Patterns for Scripted Acceptance Test-Driven Development

Osório Abath Neto (osorinho@gmail.com)
Jacques Sauvé (jacques@dsc.ufcg.edu.br)
Ayla Dantas (ayla@dsc.ufcg.edu.br)

Departamento de Sistemas e Computação (DSC),
Universidade Federal de Campina Grande (UFCG), Brazil

This paper introduces a series of related patterns to be used in an acceptance test-driven development (ATDD) approach to software development using scripts. ATDD uses executable *client-readable* acceptance tests written in the form of scripts as the key analysis artifacts that guide software development and presents a number of advantages over traditional analysis artifacts – texts and diagrams. They include a better bridging of communication gaps between clients and developers, prevention of feature creep and synchronization between analysis changes and the code being written.

Intent

There are some challenges involved with the application of ATDD so that it yields most of the benefits it offers. With this set of patterns we want to summarize some of the insights others and we have had with the approach that help cope with the challenges it imposes.

Context

Take the client, who knows software requirements and business rules, and have an analyst-tester capture those requirements and business rules directly into client-readable acceptance tests, which define problem domain, client-related interests. These will be thereafter the chief analysis artifacts for the program being developed. The tests automate software development – they can be used to verify software correctness in a regressive manner as software is developed and thus guide development. Furthermore, they are directly tied to the analysis activity: as requirements or business rules change, the change is automatically reflected in the program being written.
If you use this approach coupled with test-driven development techniques, you have ATDD – acceptance test-driven development, an extension of TDD with some modifications of its core practices (due to the more direct involvement of the client). A full discussion of the approach can be found in http://easyaccept.org/atdd.html.

Acceptance Testing Tools

In the last few years, a number of tools have emerged to help create and run client-readable acceptance tests. Any of those tools, as long as they really deal with *client-*

readable acceptance tests, can be used to yield the benefits of ATDD and executable analysis.

The most widely known of these tools is FiT (Framework for Integrated Testing) http://fit.c2.com. It uses acceptance tests represented as tables enclosed in HTML files. The rationale behind this approach is that tables are easily understood by clients and can be created in widely available spreadsheet editors. Upon running FiT, the tables are linked to the program being tested using hookup code called *Fixture* code, which guards the language of the clients from the names used in the application. Fixtures are provided by developers and are composed of attributes and methods that correspond exactly to the column titles in the test tables.

Most other tools, including EasyAccept http://easyaccept.org (whose format we employ in the pattern's examples) and Exactor http://exactor.sourceforge.net use a scripted text-driven approach, in which the tests are described using a format that resembles natural language - mostly, a sequence of sentences with verbs and nouns that serve as commands. Commands are then matched to methods in the hookup code that allow the program under test to be accessed.

Script Languages

In this paper's examples, we use acceptance tests written as textual test scripts that comply with EasyAccept's format. Acceptance testing tools have predefined "built-in" commands or structures to write the tests with, but for every specific program that will be developed, a specific set of commands must be created. We call this set a "script language", a vocabulary of the allowed verbs that are used to interact with the program being built. Tools like FiT, which do not use a textual approach but rather employ tables, have an equivalent for a script language in the form of "table templates", the basic elements that are used to write the tests with.

Business Objects

A greater number of acceptance tests involve handling business objects. These are logical entities manipulated by the program that *make sense to the client*, i.e., clients can understand what they mean and represent. They are also the domain names a client employs when discussing software requirements with the developers. A supplier, an order, a product item in a sales software; a player, a board place, a dice in a Monopoly game, etc.

Business objects may or may not (but typically do) correspond to an actual single software object. Even if they don't (e.g., if a number of interrelated objects and structures as a whole actually serve as a single business object), that remains hidden to the client. The client doesn't care *how* a particular business object is actually internally represented in the program.

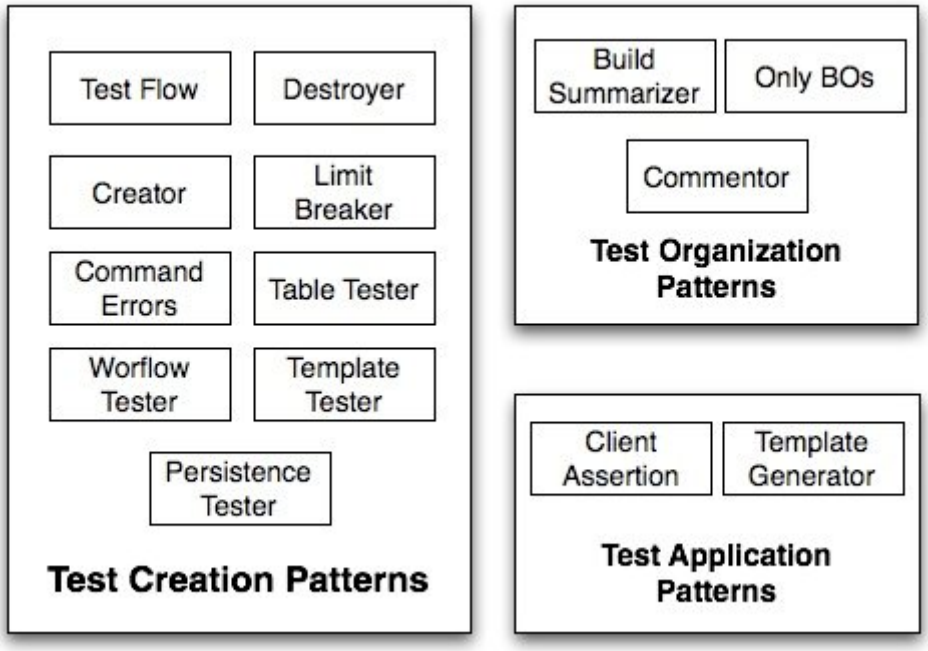
Problem

Although ATDD has a lot of benefits, there is an important challenge involved with its application, which is taken for granted in TDD that uses unit tests: the client is

generally a non-technical person who doesn't know programming or testing skills, but he must *understand and review* the tests because acceptance tests deal with problem domain, client-related interests (requirements and business rules) that evolve over time.
Making the client review tests generally requires that the client be **imbued with a testing culture** – upon reading a test script, he must not only *understand* what the tests do, but also ideally be able to *criticize* them and *suggest* new tests to improve the script. To alleviate the client's effort of reading and reviewing the tests, test writers must **create understandable tests from the start and keep them so thereafter**.
Additionally, in order for ATDD to yield most of what it offers, a learning curve is associated with the approach involving **interactions between the client and developers** and the ways in which acceptance tests are used to guide software development.

To help cope with those problems, we have defined three sets of interrelated patterns:

1) patterns to *create* scripted acceptance tests;
2) patterns to *organize* scripted acceptance tests;
3) patterns to *apply* scripted acceptance tests in an ATDD approach



Patterns in the first two groups also help with the goal of imbuing the client with a testing culture. Thus, some of them might feel familiar to some extent for anyone involved with software testing in general and test-driven development using unit tests in particular. This stems from the fact that *creating* or *organizing* tests of any kind involves general testing problems that are similar in any level.

Patterns to create acceptance tests

Test Flow

The basic flow of a test consists in the three-steps sequence: build, operate and check. *Build* a scenario you want to test; *operate* the program with the desired action and *check* the new program's state.

Context:
- You need to verify if a given software property results as expected after a given action

Problem:
- The lack of a basic structured test sequence gives rise to tests that are complicated, hard to understand and maintain

Forces:
- Clients don't have a testing background and need an approachable way to understand tests without much effort;
- Clients, as reviewers of acceptance tests, must gain a testing culture

Solution:
Test Flow provides the basic structure for an acceptance test, consisting in three steps. The first step is to build a scenario – a particular state of software taking in account the elements you want to test. In this step you use *preparer* commands to find shortcuts in the logic flow of the program to the scenario you want to test. Once the scenario is built, you use *doer* commands to perform the specific actions that modify the current state of the program to the state you want to check. Finally, you check if the program state has resulted in what you expected, comparing values of properties of interest to values you wanted. If the expected outcome of the action is an error, the check step is done coupled with the operation step. This 3 steps sequence is easy to remember, apply and identify when reading a test script; this helps alleviating the learning curve of reviewing acceptance tests for the clients.

Example:
```
# taken from tests for a Monopoly board game
# test case: checking if the luxury tax of a Monopoly game
# takes $75 from a player

# preparers: place player jacques on "Jail:Just Visiting"
# and set him with $1000
setPlayerPosition playerName="jacques" position=30
setPlayerMoney playerName="jacques" money=1000

# doer: make jacques fall on luxury tax
rollDice diceResult=8

# check: use an expect command and a getter to check
# if jacques paid $75
expect 925 getPlayerMoney playerName="jacques"
```

Related Patterns:
- **Persistence Tester** and **Template Tester** (see below) are special cases of Test Flow used for specific testing needs.
- **Creator** and **Destroyer** deal with Business Objects and also follow the basic flow sequence.

Creator and Destroyer

Context:
- You need to test if a business object has been successfully created or removed in the program;
- Business object creation and destruction are common operations in test scripts

Problem:
- The lack of a systematic repeatable sequence to test business object creation and destruction hinders test understanding and may leave these operations incompletely tested

Forces:
- In order to be used in a program, business objects must be created; often they also need to be destroyed – distracted clients and developers frequently overlook this fact;
- Clients don't have a testing background and need an approachable way to understand tests without much effort (this is especially important for common operations like these);
- Clients must gain a testing culture

Solution:
Whenever you are creating some business object (a new client, a new player, a new order) that will be used in a test, first make sure that it doesn't exist (expecting an error), then create it, and finally make sure it exists (if applicable, also validate its properties). When testing the business object's destruction, first check that it exists; then destroy it, check that it doesn't exist, and finally check that you can't remove it anymore. This is a clear-cut, easy to follow sequence of steps that contains all of the important tests for business object creation and destruction.

Example:
```
# creating a new employee in an enterprise's information system
expectError "employee doesn't exist" getEmployeeSalary \
  employeeName="John Doe"
createEmployee employeeName="John Doe" salary=1500
expect 1500 getEmployeeSalary employeeName="John Doe"

# removing John Doe
expect 1500 getEmployeeSalary employeeName="John Doe"
removeEmployee employeeName="John Doe"
expectError "employee doesn't exist" getEmployeeSalary \
  employeeName="John Doe"
expectError "employee doesn't exist" removeEmployee \
  employeeName="John Doe"
```

Limit Breaker

Context:
- You need to test a business object or program flow that has limits (ranges or sets of allowed values)

Problem:
- Unchecked limits are common sources of software bugs;
- The lack of a systematic approach to testing limits leaves room for unchecked limits

Forces:
- Testing by example, although not necessarily complete, is generally enough when it comes to communicating valid and invalid situations to the developer. From the special cases, the developer can generalize the working code.

Solution:
Acceptance testing involves giving “examples” of valid and invalid software operations or business object’s properties.
When writing tests, always precisely demarcate limits by giving examples of inner and outer bounds. Make sure values immediately out of the bounds throw errors and limit inbound values don't. In general, tests for the immediate limits are enough to indicate to the developer that further values aren’t acceptable, even if the tests don’t explicitly list all of them.

Example:

```
# a Monopoly game only accepts 2 to 8 players;
# errors are thrown for 1 and 9, and games are created
# successfully for 2 and 8
expectError "Too few players" createMonopolyGame \
  numPlayers=1 playerNames={John Doe} tokenColors={black}
expectError "Too many players" createMonopolyGame \
  numPlayers=9 playerNames={a,b,c,d,e,f,g,h,i}\
  tokenColors={black,white,green,red,blue,yellow,brown,gray,pink}
createMonopolyGame numPlayers=2 \
  playerNames={John Doe,Mary Donna} tokenColors={black,white}
finishGame
createMonopolyGame numPlayers=8 \
  playerNames={a,b,c,d,e,f,g,h}\
  tokenColors={black,white,green,red,blue,yellow,brown,gray}
```

Related patterns:
- You often use Limit Breaker in conjunction with **Command Errors** to increase test coverage.

Command Errors

Context:
- You have created a new command to incorporate in the script language and will start writing tests with it

Problem:
- Addition of new commands involves interacting with new, untested software operations that can be the source of bugs;
- The new commands can be used in the wrong way

Forces:
- Test coverage is difficult to attain if you don't approach testing systematically;
- When thinking on a new command to add to the script language, you often think about tests that will involve this command, including anomalous uses;
- A system should be able to identify a misuse of one of its operations and may even give clues about the problem cause

Solution:
Every time you create a new command to incorporate in the script language, you must systematically write tests that devise a "fire curtain" of error tests that wear out the possibilities of errors: think about the command's scope, limits, restrictions and invalid situations that it could make arise. Additionally, put the command error testing in a separate section of the script for clarity.

Example:

```
# We have just devised a command, setEmployeeSalary
# setEmployeeSalary should throw errors when an employee
# doesn't exist or when invalid values are chosen
expectError "employee doesn't exist" setEmployeeSalary \
  employeeName="Nonexistent John" salary=1000
expectError "salary can't be negative" setEmployeeSalary \
  employeeName="John Doe" salary=-15
expectError "salary can't be null" setEmployeeSalary \
  employeeName="John Doe" salary=0
```

Related patterns:
- You often use Command Errors in conjunction with **Limit Breaker** to increase test coverage.

Table Tester

Context:
- You need to test extensive lists of features for multiple business objects of the same kind, or multiple examples to test *formulas* (calculations);
- Clients want to declare business rules in a formulaic manner;

Problem:
- Sequences of commands become too verbose when it comes to testing extensive lists of properties or calculation examples. This makes a script distracting, hard to review and maintain.

Forces:
- Developers need an algorithmic translation of business rules so that they can be tested in an automated way;

Solution:
Use tables for testing in the fashion table based tools do, either by using an IDE that translates tables to scripts automatically, or by employing tailored built-in commands that simulates tables (like EasyAccept's *expectTable* command).
This eases up reading long-winded lists of properties and formulae examples and, in the case of formulae, hides its associated algorithm from the script while still keeping the algorithm accessible to the developers in the hookup code. Formulaic statements should be added to the script as a comment for the sake of understandability.

Example:
A client would state that "employees are paid a base salary plus 1% of his sales in the month minus the social security, which is a base value times the number of members of his family". This formulaic statement would become a comment in the tests. Then you need examples:
"if the salary is $1500, employee sells $50000, and has 2 kids, with a base social security value of $20, we owe him $1940"
"if the salary is $1000, employee sells $20000, and has 3 kids, with a base social security value of $20, we owe him $1120"
These examples (and many others more) could be written into a table. Using a table-driven test approach (for example, with EasyAccept's *expectTable* command), the algorithmic translation of the table into tests would be enclosed in the hookup code, hidden from the client.

employeeName	salary	maritalStatus	numberOfKids	socialSecurity	Sells	wage?
John Doe	1500	married	2	20	50000	1940
John Doe	1000	married	3	20	20000	1120

If you don't use a table, you must describe the algorithm in the tests explicitly, and the tests become more verbose (you need a series of commands for every line of the table), as below.

```
# first example
createEmployee employeeName="John Doe" \
  salary=1500 maritalStatus=married numberOfKids=2 socialSecurity=20
# for a more typical scenario, there could be multiple
# employeeSells, but let's say the total was from a single sale
employeeSells sellValue=50000 date=05/02/06
expect 1940 calculateEmployeeWage employeeName="John Doe"

# second example
createEmployee employeeName="John Doe" \
  salary=1000 maritalStatus=married numberOfKids=3 socialSecurity=20

# for a more typical scenario, there could be multiple employeeSells, # but
let's say the total was from a single sale
employeeSells sellValue=20000 date=05/02/06
expect 1120 calculateEmployeeWage employeeName="John Doe"
```

Related Patterns:
- **Workflow Tester** is a pattern used for reasons opposite as those for Table Tester; you need to test sequences of commands, but tables come in the way.

Workflow Tester

Context:
- You need to test a workflow - an often-extensive sequence of operations;

Problem:
- Tables used to test workflows become too verbose and make a script hard to understand and maintain.

Forces:
- A workflow is sequential in nature and its testing matches specifically the way scripted testing tools operate;

Solution:
Use commands in sequence instead of tables.

Example:
In a Monopoly game, you need to test if a player is thrown out of the game after falling on a series of tax places and taking a series of actions that decrease his money. If you use tables, you will need a separate table for every command you use.

Template Tester

Context:
- You need to test massive textual or non-textual content;

Problem:
- Massive textual content is cumbersome when included in a script because it may render the script unintelligible;
- Non-textual content can only be directly included in a script through an IDE;

Forces:
- Templates for test cases can be generated automatically from the customer using partially complete software.

Solution:
Divert the contents that need to be tested to outside the script (on another file) and use a template to compare it with.

Example:

```
generateReport output="report.txt"
# equalFiles is a command in EasyAccept that tests if
# two files have the exact same content
equalFiles templates\reportTemplate.txt report.txt
```

Related Patterns:
- Template Tester is a special case of **Test Flow**

Persistence Tester

Context:
- You need to test if data entered in the program persists in future sessions.

Problem:
- The lack of a systematic approach to testing persistence can hinder understanding (particularly by the client) and test coverage.

Forces:
- Setting up a scenario for persistence can often be reused as other tests;

Solution:
Testing persistence consists in two parts: in the first, the program under test is run, data is cleared, some information is entered and checked, and then the program is closed. The second part runs the program and checks if the information entered in the first part is still there. The first part is a meaningful test by itself and could serve as a setup for other tests. The second part is peculiar because, although test independence is something one must aim for, it depends on the first and should be run only after it.

Example:

```
# creating an employee - script1.txt
clearEmployeeDatabase
expecterror "employee doesn't exist" getEmployeeSalary \
  employeeName="John Doe"
createEmployee employeeName="John Doe" salary=1500
expect 1500 getEmployeeSalary employeeName="John Doe"
saveEverythingAndCloseProgram

# restart is an EasyAccept command
restart

# checking persistence
expect 1500 getEmployeeSalary employeeName="John Doe"
```

Related Patterns:
- Persistence Tester is a special case of **Test Flow**

Patterns to organize acceptance tests

Build Summarizer

Context:
- Too many tests in the script begin the same way

Problem:
- Repetition of the same test sequence in multiple tests hinders understanding and makes the scripts harder to maintain

Forces:
- Much of what is commonly included in the build sequence of a test can be summarized in an expressive command without lack of understanding

Solution:
If multiple tests have the same setup or preparer commands sequence, set it aside as a separate single test sequence and have all tests that begin with it refer to it. Alternatively, create a preparer command if you need to hide its content from the script (only developers will have access to what it does, in the hookup code).

Example:
Suppose you have a sequence of commands that create a sample database of employees for subsequent testing. The same database is used throughout the scripts or within the same script. Put it aside in a separate script and refer to it using an `executeScript` command.

```
# script 1
executeScript sampleDatabaseSetup.txt
… and then do the tests you want

# script 2
executeScript sampleDatabaseSetup.txt
… and then do a number of different tests
```

Only Business Objects

Context:
- Non-business objects are included in the tests

Problem:
- When non-business objects are included in the script, the tests related to them will be unintelligible for the client, who typically is not a technical person and doesn't understand non-business objects

Forces:
- When clients don't understand tests, their commitment to keep reviewing tests is lowered;

- Testing non-business objects is important and must be done as much as the testing of business objects

Solution:
Only business objects should be tested in an acceptance test script. Remove all tests for non-BOs and make them unit tests instead. That way, such objects are still tested (in TDD, everything must be tested) but, as they are not a concern for the client, they should be hidden from him.

Example:
```
# the tests below should be unit tests
expect 25 getBQueueNodeFatherIndex
expectError “Code #432653: unexpected fault for the middleware connection”
getCORBASkeletonMiddlwareProperty
```

Commentor

Context:
- Developers can’t understand exactly what some tests are doing.

Problem:
- If developers don’t understand a test, either development stalls or a bug or wrong requirement may be introduced in the program;
- Comments become inconsistent with the associated tests if they are not updated when the tests change.

Forces:
- Comments in the tests serve an important role of communication between clients and developers, as they further clarify the script that reconciles the languages of both.

Solution:
Ask the client for clarification and include it as a comment in the script, explaining the test. Additionally, whenever a test changes (after being clarified), the associated comments must be updated. Comments are an integral part of the test base and serve as the means of communicating how the program should behave. As the understanding of requirements and business rules evolve (including client's decisions) or doubts are cleared, comments must be added to be tests.

Patterns to Apply Acceptance Tests during Development

Client Assertion

Context:
- A developer has found a potential wrong test (test bug).

Problem:
- If developers change what they think might be test bugs by themselves, potential feature creep will emerge in the program;
- If test bugs are not discovered the system may not be what the clients expect

Forces:
- Clients must frequently review tests to assure they are correct

Solution:
Every time a doubt arises over a test (i.e., involving requirements and/or business rules), developers should ask the client for clarification. No test should be modified without the client's consent. This avoids developers introducing errors in the test base and can serve as a means of impelling the client towards reviewing the tests. Additionally, the clarification should be included as comments in the tests.

Template Generator

Context:
- Development is under way and partially working software is available. You need to find more test cases.

Problem:
- As development progresses, it becomes harder to find test cases other than the more direct examples of software functions.

Forces:
- Automation speeds up test creation;
- Software usage by the client can reveal bugs that would otherwise not be considered;
- Providing a recording mechanism requires extra effort by the developers.

Solution: have the client or end user operate the partially working software and provide a background mechanism to automatically generate a test script based on his actions (by recording the sequence of actions). When the client is done with a given operation, he examines the results that were presented and either accepts or rejects it. This result then functions as a template for a test consisting in the sequence of actions performed by the client.