
- NFC-led light which produces light when near NFC field, indicating whether the NFC power-field is on or off.

5.5 Investigating the Research Questions
5.5.1 What is the difference in Gherkin readability compared to PyUnit from a testers perspective?
It is a known fact that most programming languages have a high learning curve, it is not something that you can easily master in one day. Although programming languages differ in their human readability, e.g. C++ vs Python, the concepts in programming such as variables, loops, lists, functions, statements, etc are all complicated logical operations that have less to do with the learning curve of just reading a programming language. Businesses can vary in their search for testers, some business require testers with high programming knowledge while others can deal with tester which have less programming knowledge. I will assume from a business perspective that a tester only possesses basic programming knowledge to broaden protentional testers.

I think that the biggest difference when comparing Gherkin and PyUnit is that Gherkin is not a programming language. Gherkin is a DSL which functions as a business readable language. A language that explains and specifies behaviour but doesn\’t necessarily demonstrate them or automate them, it only specifies the expected behaviours. Gherkin is the sugar on top of tests. Once the systems behaviour specifications are written in Gherkin, the expected behaviour in cleartext is mapped to code, which is written for the tests to pass, and once we are in the domain of writing python code, we are in PyUnit\’s domain of readability.

Therefore, instead of looking at the readability in the sense of understanding what the test behind the specification does, one can instead look at the readability on which a tester can understand the expected behaviour of a test.

When looking at PyUnit, in some sense, the expected behaviour for a test could either be comments with some extra context and explanation, or it could be that the variables, class names and functions all have names which hints to what behaviour they are following. In its core, the expected behaviour is the assert statement for which a conditions outcome should follow. But does a test in PyUnit really explain its expected behaviour on such a detail level that a tester can see the given state the system is in, what changes the state and what the state is expected to be in after the change? Like the Given, When and Then does in Gherkin?

In the SBTS, Behave won in Legibility and Self-Descriptiveness. In PyUnit it is up to the one writing the test whether to make the test easy to follow, understand the state the system is in, when the change occurs, and what the outcome is expected to be.

Figure 5 demonstrates a simple Yubikey connection in Gherkin. Figure 6 demonstrates the same test in PyUnit while Figure 7 and Figure 8 demonstrates how PyUnit can give uphold to drastically differences in readability.

To really understand readability, I think that the underlaying question really is; Gherkin or PyUnit, which one is the
easiest to learn how to follow? The definition of cumulative learning is: “Intelligent System, human or artificial, accumulate knowledge and abilities that serve as building blocks for subsequent cognitive development. [9]”. Which indicates that the test readability which is mostly linked to some previous knowledge is easier to learn than the other. In this case, the link for previous knowledge would lie in the names of classes, functions and names for a tester. Gherkin explains the system in natural language, which makes it easier to learn how the system works. With that said, Gherkin forces readability which in turn explains the expected behaviour better than PyUnit.

def test_connect_to_scl3711(self):
    """Scenario: connect to the Yubikey Neo via nfc-reader SCL3711"
    # Given there is a SCL3711 reader
c3711 = SCL3711()
    assert c3711 is not None

    # When connecting the Yubikey Neo
cy_neo = YubikeyNeo()
    assert cy_neo is not None

    # Then the Yubikey Neo should be present
    c3711.connect(cy_neo)

Figure 7. A commented function to improve readability

Figure 8. A Test example with low readability

5.5.2 How can a tester modify existing tests in Behave compared to PyUnit?
Both PyUnit and Behave support modular tests. In PyUnit one must inherit from tests or setup a test-suite. In Behave it is as simple as using the same sentence or map a new sentence to an already existing step-function, given the step-function is not dependent on any context. One of the simplest test modification in Behave is swapping the place of two sentences, e.g. in Figure 5 one can swap there is a SCL3711 reader and there is a Yubikey Neo. For PyUnit the comment, initiation of the class and assert-statement for the scl3711 would need to be swapped with the yk_neo, see Figure 7. For a test to be modified in PyUnit one needs to have some basic knowledge in code, and as mentioned in section 5.5.1, the test-cases readability very much depends on the one writing the test. It would be harder to know where to swap the code in e.g. the worst-case scenario seen in Figure 8.

5.5.3 How can a tester write new tests in Behave compared to PyUnit?
As mentioned in section 5.5.1, Gherkin is not a programming language, it is a business readable DSL, the sugar on top of test code. To avoid the programming part when writing a new test-case in Behave, a tester would have to write Gherkin code that maps to already existing tests for new features or scenarios. This limits the scope of writing new scenarios, although if a tester writes a new scenario or feature in Gherkin which isn’t implemented yet, that Gherkin code serves as a great specification on what new behaviour the tester wants to be tested. In PyUnit one needs to understand the programming language. In Gherkin case, the tester can swap around already existing test-cases or test-case
functions. For new test-cases, the tester would either implement the test themselves or explain in detail what the new test-case should test for another more involved or more skilled programmer.

6. RESULT

6.1 SBTS – Result
The overall result is PyUnit wins 2/9 rounds, Behave 6/9 rounds and 1/9 resulted in a draw. In the Testability branch PyUnit scored 2/5, Behave scored 3/5. For the Understandability branch PyUnit scored 0/5, Behave scored 5/5. Lastly, in the Modifiability branch PyUnit scored 0/5 while Behaved scored 5/5.

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>PyUnit</th>
<th>Behave</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessibility</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Accountability</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Augmentability</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Communicativeness</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Consistency</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Legibility</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Self-Descriptiveness</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Structuredness</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

Figure 9. Results of SBTS – PyUnit vs Behave

7. DISCUSSION

7.1 Result
The biggest difference in the compared testing frameworks I found is that PyUnit does not strictly encourage writing test code which explains the systems behaviour. Another interesting finding is that PyUnit excels in accessibility and accountability. This suggests that BDD can’t replace all the intended effects that TDD provides because TDD have a finer grained, low level, input-output specific test-cases that goes outside the scope of behaviour. Another finding is that Gherkin in Behave does not help the testers to keep track on what context a step requires, at least not with complete certainty. Although in that case, one could use fixtures that suggests what context is needed e.g. the @fixture.nfc.use.ykneo tag in Figure 5 suggest that the steps in the scenario depends on a Yubikey Neo.

The most interesting finding that surprisingly didn’t struck me as something obvious is that Gherkin can also teach a person how a system works since it clearly explains the behaviour of the system, which statement is supported by “Learning is a hypothetical construct: it cannot be directly observed, but only inferred from observable behaviour.” [15].

7.2 Method
The biggest flaw in this article is the improvised method SBTS. Here, logical arguments are laid out for every characteristics, but those logical arguments in its turn are derived from what I, the author, possesses in knowledge on the subject.

An improvement in the current method would be some defined testing framework guideline which would cover all the maintainability characteristics, which could prevent vague logical arguments.

I think the environment chapter is sufficient for one to replicate setting up test-cases for the Yubikey with both PyUnit and Behave. The chapter have versions and names of every tools used for the project, it also provides concrete commands for one to execute to setup the environment. Although one flaw is that there is no guide on how to install virtual environment in python nor how to execute the tests. Although when it comes to the reliability of the SBTS one might have more knowledge than me and could therefore get other results regarding which testing framework won in the different characteristics.

Since the SBTS method is so simple and yet generalizes, one could also question the validity of the result. With more time, a deeper dive into the testing frameworks might hint to other answers than what this article finding provides.

7.3 Ethical and Social Aspects
The social aspect can be seen in the planning phase of BDD, for a company, such a tool is great to minimize otherwise miscommunications in expected behaviour which can leads to angry customers or extensive time in refactoring which can lead to expensive programmers. I have yet to find any ethical aspects regarding testing frameworks.

8. CONCLUSION

8.1 Answer to research questions

8.1.1 What is the difference in Gherkin readability compared to PyUnit from a testers perspective?
Gherkin have easier readability and widens the scope of protentional testers.

8.1.2 How can a tester modify existing tests in Behave compared to PyUnit?
Behave support executable cleartext functions which can be implemented on a highly modular level. Modifying tests in Behave also requires less refactoring.

8.1.3 How can a tester write new tests in Behave compared to PyUnit?
Writing new tests in Behave requires programming knowledge if already written step-functions can’t be reused. Although if the step-functions have a high modularity level and is not bound to specific a context, new test cases have the protentional to not require any programming skills. In PyUnit one always need basic programming knowledge.

9. REFERENCES


