

2008

PACIFIC NW SOFTWARE
QUALITY
CONFERENCE



COLLABORATIVE
QUALITY

OCTOBER 13-15, 2008

CONFERENCE PAPER EXCERPT
FROM THE

CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

Test Automation Design Pattern

A pattern for automatic test case generation

Lian Yang
liany@microsoft.com

Abstract

Test automation itself is a software engineering project and its effectiveness, efficiency, and maintainability quite often present the same engineering challenges as other software projects. The development community has been embracing the “design pattern” concepts for 15 years, in an effort to address software engineering difficulties and formalizing design and coding process. While mentioned in various papers ([3]), design pattern practice is still not well practiced in the software test automation community. Ill-planned, and roughly designed test automation projects are still wide-spread. This paper tries to define a “test case automatic generation” design pattern, which addresses the most common and fundamental test automation engineering issues facing most testers today. The concepts presented in this paper has been implemented by the author and his team in testing several Microsoft© products.

Glossary

Terminology or Acronym	English
SUT	Software Under Test
SPEC	(product) Specification
TTS	The test state
TCGEP	Test Case Generation and Execution Pattern

1. Test Automation

1.1. Definition

A Test automation system is a software system that exercises a target software product (or SUT) in order to detect defects in SUT. Software defects can be any of the following things:

- Mal-functions that cause the user to be unable to use the software, such as hanging, Access Violation (in Windows), and system crashing
- Not SPEC compliant
- Bad user experience that may or may not be in conflict with the SPEC.
 - Slow response
 - Using too much resource

- All kind of things that don't make users job easy
- Unwanted side effects
 - Data corruption
 - Security issues

Most of software defects (or bugs) fall into the above categories. An ideal test automation system should be able to catch a fair percentage of the above kind of bugs. From the above definition, test automation should consist of the following elements (in red):

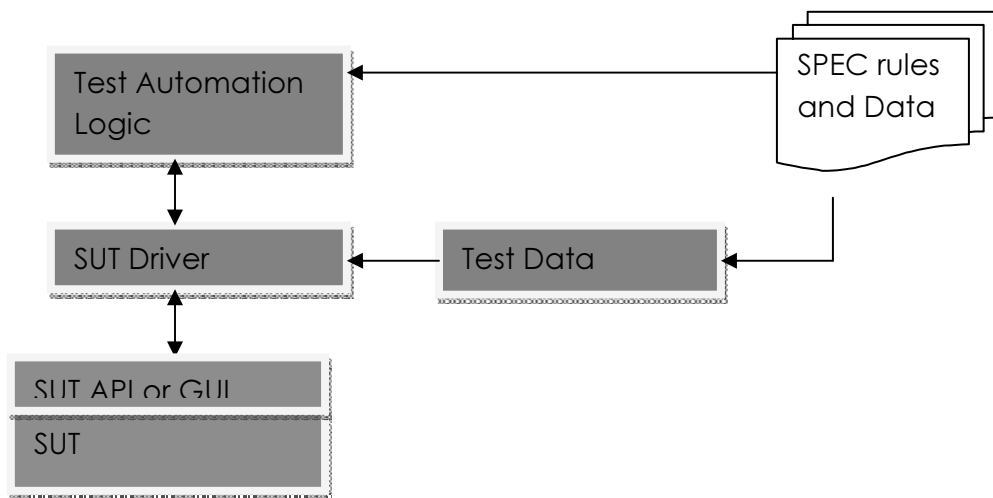


Fig. 1: components for a generic test automation

The above diagram can be thought as a high-level design pattern, from which other design patterns can be derived. The most important concepts introduced in the above generic pattern are:

- 1) Separation between test logic and SUT driver
This allows the design patterns such as TCGEP to express the design concepts in abstract terms while leaving all SUT specific details to the SUT Driver
- 2) Derive test data directly from SPEC in form of data template or using advanced SPEC to case generation tools (such as Microsoft Research's SPEC#)
Hard-coded test data is as bad as hard-coded magic numbers or strings in product code. There is really very little justification for test automation applications to continue using hard-coded magic data.

1.2. A sample SUT application

We need to use a sample SUT application to illustrate concepts and points throughout the paper. Our sample application is code named **Happy-Fish**. Happy-fish is a simple on-line banking system with the following components:

- A WEB UI Front End
 - Customer pages
 - A logon page with user name, password fields, and an OK button
 - A user page with account names as link buttons, and a logoff button
 - A user account detail page with a data grid control and a logoff button
 - Admin pages with a button “add user”, that leads to a “Add New User Page”
 - Assume that admin pages are only known by the Admin and privately accessed (without password protection) in order to simplify the demo application
 - Add New User Page with user name, password fields, and an Add button
- A business logic mid-layer behind the front end on the web server
- A SQL DB that stores customer account information and personal data

The following diagram shows the components of the system:

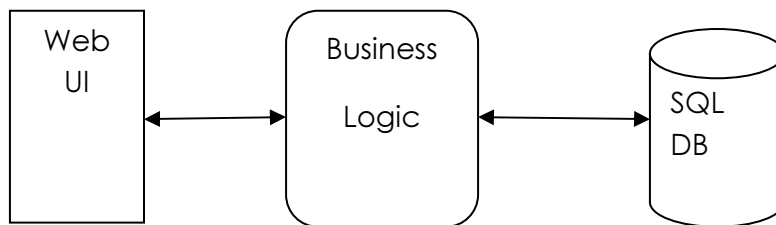


Fig. 2: components for a sample SUT application code name

2. Design Pattern

First introduced by the “gang of four” in 1994 [1], Design Pattern Concepts have received wide acceptance in software development community. In software engineering, a design pattern is a **general reusable solution** to a **commonly occurring problem** in software design ([WIKIPEDIA](#)). In this paper, the essence of Design Pattern contains the following three elements:

- The intent

States the purpose, problem domain, applicability, and limitation of the pattern (problem)

- Design constraints

States the implementation requirements, adaptability, and extensibility of the pattern (reusability prerequisites)

- Supporting framework

States the utility, helper functions, and inherited attributes, which helps pattern user to implement the working system (reusable objects and concepts)

I will use the above elements in describing the “Test Case Generation Pattern”.

3. Test Case Generation and Execution Pattern (TCGEP)

Think of your automation as a baseline test suite to be used in conjunction with manual testing, rather than as a replacement for it. [2] To make the automation worthwhile, we need to understand what test cases really are, come up with an abstract and formal description, and form some basic test case generation design patterns. We will not cover specific test tool areas such as code coverage data collection.

3.1. Test case definition

3.1.1. Test Case Template

A test case in test automation terms can be defined as the following two elements:

- 1) *Applying a set of actions in form of function calls to SUT. The functions are called in a given order with their parameters precisely defined.*
- 2) *The result of the actions can be clearly predicated and measured by a set of other function calls to SUT.*

In reality, a test case resembles a state transition in a finite state machine. However, in describing this pattern, we use results from function calls to signal the pass or fail of a test case, rather than use state machine. This model makes it simple to model a test case and is also effective when combining with run time “test state”.

Also note that the second element of test case is the “testability” requirement of a test case. In reality, it can be done as black box or white box depending on the nature of SUT and its testability level.

For our pattern to work, we have defined the following “test case template” class, based on which one or more actual test case can be formed:

A **test case template** is a class that contains a set of Action instances, and can be executed by calling the *DoTest* method of each action until one of the action signals failure or state change. The class definition as well as pseudo code for **execute** is shown below:

```

class MyTestCaseTemplate: BaseTestCaseTemplate
{
    List<Action> _myActionList;
    Void AddAction(Action action) {...}
    Virtual GetNextAction() (//default get test case sequentially ....)
    ....

    Void Execute()
    {
        While (true)
        {
            Action action;
            While (action = GetNextAction())
            {
                Try
                {
                    action.PrepareTestData();
                    action.DoTest();
                    action.Validate();
                }
                Catch(TestStateChange ex)
                {
                    // handle the state change
                    StateChanged();
                    break;
                }
                Catch(TestException ex)
                {
                    // handle error
                    HandleError(ex);
                    // decide to continue or quite
                    .....
                }
            }
        }
    }
    .....
}

```

Table 3.1: pseudo code for test case template

The above class is named as **TestCaseTemplate**, instead of **TestCase** for a reason. To satisfy the 2nd elements of a test case definition, the above function *Execute* shall be predictable and can be measured for state change, pass, or failure accurately. This would not be possible given that each action has to prepare for its data at runtime. At the same time, **GetNextAction** may not return the same action as the last time the test was executed. Hence the “predictable” test case is only available after you play back the previously generated action/data sequence. This is why in the pseudo code, it’s extremely important to log every action (**LogAction**) after running “**DoTest**”.

The essence of the test case template idea can be summarized by the following points:

- 1) Avoid using hard coded test steps and test data early in test case generation by randomizing the actual steps and data as much as possible. For example, to test the user logon page of Happy-fish, your test steps can be either enter user name, enter password, and click OK, or enter password, enter user name, and click OK. On the other hand, you should randomly select a legal or illegal user name, rather than always use the same user name.
- 2) Separate test steps (actions) from test data and dynamically generate data at runtime
- 3) Log (serialize) the test case run and make it possible to replay test cases based on previously logged step sequence and data
- 4) Test case is confined by the SPEC (step and data requirement) but by no means static and inflexible. This makes test automation not only a perfect tool for regression but also introduces some explorative testing elements to test automation. For example, Happy-fish "SPEC" dictates that a user name must be longer than 4 characters and shorter than or equal to 10 characters. This information is used to generate both positive and negative cases for "Add New User Page".

3.1.2. Action definition

A test action for a test case can be defined as the following three elements:

- 1) A specific function call made to SUT.
- 2) A group of parameters, with each parameter a set of values
- 3) Each parameter defines a *template* from which real value derives. The template is derived from the product SPEC.

The following pseudo code describes a test case action:

```
class TestCaseStep: BaseTestStep
{
    // Will call SUT API f1 with two parameters: an integer and a string
    public void DoTest() { ....}

    // Log this action
    public void LogAction();

    // Derive actual test data from data provider
    void PrepareTestData();

    // Validate the action
    void Validate();

    // Can the action be run now?
    public virtual bool ActionCanRun() {...}

    // first parameter needed for calling SUT function
    String _param1_template = "integer:0-100";

    // Second parameter needed for calling SUT function
```

```
String _param2_template="string:charset:all illegal:~!@# length:1-256"  
...  
}
```

Table 3.2: pseudo code for test action

From the above test action pseudo case, we learn that the actual action will be realized only when we run "**PrepareTestData**" because the parameter values are not hard-coded and only their templates are given. Regardless of the result of the test run, the action is logged and serialized to an XML file, in which the entire test case is serialized, makes it a concrete step within a concrete test case.

3.2. Test Case Generator

In the pseudo code listed in Table 3.1, **GetNextAction** as a virtual method, is also a variant factor in test case generation. It is difficult to predict what factor at run time will influence the decision of picking an action from a set of predefined actions. But to make it simple and practical, the pattern provides 3 implementation choices:

- Sequential action
User is responsible to make sure the order of the action is the same as the order of the action list. This is suitable for simple test cases in which a set of well defined test steps can be used to simulate a usage scenario and the SUT behavior is predictable.
- Action/Data combined Permutation
This method uses multiple-dimension permutation algorithm to calculate the action/data combination for each test case run. This is suitable for small steps/data value combination and guarantees to hit most if not all test cases automatically. It is also a great stress, long-haul test case generator.
- Randomly picking an action
This method randomly picks up an action from an existing action list and randomly generates input data from data template.

No matter what choices you use, the derived class could override the decision making by calling **ActionCanRun** function to determine whether an action is relevant at the moment. This requires us to define the runtime decision making mechanism as a restriction. "Test State" is a simple interface we define for this purpose. It is a bridge between the framework and the SUT Test Drivers and helps the test action interact well with real world usage scenarios.

3.3. Test State

We have used a notion of test context or test state, to capture the SUT state, with respects to test applications. It is implemented as a simple string-object Hash-table. The idea is powerful even though it is extremely simple:

- The captured SUT states can be serialized to an XML file at any moment, making it easy to debug the test code as well as SUT code in case of test failure.
- The SUT states serve as an important mechanism for choosing the next test action. The following pseudo code from one of our test application demonstrates it:

```

Action GetNextAction()
{
    TargetList tgtLst = _testStates.GetNamedValue("TargetList") as TargetList;
    If (tgtLst.Count > 0)
    {
        return new ActionCreateVDS();
    }
    return new ActionCreateTarget();
}

```

Table 3.3: Pseudo Test Step Generation Code

In the above pseudo-code, we choose "CreateVDS" action if there exists a "TargetList" object in the test state, while choosing "CreateTarget" action otherwise.

The Test State (TTS) can be viewed as an in-memory object model maintained by test automation which reflects the real SUT state from test automation's point of view. Whenever, there is a conflict between TTS and SUT, it's either a SUT product bug or a test bug and if it's latter we need to improve our test automation.

3.4. Pattern Description

Name

Test Case Generation and Execution Pattern

Intent

- Reusability
- Standardize Basic Test Automation Components to guarantee the basic test coverage regardless of SUT's problem domain
- Maximize the randomness in test case generation thus maximize the opportunity of finding real SUT bugs as early and as much as possible using the minimum efforts
- Possibility to auto-generate test program for the upper layer of test automation logic
- Expedite test automation development process

Constraints

- Data provider interface
- Test action interface
- Runtime state interface

Support Framework

- Standard data providers
- Standard error handling mechanism
- Basic permutation and combined permutation algorithm implementation

3.5. Pattern Framework Design

Although this paper mainly presents the concepts and ideas behind TCGEP pattern, we did provide an implementation and a framework for further expanding it. The following is the diagram which describes the basic OOP design architecture for TCGEP.

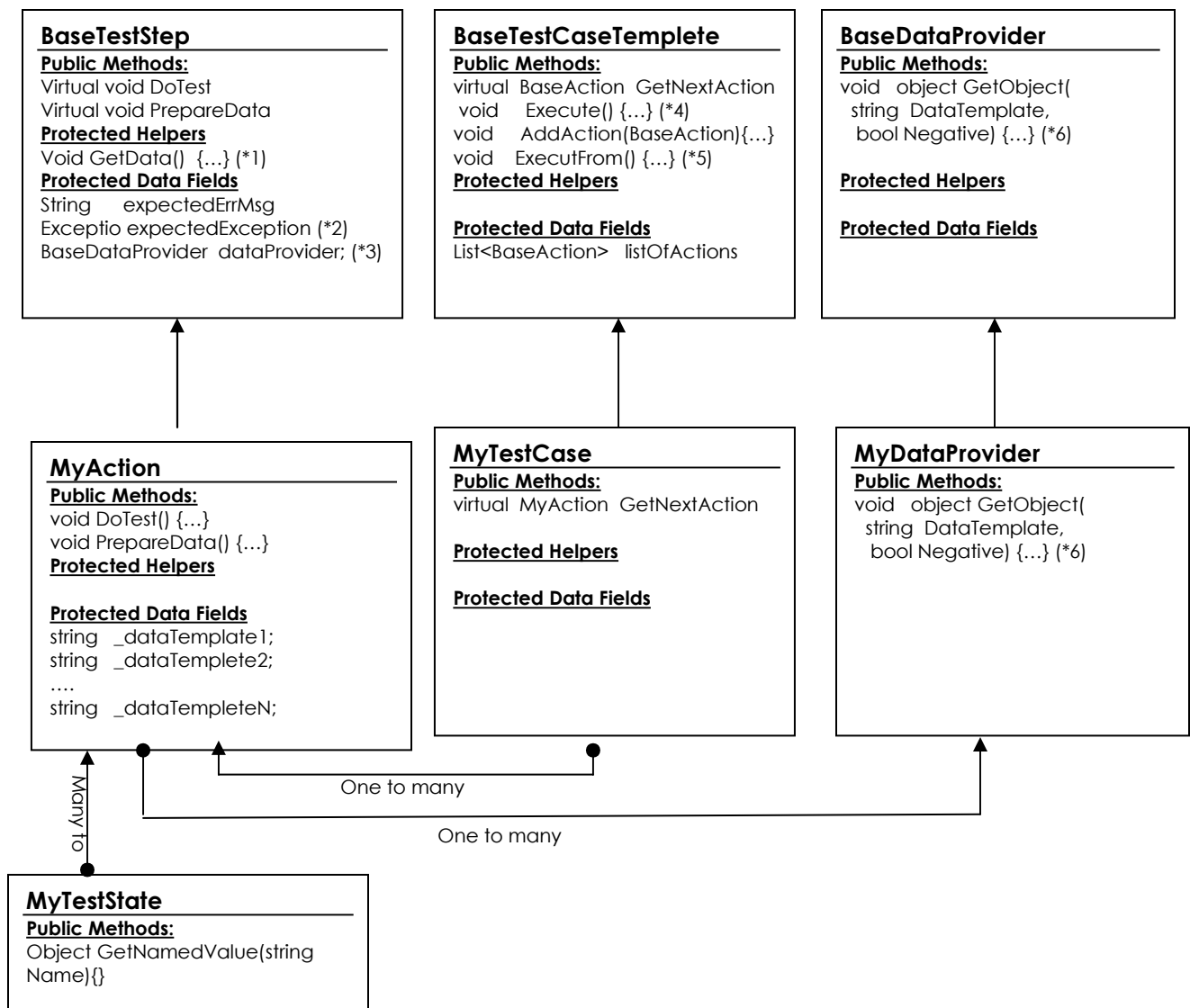
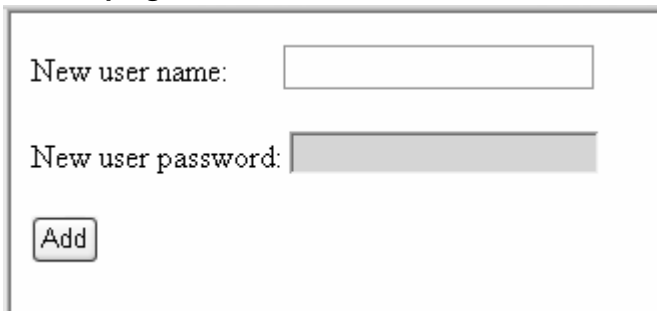


Fig. 3: TCGEP support framework class hierarchy

4. Demo Test Cases for Happy-fish Test Application

We will focus on our test efforts on admin page and user logon page. The following two UI pictures show admin page and logon page:

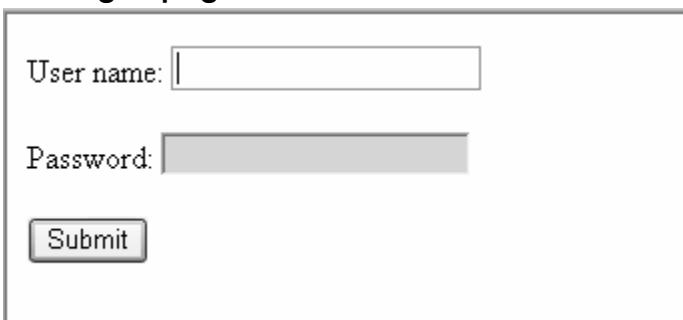
Admin page:



New user name:

New user password:

User logon page:



User name:

Password:

Although from UI perspective, the above two pages looks very similar, they are very different web pages in terms of functionality and testability:

- Admin page has much less restriction for name and password fields, as long as they meet the product SPEC regarding user name format and password requirement.
- User login page name and password fields can use random data only for fuzzing and security verification.
- User logon page functionality test data is provided by Admin page test and the data can only be retrieved from the "test state".

The following pseudo code illustrates the test steps for admin page:

```
Class AdminStep_PutName : BaseTestStep
{
    // test data template:
    string nameTemplate = "[char: any] [forbit:@%$&*()] [length:4-10]";
    void DoTest()
```

```

    { // get the name from pattern and put it into the name field:
      String name = _myDataProvider.GetString(nameTemplate);
      // UI call
      SetData(_contriDName, name);
    }
  }
}
Class AdminStep_PutPassword: BaseTestStep
{ // test data template
  string pwTemplate = "[provider: cusPWPriv.dll]";
  // similar to the above DoTest function
}

```

```

Class AdminStep_Add: BaseTestStep
{
  void DoTest()
  {
    // call UI driver tool:
    clickOn("ctrlID_add");
  }
}

```

In the above pseudo case implementation, the Admin page has lots of freedom to randomly generate test data based on a well defined data pattern for user Name and uses a customer data provider for password. On the user logon page, the data source is more restricted and can come from the three sources:

1. From previously generated Admin test pages for "good" user credentials
2. From randomly generated data provider for random (mostly bad) data
3. Mixed data sources

The following lists the pseudo code for log on page (with a simplified one step):

```

class UserLogon: BaseTestStep
{
  void DoTest()
  {
    // get name from two sources
    Name1 = _testState.GetAddedUserName();
    Name2 = _myDataProvider.GetString("any");
    // get PW from two sources
    Pw1 = _testState.GetPWForUser(Name1);
    Pw2 = _myDataProvider.GetString("any");
    // Randomly choose:
    ChooseNameAndPWCombo(out Name, out PW);

    // call UI driver tool:
    LogOn(Name, PW);
  }
}

```

In reality, we can have three test step classes from the above code to randomly choose one method. We mixed the three methods into one function just for demo purpose.

4.1. Test case generation

Some test automation applications only generate the obvious test cases due to lack of design and time. For example, the following would be the test case steps for testing the user log on page:

Step1: admin add a new user

Step2: the new user use his name and password to log on and he should be able to log on to "Account" page

Step3: log off from account page and back to user log on page

We use the default test case generator – "random test case generator", which form test cases by randomly picking a test step defined above. On the surface, this is less efficient than the "basic case", however, we have covered much more basic usage scenarios and discovered several Happy-fish bugs. The following user scenarios are some of the generated cases:

- Enter random user name/password before there is any user added by Admin user (a crashing bug found for this case).
- Enter a valid user to the DB; then from the user-logout page enter the right name but invalid password
- On the user login page enter a invalid user name and a valid password of another user
- Enter a valid user and password and we succeeded in landing in user account page
- Add a new user while an old user is trying to log on
- Remove a user and the same user tries to log on again
- Enter invalid user name or password format (for both admin and user login pages)

4.2. Conclusion

From the above example and pseudo code, we can observe the following important principals for using the test case generation design pattern:

- Focus on creating reusable actions with data template
- Use a standard test case generator for simple and effective test case generation by calling the defined action as test steps
- Use test state manager to keep track of SUT state and help decision making during runtime
- Log test state as well as test action details for regression

The practice has taught us some valuable lessons in implementing the design pattern. Nevertheless, we found that this design pattern is very effective in test coverage, stress, and explorative test case generation.

5. References

- [1] Gamma, Helm, Johnson, Lissides. 1994. Design Patterns: Elements of Reusable Object-Oriented Software
- [2] Bach, James. 1996. "Test Automation Snake Oil." *Windows Technical Journal* , (October): 40-44. http://www.satisfice.com/articles/test_automation_snake_oil.pdf
- [3] Robert V. Binder "Testing Object-Oriented Systems: Models, Patterns, and Tools".

6. About the Author

Lian Yang has been a developer, tester, and test lead at Microsoft for 13 years. He owns two patents in test automation areas and helped Microsoft © shipped products such as Windows Media, Smartphone, Vista, and Windows Storage Server.