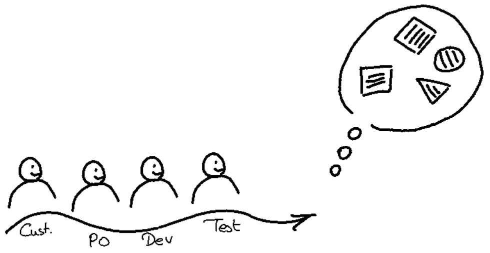
Behaviour-Driven Development for better software products

31 May 2021



We need to minimize friction due to misunderstandings between software development and software specification teams, proposing an innovative approach for software specification. This one should facilitate the common understanding and encourage each stakeholder's accountability, product person, software engineer, test engineer, in the quality of the delivered increments.

Structure

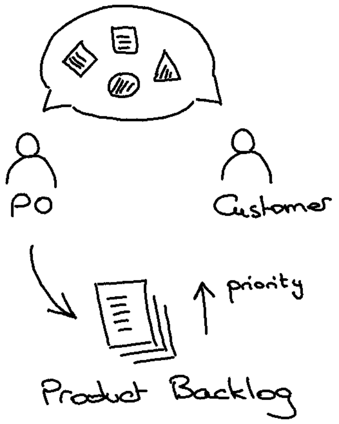
1. The **first** part sets the current context with the traditional approach based on the usual agile tools and roles, illustrating some issues and suggesting remedies.
2. The **second** part outlines Behaviour-Driven Development (BDD) as a recognised software crafting technique that can address the issues listed.
3. The **third** part explains the concrete implementation of this technique.
4. The **fourth** part provides immediately usable material for trying it out.

Context

Traditional approach

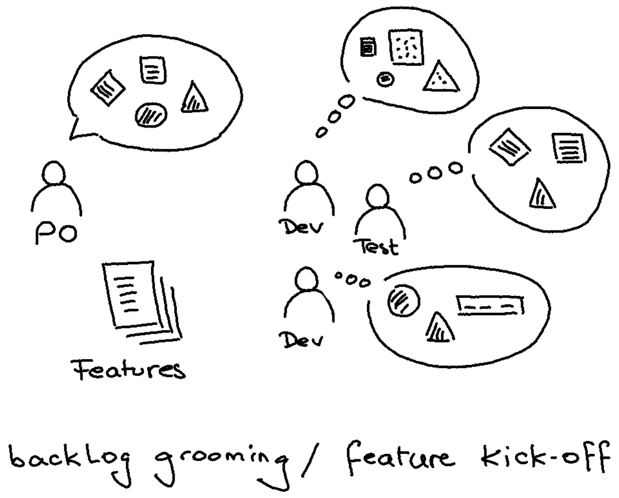
Backlog building

The product owner listens to customer needs, he owns and manages a product backlog composed of features that extend the current product capabilities or current system behaviour.



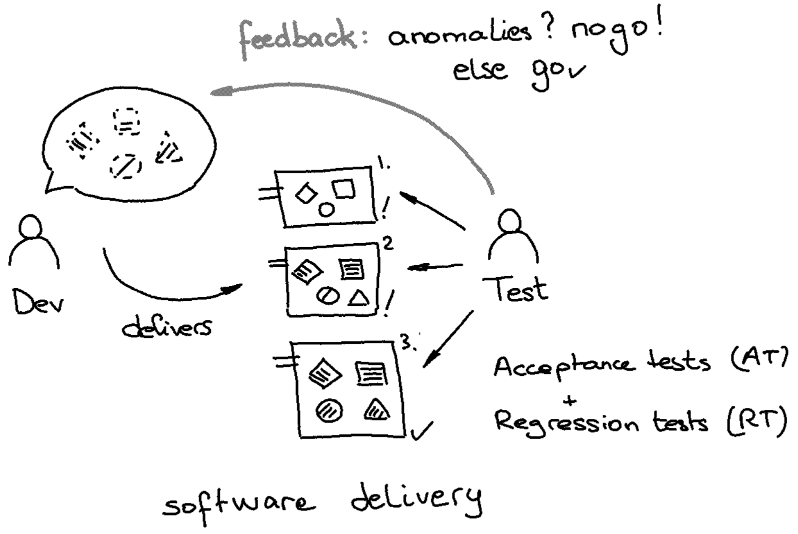
Features understanding

The product owner explains the expected system behaviour to software and test engineers; they try to align their mental representation with the product owner's one challenging him with their perspective during backlog grooming or feature kick-off sessions.



Software delivery

Every delivered functional increment is verified by the test engineer; he checks the completeness and the correctness, and with the best effort, the absence of regression. Anomalies are returned to the development team for the next delivery.

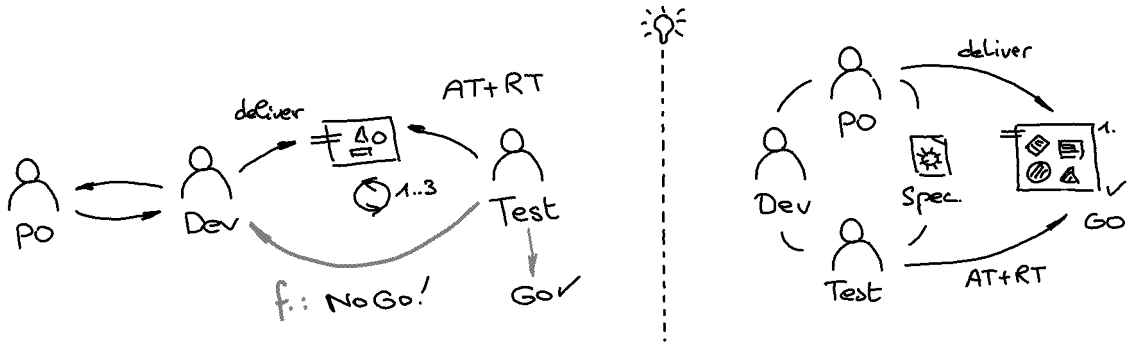


Some weaknesses and remedies

Anomalies

Anomalies due to misalignments or lack of quality in the realisation can cause time-consuming ping-pong loops.

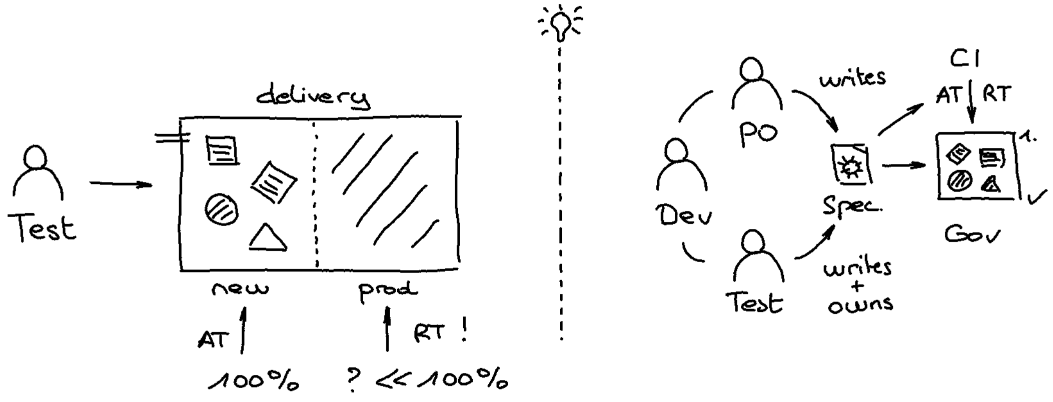
Could an innovative approach that re-establishes a contractual element encourage greater accountability of each stakeholder for *the quality of delivery?*



Cost of regression tests

The testing activity usually focuses on checking newly added features and regressions. Regression testing activity is time-consuming and should be automatized, or unfortunately, may be shortened to well-selected tests.

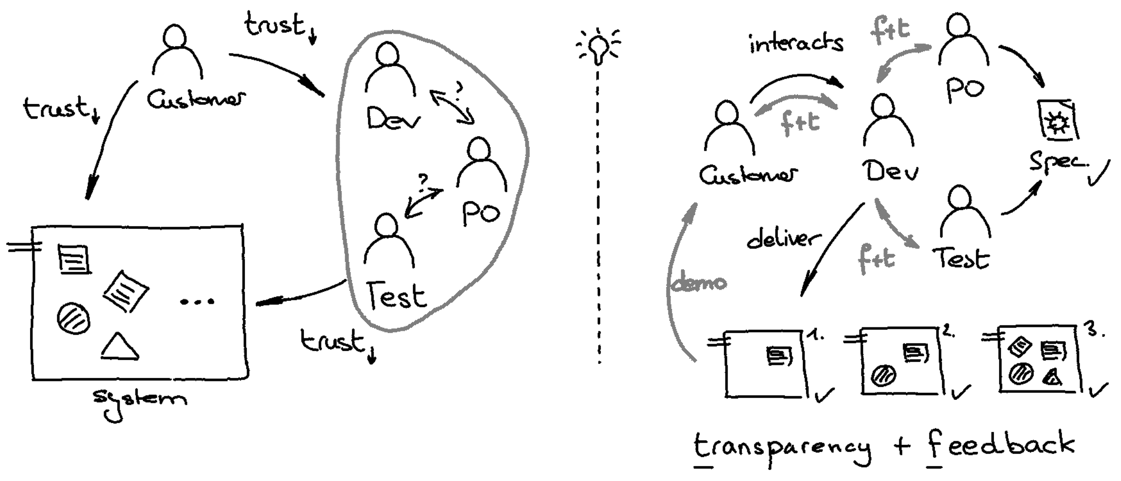
Could we innovate on the tester's activity by automating it and giving him more impact upstream?



Loss of trust

Anomalies, regressions, misalignments can have a bad impact on the already established trust towards the product itself, outside or even inside the product team.

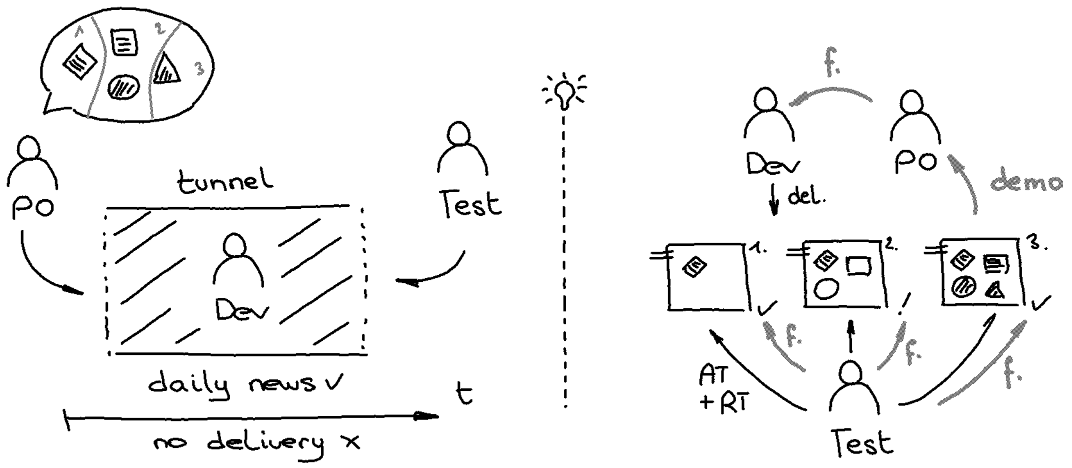
Could a collaborative approach that promotes shared accountability, transparency and regular feedback during implementation build trust?



Tunnel effect

The absence of testable delivery for a couple of time means less feedback, hence less opportunity to react to any misalignment.

Could the division of a feature into several deliverables containing one or more scenarios facilitate deliveries and thus feedback *possibilities?*

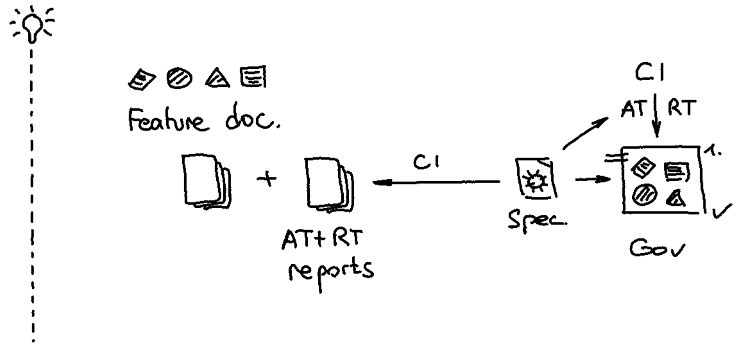


Lack of documentation

What does the system solve? What are the features? What are the nominal and edge scenarios? What about testing evidence?

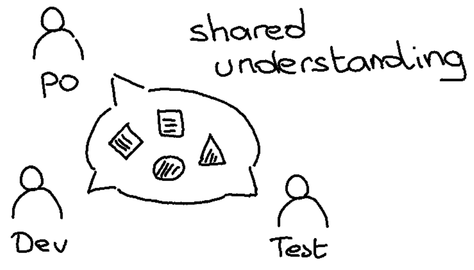
Agile software development manifesto aims to privilege working software over comprehensive documentation. Therefore, the sources of documentation about **what** the developed system solves are production code, test code and mental and often distributed knowledge. Consequently, more pressure is put on well-crafted code. In addition, narrative sources of documentation can still be valuable for setting the context, but they are usually written at a high level and may no longer be aligned with the behaviour of the completed system.

Could a collaborative approach restore a lightweight, human-readable digital artefact that contractually formalises common understanding and be a source of living documentation that provides features and testing evidence?



BDD

Behaviour-Driven Development (BDD) is a software development technique with the main promise to facilitate shared understanding and optimise software deliveries' functional quality.



Secure software development

This technique helps to develop securely **the right thing**, which is the expected system behaviour, and without causing functional regression.

Contract-driven

This technique proposes a context for **collaborative specification** which aims to establish shared understanding of the expected system behaviour, and formalising this latter as a readable **contractual and executable specification** on which acceptance tests can be defined.

Domain-driven

The issued specification artefact uses a natural language based on the **domain language** which is naturally understandable by every stakeholder: product person, software and test engineers.

Documentation-driven

This artefact written with Gherkin syntax explains the system behaviour **based on examples** grouped in scenarios that describe the system use cases and the business rules.

Here is an example that uses a generic scenario to describe the behaviour of a calculation function; the *Examples* section defines the input and output values:

```
Rule: The stress amplitude is the ratio between shock valu
es of two stress driver shocks.

    Scenario Outline: Return expected stress amplitude for
compatible shock types

      Given a stress shock
        | driver-key   | shock-type   | shock-value   |
        | <driver-key> | <shock-type> | <shock-value> |
      And another stress shock
        | driver-key   | shock-type       | shock-value
|
        | <driver-key> | <ref-shock-type> | <ref-shock-val
ue> |
      When calculating amplitude
      Then the returned amplitude is <amplitude>

      Examples:
        | driver-key | shock-type | shock-value | ref-shoc
k-type | ref-shock-value | amplitude |
        | Equity-USA | RELATIVE   | 0.15        | RELATIVE
| 0.10            | 1.5       |
        | Equity-USA | RELATIVE   | 0.25        | RELATIVE
| 0.10            | 2.5       |
        | Equity-USA | RELATIVE   | -0.15       | RELATIVE
| 0.10            | -1.5      |
        | Equity-USA | ABSOLUTE   | 3          | ABSOLUTE
| 2              | 1.5          |
        | Equity-USA | ABSOLUTE   | 2          | ABSOLUTE
| 3              | 0.6667       |
        | Equity-USA | ABSOLUTE   | 2          | RELATIVE
| 0.10           | -            |
        | Equity-USA | RELATIVE   | 0.15       | ABSOLUTE
| 2              | -            |
        | Equity-USA | RELATIVE   | 0.15       | RELATIVE
| 0              | -            |
        | Equity-USA | ABSOLUTE   | 2          | ABSOLUTE
| 0              | -            |
```

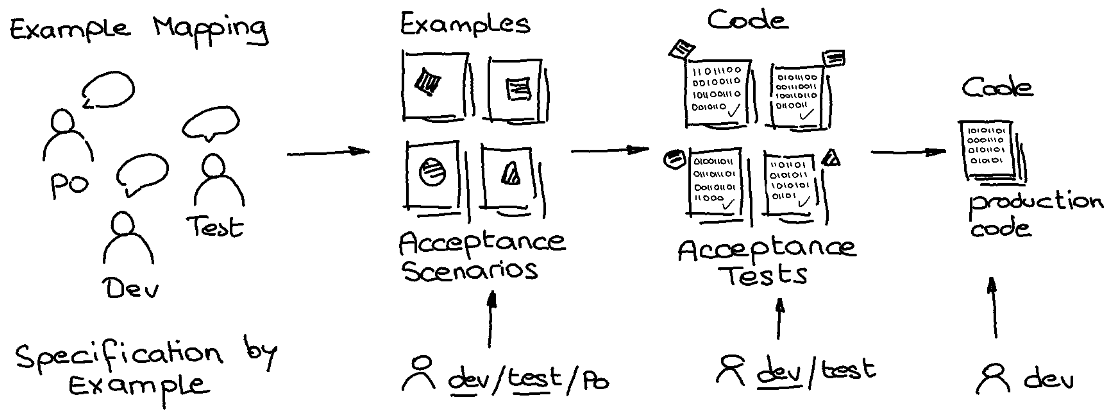
> More BDD scenarios: Blueprint API (Kotlin),

Test-driven

Coupled with Acceptance Test-Driven Development (ATDD), an outside-in software development technique, the software engineer automates the test cases and can be driven to the completeness, correctness, and the absence of functional regression when developing a feature increment.

BDD was originally named in 2003 by Dan North as a response to test-driven development (TDD), including acceptance test or customer test-driven development practices as found in extreme programming.

BDD in practice



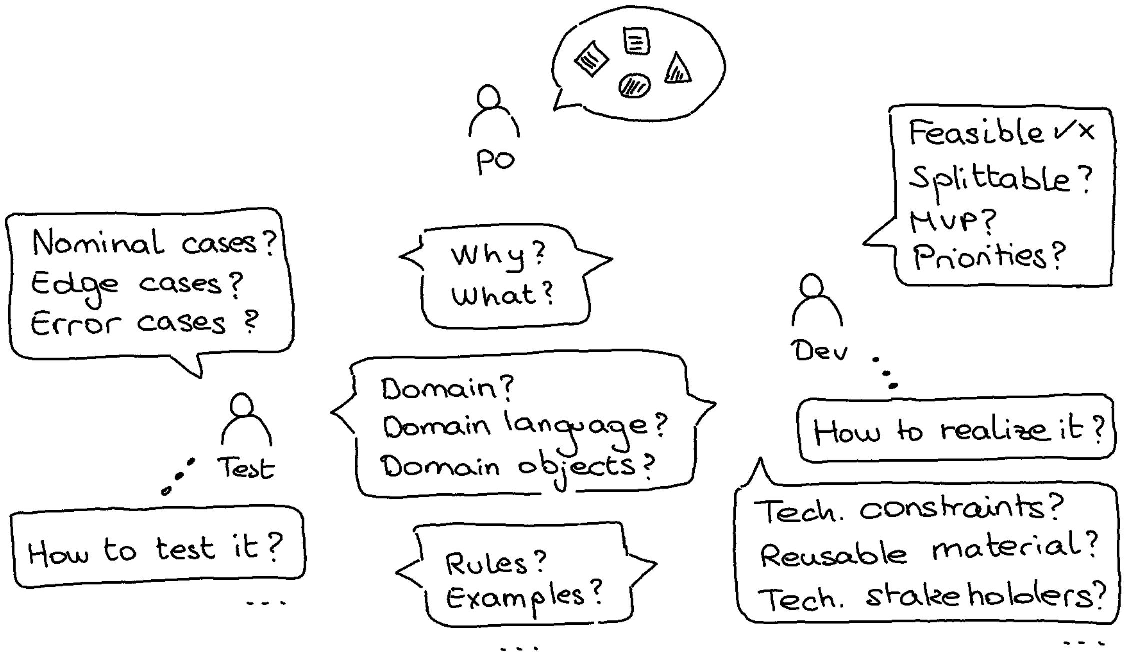
Specification by example activity facilitates a common understanding of expected behaviour based on examples. It applies to a large set of use cases of any domain: calculation, aggregation, orchestration, eventing, management, workflow.

- Specification by example activity is done during **Example Mapping** sessions.
- Formalisation of expected system behaviour is done with **acceptance scenarios** expressed with **domain language** and formalized with Gherkin syntax.
- It requires practice; finding **good wording** have to be learned by doing; efficient wording can be easily understood, can be validated, and can be automatized.

- Automatization means the development of **automatized acceptance tests** enabled by using **glue code**.
- Development of production code is then **driven by existing acceptance tests** (ATDD).

Three amigos

The conversations in the Specification by Example activity are the result of the interaction of three actors with different perspectives, brought together for the same purpose: **Product person** or domain expert, **software engineer, and test engineer**.

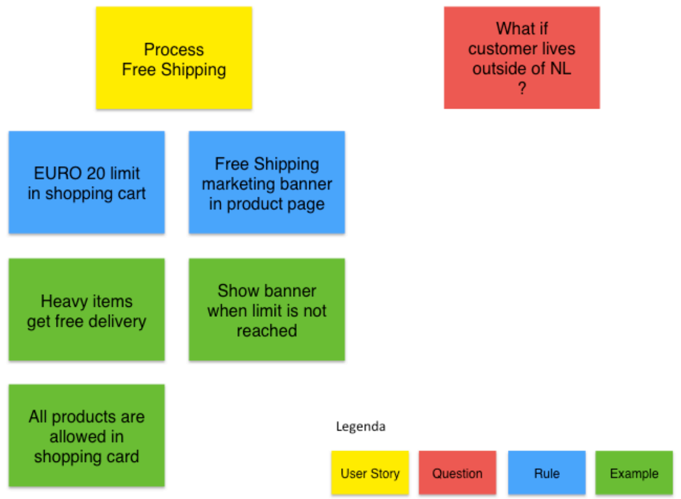


The Product person should be the product owner; the Domain expert (business analyst, domain architect, or business owner) can be a substitute of the Product person; the Customer can be an optional stakeholder.

Example mapping

An Example mapping session facilitates **structured conversations** between the three amigos.

It allows identifying user stories, open questions, business rules, and examples.


Process Free Shipping
What if customer lives outside of NL ?
EURO 20 limit in shopping cart
Free Shipping marketing banner in product page
Heavy items get free delivery
Show banner when limit is not reached
All products are allowed in shopping card
Legenda
User Story
Question
Rule
Example

> xebia.com/blog/example-mapping-steering-the-

> conversation

Feature, story, business rule, scenario, example

- One **feature** is explained by one or more scenarios grouped in stories.
- One **story** usually groups one or more scenarios and represents a feature increment with business value.
- One **business rule** is usually supported by one or more scenarios or examples.
- One **scenario** is supported by one or more **examples**.
- One scenario by nominal case.
- One scenario by edge case.
- One scenario by negative or error case.

Scenarios writing

Writing scenarios helps to build up the common ubiquitous language in a way that **everyone can understand and validate**.

As it requires some experience and a particular way of thinking, it is recommended to let this activity to the test engineer or the software engineer. Feedback by the Product person can be done once they have drafted the Gherkin specification.

A scenario is composed of a set of **preconditions** about the initial state or context, then a set of **actions** (usually one), then a set of **assertions** (or postconditions) about the final state or context.

```
Feature: To be able to manage a set of existing clients in
a persistent way

  Background:
    Given a following set of existing clients
      | model-id | id                                   |
name    | internal | prospect | owners | tags   | creation
-date       |
      | 1        | ce751f30-217a-422c-b81b-8f75df4917b6 |
client1 | true     | false    | x      | key1:a | 2020-10-
10T12:00:00 |
      | 2        | 29e364b9-f5ef-43d9-9f30-e07a30b73e01 |
client2 | true     | true     | -      | -      | 2020-10-
09T12:00:00 |

  Rule: An existing client is a persisted resource in the
system.

    Scenario: Add a new client to the existing clients

      Given a following set of client attributes
        | name | prospect | owners | tags            |
        | test | true     | owner1 | key:test,key1:a |
      And the next identifier is afd9ce9f-ee0e-4547-8c77-3
cc43ec85dbc
      And the next timestamp is 2020-10-11T12:00:00
      When registering the new client
      Then the response status is CREATED
      And the attributes of the returned client are the fo
llowing
        | id                                   | name | pr
ospect | owners | tags             | creation-date       |
        | afd9ce9f-ee0e-4547-8c77-3cc43ec85dbc | test | tr
ue     | owner1 | key:test,key1:a | 2020-10-11T12:00:00 |
      And the returned client is added to the set of exist
ing clients
      And no export of the configuration has been triggere
d
      And a Slack notification has been sent
```

> More BDD scenarios: Blueprint API (Kotlin),

Scenarios validation

Every scenario has to be validated by every stakeholder or amigo, that it describes an expected facet of the system behaviour. As a result, shared understanding is materialized into a digital set of acceptance scenarios that establish a **contract between all stakeholders**.

These **acceptance scenarios** are the foundation of an executable specification and the building blocks for the definition of acceptance tests.

Acceptance testing

Acceptance tests support **high-level functional testing** which includes testing of new feature increment and regression testing.

This activity is usually done by a test engineer of a quality insurance team to validate a delivered feature increment before its deployment to production.

Non-functional requirements could be tested by specific acceptance tests.

ATDD

ATDD is a **test-driven development technique** based on **acceptance tests** used by the software engineer for driving its development to the expected system behaviour.

> Given an acceptance scenario, a failing acceptance test is written first, then the software engineer writes the minimal production code to make it pass. This action is repeated with a next acceptance scenario, coupled with a refactoring phase applied to both test and production code, to take care of well-crafted code and design.

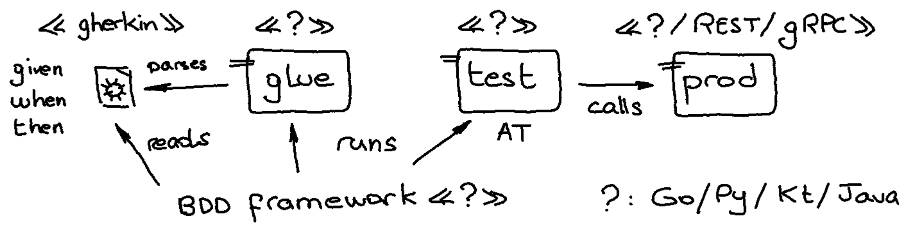
> This cycle executed at the feature level contains an inner TDD cycle performed at the component level, this technique is named Outside-in TDD.

BDD and ATDD

ATDD is used with BDD to **automate acceptance scenarios**: one scenario gives one acceptance test.

BDD glue code, test code, and production code

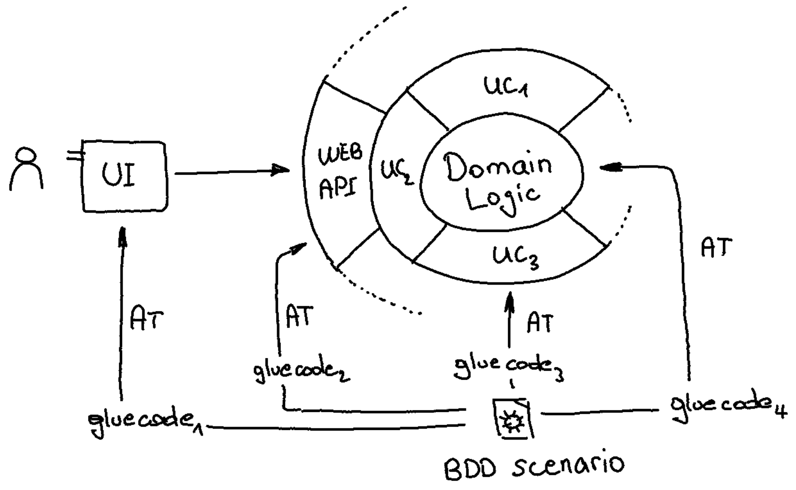
- **Test code** implements acceptance tests and interacts with the **production code**.
- BDD **glue code** implements the mapping of BDD steps writen in natural language into test code.
- BDD steps are mapped by BDD glue code into test code writen in a given technology.
- BDD framework (Cucumber, Behave) traverses the steps and automatizes the execution of the test code.



```
@Given("a stress shock") // glue code
fun oneShock(shock: List<TestShock>) { // glue code
 // test code: fixture
    ctx.put("one-stress-shock", shock.first())
}

@Given("another stress shock") // glue code
fun anotherShock(shock: List<TestShock>) { // glue code
 // test code: fixture
    ctx.put("other-stress-shock", shock.first())
}

@When("calculating amplitude") // glue code
fun calculateAmplitude() { // glue code
 // test code: action
    val one: TestShock = ctx.byId("one-stress-shock")
    val other: TestShock = ctx.byId("other-stress-shock")
    runCatching {
        AmplitudeComputer().compute(one.toShock(), other.t
oShock()) // production code
         }.getOrNull()
    ?.let { ctx.put("result-amplitude", it) }
}

@Then("the returned amplitude is {word}") // glue code
fun returnedAmplitudeValueIs(expected: String) { // glue c
ode
// test code: assertion
    val resultAmplitude: Double? = ctx.byIdOpt("result-amp
litude")
    val expectedAmplitude = expected.toNullable()?.toDoubl
e()
    if (resultAmplitude != null)
        assertThat(resultAmplitude).isCloseTo(expectedAmpl
itude, offset(0.001))
    else
        assertThat(expectedAmplitude).isNull()
}
```

BDD with acceptance test, scope

Well-written BDD scenarios are done using domain language which is naturally high-level so that one scenario could be automated for targetting either a user interacting with a frontend application, or a web API, or a component that supports application logic or domain logic. A specific glue code developed for each target enables this decoupling.



Examples:

- Blueprint API - AT at API level
- Blueprint API - AT at domain level

BDD with acceptance test vs xUnit tests

Acceptance tests are high-level integration tests defined at feature-level. Supported by the natural language and the Gherkin syntax, their self-documentation is accessible to every stakeholder and emphasizes the implemented and tested behaviour with its preconditions and postconditions. Acceptance tests usually cover functional requirements, and could check non-functional ones as well. **Smoke tests** can be supported by a subset of acceptance tests.

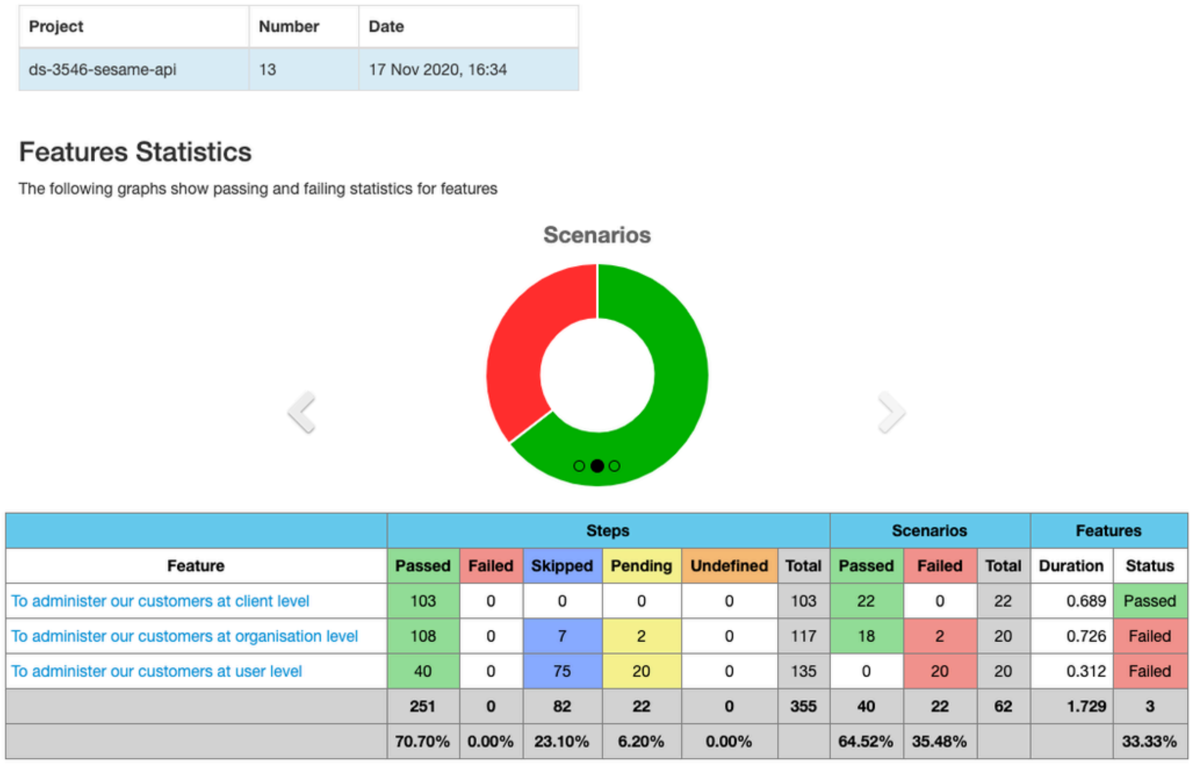
xUnit tests, due to their technical aspect, do not connect every stakeholder, hence do not provide evidence of which functional requirements are implemented and tested.

BDD and agility

Agile ideology influences methodologies for iterative development of small increments, enabling quick feedback and adapt.

Delivering the most valuable scenarios inside one or more stories has a natural priority, then further valuable set of scenarios can be delivered in other increments.

Scenarios are developed against single acceptance tests which can be executed by a continuous integration tool. This tooling can be configured to monitor which scenarios have been delivered and which ones are still under development. This promotes transparency and enables feedback, early reaction, and prediction towards the deadline.



Project	Number	Date
ds-3546-sesame-api	13	17 Nov 2020, 16:34

Features Statistics

The following graphs show passing and failing statistics for features

	Steps						Scenarios			Features	
Feature	Passed	Failed	Skipped	Pending	Undefined	Total	Passed	Failed	Total	Duration	Status
To administer our customers at client level	103	0	0	0	0	103	22	0	22	0.689	Passed
To administer our customers at organisation level	108	0	7	2	0	117	18	2	20	0.726	Failed
To administer our customers at user level	40	0	75	20	0	135	0	20	20	0.312	Failed
	251	0	82	22	0	355	40	22	62	1.729	3
	70.70%	0.00%	23.10%	6.20%	0.00%		64.52%	35.48%			33.33%

> plugins.jenkins.io/cucumber-reports/

Living documentation

Living documentation is made possible based on BDD-driven specification with tooling provided by Serenity-BDD.

Evidence of system well-being

- With Serenity-BDD test execution reports tooling, proof of evidence can be provided to auditors that all requirements covered by the system in place are supported by documented and continuous acceptance tests. It supports Java technology only.

- With Allure test execution reports tooling, more evidence can be provided to auditors that testing is done in-depth, even at the lower levels of the test pyramid i.e component integration testing and component unit testing, and uniformly for a couple of technologies (Python/Go/Java/Kotlin).

Summary

Outcomes

- **Common understanding** of expected system behaviour expressed in a domain-oriented ubiquitous language.
- Enhanced probability that the **implemented thing is the right one**.
- Human-readable, **contractual and executable specification** directly useable by every actor of the development team.
- Scenario-based, use-case-driven and domain-driven **documentation of system behaviour**.
- Scenario-based organization and **survey of feature development**.
- Innovation brought to the **tester role**, the test engineer can now have a significant impact on the development.

Costs

- Time: to attend, organize and facilitate Example mapping meetings
- Time: to learn one new technique
- Time: to learn how to write BDD-scenarios
- Time: to setup a BDD-framework
- Time: to craft reusable glue code and test code
- Time + Money: to maintain Gherkin phrasing, glue code and test code

This technique should reveal itself as a sustainable investment in product quality.

BDD material

Technical setup

Ready to use **blueprints** are available as BDD starters for some technology stack:

- Java BDD blueprint (Cucumber)
- Kotlin BDD blueprint (Cucumber)

References

Literature

- Example Mapping
- BDD in Action
- Serenity-BDD

Links

- Cucumber BDD framework
- Behave BDD framework
- Allure test reporting

Related posts

About me 14 Apr 2025

Technical Excellence or the mastery of practices 22 Feb 2020

Software Craftsmanship is above all a mindset 22 Feb 2020