V Skills
Skills for a secure future

Certified Capybara Testing Professional
VS-1164

V-Skills Certifications

A Government of India
&
Government of NCT Delhi Initiative

V-Skills

Skills for a secure future

1. CAPYBARA BASICS

Before starting with Capybara, this section illustrates the basic concepts like TDD, BDD, DSL, etc.

1.1. TDD

Test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards. Kent Beck, who is credited with having developed or 'rediscovered' the technique, stated in 2003 that TDD encourages simple designs and inspires confidence.

Test-driven development is related to the test-first programming concepts of extreme programming, begun in 1999, but more recently has created more general interest in its own right.

Programmers also apply the concept to improving and debugging legacy code developed with older techniques.

Test-driven development cycle

1. Add a test
In test-driven development, each new feature begins with writing a test. To write a test, the developer must clearly understand the feature's specification and requirements. The developer can accomplish this through use cases and user stories to cover the requirements and exception conditions, and can write the test in whatever testing framework is appropriate to the software environment. It could be a modified version of an existing test. This is a differentiating feature of test-driven development versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code, a subtle but important difference.

2. Run all tests and see if the new one fails
This validates that the test harness is working correctly, that the new test does not mistakenly pass without requiring any new code, and that the required feature does not already exist. This step also tests the test itself, in the negative: it rules out the possibility that the new test always passes, and therefore is worthless. The new test should also fail for the expected reason. This step increases the developer's confidence that the unit test is testing the correct constraint, and passes only in intended cases.
3. Write some code
The next step is to write some code that causes the test to pass. The new code written at this stage is not perfect and may, for example, pass the test in an inelegant way. That is acceptable because it will be improved and honed in Step 5.

At this point, the only purpose of the written code is to pass the test; no further (and therefore untested) functionality should be predicted nor 'allowed for' at any stage.

www.vskills.in

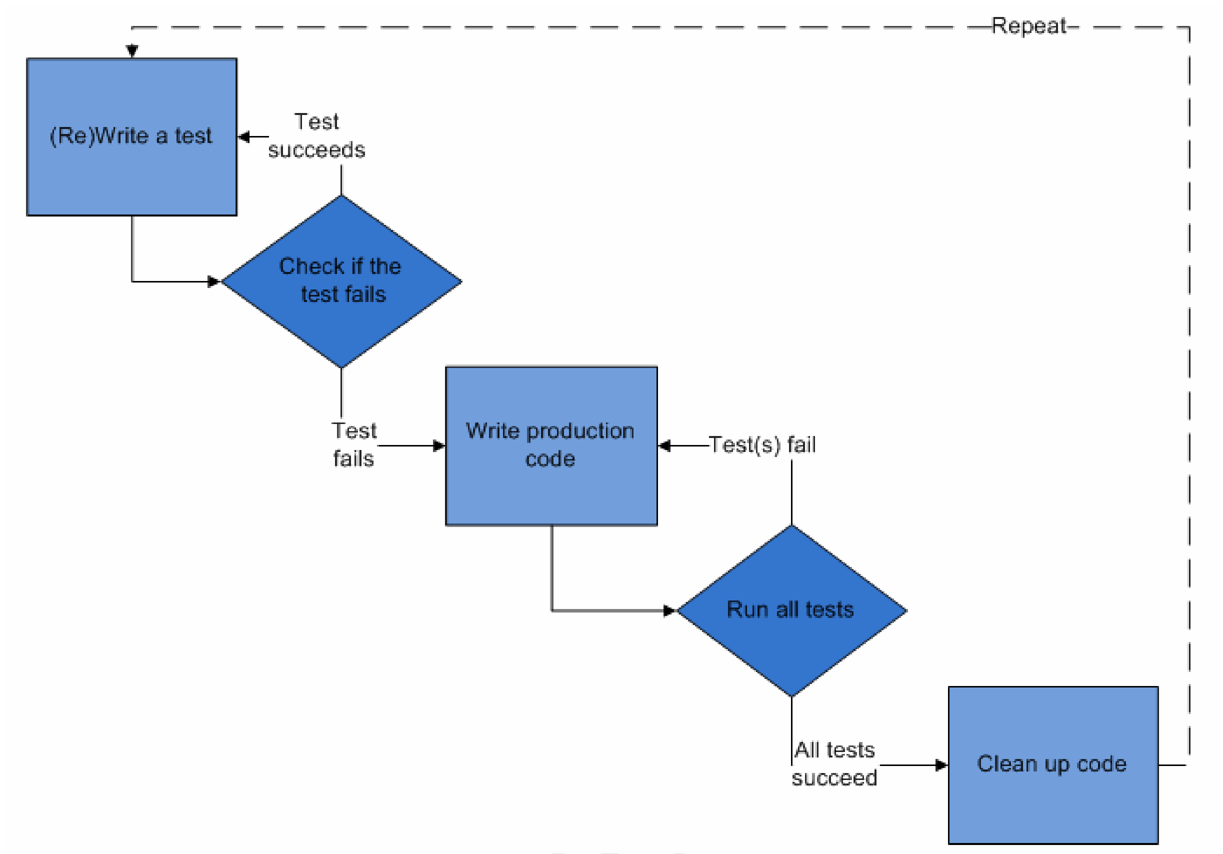
4. Run tests
If all test cases now pass, the programmer can be confident that the new code meets the test requirements, and does not break or degrade any existing features. If they do not, the new code must be adjusted until they do.

5. Refactor code
The growing code base must be cleaned up regularly during test-driven development. New code can be moved from where it was convenient for passing a test to where it more logically belongs. Duplication must be removed. Object, class, module, variable and method names should clearly represent their current purpose and use, as extra functionality is added. As features are added, method bodies can get longer and other objects larger. They benefit from being split and their parts carefully named to improve readability and maintainability, which will be increasingly valuable later in the software lifecycle. Inheritance hierarchies may be rearranged to be more logical and helpful, and perhaps to benefit from recognised design patterns. There are specific and general guidelines for refactoring and for creating clean code. By continually re-running the test cases throughout each refactoring phase, the developer can be confident that process is not altering any existing functionality.

The concept of removing duplication is an important aspect of any software design. In this case, however, it also applies to the removal of any duplication between the test code and the production code—for example magic numbers or strings repeated in both to make the test pass in Step 3.

Repeat
Starting with another new test, the cycle is then repeated to push forward the functionality. The size of the steps should always be small, with as few as 1 to 10 edits between each test run. If new code does not rapidly satisfy a new test, or other tests fail unexpectedly, the programmer should undo or revert in preference to excessive debugging. Continuous integration helps by providing revertible checkpoints. When using external libraries it is important not to make increments that are so small as to be effectively merely testing the library itself, unless there is some reason to believe that the library is buggy or is not sufficiently feature-complete to serve all the needs of the software under development.

www.vskills.in



Best Practices

Test structure

Effective layout of a test case ensures all required actions are completed, improves the readability of the test case, and smoothens the flow of execution. Consistent structure helps in building a self-documenting test case. A commonly applied structure for test cases has (1) setup, (2) execution, (3) validation, and (4) cleanup.

- ✓ Setup: Put the Unit Under Test (UUT) or the overall test system in the state needed to run the test.
- ✓ Execution: Trigger/drive the UUT to perform the target behavior and capture all output, such as return values and output parameters. This step is usually very simple.
- ✓ Validation: Ensure the results of the test are correct. These results may include explicit outputs captured during execution or state changes in the UUT & UAT.
- ✓ Cleanup: Restore the UUT or the overall test system to the pre-test state. This restoration permits another test to execute immediately after this one.

Individual best practices

Individual best practices states that one should:

- ✓ Separate common set-up and teardown logic into test support services utilized by the appropriate test cases.
- ✓ Keep each test oracle focused on only the results necessary to validate its test.

www.vskills.in

- ✓ Design time-related tests to allow tolerance for execution in non-real time operating systems. The common practice of allowing a 5-10 percent margin for late execution reduces the potential number of false negatives in test execution.
- ✓ Treat your test code with the same respect as your production code. It also must work correctly for both positive and negative cases, last a long time, and be readable and maintainable.
- ✓ Get together with your team and review your tests and test practices to share effective techniques and catch bad habits. It may be helpful to review this section during your discussion.

Practices to avoid, or "anti-patterns"

- ✓ Having test cases depend on system state manipulated from previously executed test cases.
- ✓ Dependencies between test cases. A test suite where test cases are dependent upon each other is brittle and complex. Execution order should not be presumed. Basic refactoring of the initial test cases or structure of the UUT causes a spiral of increasingly pervasive impacts in associated tests.
- ✓ Interdependent tests. Interdependent tests can cause cascading false negatives. A failure in an early test case breaks a later test case even if no actual fault exists in the UUT, increasing defect analysis and debug efforts.
- ✓ Testing precise execution behavior timing or performance.
- ✓ Building “all-knowing oracles.” An oracle that inspects more than necessary is more expensive and brittle over time. This very common error is dangerous because it causes a subtle but pervasive time sink across the complex project.
- ✓ Testing implementation details.
- ✓ Slow running tests.

Benefits

A 2005 study found that using TDD meant writing more tests and, in turn, programmers who wrote more tests tended to be more productive. Hypotheses relating to code quality and a more direct correlation between TDD and productivity were inconclusive.

Programmers using pure TDD on new ("greenfield") projects reported they only rarely felt the need to invoke a debugger. Used in conjunction with a version control system, when tests fail unexpectedly, reverting the code to the last version that passed all tests may often be more productive than debugging.

Test-driven development offers more than just simple validation of correctness, but can also drive the design of a program. By focusing on the test cases first, one must imagine how the functionality is used by clients (in the first case, the test cases). So, the programmer is concerned with the interface before the implementation. This benefit is complementary to Design by Contract as it approaches code through test cases rather than through mathematical assertions or preconceptions.

Test-driven development offers the ability to take small steps when required. It allows a programmer to focus on the task at hand as the first goal is to make the test pass. Exceptional cases and error handling are not considered initially, and tests to create these extraneous circumstances are implemented separately. Test-driven development ensures in this way that all written code is

www.vskills.in

covered by at least one test. This gives the programming team, and subsequent users, a greater level of confidence in the code.

While it is true that more code is required with TDD than without TDD because of the unit test code, the total code implementation time could be shorter based on a model by Müller and Padberg. Large numbers of tests help to limit the number of defects in the code. The early and frequent nature of the testing helps to catch defects early in the development cycle, preventing them from becoming endemic and expensive problems. Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project.

TDD can lead to more modularized, flexible, and extensible code. This effect often comes about because the methodology requires that the developers think of the software in terms of small units that can be written and tested independently and integrated together later. This leads to smaller, more focused classes, looser coupling, and cleaner interfaces. The use of the mock object design pattern also contributes to the overall modularization of the code because this pattern requires that the code be written so that modules can be switched easily between mock versions for unit testing and "real" versions for deployment.

Because no more code is written than necessary to pass a failing test case, automated tests tend to cover every code path. For example, for a TDD developer to add an else branch to an existing if statement, the developer would first have to write a failing test case that motivates the branch. As a result, the automated tests resulting from TDD tend to be very thorough: they detect any unexpected changes in the code's behaviour. This detects problems that can arise where a change later in the development cycle unexpectedly alters other functionality.

Madeyski provided an empirical evidence (via a series of laboratory experiments with over 200 developers) regarding the superiority of the TDD practice over the classic Test-Last approach, with respect to the lower coupling between objects (CBO). The mean effect size represents a medium (but close to large) effect on the basis of meta-analysis of the performed experiments which is a substantial finding. It suggests a better modularization (i.e., a more modular design), easier reuse and testing of the developed software products due to the TDD programming practice. Madeyski also measured the effect of the TDD practice on unit tests using branch coverage (BC) and mutation score indicator (MSI), which are indicators of the thoroughness and the fault detection effectiveness of unit tests, respectively. The effect size of TDD on branch coverage was medium in size and therefore is considered substantive effect.

Limitations

Test-driven development does not perform sufficient testing in situations where full functional tests are required to determine success or failure, due to extensive use of unit tests. Examples of these are user interfaces, programs that work with databases, and some that depend on specific network configurations. TDD encourages developers to put the minimum amount of code into such modules and to maximize the logic that is in testable library code, using fakes and mocks to represent the outside world.

Management support is essential. Without the entire organization believing that test-driven development is going to improve the product, management may feel that time spent writing tests is wasted.

www.vskills.in

Unit tests created in a test-driven development environment are typically created by the developer who is writing the code being tested. Therefore, the tests may share blind spots with the code: if, for example, a developer does not realize that certain input parameters must be checked, most likely neither the test nor the code will verify those parameters. Another example: if the developer misinterprets the requirements for the module he is developing, the code and the unit tests he writes will both be wrong in the same way. Therefore, the tests will pass, giving a false sense of correctness.

A high number of passing unit tests may bring a false sense of security, resulting in fewer additional software testing activities, such as integration testing and compliance testing.

Tests become part of the maintenance overhead of a project. Badly written tests, for example ones that include hard-coded error strings or are themselves prone to failure, are expensive to maintain. This is especially the case with fragile tests. There is a risk that tests that regularly generate false failures will be ignored, so that when a real failure occurs, it may not be detected. It is possible to write tests for low and easy maintenance, for example by the reuse of error strings, and this should be a goal during the code refactoring phase described above.

Writing and maintaining an excessive number of tests costs time. Also, more-flexible modules (with limited tests) might accept new requirements without the need for changing the tests. For those reasons, testing for only extreme conditions, or a small sample of data, can be easier to adjust than a set of highly detailed tests. However, developers could be warned about overtesting to avoid the excessive work, but it might require advanced skills in sampling or factor analysis.

The level of coverage and testing detail achieved during repeated TDD cycles cannot easily be re-created at a later date. Therefore these original, or early, tests become increasingly precious as time goes by. The tactic is to fix it early. Also, if a poor architecture, a poor design, or a poor testing strategy leads to a late change that makes dozens of existing tests fail, then it is important that they are individually fixed. Merely deleting, disabling or rashly altering them can lead to undetectable holes in the test coverage.

TDD and ATDD

Test-Driven Development is related to, but different from Acceptance Test-Driven Development (ATDD). TDD is primarily a developer's tool to help create well-written unit of code (function, class, or module) that correctly performs a set of operations. ATDD is a communication tool between the customer, developer, and tester to ensure that the requirements are well-defined. TDD requires test automation. ATDD does not, although automation helps with regression testing. Tests used In TDD can often be derived from ATDD tests, since the code units implement some portion of a requirement. ATDD tests should be readable by the customer. TDD tests do not need to be.

TDD and BDD

BDD (Behavior-driven development) combines practices from TDD and from ATDD. It includes the practice of writing tests first, but focuses on tests which describe behavior, rather than tests which test a unit of implementation. Tools such as Mspec and Specflow provide a syntax which allow non-programmers to define the behaviors which developers can then translate into automated tests.

www.vskills.in

1.2. BDD

In software engineering, behavior-driven development (BDD) is a software development process that emerged from test-driven development (TDD). Behavior-driven development combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software development and management teams with shared tools and a shared process to collaborate on software development.

Although BDD is principally an idea about how software development should be managed by both business interests and technical insight, the practice of BDD does assume the use of specialized software tools to support the development process. Although these tools are often developed specifically for use in BDD projects, they can be seen as specialized forms of the tooling that supports test-driven development. The tools serve to add automation to the ubiquitous language that is a central theme of BDD.

BDD is largely facilitated through the use of a simple domain specific language (DSL) using natural language constructs (e.g., English like sentences) that can express the behavior and the expected outcomes. Test scripts have long been a popular application of DSLs with varying degrees of sophistication.

Cucumber is a behavior-driven development (BDD) acceptance test framework

- ✓ Capybara, Acceptance test framework for Ruby web applications
- ✓ Behat, BDD acceptance framework for PHP
- ✓ Lettuce, BDD acceptance framework for Python

History

Behavior-driven development is an extension of test-driven development: development that leverages a simple, domain-specific scripting language. These DSLs convert structured natural language statements into executable tests. The result is a closer relationship to acceptance criteria for a given function and the tests used to validate that functionality. As such it is a natural extension of TDD testing in general.

BDD focused on:

- ✓ Where to start in the process
- ✓ What to test and what not to test
- ✓ How much to test in one go
- ✓ What to call the tests
- ✓ How to understand why a test fails

At the heart of BDD is a rethinking of the approach to the unit testing and acceptance testing that naturally arise with these issues. For example, BDD suggests that unit test names be whole sentences starting with a conditional verb ("should" in English for example) and should be written in order of business value. Acceptance tests should be written using the standard agile framework of a User story: "As a [role] I want [feature] so that [benefit]". Acceptance criteria should be written

www.vskills.in

in terms of scenarios and implemented as classes: Given [initial context], when [event occurs], then [ensure some outcomes].

Starting from this point, many people developed BDD frameworks over a period of years, finally framing it as a communication and collaboration framework for developers, QA and non-technical or business participants in a software project. During the "Agile specifications, BDD and Testing eXchange" in November 2009 in London, Dan North gave the following description of BDD:
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 created a BDD framework, JBehave, followed by a story-level BDD framework for Ruby called RBehave which was later integrated into the RSpec project. He also worked with David Chelimsky, Aslak Hellesøy and others to develop RSpec and also to write "The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends". The first story-based framework in RSpec was later replaced by Cucumber mainly developed by Aslak Hellesøy.

Principles of BDD

At its core, behavior-driven development is a specialized form of Hoare logic applied to test-driven development which focuses on behavioral specification of software units using the Domain Language of the situation.

Test-driven development is a software development methodology which essentially states that for each unit of software, a software developer must:

- ✓ define a test set for the unit first;
- ✓ then implement the unit;
- ✓ finally verify that the implementation of the unit makes the tests succeed.

This definition is rather non-specific in that it allows tests in terms of high-level software requirements, low-level technical details or anything in between. One way of looking at BDD therefore, is that it is a continued development of TDD which makes more specific choices than TDD.

Behavior-driven development specifies that tests of any unit of software should be specified in terms of the desired behavior of the unit. Borrowing from agile software development the "desired behavior" in this case consists of the requirements set by the business — that is, the desired behavior that has business value for whatever entity commissioned the software unit under construction. Within BDD practice, this is referred to as BDD being an "outside-in" activity.

Behavioural specifications
Following this fundamental choice, a second choice made by BDD relates to how the desired behavior should be specified. In this area BDD chooses to use a semi-formal format for behavioral specification which is borrowed from user story specifications from the field of object-oriented analysis and design. BDD specifies that business analysts and developers should collaborate in this area and should specify behavior in terms of user stories, which are each explicitly written down in a dedicated document. Each user story should, in some way, follow the following structure:

www.vskills.in

Title: The story should have a clear, explicit title.
Narrative
A short, introductory section that specifies

- ✓ who (which business or project role) is the driver or primary stakeholder of the story (the actor who derives business benefit from the story)
- ✓ what effect the stakeholder wants the story to have
- ✓ what business value the stakeholder will derive from this effect

Acceptance criteria or scenarios
a description of each specific case of the narrative. Such a scenario has the following structure:

- ✓ It starts by specifying the initial condition that is assumed to be true at the beginning of the scenario. This may consist of a single clause, or several.
- ✓ It then states which event triggers the start of the scenario.
- ✓ Finally, it states the expected outcome, in one or more clauses.

BDD does not have any formal requirements for exactly how these user stories must be written down, but it does insist that each team using BDD come up with a simple, standardized format for writing down the user stories which includes the elements listed above. However, in 2007 Dan North suggested a template for a textual format which has found wide following in different BDD software tools. A very brief example of this format might look like this:

Story: Returns go to stock

In order to keep track of stock
As a store owner
I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock.
When he returns the garment for a replacement in black,
Then I should have three blue garments in stock
And two black garments in stock

The scenarios are ideally phrased declaratively rather than imperatively — in the business language, with no reference to elements of the UI through which the interactions take place.

www.vskills.in

This format is referred to as the Gherkin language, which has a syntax similar to the above example. The term Gherkin, however, is specific to the Cucumber, JBehave and Behat software tools.

Specification as a ubiquitous language
Behavior-driven development borrows the concept of the ubiquitous language from domain driven design. A ubiquitous language is a (semi-)formal language that is shared by all members of a software development team — both software developers and non-technical personnel. The language in question is both used and developed by all team members as a common means of discussing the domain of the software in question. In this way BDD becomes a vehicle for communication between all the different roles in a software project.

A common risk with software development includes communication breakdowns between Developers and Business Stakeholders. BDD uses the specification of desired behavior as a ubiquitous language for the project team members. This is the reason that BDD insists on a semi-formal language for behavioral specification: some formality is a requirement for being a ubiquitous language. In addition, having such a ubiquitous language creates a domain model of specifications, so that specifications may be reasoned about formally. This model is also the basis for the different BDD-supporting software tools that are available.

The example given above establishes a user story for a software system under development. This user story identifies a stakeholder, a business effect and a business value. It also describes several scenarios, each with a precondition, trigger and expected outcome. Each of these parts is exactly identified by the more formal part of the language (the term Given might be considered a keyword, for example) and may therefore be processed in some way by a tool that understands the formal parts of the ubiquitous language.

Specialized Tooling Support

Much like test-driven design practice, behavior-driven development assumes the use of specialized support tooling in a project. Inasmuch as BDD is, in many respects, a more specific version of TDD, the tooling for BDD is similar to that for TDD, but makes more demands on the developer than basic TDD tooling.

Tooling principles
In principle a BDD support tool is a testing framework for software, much like the tools that support TDD. However, where TDD tools tend to be quite free-format in what is allowed for specifying tests, BDD tools are linked to the definition of the ubiquitous language discussed earlier.

As discussed, the ubiquitous language allows business analysts to write down behavioral requirements in a way that will also be understood by developers. The principle of BDD support tooling is to make these same requirements documents directly executable as a collection of tests. The exact implementation of this varies per tool, but agile practice has come up with the following general process:

✓ The tooling reads a specification document.

www.vskills.in

- ✓ The tooling directly understands completely formal parts of the ubiquitous language (such as the Given keyword in the example above). Based on this, the tool breaks each scenario up into meaningful clauses.
- ✓ Each individual clause in a scenario is transformed into some sort of parameter for a test for the user story. This part requires project-specific work by the software developers.
- ✓ The framework then executes the test for each scenario, with the parameters from that scenario.

Dan North has developed a number of frameworks that support BDD (including JBehave and RBehave), whose operation is based on the template that he suggested for recording user stories. These tools use a textual description for use cases and several other tools (such as CBehave) have followed suit. However, this format is not required and so there are other tools that use other formats as well. For example, Fitnesse (which is built around decision tables), has also been used to roll out BDD.

Tooling examples
There are several different examples of BDD software tools in use in projects today, for different platforms and programming languages.

Possibly the most well-known is JBehave, which was developed by Dan North. The following is an example taken from that project:

Consider an implementation of the Game of Life. A domain expert (or business analyst) might want to specify what should happen when someone is setting up a starting configuration of the game grid. To do this, he might want to give an example of a number of steps taken by a person who is toggling cells. Skipping over the narrative part, he might do this by writing up the following scenario into a plain text document (which is the type of input document that JBehave reads)

```
Given a 5 by 5 game
When I toggle the cell at (3, 2)
Then the grid should look like
.....
.....
.....
..X..
.....
When I toggle the cell at (3, 1)
Then the grid should look like
.....
.....
.....
..X..
..X..
When I toggle the cell at (3, 2)
Then the grid should look like
.....
.....
```

www.vskills.in

```
.....
.....
..X..
```

The bold print is not actually part of the input; it is included here to show which words are recognized as formal language. JBehave recognizes the terms Given (as a precondition which defines the start of a scenario), When (as an event trigger) and Then (as a postcondition which must be verified as the outcome of the action that follows the trigger). Based on this, JBehave is capable of reading the text file containing the scenario and parsing it into clauses (a set-up clause and then three event triggers with verifiable conditions). JBehave then takes these clauses and passes them on to code that is capable of setting a test, responding to the event triggers and verifying the outcome. This code must be written by the developers in the project team (in Java, because that is the platform JBehave is based on). In this case, the code might look like this

```
private Game game;
private StringRenderer renderer;

@Given("a $width by $height game")
public void theGameIsRunning(int width, int height) {
    game = new Game(width, height);
    renderer = new StringRenderer();
    game.setObserver(renderer);
}

@When("I toggle the cell at ($column, $row)")
public void iToggleTheCellAt(int column, int row) {
    game.toggleCellAt(column, row);
}

@Then("the grid should look like $grid")
public void theGridShouldLookLike(String grid) {
    assertThat(renderer.asString(), equalTo(grid));
}
```

The code has a method for every type of clause in a scenario. JBehave will identify which method goes with which clause through the use of annotations and will call each method in order while running through the scenario. The text in each clause in the scenario is expected to match the template text given in the code for that clause (for example, a Given in a scenario is expected to be followed by a clause of the form "a X by Y game"). JBehave supports the matching of actual clauses to templates and has built-in support for picking terms out of the template and passing them to methods in the test code as parameters. The test code provides an implementation for each clause type in a scenario which interacts with the code that is being tested and performs an actual test based on the scenario. In this case

- ✓ The theGameIsRunning method reacts to a Given clause by setting up the initial game grid.
- ✓ The iToggleTheCellAt method reacts to a When clause by firing off the toggle event described in the clause.

www.vskills.in

- ✓ The theGridShouldLookLike method reacts to a Then clause by comparing the actual state of the game grid to the expected state from the scenario.

The primary function of this code is to be a bridge between a text file with a story and the actual code being tested. Note that the test code has access to the code being tested (in this case an instance of Game) and is very simple in nature. The test code has to be simple, otherwise a developer would end up having to write tests for his tests.

Finally, in order to run the tests, JBehave requires some plumbing code that identifies the text files which contain scenarios and which inject dependencies (like instances of Game) into the test code. This plumbing code is not illustrated here, since it is a technical requirement of JBehave and does not relate directly to the principle of BDD-style testing.

Story versus specification

A separate subcategory of behavior-driven development is formed by tools that use specifications as an input language rather than user stories. An example of this style is the RSpec tool that was also developed by Dan North. Specification tools don't use user stories as an input format for test scenarios but rather use functional specifications for units that are being tested. These specifications often have a more technical nature than user stories and are usually less convenient for communication with business personnel than are user stories. An example of a specification for a stack might look like this

Specification: Stack

When a new stack is created
Then it is empty

When an element is added to the stack
Then that element is at the top of the stack

When a stack has N elements
And element E is on top of the stack
Then a pop operation returns E
And the new size of the stack is N-1

Such a specification may exactly specify the behavior of the component being tested, but is less meaningful to a business user. As a result, specification-based testing is seen in BDD practice as a complement to story-based testing and operates at a lower level. Specification testing is often seen as a replacement for free-format unit testing.

Specification testing tools like RSpec and JDave are somewhat different in nature from tools like JBehave. Since they are seen as alternatives to basic unit testing tools like JUnit, these tools tend to favor forgoing the separation of story and testing code and prefer embedding the specification directly in the test code instead. For example, an RSpec test for a hashtable might look like this

```
describe Hash do
  let(:hash) { Hash[:hello, 'world'] }
```

www.vskills.in

```
  it { expect(Hash.new).to eq({}) }

  it "hashes the correct information in a key" do
    expect(hash[:hello]).to eq('world')
  end

  it 'includes key' do
    hash.keys.include?(:hello).should be true
  end
end
```

This example shows a specification in readable language embedded in executable code. In this case a choice of the tool is to formalize the specification language into the language of the test code by adding methods named it and should. Also there is the concept of a specification precondition – the before section establishes the preconditions that the specification is based on.

The result of test will be:

```
 Hash
   should eq {}
   includes key
   hashes the correct information in a key
```

1.3. DSL

A domain-specific language (DSL) is a computer language specialized to a particular application domain. This is in contrast to a general-purpose language (GPL), which is broadly applicable across domains, and lacks specialized features for a particular domain. There are a wide variety of DSLs, ranging from widely used languages for common domains, such as HTML for web pages, down to languages used by only one or a few pieces of software, such a Emacs Lisp for GNU Emacs and XEmacs. DSLs can be further subdivided by the kind of language, and include domain-specific markup languages, domain-specific modeling languages (more generally, specification languages), and domain-specific programming languages. Special-purpose computer languages have always existed in the computer age, but the term "domain-specific language" has become more popular due to the rise of domain-specific modeling. Simpler DSLs, particularly ones used by a single application, are sometimes informally called mini-languages.

The line between general-purpose languages and domain-specific languages is not always sharp, as a language may have specialized features for a particular domain but be applicable more broadly, or conversely may in principle be capable of broad application but in practice used primarily for a specific domain. For example, Perl was originally developed as a text-processing and glue language, for the same domain as AWK and shell scripts, but has since become a general-purpose programming language. By contrast, PostScript is a Turing complete language, and in principle can be used for any task, but in practice is narrowly used as a page description language.

A domain-specific language is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside it (although that may be technically possible). In

www.vskills.in

contrast, general-purpose languages are created to solve problems in many domains. The domain can also be a business area. Some examples of business areas include:

- ✓ domain-specific language for life insurance policies developed internally in large insurance enterprise
- ✓ domain-specific language for combat simulation
- ✓ domain-specific language for salary calculation
- ✓ domain-specific language for billing

A domain-specific language is somewhere between a tiny programming language and a scripting language, and is often used in a way analogous to a programming library. The boundaries between these concepts are quite blurry, much like the boundary between scripting languages and general-purpose languages.

Domain-specific languages are languages (or often, declared syntaxes or grammars) with very specific goals in design and implementation. A domain-specific language can be one of a visual diagramming language, such as those created by the Generic Eclipse Modeling System, programmatic abstractions, such as the Eclipse Modeling Framework, or textual languages. For instance, the command line utility grep has a regular expression syntax which matches patterns in lines of text. The sed utility defines a syntax for matching and replacing regular expressions. Often, these tiny languages can be used together inside a shell to perform more complex programming tasks.

The line between domain-specific languages and scripting languages is somewhat blurred, but domain-specific languages often lack low-level functions for filesystem access, interprocess control, and other functions that characterize full-featured programming languages, scripting or otherwise. Many domain-specific languages do not compile to byte-code or executable code, but to various kinds of media objects: GraphViz exports to PostScript, GIF, JPEG, etc., where Csound compiles to audio files, and a ray-tracing domain- specific language like POV compiles to graphics files. A computer language like SQL presents an interesting case: it can be deemed a domain-specific language because it is specific to a specific domain (in SQL's case, accessing and managing relational databases), and is often called from another application, but SQL has more keywords and functions than many scripting languages, and is often thought of as a language in its own right, perhaps because of the prevalence of database manipulation in programming and the amount of mastery required to be an expert in the language.

Further blurring this line, many domain-specific languages have exposed APIs, and can be accessed from other programming languages without breaking the flow of execution or calling a separate process, and can thus operate as programming libraries.

Programming Tools

Some domain-specific languages expand over time to include full-featured programming tools, which further complicates the question of whether a language is domain-specific or not. A good example is the functional language XSLT, specifically designed for transforming one XML graph into another, which has been extended since its inception to allow (particularly in its 2.0 version) for various forms of filesystem interaction, string and date manipulation, and data typing.

www.vskills.in

In model-driven engineering many examples of domain-specific languages may be found like OCL, a language for decorating models with assertions or QVT, a domain-specific transformation language. However languages like UML are typically general purpose modeling languages.

To summarize, an analogy might be useful: a Very Little Language is like a knife, which can be used in thousands of different ways, from cutting food to cutting down trees. A domain-specific language is like an electric drill: it is a powerful tool with a wide variety of uses, but a specific context, namely, putting holes in things. A General Purpose Language is a complete workbench, with a variety of tools intended for performing a variety of tasks. Domain-specific languages should be used by programmers who, looking at their current workbench, realize they need a better drill, and find that a particular domain-specific language provides exactly that.

Usage Patterns

There are several usage patterns for domain-specific languages:

- ✓ processing with standalone tools, invoked via direct user operation, often on the command line or from a Makefile (e.g., grep for regular expression matching, sed, lex, yacc, the GraphViz tool set, etc.)
- ✓ domain-specific languages which are implemented using programming language macro systems, and which are converted or expanded into a host general purpose language at compile-time or read-time
- ✓ embedded (or internal) domain-specific languages, implemented as libraries which exploit the syntax of their host general purpose language or a subset thereof, while adding domain-specific language elements (data types, routines, methods, macros etc.). (e.g. Embedded SQL, LINQ)
- ✓ domain-specific languages which are called (at runtime) from programs written in general purpose languages like C or Perl, to perform a specific function, often returning the results of operation to the "host" programming language for further processing; generally, an interpreter or virtual machine for the domain-specific language is embedded into the host application (e.g. format strings, a regular expression engine)
- ✓ domain-specific languages which are embedded into user applications (e.g., macro languages within spreadsheets) and which are (1) used to execute code that is written by users of the application, (2) dynamically generated by the application, or (3) both.

Many domain-specific languages can be used in more than one way. DSL code embedded in a host language may have special syntax support, such as regexes in sed, AWK, Perl or JavaScript, or may be passed as strings.

Design Goals

Adopting a domain-specific language approach to software engineering involves both risks and opportunities. The well-designed domain-specific language manages to find the proper balance between these.

Domain-specific languages have important design goals that contrast with those of general-purpose languages:

- ✓ domain-specific languages are less comprehensive.
- ✓ domain-specific languages are much more expressive in their domain.

www.vskills.in

- ✓ domain-specific languages should exhibit minimum redundancy according to the following subjective definition.

Redundancy of a program is defined as the average number of textual insertions, deletions, or replacements necessary to correctly implement a single stand-alone change in requirements. For a language, this is averaged over programs in the problem domain. This measure is useful because, the smaller it is, the less likely that bugs can be introduced by incompletely implementing changes.

Idioms

In programming, idioms are methods imposed by programmers to handle common development tasks, e.g.:

- ✓ Ensure data is saved before the window is closed.
- ✓ Before conducting expensive tests, perform cheap tests that can rule out need for expensive tests.
- ✓ Edit code whenever command-line parameters change because they affect program behavior.

General purpose programming languages rarely support such idioms, but domain-specific languages can describe them, e.g.:

- ✓ A script can automatically save data.
- ✓ A smart test harness can learn what good tests are.
- ✓ A domain-specific language can parameterize command line input.

Advantages and disadvantages

Some of the advantages

- ✓ Domain-specific languages allow solutions to be expressed in the idiom and at the level of abstraction of the problem domain. The idea is domain experts themselves may understand, validate, modify, and often even develop domain-specific language programs. However, this is seldom the case.
- ✓ Domain-specific languages allow validation at the domain level. As long as the language constructs are safe any sentence written with them can be considered safe.
- ✓ Domain-specific languages can help to shift the development of business information systems from traditional software developers to the typically larger group of domain-experts who (despite having less technical expertise) have deeper knowledge of the domain.

Some of the disadvantages

- ✓ Cost of learning a new language vs. its limited applicability
- ✓ Cost of designing, implementing, and maintaining a domain-specific language as well as the tools required to develop with it (IDE)
- ✓ Finding, setting, and maintaining proper scope.
- ✓ Difficulty of balancing trade-offs between domain-specificity and general-purpose programming language constructs.
- ✓ Potential loss of processor efficiency compared with hand-coded software.

www.vskills.in

- ✓ Proliferation of similar non-standard domain-specific languages, for example, a DSL used within one insurance company versus a DSL used within another insurance company.
- ✓ Non-technical domain experts can find it hard to write or modify DSL programs by themselves.
- ✓ Increased difficulty of integrating the DSL with other components of the IT system (as compared to integrating with a general-purpose language).
- ✓ Low supply of experts in a particular DSL tends to raise labor costs.
- ✓ Harder to find code examples.

1.4. Capybara

Capybara is a web-based automation framework used for creating functional tests that simulate how users would interact with your application. Today, there are many alternatives for web-based automation tools, such as Selenium, Watir, Capybara, etc. All of these tools have the same purpose, but there are slight differences that make each of them more or less suitable.

The main characteristic that developers are aiming for is the ability to have tests that are modular, easy to write, and easy to maintain. This is especially true in Agile/TDD environments where writing tests is second nature. These tests are expected to give good and fast feedback on code quality. As time goes by, the number of tests grows and it can be a real nightmare to maintain the tests, especially when the tests are not modular and simple enough.

Capybara is an awesome Ruby gem that drives a browser over your code so you can exercise your entire application. It has a variety of methods like visit(url) to visit a page, or click_link("Sign in") to click a link. It also has an amazing feature which is to retry a command when it's driving a real browser like Selenium or PhantomJS to handle asynchronous issues. So, if you say click_link("Sign in") but there's no link with the text "Sign in" then Capybara will try to find it over and over until it finds it or it reaches Capybara's default_wait_time.

This is great when we're working on a test and it's not passing yet because our code isn't done. We want it to try and fail if the link isn't there. And if that link pops up via a jQuery plugin, we want Capybara to wait for it to show up, not fail fast.

Key benefits

No setup necessary for Rails and Rack application. It works out of the box.

- ✓ Intuitive API which mimics the language an actual user would use.
- ✓ Switch the backend your tests run against from fast headless mode to an actual browser with no changes to your tests.
- ✓ Powerful synchronization features mean you never have to manually wait for asynchronous processes to complete.

1.5. Capybara Anatomy

Capybara is a library/gem built to be used on top of an underlying web-based driver. It offers a user-friendly DSL (Domain Specific Language) which is used to describe actions that are executed by the underlying web driver.

www.vskills.in

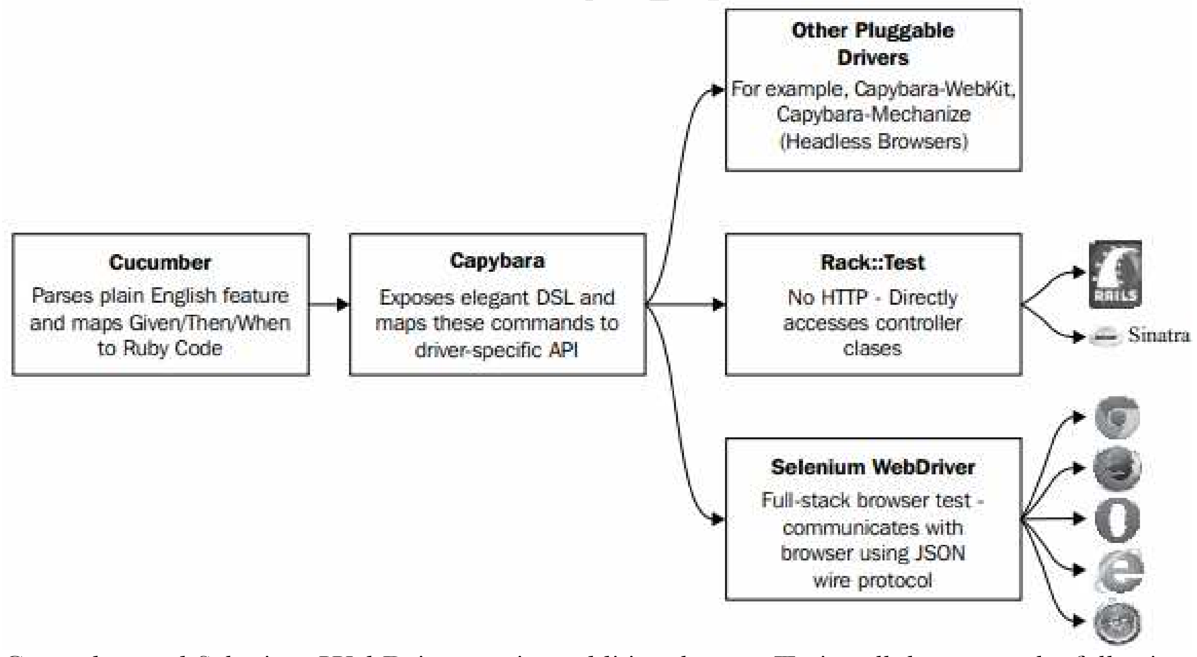
When the page is loaded using the DSL (and underlying web driver), Capybara will try to locate the relevant element in the DOM (Document Object Model) and execute the action, such as click button, link, etc.

Here are some of the web drivers supported by Capybara:

rack::test:
By default, it works with rack::test driver. This driver is considerably faster than other drivers, but it lacks JavaScript support and it cannot access HTTP resources outside of the application for which the tests are made (Rails app, Sinatra app).

selenium-webdriver:
Capybara supports selenium-webdriver, which is mostly used in web-based automation frameworks. It supports JavaScript, can access HTTP resources outside of application, and can also be setup for testing in headless mode which is especially useful for CI scenarios. To setup usage of this driver you need to add:
Capybara.default_driver = :selenium

capybara-webkit:
For true headless testing with JavaScript support, we can use the capybara-webkit driver (gem). It uses QtWebKit and it is significantly faster than selenium as it does not load the entire browser. To setup usage of this driver, you need to add:
Capybara.default_driver = :webkit


Other Pluggable Drivers
For example, Capybara-WebKit, Capybara-Mechanize (Headless Browsers)
Cucumber
Parses plain English feature and maps Given/Then/When to Ruby Code
Capybara
Exposes elegant DSL and maps these commands to driver-specific API
Rack::Test
No HTTP - Directly accesses controller clases
RAILS
Sinatra
Selenium WebDriver
Full-stack browser test - communicates with browser using JSON wire protocol

Cucumber and Selenium WebDriver are just additional gems. To install them, run the following gem install cucumber selenium-webdriver

vskills.in

1.6. Capybara vs Cucumber

Capybara is a tool that interacts with a website the way a human would (like visiting a url, clicking a link, typing text into a form and submitting it). It is used to emulate a user's flow through a website. With Capybara you can write something like this:

```
describe "the signup process", :type => :feature do
  before :each do
    User.make(:email => 'user@example.com', :password => 'caplin')
  end

  it "signs me in" do
    visit '/sessions/new'
    within("#session") do
      fill_in 'Login', :with => 'user@example.com'
      fill_in 'Password', :with => 'password'
    end
    click_link 'Sign in'
    page.should have_content 'Success'
  end
end
```

Capybara is an automated testing tool (often used) for ROR applications.

Cucumber

Cucumber is a BDD tool that expresses testing scenarios in a business-readable, domain-specific language. Cucumber is a general-purpose BDD tool. It knows nothing about web apps. So Cucumber step definitions call Capybara in order to test web apps. Cucumber is a tool to write human-readable tests that are mapped into code. With it, you can rewrite the above example like this:

```
Scenario: Signup process

Given a user exists with email "user@example.com" and password "caplin"
When I try to login with "user@example.com" and "caplin"
Then I should be logged in successfully
```

The almost plain-text interpretation is useful to pass around non-developers but also need some code mapped into it to actually work (the step definitions).

Usually you will use Capybara if you testing a website and use Cucumber if you need to share those tests with non-developers. These two conditions are independent so you can use one without the other or both or none.

In the code snippet there is some RSpec as well. This is needed because Cucumber or Capybara by themselves cannot test something. They rely on RSpec, Test::Unit or minitest to do the actual "Pass or Fail" work.

www.vskills.in

Certifications

Accounting, Banking & Finance
- Certified AML- KYC Compliance Officer
- Certified Business Accountant
- Certified Commercial Banker
- Certified Equity Research Analyst
- Certified Foreign Exchange Professional
- Certified Hedge Fund Manager
- Certified Merger and Acquisition Analyst
- Certified Tally 9.0 Professional
- Certified Treasury Markets Professional
- Certified Wealth Manager

Foreign Trade
- Certified Export Import (Foreign Trade) Professional

Hospitality
- Certified Restaurant Team Member (Hospitality)

Human Resources
- Certified HR Compensation Manager
- Certified HR Staffing Manager
- Certified Human Resources Manager
- Certified Performance Appraisal Manager

Logistics & Supply Chain Management
- Certified International Logistics Professional
- Certified Logistics & SCM Professional
- Certified Purchase Manager

Law
- Certified IPR & Legal Manager

Life Skills
- Certified Business Communication Specialist
- Certified Public Relations Officer

Media
- Certified Advertising Manager
- Certified Advertising Sales Professional

Office Skills
- Certified Data Entry Operator
- Certified Office Administrator

Project Management
- Certified Project Management Professional

Real Estate
- Certified Real Estate Consultant

Information Technology
- Certified Android Apps Developer
- Certified ASP.NET Programmer
- Certified Basic Network Support Professional
- Certified Business Intelligence Professional
- Certified C# Professional
- Certified CAD Professional
- Certified Cloud Computing Professional
- Certified Computer Fundamentals (MS Office) Professional
- Certified Core Java Developer
- Certified CSS Designer
- Certified Data Mining and Warehousing Professional
- Certified DHTML & Javascript Developer
- Certified Django Developer
- Certified DTP operator
- Certified E-commerce Professional
- Certified E-Governance Professional
- Certified Enterprise Applications Integration Specialist (Biztalk)
- Certified Ethical Hacking and Security Professional
- Certified Facebook Apps Developer
- Certified Grid Computing Professional
- Certified Hadoop and Mapreduce Professional
- Certified HTML Designer
- Certified HTML5 Developer
- Certified iPhone Apps Developer
- Certified IT Support Professional
- Certified J2ME Programmer
- Certified Joomla Developer
- Certified Linux Administrator
- Certified Magento Professional
- Certified MySQL DB Administrator
- Certified Network Security Professional
- Certified Open Source CMS (Drupal) Professional
- Certified PHP Professional
- Certified PL/SQL Developer
- Certified Python Professional
- Certified Router Support Professional
- Certified Selenium Professional
- Certified SEO Professional
- Certified Software Quality Assurance Professional
- Certified Software Security Professional
- Certified Software Testing Professional
- Certified SQL Server 2008 Programmer
- Certified WiMax(4G) Professional
- Certified Wordpress Developer
- Certified XML Developer

Sales, BPO
- Certified Telesales Executive

Contact us at:
V-Skills
011-473 44 723 or info@vskills.in
Intelligent Communication Systems India Limited
DSIIDC Administrative Building,
Okhla Industrial Estate-III, New Delhi-110020
www.vskills.in