ACCEPTANCE TEST-DRIVEN DEVELOPMENT WITH ROBOT FRAMEWORK

by Craig Larman and Bas Vodde

Version 1.1

Acceptance test-driven development is an essential practice applied by successful Agile and Scrum teams. It changes the purpose of testing by using examples/tests for clarifying and documenting requirements. This short paper is an extract from the Test *chapter found in* Practices for Scaling Lean & Agile Development: Large, Multisite, and Offshore Product Development with Large-Scale Scrum.

INTRODUCTION TO ACCEPTANCE TEST-DRIVEN DEVELOPMENT

Acceptance test-driven development (A-TDD)[1] is a collaborative requirements discovery approach where examples and automatable tests are used for specifying requirements—creating *executable specifications*. These are created with the team, Product Owner, and other stakeholders in requirements workshops.

Tests as requirements, requirements as tests—According to Melnik and Martin, "*As formality increases, tests and requirements become indistinguishable. At the limit, tests and requirements are equivalent.*" Tests must be precise in order to be automatable. A-TDD exploits this formality and formulates requirements by writing automatable tests.

Workshops for clarifying requirements—Face-to-face requirement clarifications in workshops have been used since the invention of Joint Application Design (JAD). A-TDD similarly exploits face-to-face conversation by using workshops for formulating requirements-as-tests.

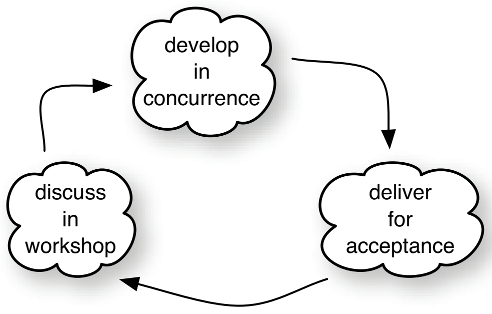
Concurrent engineering—The most common iteration length is two-weeks. That's fast and therefore the team needs to conceive a way to work concurrently—sequential development in a short iteration does not work. We have seen teams *invent* A-TDD again and again simply because they had to answer the question: "*How can we perform our work at the same time.*"

1. Acceptance test-driven development is also known as agile acceptance testing or story test- driven development.

Prevention rather than detection—When including people specialized in test in the requirements workshop, they can ask the test-related questions or use normal test strategies such as border value analysis. In that way, they help to improve the requirements and *prevent* defects.

How does A-TDD work? Figure 1.1 presents an overview.

Figure 1.1 A-TDD overview



A-TDD consists of three steps:

1. Discuss the requirements in a workshop.
2. Develop them concurrently during the iteration.
3. Deliver the results to the stakeholders for acceptance.

Discuss—Requirements are discovered through discussion in a requirements workshop[2]. Participants of a workshop are the cross-functional team, the Product Owner or representative, and any other stakeholder who potentially has information about the requirements. A common question to ask during such workshops is "*Imagine the system to be finished. How would you use it and what would you expect from it?*" Such a question results in examples of use, and these examples can be written as tests—the requirements. The workshop focus ought to be on discussion and discovery of requirements more than on the actual tests.

Develop—At the end of the workshop, the examples are *distilled*[3] into tests. All activities needed to implement the requirement are done concurrently. These include

- ❏ making the glue code between the tests and the system under test ("test libraries" and "lower-level tables" in Robot Framework or 'fixtures' in Fit)

2. Gojko Adzic calls these *specification workshops*.
3. Elisabeth Hendrickson considers *distill* a separate step in A-TDD.

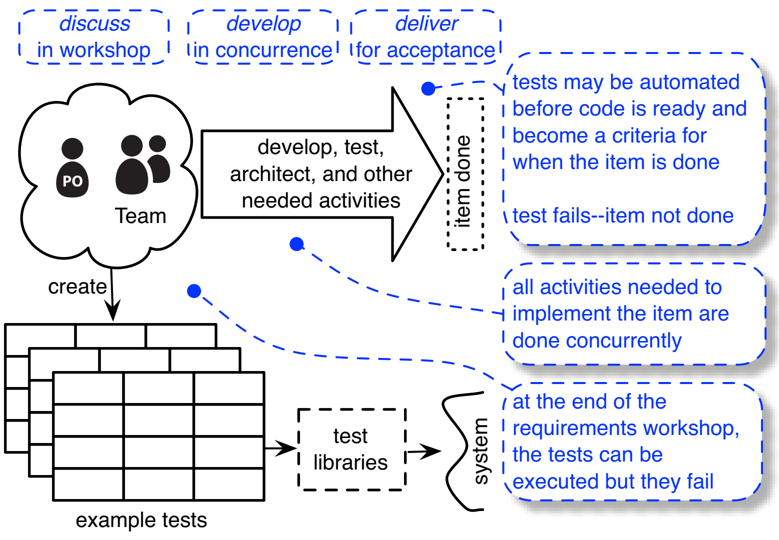
- ❑ implementing the requirement so that the tests pass
- ❑ updating architectural and other internal documentation according to the working agreement of the team
- ❑ writing customer documentation for the requirement
- ❑ additional exploratory testing

The exact list depends on the product, context, working agreements, and the Definition of Done[4].

Deliver—When the tests pass, the requirement is reviewed with the Product Owner and other stakeholders. This might lead to new requirements or a change in the existing tests.

A more detailed way of describing A-TDD is shown in Figure 1.2.

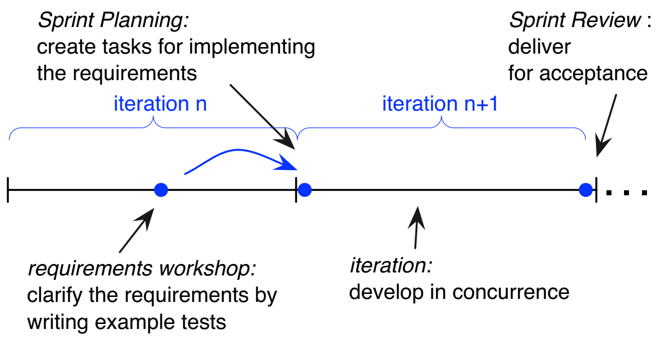
Figure 1.2 A-TDD in more detail


discuss in workshop
develop in concurrence
deliver for acceptance
PO
Team
develop, test, architect, and other needed activities
item done
tests may be automated before code is ready and become a criteria for when the item is done
test fails--item not done
create
all activities needed to implement the item are done concurrently
test libraries
system
at the end of the requirements workshop, the tests can be executed but they fail
example tests

The steps in A-TDD map nicely to the Scrum iteration cycle (Figure 1.3).

4. The “Definition of Done” is a Scrum term for the agreement between the Product Owner and Team on what is included inside the iteration. A short introduction to Scrum—the Scrum Primer— can be found at http://www.scrumprimer.org

Figure 1.3 A-TDD steps mapped to Scrum iteration



Discuss in workshop—Before the detailed Sprint Planning[5], the team, Product Owner, and other stakeholders clarify the requirements collaboratively in a workshop.

Develop in concurrence—Tasks for implementing the tests/requirements are created in the detailed Sprint Planning and implemented during the iteration. All activities happen "at about the same time."

Deliver for acceptance—The working product increment—the passing acceptance tests—are delivered for acceptance to stakeholders and discussed together in the Sprint Review.

EXAMPLE: ROBOT FRAMEWORK

This section presents a brief example of Robot Framework that uses a system we worked on years ago. The original development did not use A-TDD or Robot Framework—the example is new, the system is old. The system is medium-sized,[6] yet the example demonstrates the key points that are also valid in larger development.

Robot Framework is a *keyword-driven*[7] test automation framework created by Pekka Klärck[8] at Nokia Siemens Networks in 2005. One of its early design goals was A-TDD support. It was open-sourced in 2008 and is available at www.robotframework.org.

The product used in this case study is a conference information system built for a convention center. Access to the system is available at "information pillars" distributed through-

5. In Sprint Planning part one or Product Backlog refinement.
6. Larger products tend to come from a more complex or unfamiliar domain—making it hard to use as an example in a few pages.
7. Keyword-driven test frameworks use keywords in data to determine the action to take on the data. Keywords are sometimes called action words.
8. Formerly known as Pekka Laukkanen.

out the convention center. Visitors can use it to find information about the current conference or the convention center, such as

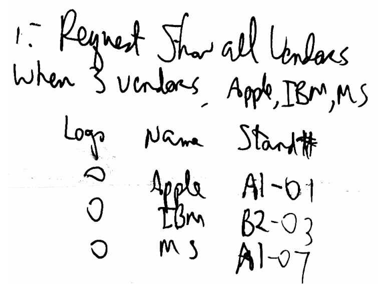
- ❑ Where can I find the booth of vendor X?
- ❑ How do I get there?
- ❑ Where is the nearest restaurant?
- ❑ What did other visitors have to say about the conference?

These pillars are maintained and controlled centrally. The information inside the system *must* come from the *existing* conference preparation process. Not much additional preparation work is allowed when the system is updated for a new conference—the new system must adapt to the existing systems and ways of working. And, of course, the system must be *flashy* and *shiny*—sound, graphics, and other bells and whistles.

Example One: The Vendor List

The first example is simple but demonstrates some key points. The customer requested the ability to list all the vendors and display them in "a nice list." After abstract discussion about what "a nice list" means, *we asked for an example*. The first example was a workflow, one (Figure 1.4) with three vendors in the database shown in an alphabetically ordered list with three columns.

Figure 1.4 workflow example for selecting vendors



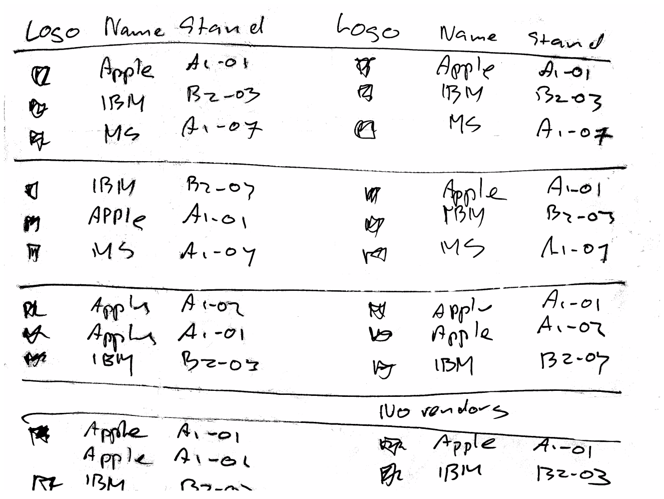
1: Request Show all Vendors
When 3 vendors, Apple, IBM, MS

Logo	Name	Stand#
○	Apple	A1-01
○	IBM	B2-03
○	MS	A1-07

After asking for more examples (not shown), we discover that the provided data in the supplied database is not consistent—the information system will have to compensate for that because we have no control over the database. For example, there can be duplicate entries with minor differences (with and without a logo).

Describing all the data compensations with *workflow examples leads to myriad similar tests*, so we switch to *data-driven tests* for these particular *business rules* by asking, "What data in the database will lead to what *nice* vendor list?" The results are shown in Figure 1.5 and include examples for alphabetical sorting, removing duplications, and preferring logo over no-logo entries.

Figure 1.5 data-driven test for vendor lists



Logo	Name	Stand	Logo	Name	Stand
	Apple	A1-01		Apple	A1-01
	IBM	B2-03		IBM	B2-03
	MS	A1-07		MS	A1-07
	IBM	B2-07		Apple	A1-01
	APPle	A1-01		IBM	B2-03
	MS	A1-07		MS	A1-07
	Apple	A1-02		Apple	A1-01
	Apple	A1-01		Apple	A1-02
	IBM	B2-03		IBM	B2-07
				No vendors	
	Apple	A1-01		Apple	A1-01
	Apple	A1-01		IBM	B2-03
	IBM	[illegible]			

Intermission—Robot Framework overview

The next step is to *distill* these examples into Robot Framework tests. But how does Robot Framework work?

Robot tests are written in tables with HTML, TSV, reST[9] or plain text. HTML and plain text are the most commonly used format. In HTML, Robot Framework uses only the tables and ignores all the additional text—which can be used for documentation. There are four types of tables:

- *Test case tables*—contain the actual test cases. The first field of these tables must contain "Test Case" or "Test Cases."
- *Keyword tables*—contain lower-level user keywords that can be used to construct test cases. The first field must contain 'Keyword' or "User Keywords."
- *Settings table*—allows for importing files and defining metadata. The first field must contain 'Setting,' or 'Settings.'
- *Variable table*—declares variables containing global data that can be used elsewhere. The first field must contain 'Variable,' or 'Variables.'

9. TSV are tab-separated-values files. ReStructuredText (reST) is a markup language commonly used for documentation in Python projects.

Test Cases	Action	Arg
Drink coffee	Drink coffee in liters	10
	Is physical health	OK
Drink coffee over capabity	Drink over the max amount of coffee	
	Is physical health	NOK

The table to the left shows a test case table with two test cases in it—the first column. The second column contains the keywords, and the remaining columns are for passing arguments to these keywords.

Robot Framework has two types of keywords: *user keywords* and *library keywords.* A user keyword is implemented in a keyword table, whereas a library keyword is implemented in a piece of glue code between the test and the system under test—called a ***test library***. Executing the table fails because it contains three undefined keywords—Drink coffee in liters, Is physical health, and Drink over max amount of coffee.

Keywords	Action	Arg
Drink over the max amount of coffee	Drink coffee in liters	${MAX COFFEE}
	Drink coffee in liters	1

The table to the left implements the "Drink over the max amount of coffee" user keyword as drinking one liter more than the maximum. ${MAX COFFEE} is a variable defined in a variable table (not shown).

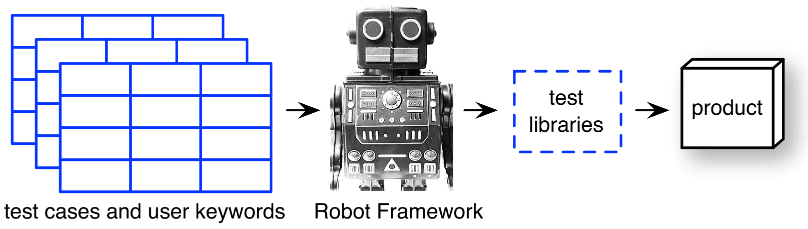
Executing the table fails because two keywords are undefined. These are low level and implemented as library keywords in Java (or alternatively, in Python).

```
public class CoffeeTestLibrary
{
   Human humanUnderTest = new Human("Bas");

   public void drinkCoffeeInLiters(Integer liters) {
      humanUnderTest.drinkCoffee(liters);
   }

   public void
   isPhysicalHealth(String expectedHealth) throws Exception
   {
      if (!expectedHealth.equals(humanUnderTest.checkHealth()))
         throw new Exception("Health problem. Expected health: " +
               expectedHealth + " but actual health was " +
               humanUnderTest.checkHealth());
   }
}
```

The figure below is an overview of Robot Framework. Test case and user keyword tables are fed to Robot Framework. The framework calls the test libraries, and they call the system under test. More information can be found in the Robot Framework User Guide[10].

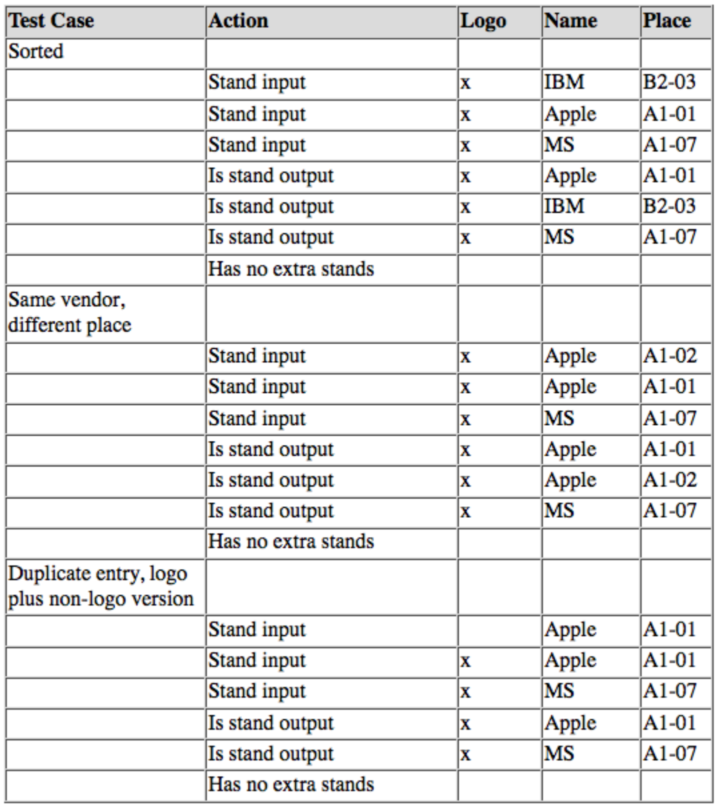


Continuation—List the vendors

We *distill* the test from the example on the whiteboard. The 'normal' example is removed because it is of the same *equivalence class* as the 'sorted' test.[11]

These test cases are executable—but they fail. Robot Framework complains that the keywords "Stand input," "Is stand output," and "Has No Extra Stands" are undefined.

Figure 1.6 lists all vendors test case table

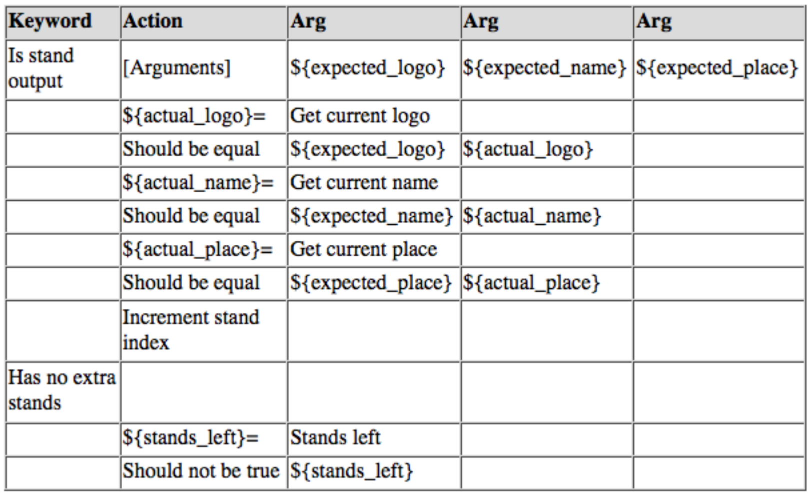


Test Case	Action	Logo	Name	Place
Sorted				
	Stand input	x	IBM	B2-03
	Stand input	x	Apple	A1-01
	Stand input	x	MS	A1-07
	Is stand output	x	Apple	A1-01
	Is stand output	x	IBM	B2-03
	Is stand output	x	MS	A1-07
	Has no extra stands			
Same vendor, different place				
	Stand input	x	Apple	A1-02
	Stand input	x	Apple	A1-01
	Stand input	x	MS	A1-07
	Is stand output	x	Apple	A1-01
	Is stand output	x	Apple	A1-02
	Is stand output	x	MS	A1-07
	Has no extra stands			
Duplicate entry, logo plus non-logo version				
	Stand input		Apple	A1-01
	Stand input	x	Apple	A1-01
	Stand input	x	MS	A1-07
	Is stand output	x	Apple	A1-01
	Is stand output	x	MS	A1-07
	Has no extra stands			

The keywords can be implemented as user or library keywords. We chose to implement the "Is stand output," and the "Has No Extra Stands" as user keywords (Figure 1.7).

10. The user guide can be found at www.robotframework.org
11. The keyword-driven nature of Robot Framework is evident from the repeating keywords in data-driven tests. In Fit this would not be necessary. This will be enhanced in the future versions.

Figure 1.7 list all vendors user keyword table



Keyword	Action	Arg	Arg	Arg
Is stand output	[Arguments]	${expected_logo}	${expected_name}	${expected_place}
	${actual_logo}=	Get current logo		
	Should be equal	${expected_logo}	${actual_logo}	
	${actual_name}=	Get current name		
	Should be equal	${expected_name}	${actual_name}	
	${actual_place}=	Get current place		
	Should be equal	${expected_place}	${actual_place}	
	Increment stand index			
Has no extra stands				
	${stands_left}=	Stands left		
	Should not be true	${stands_left}		

The first row declares the three arguments of the “Is stand output” keyword. The next rows assign the output of the “get current logo” keyword to the ${actual_logo} variable and compare that with the expected_logo. The same steps are repeated for the other expected values. At the end, we increment the stand index.[12]

When we now run the tests, Robot Framework complains about five undefined keywords: stand input, get current logo, get current name, get current place, and stands left. These keywords will be implemented in a test library.

Our system under test is written in C. We can call it through the user interface (not recommended), or we can call C code directly. We chose the latter and implemented the test library in Python because we can easily call C code directly from Python by using the ctypes foreign library.[13] The test library code:

```
from ctypes import *

class conferencekeywords:

  def __init__(self):
    self.conf = cdll.LoadLibrary("stands.so")
    self.conf.init()
    self.stand_index = 0
    self.logo_mark = ["", "x"]
    self.conf.get_place_at_index.restype = c_char_p
    self.conf.get_name_at_index.restype = c_char_p
```

12. This is a side effect in the keyword implementation which should be there but is there to keep the higher-level table simpler
13. Works excellently for C. For C++ you may prefer to use SWIG or Boost Python.

```
def increment_stand_index(self):
  self.stand_index = self.stand_index + 1

def stands_left(self):
  return self.conf.stand_output_at(self.stand_index)

def stand_input(self, logo, name, stand):
  has_logo = logo == "x"
  self.conf.add(has_logo, c_char_p(name), c_char_p(stand))

def get_current_logo(self):
  logo = int(self.conf.get_logo_at(self.stand_index))
  return self.logo_mark[logo]

def get_current_name(self):
  return self.conf.get_name_at_index(self.stand_index)

def get_current_place(self):
  return self.conf.get_place_at_index(self.stand_index)
```

The code is trivial—the only thing it does is call the C interface.

The system under test is linked into a shared library and loaded by the *cdll* ctypes call. The output is assigned to the *conf* variable and is used to call the C functions in the shared library. The additional code specifies the parameter and return types; this must be done when they are not integers—the default.

When we run the tests, they will pass—if the functionality is implemented.

Example Two: Importing AutoCAD Files

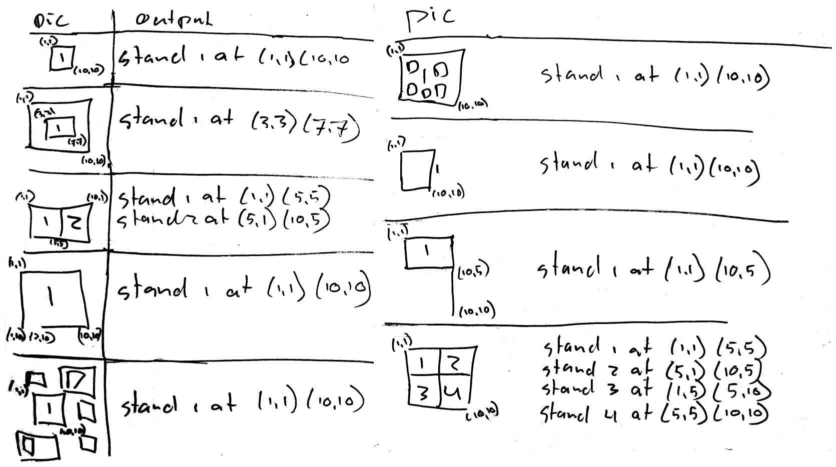
The floorplan department uses AutoCAD to create conference layouts. These contain the location of the stands and everything else related to the conference—including weird symbols and electrical outlets. These CAD drawings, however, *were made for humans to read*.

A requirement for the conference information system was to show a conference map. Visitors can select a location on the map and the system will show them the vendor. The convention center required the system to import the AutoCAD DWG files that were created by the floorplan department. Reading AutoCAD files is slow, so the files are converted to an internal format that stores only the needed information—it discards *noise* such as electrical outlets.

The DWG contains lines that form shapes. A shape with a number written in it is probably a stand. The system needs to read the file, form the shapes, recognize the stands, and discard irrelevant information. This is not too hard… except that the data is inconsistent—not made for computers. The shapes are not always *exactly* closed, the number is not always *exactly* in the stand, and so forth.

In a requirement workshop, we discussed the different possible inconsistencies that the system should support. We started with *abstract descriptions* and moved toward *examples*, as shown in Figure 1.8.

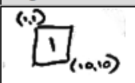
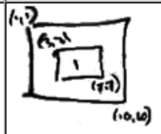
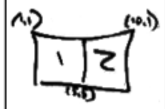
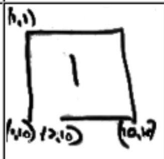
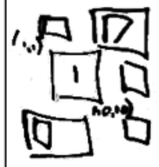
Figure 1.8 examples of rectangular stands



The examples are drawings of shapes. The *discussion* was about distinguishing stands from noise and dealing with inconsistencies in the drawing.

The tests *distilled* from the examples are shown in Figure 1.9. The tests contain an AutoCAD input file and the expected output file. We added the pictures to the tests for documentation. Robot Framework ignores all information except for the text inside the table. This allows us to add documentation to our specifications tests.

Figure 1.9 tests for stand recognition

Test Case	Action	Original	Result
Recognize rectangular stands	Detect and Check stands	 simple_rectangle.dwg	simple_rectangle.fpm
	Detect and Check stands	 rect_in_rect.dwg	rect_in_rect.fpm
	Detect and Check stands	 two_rects.dwg	two_rect.fpm
	Detect and Check stands	 non_closed_rect.dwg	non_closed_rect.fpm
	Detect and Check stands	 rect_with_noise.dwg	rect_with_noise.fpm

The "Detect and check stands" keyword is implemented as a user keyword shown in Figure 1.10.

Figure 1.10 stand recognition user keywords

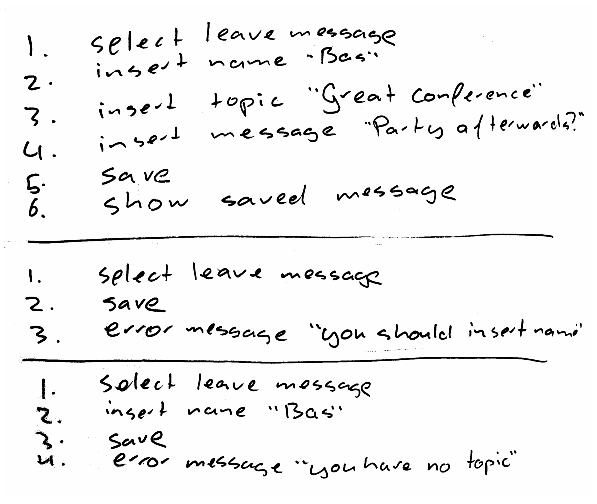
Keyword	Action	Arg	Arg
Detect and Check stands	[Arguments]	${dwg_drawing}	${fpm_result}
	${fpm_output_file}=	Convert to FPM	${dwg_drawing}
	Compare FPM files	${fpm_output_file}	${fpm_result}

The test library calls the C interface of the system under test (not shown).

Example Three: Leaving Messages

This last example is an example of a workflow test. Visitors need to be able to leave messages for one another on the conference information system. During a *discussion* in a requirement workshop, we moved from the abstract description into *workflow examples*. The results are shown in Figure 1.11.

Figure 1.11 examples of messaging workflow



1. select leave message
2. insert name "Bas"
3. insert topic "Great conference"
4. insert message "Party afterwards?"
5. save
6. show saved message

---

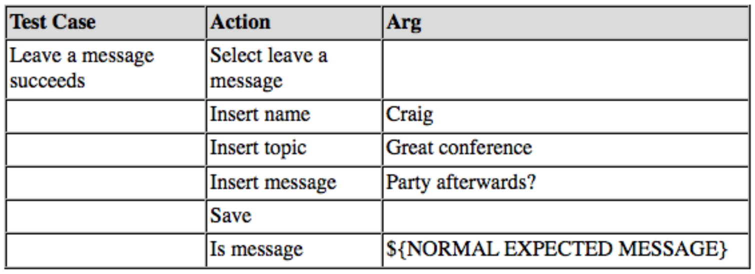
1. select leave message
2. save
3. error message "you should insert name"

---

1. select leave message
2. insert name "Bas"
3. save
4. error message "you have no topic"

The *distilled* tests ended up the same as the wall-workflow examples. They are shown in Figure 1.12.

Figure 1.12 workflow example



Test Case	Action	Arg
Leave a message succeeds	Select leave a message	
	Insert name	Craig
	Insert topic	Great conference
	Insert message	Party afterwards?
	Save	
	Is message	${NORMAL EXPECTED MESSAGE}

These actions were each implemented as user keywords so that they can be reused for future workflow tests. This minimizes the duplication between workflow tests and reduces the maintenance effort. We skip the user keywords and the test libraries; they are similar to the previous examples.

Robot Framework conclusion

These examples showed some core features of Robot Framework. Some other features worth mentioning are

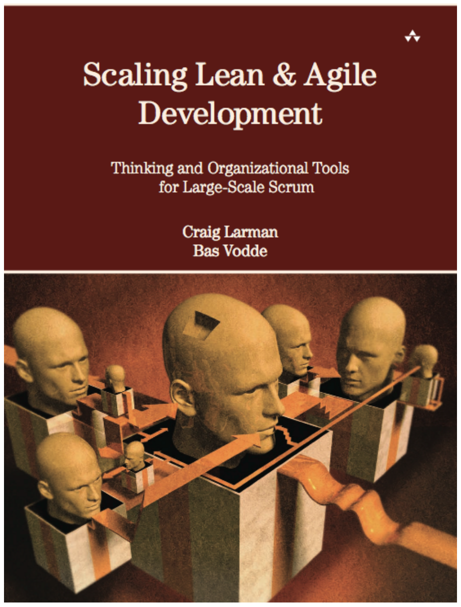
- Ability to classify tests by using tags. These can be used for reporting, statistics, or selective test runs.
- Availability of generic test libraries such as Swing library and Selenium library.
- Clear logs and reports that make it easy to discover what happened.
- Easy integration with other systems such as SCM systems or CI systems.
- An IDE for developing the tests—RIDE.

CONCLUSION

A-TDD is a collaborative requirement clarification technique that uses executable examples for exploring requirements. Its a team technique that creates a collaborative environment and parallelizes all activities within an iteration. It helps blurring traditional roles and create a cooperative environment.

RobotFramework is a test framework build with A-TDD in mind. It uses tablular format for describing examples which become executable by implementing a piece of glue code called test libraries. The multi-layer table structure encourages the highest level tables to be in natural language making the requirements and the example tests equal.

References

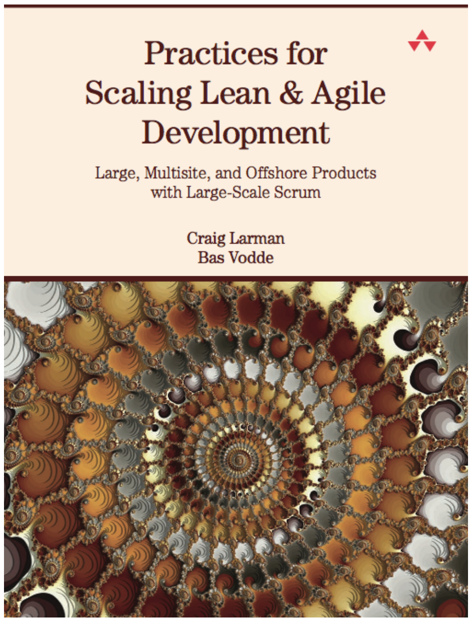

Scaling Lean & Agile Development

Thinking and Organizational Tools for Large-Scale Scrum

Craig Larman
Bas Vodde

Chapters:

- ❑ Introduction
- ❑ Systems Thinking
- ❑ Lean
- ❑ Queueing Theory
- ❑ False Dichotomies
- ❑ Be Agile
- ❑ Feature Teams
- ❑ Teams
- ❑ Requirement Areas
- ❑ Organization
- ❑ Large-Scale Scrum


Practices for Scaling Lean & Agile Development

Large, Multisite, and Offshore Products with Large-Scale Scrum

Craig Larman
Bas Vodde

Chapters:

- ❑ Large-Scale Scrum
- ❑ Test
- ❑ Product Management
- ❑ Planning
- ❑ Coordination
- ❑ Requirements
- ❑ Design
- ❑ Legacy Code
- ❑ Continuous Integration
- ❑ Inspect & Adapt
- ❑ Multisite
- ❑ Offshore
- ❑ Contracts