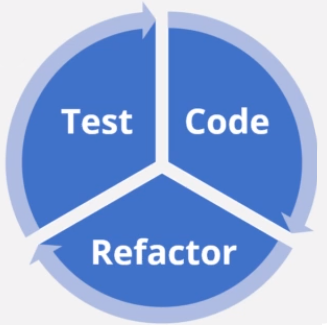
**TDD**

Test driven development - it is primarily used for unit testing by the development team. It is not a methodology like waterfall or Agile that talks overall (общий) software development cycle, but **it's more of programming style or practice** that talks about how to write the code in iterations (iterations - means during software development iteration). There are mainly three steps in TDD: **Test, Code, Refuctor**.

1)- The developer would first write an unit test that would describe the functionality to be developed. The test will execute and it will fail as expected.

2) - The developer would then write just enough code to make the test pass.

3) - The developer than would refactor the code, to remove any code duplication or any inconsistency (непоследовательности) in the code. 

Then the same process is repeated.

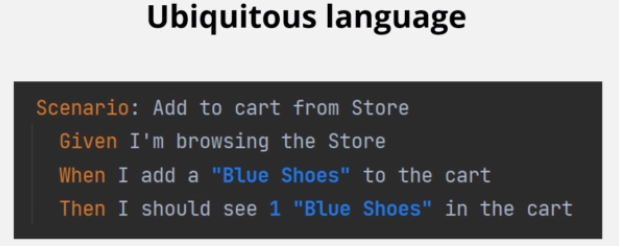
This way the developer keeps accumulating unit tests that form the safety net (подстраховку) for the code quality. This practice helps developers to identify the defects (bags) in the developers phase itself. That can reduce the amount of defects that would escape to the later phases of software development, where it can become very costly to fix them. Overall software quality increases because now any change in the code may fail the unit test, and the defects will be caught during development (code phase) itself. In the end, this helps reduce the overall cost at the expense of a slightly delayed development cycle.

**BDD**  <- - этот шаг не очень информативен

One area where TDD can fall short (не дотягивать) is that the unit tests would be often difficult to read and comprehend (постигать/понимать). It is possible that the developer would not write descriptive test method names. In that case it would be very difficult, especially if many unit tests are written in this manner.

The North, who noticed this (it is a creator of BDD), he has an opinion that if test method names were a descriptive enough, he could derive (вывести/определить) the specification directly from the unit tests.

This was the moment when he realized that it was indeed possible to write specs from the tests. Другими словами если названия методов/тестов будут достаточно информативны то можно создать документ для заказчика где он бы мог отслеживать этапы разработки без знания технической части, но опираясь на эти названия методов было бы понятно что происходит на данном этапе разработки.

Then North discovered that this could help a lot for developers if they are able to automatically document their unit test in this manner. And if the method name would be written in the language known for everybody in the company, then the generated document would make more sense for business users, analysts, and testers as well. Then North started thinking more of behavior and not tests. He started writing unit tests in terms of **should be able to**. Which talks more about the behavior of the class. Basically each class should be able to do something specific. This sounds more profound (более глубокий) than merely writing it in terms of a test. Then he created a tool, jbehave, that would help write unit test in the form behaviors. But it was helpful only for developers, so he was thinking about consistent vocabulary (какой то постоянно используемый словарь)for all the stakeHolders (developers, business, testers…), so it would increase communication and understanding between all company. So this thought gave birth what we call **Behavior Driven Development - it is an agile software development practice that intends to close the gap between business people and technical people by encouraging collaboration between them** . By building shared understanding of the problem to be solved, by working in small rapid iterations to increase feedback and the flow of value, and by producing system documentation that is automatically checked against the system`s behavior. So in BDD requirements are written in a very simple, business focused language called as ubiquitous (вездесущий). that is understood by all stakeholders. This effectively bridges the gap between technical and not technical people.

The business analysts have conversations with the developers and testers around the user stories (чё это?), scenarios, examples, this can happen right from the requirements discovery phase.The scenarios written in the structured way. The most preferred way is the gherkin syntax that is uses: Given, then, and… statements. The scenarios are backed (поддержанный) by concrete, real world examples that illustrate how the system should behave. This eliminates any miscommunications and any hidden assumptions among the stakeholders about the behavior of the system to be developed. This scenarios and examples form something that is called **acceptance criteria**. Acceptance criteria helps to understand when a feature can be marked as done. It gives the clearer picture of the developer what should be developed. Testers use it to test the feature.

Acceptance criteria are converted to executable specifications that can be automated (здесь речь идёт о реализации feature файла в java классе). Cucumber can be used to automate executable specification. These can be executed in the continuous integration environment, ensuring new changes have not broken any existing feature. Automation generates a living documentation that forms a single source of truth for all stakeholders. The reports generated by automation form product documentation that is expressed in a consistent vocabulary and understood by all the stakeholders. The documentation is always up to date hence called **living documentation**. Since the automation runs for every build, this documentation effectively illustrates the current state of the application, which features are working, which are broken, which are pending (в ожидании) and so on. The documentation serves as a single source of truth for all stakeholders.

In a nutshell BDD is all about conversations/communications.

**BDD/ TDD/ AGILE**

BDD - is all about collaboration between business teams and technical teams. The business team understands the user`s scenarios and how users interact with the application, on the other hand we have developers, testers, automation engineers who do the implementation part. And usually there's a gap in understanding between them. BDD generates a common documentation that can be understood by all teams and this documentation is the source of truth for all the teams.

BDD assumes that the team uses some kind of agile methodology, for example **scrum**, work is planned in small increments of value called as user stories (informal/неофициальный, general explanation of software feature written from the perspective of the end user). BDD can starts when the team is about to develop a user story.

-The team should start conversations around the user story/feature to come up with concrete examples that illustrate the behavior of the feature to be developed. This phase is called **discovery**.

-The team then converts the examples to executable specification (feature fail), typically using the gherkin syntax. This phase is called **formulation**.

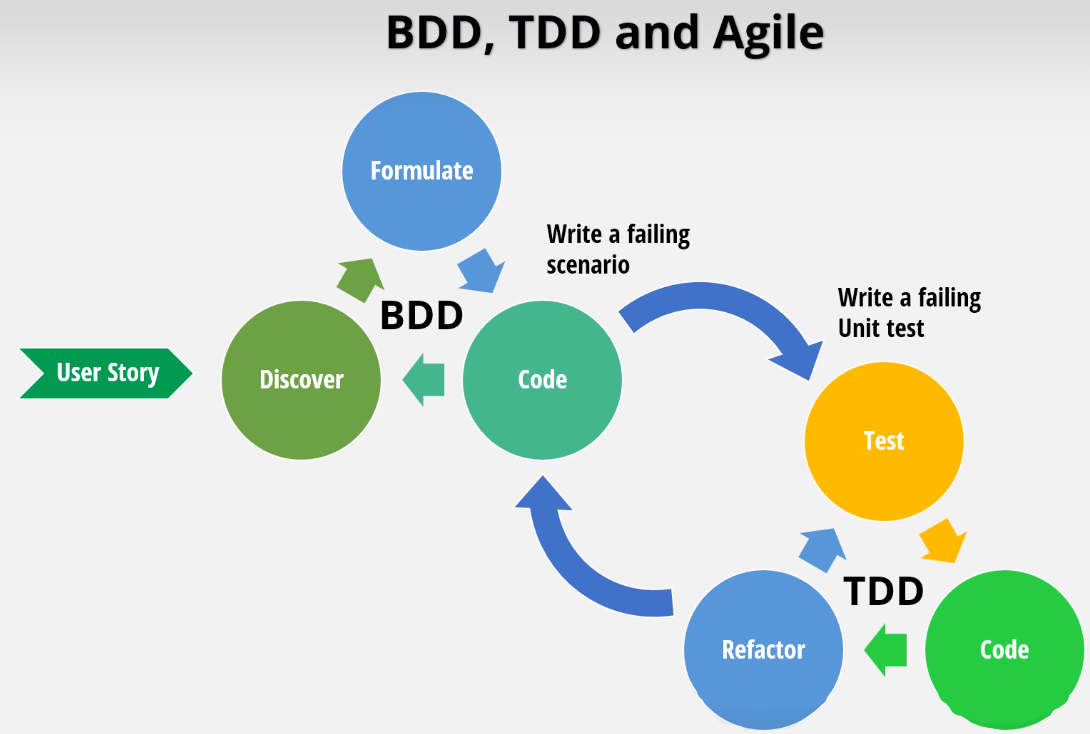
-Than just enough code is written to fill the scenario, this phase is called **automation/code**.

All these phases can happen in iteration. The conversations would start multiple times, depending on the complexity of the user story, and until the team is able to establish concrete examples. Defining clear acceptance criteria which will be used as input for the development team, and this is where TDD process would start. The developer would than start design and write unit tests that would fail - phase **test.**

- Then code will be written just enough to pass unit tests- phase **code.**

- Then **code refactor**

The final code will then serve as an input back to BDD specification that can be executed automatically against the code developed. This is where the BDD scenario will pass, the acceptance criteria would be fulfilled and the user story would be declared done. In the case of TDD there can be as well multiple iterations until the code is ready and valid.



**So teams that follow BDD efficiently usually follow 3 important practices.**

**discovery** - the technical people and non technical people pick up user story/feature to be developed and have structured conversations also called **discovery workshop**, these conversations focus on:

1)-building the shared understanding of the needs of the users

2)-identifying and understanding the business rules that govern how the system should function

3)- scope (объем) of what should be done

The outcome is: agreed behavior of the system.

There are different models to conduct discovery workshops. One of them is **example mapping.** In this activity the participants/патисипэнтс don't really have to write the examples in Gherkin.

**formulation**- the team would document those examples in away that can be automated and checked for agreement. typically gherkin is used in this phase. (executable specifications)

**automation** - the scenarios are glued to the code. Initially these scenarios would fail. At the same time the developer would develop the code. The important note is that the automated test would guide the development of the application.

The main idea to make change small and iterate rapidly, moving back if needed more information

**Example mapping !!!**

It is designed to make conversations short and productive. Basically 4 indexed colored cards are used and arranged in a map.

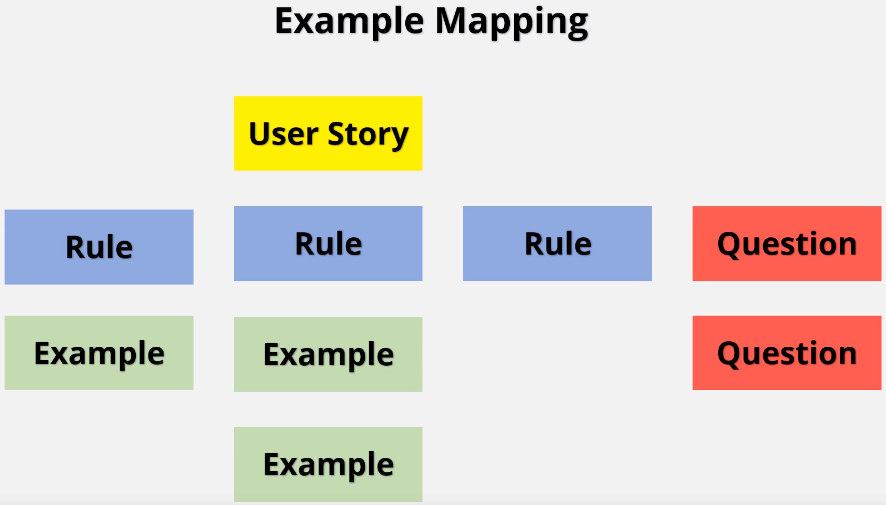
1) - User story/feature is written on the yellow card and placed on top

2) - Business rules that governs the user story are written on a blue card and placed beneath the yellow user story card.

3) - Examples to illustrate these rules are written on a green card and placed under the relevant (соответствующий) rule card

4) - Questions that can not be answered during the session are captured on a red card so that team might move on with the conversation.

The conversation keeps on going until the amigos are satisfied that the scope of the user story is clear or the team runs out of time.



Rules to remember while doing Example mapping:

1)- If there are too many questions, this may be the indication that there are too many known unknowns. The team might use research and come back with another session,

2)- If there are too many rules, this may be an indication that the story is too big and complicated. The team might want to split it to smaller.

3)- If there is a rule with too many examples it might be an indication that there are multiple rules.

4)- You might find that some rules are so obvious, you don't need examples at all.

5)- It is ok if you can not scope the story in 25-30 minutes. This may be an indication that some more work is required before the story can be taken up.

6)- Avoid using Gherkin for the examples.