Using AOP Techniques as an Alternative Test Strategy

Lian Yang

ly8838@gmail.com

Abstract

Ever since its inception [**1**], researchers and software professionals saw that Aspect-oriented Programming (**AOP**) has huge potential to become a powerful new approach for software development and software testing. There have been many papers in recent years on the subject of Aspect-oriented Test, exploring various interesting ideas and techniques in software test automation [**2**], [**3**]. This paper presents a strategic new approach, using AOP, for software test automation and attempts to incorporate test automation filters (or **interceptors** in **AOP** jargon) in real applications through-out an application's life cycle.

Biography

Mr. Lian Yang is an independent software solution and quality consultant based at Redmond Washington. He is the principal solution architect for Yachi Technical Consultant, Inc. His passion lies in cloud computing, SOA, next generation programming languages, software performance issues, application security, and software quality controls.

Since 1991, Lian has been working in software industry. As soon as he graduated from Portland State University with an M.S. degree, he became a Software Engineer at ImageBuilder Software in Portland, Oregon, working for award-winning PC games for kids.

He later joined Microsoft Corporation in Redmond Washington as a Software Developer in 1995 and worked at several software testing and performance analysis tools. He then became a Lead Software Developer in Test and led QA teams for MSN and Windows Storage Server.

After 2008, he worked as an independent consultant and served as lead developer for various consulting companies, building and testing commercial web site for high profile clients such as Microsoft and Avanade. He also worked as solution architect for start-up companies in cloud computing areas.

Lian has an M.S. in Computer Engineering from the Portland State University.

1. INTRODUCTION

The software testing landscape has changed dramatically in the past decades amid the development paradigm shift from traditional waterfall to an agile-style test-driven development. The wide acceptance of developer-driven unit test practices has made certain test automation practices obsolete. On the other hand, the emerging cloud-based and mobile-based applications have changed the core of the traditional software development life cycle, causing software test practice changes. Emphasis on rapid test turn around, on-line test, and non-interference testing are more and more prevalent. As a result of this development, the traditional software test methodology simply cannot meet the software application's development demand and the dynamic life cycle of average cloud and mobile application.

This article proposes an AOP-based white-box test strategy, which treats software quality assurance as an aspect of software functionality and thus employs certain aspect oriented programming (AOP) principals to software testing.

The goal is to make the always sought after white-box testing less obstructive and more dynamic. Upon achieving this goal, the following effect will be shown:

- Testing and maintainability concerns would be treated as an essential feature, improving software quality in today's agile and fast paced software development arena.
- Making test automation non-obstructive by addressing testing concern at the architectural level and defining these concerns at various *point-cuts*.
- Running and improving test cases and runtime diagnostics can be conducted at runtime in parallel, without the need for re-compiling and taking your application offline, thus making test more relevant to the entire life cycle of application.

1.1. ISSURES FACING SOFTWARE TESTING TODAY

With today's trend for test-driven development, some testers feel a little lost as to what their roles have become. Traditional test automation uses a black box test strategy and does a good job covering key functional areas. However, it overlaps, to certain degree, with developer-driven unit test framework. This could result in resource waste and cause under-testing in integration, performance, and more complex test scenarios.

Your test team should adapt to the new challenge and focus more on integration, performance, and live incidence diagnostics, which require more white-box testing methods over black-box testing.

Traditionally, we can conduct white-box tests by adding test hooks and logs. Although they are still widely used by developers as diagnostic and maintenance tools, they are rarely systematic and not widely accepted as best practices. The side effects they cause to application are:

1) Loss of code brevity
   Mixing test code and application logic produces ugly code and will likely cause long-term code maintenance difficulties.
2) Performance
   Even the test hooks and logs are usually turned off by conditional statements, they nevertheless count on overall code size and their tiny runtime effects could accumulate and result in noticeable performance loss.
3) Security
   Mixing test code and development code together is always a security concern.

It would be nice to separate development concerns and test concerns entirely and apply test concern across functions, classes, and modules, with minimal impact on development processes and code. Aspect-oriented programming (AOP) provides a good alternative.

1.2. AOP CAN PROVIDE A NEW TEST STRATEGY

According to Wikipedia, **aspect-oriented programming** (**AOP**) is a programming paradigm that aims to increase modularity by allowing ***the separation of cross-cutting concerns***. AOP forms a basis for aspect-oriented software development.

What are cross-cutting concerns? They are the concerns shared across different types, functions, and modules, thus are not very natural and efficient to be expressed by traditional OOP concepts, such as class inheritance and polymorphism. The latter is best for expressing families of entities, while the former is better expressed by cross-cutting points, joint points, and point-cuts.
The following diagram illustrates the difference:

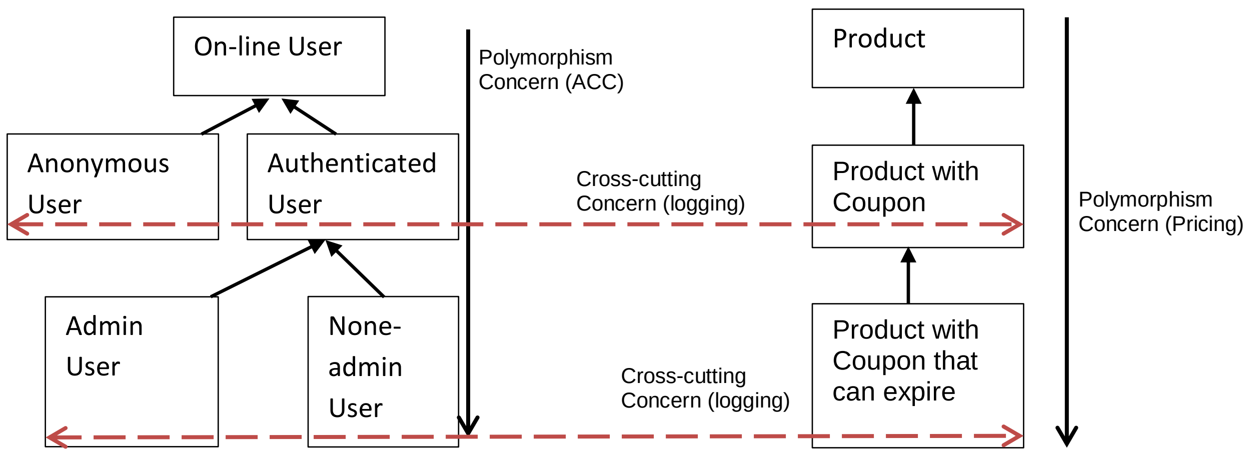
On-line User
Anonymous User
Authenticated User
Admin User
None-admin User
Polymorphism Concern (ACC)
Cross-cutting Concern (logging)
Cross-cutting Concern (logging)
Product
Product with Coupon
Product with Coupon that can expire
Polymorphism Concern (Pricing)

Fig. 1: Polymorphic concerns (↓) vs. Cross cutting concerns (←– →)

In the above on-line shopping example, we have polymorphic concerns for both user and product family (solid vertical line). For the user family, the concern is Access Control while for the product family the concern is pricing. The family-specific concerns are better expressed by OOP polymorphic functions defined in each class family roots.

We also have common (cross-cutting) concerns for both class families for logging. For these concerns, OOP polymorphic functions would work, although requiring lots of ad-hoc coding: we could add another layer as the root class for both user and product families and use another polymorphic function to support the cross-cutting concern for logging. However, that would be inefficient and not extensible because we cannot expect that the newly added root would support all the future unrelated cross-cutting concerns. More than anything else, the solution would possibly violate OOP's single responsibility principal (SRP) and open-close principal (OCP).

In essence, the OOP design philosophy dictates that the design changes should only extend vertically within the class hierarchy, not horizontally across different class families.

2. AOP FOR SOFTWARE TESTING

Aspect-oriented programming is exactly proposed for solving the above cross-cutting concern design issues. At the first thought, we could use “global functions” for cross-cutting concerns and apply these functions across the types and modules. Conceptually, it is not wrong technically to realize AOP principals. But that would hardly be considered as a new programming “methodology”. For AOP to become useful beyond a talking point, we need some technology and tools to help us expressing cross-cutting concerns without resorting to tedious ad-hoc coding. AOP does have a novel way and it is called an “***interceptor***”.

An AOP interceptor is a design pattern which uses ways to inject code between two components and alter the execution flow. Places where the interceptor can inject itself in are called ***joint points*** and the filters with which the joint points are identified are called ***point-cuts***, figuratively describing cutting a line in between two components where code can be injected.

For the example, if we use AOP interceptor to implement logging for “all” the required class methods at the *before or after function call joint points*, we can save lots of coding efforts and conceptual complexity. The following diagram shows the logging example using interceptors to decorate function foo with a pre-call logging and a post-call logging:

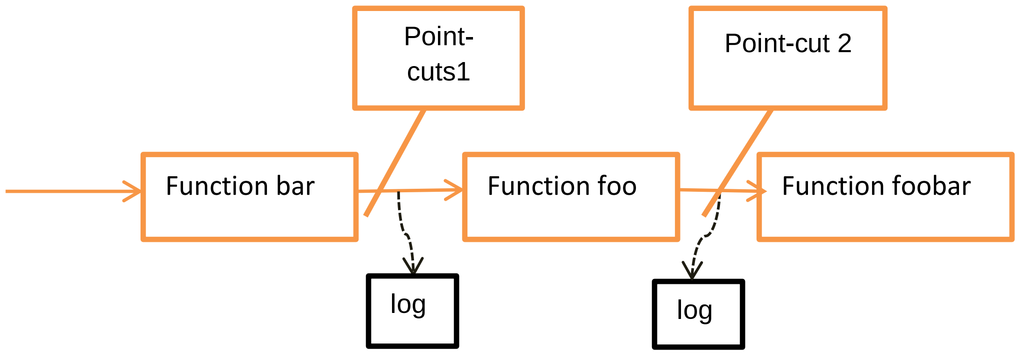

Fig. 2: an illustration of an AOP interceptor.

2.1. Design Goals

Using AOP in test automation, we are aiming to accomplish the following design goals:

- Rapid development evolution cycle
- More white-box testing for maintainability and runtime diagnostics
- Minimum performance impact
- No service interruption
- No security impact
- No interference with app development
- Not introducing a huge infrastructural change

2.2. AOP based White-box testing

To accomplish the above goals, we first take a look at what AOP can help in developing white-box test automation:

Traditional black box testing does not need to know the internal states of the target application. But today, lots of complex bugs can only be understood and fixed by examining an app's internal states, such as resource usage, logged on users, connection status, and other domain-specific states. Those bugs are "hot" bugs in that they can best be diagnosed during live sessions. It would help diagnostics of live bugs by peeking into the internal state.

AOP can help us obtain above states by identifying point-cuts and intercepting function calls at various joint points and gathering the information.

2.3. Identifying Cross-cutting Concerns

There is no way to foresee all the joint points that an interceptor is needed for, nor the kind of interceptor is needed. Therefore the first design and architectural decision is to identify these elements:

1) Address Cross-cutting test concerns
- Verification of an internal state at a joint point (for both pre and post release test)
- Logging of the internal state for later analysis
- Injecting error condition at an internal joint point (more applicable for pre-release test)

2) Connect test automation with app logic by interceptors
3) Develop the test automation against interceptors separately along with app development

2.4. Code-Interception Techniques

There are built-in supports for AOP interceptors from various programming languages. For languages that do not provide interception support, there are design patterns such as delegates or decorators as "poor-men's AOP" as well.

- Attributes
  Attributes are the most popular ways to "decorate" a function or class. Together with reflection, they are widely used ways to intercept a function call or type construction.
  Pros: easy to use and proven to be an effective way to intercept any functions calls or class constructions.
  Cons: the attributes have to be fixed at design time and there is no way to add/remove them at runtime.

- Filter infrastructure
  Filter infrastructure provides a systematical way to "filter" a function call before or after a function is called. A good example of the filter infrastructure is the ASP.NET MVC filters, which provides an infrastructure to intercept any "controller actions" at different processing stages. This is a very powerful technique for modifying a page output or monitoring certain server status before the page is rendered. It can be combined with attribute to turn on/off the interception for individual controller actions.

- Source code re-write or injecting new code
  This is the ultimate way to intercept a call. This was impossible to do for traditional statically typed languages such as C/C++. With the popularity if dynamical typed programming languages such JavaScript (NODE JS as its server side version) and C# (still statically typed with support for dynamics and runtime code ejection).
  Pros: this technology goes beyond AOP and brings programming to an unseen new level.
  Cons: lacks support from statically typed programming languages

- Decorator design pattern
  Statically typed programming languages use this design pattern to achieve AOP interception. Obviously, this technique requires lots of coding and cannot support runtime code injection and requires recompilation for adding/removing decorators.

- Runtime configuration
  Using runtime configuration, we can dynamically turn on/off a piece of code inside the target application, thus achieves the enabling of test behavior without affecting app running. The drawback of this approach is obvious in that we have to add code at the point-cuts.

2.5. More on Dynamic code generation

Dynamical programming gives us greater support for injecting dynamical code at runtime. For example, in JavaScript it is pretty easy to "re-write" an existing function and create an entirely new function based on the function:

```
function insertCode(func, replacer, pattern) {
        var newFunc = func.toString().replace(pattern, replacer);
        eval(newFunc);
}
```

JIT compiled code allows more dynamic code generation as well.

We could potentially generate an entirely new test automation application by altering the target app code and inserting test code at different joint points. AOP.JS is such a tool.

This kind of tool serves as a base idea for future system which can dynamically "re-write" the target system into a new system with full-fledged white-box testing automation capability.

Although this approached sounds like a far-fetched idea and not exactly aspect-oriented, it is inspired by AOP and not too hard to achieve under dynamical programming systems such as JavaScript or .NET platform. The potential of this technique sees many cutting edge usages and one of them is a completely innovative test automation framework.

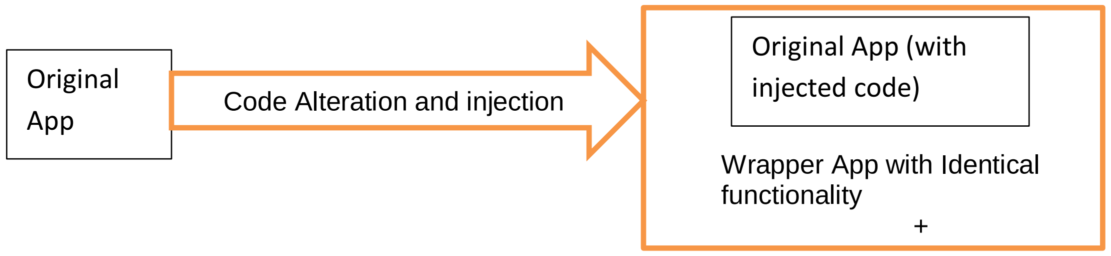
Original App
Code Alteration and injection
Original App (with injected code)
Wrapper App with Identical functionality
+

Fig. 3: illustration for a code generation system, which is not only an AOP interceptor but a turn-key solution for dynamically generates a new application based on an existing application – a test automation framework for any target application in a runtime environment that supports dynamic code injection and source code generation.

3. CASE STUDY

3.1. An ASP.NET MVC 4 Web Application

Microsoft ASP.NET MVC provides great AOP supports via attributes and application filters. This was one of the most important reasons for us to choose it as our web development platform.

Our web site is a political polling application, supporting localization, role-based security, and location-based advertising. We started using an attribute-based filter to test certain aspect of test that would be hard to automate. For example, we need to test location based resources such as style and images, as well as localized content based on user roles and user locations. We found that it is extremely useful to use a filter inside the web server so we can execute a test function when a user with “test role” submitted a request. It has since been developed into a full-fledged test automation infrastructure, which can be turned on/off in live site and conduct essential test in areas of security, localization, and content.

The following diagram shows the idea and infrastructure:

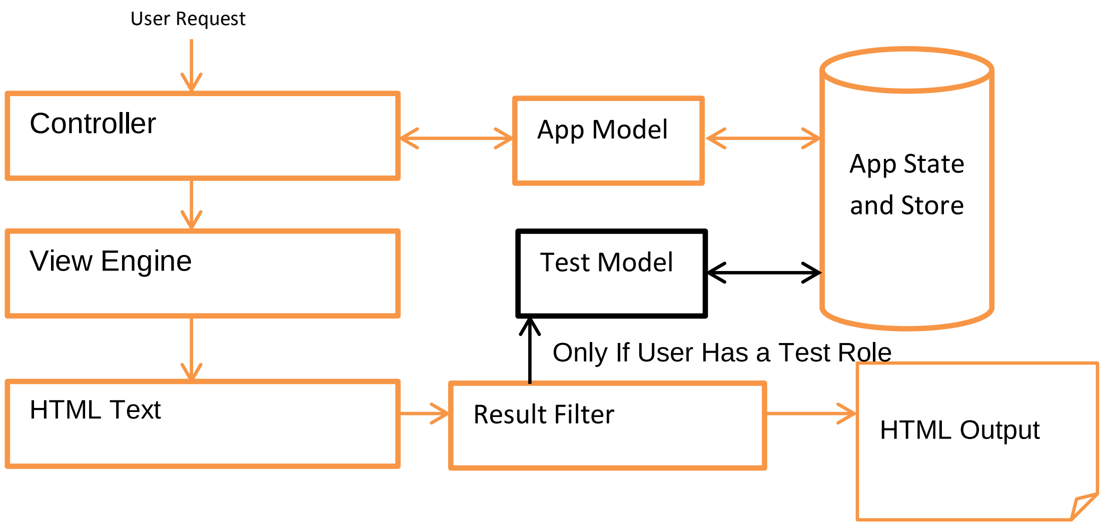

Fig. 4: an actual web application using AOP as test infrastructure. Test model can be turned on/off at live site and invoked via a global request filter.

3.2. Advantage

1. Test automation does not interfere with app development as long as the “result filter” infrastructure is considered and decided at the design time.
2. Test automation works with live system and support application operation / maintenance via test role (using ASP.NET role based security). This is also secure since the “test role” is approved and provisioned only by admin through managed deployment process and will not be exposed to end users ever.
3. Can do white box test through-out the application life cycle.

3.3. Issues

1. Impact of performance
   This should be minimal since the test model will not be called unless the request coming from a test role.
2. Security
   It should be well documented and the deployment is well managed so that no end user will ever be granted test role.
3. Microsoft ASP.Net MVC platform centric
   We are biased since we use ASP.Net MVC, which has first class AOP support. However, its idea and philosophy shall apply to other platform that supports some sort of role based security and server side filters.

3.4. Code Snippet

Create a global filter for test-only

```
 [AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
 public sealed class TestAuotmationAttribute : ActionFilterAttribute
{
        /* Some state info is maintained here for HTML output capture
           (omitted)
       */

        // called when a request is processed by controller
    public override void OnActionExecuting(ActionExecutingContext filterContext)
    {
      // Do nothing unless used as test automation by testers
       if (filterContext.HttpContext.User.IsInRole("testAutomation") == false)
       {
           base.OnActionExecuting(filterContext);
           return;
       }
       /*
       Details for capturing output omitted…
       */
    }

    // called prior to HTML rendering. It is an ideal place for
    // capturing HTML output and context for verification and analysis
    public override void OnResultExecuted(ResultExecutedContext filterContext)
    {
       if (filterContext.HttpContext.User.IsInRole("admin") == false)
       {
           base.OnResultExecuted(filterContext);
           return;
       }
```

```
        /*
        Details for capturing output omitted…
        */
           // Test Automation method, passing cached app states, HTTP context, and
           // generated HTML output
           DoTest(CachedState, filterContext.RequestContext.HttpContext, htmlOutput);
    }
  }
```

Register a global result filter at App Start-up

```
        public class FilterConfig
        {
          public static void RegisterGlobalFilters(GlobalFilterCollection filters)
         {
           filters.Add(new HandleErrorAttribute());
           filters.Add(new TestAuotmationAttribute ());
         }
        }
```

3.5. Conclusion

As you can see from the above example, we have a simple and clean solution for white-box test automation infrastructure with the AOP supports from ASP.NET MVC web platform.

We have created a test automation engine that can be used throughout product development cycle. In particular, it can be used after the product has been published with ***negligible*** impact on the live site.

However, I want to remind you that the support for AOP is not without issues. In the above example, there is still significant expertise needed to design and maintain the solution. It is not entirely dynamic either.

We expect that with more advanced AOP support on the way, we will see more advanced test automation solutions taking advantage of AOP methodologies.

REFERENCES

[1] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.M. and Irwin J., Aspect-oriented Programming. http://cseweb.ucsd.edu/~wgg/CSE218/aop-ecoop97.pdf

[2] Rajan H. AND Sullivan K., Generalizing AOP for Aspect-Oriented Testing

[3] Misra A., Mehra R.: Novel Approach to Automated Test Data Generation for AOP. International Journal of Information and Education Technology, Vol. 1, No. 2, June 2011