



Dynamic Hook Points

Eli Acherkan^{1,2}, Atzmon Hen-Tov¹, Lior Schachter²,

David H. Lorenz², Rebecca Wirfs-Brock³, Joseph W. Yoder⁴

¹Pontis Ltd., Glil Yam 46905, Israel

²The Open University of Israel, Raanana 43107, Israel

³Wirfs-Brock Associates

⁴The Refactory, Inc.

eliac@pontis.com, atzmon@pontis.com,
liorsav@gmail.com, lorenz@openu.ac.il,
rebecca@wirfs-brock.com, joe@refactory.com

***Abstract.** When building dynamic systems, it is often the case that new behavior is needed which is not supported by the core architecture. One way to vary the behavior quickly is to provide well-defined variation points (hook-points) at different places in the systems and have a means to dynamically lookup and invoke new behavior at runtime when desired.*

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.2 [Design Tools and Techniques]: Object-oriented design methods; D.2.11 [Software Architectures]: Patterns

General Terms

Design

Keywords

Adaptive Object-Models, Dynamic systems, Reflection, Variation points, Hook points

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission. Preliminary versions of these papers were presented in a writers' workshop at the 2nd Asian Conference on Pattern Languages of Programs (AsianPLoP). AsianPLoP'2011, March 17-18, Tokyo, Japan. Copyright 2011 is held by the author(s). ACM 978-1-XXXX-XXXX-X.

Introduction

It is often necessary to adapt the behavior of an Adaptive Object-Model (AOM) system [YBJ01; YJ02] in a way not supported by the core AOM architecture. One way to change the behavior quickly and without a full-blown software delivery is to provide well-defined hook-points at different places in the running systems where custom behavior can be defined.

Natural candidate sites for placing hook-points in an AOM are Entity-Type definitions. An AOM engineer can extend the system behavior by defining a new Entity-Type and implementing the new behavior using the Entity-Type's hooks, in a manner similar to using template-methods [GHJ+95] in classic Object-Oriented inheritance.

It is highly desirable that, when possible, the custom behavior can be defined by a non-programmer (an AOM engineer, who isn't necessarily a programmer). This can be achieved by providing the AOM engineer with the ability to define Business Rules [ref TBD]; that is, specify actions to be taken when certain conditions are met.

However, sometimes the use-case requires complex custom behavior, possibly involving technical capabilities, such as interacting with external systems or data repositories. In such cases, writing code cannot be avoided. In other cases, the extra-functional requirements of a particular use-case demand highly optimized performance, which can only be satisfied by imperative code (e.g., performance-tuned Java code).

Figure 1 shows several implementation alternatives in terms of complexity and expressiveness.

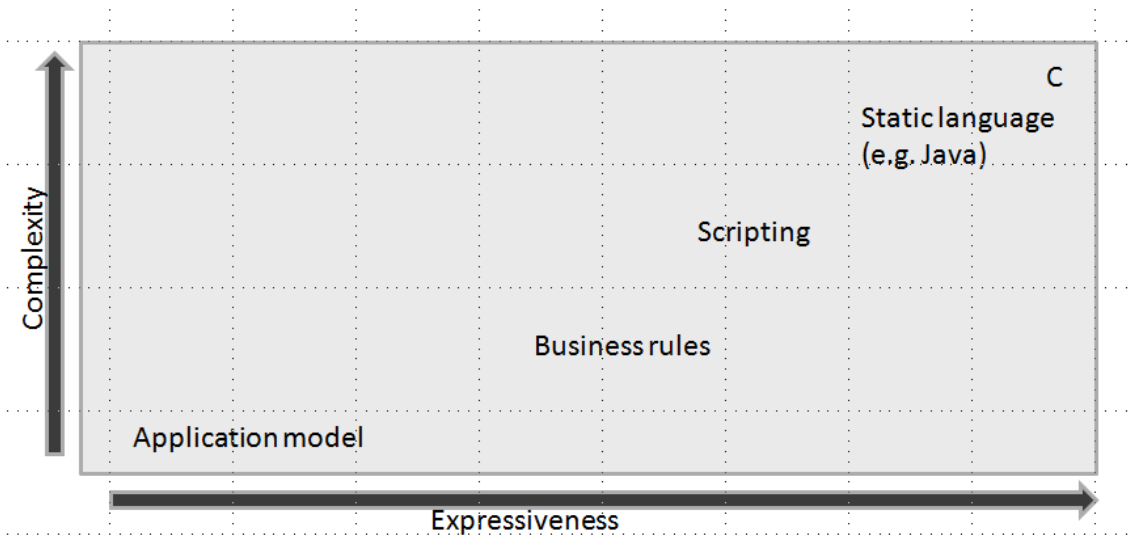


Figure 1 - Expressiveness versus complexity

The pattern presented in this paper is primarily intended for those that are building dynamic or flexible systems and need ways to incorporate variation points in the architecture where new behavior can easily be added

Dynamic Hook Point

Context

The system is changing quickly in a somewhat predictable but not restricted way. There are known spots in the code that need to support a lot of variation. These variations are not defined well enough to be known in advance. In fact some can come from different business requirements as we evolve and add new customers.

Problem

How can you allow a system that is changing in specific locations to adapt to unpredicted behaviors without changing the AOM core? These changes are in well-known places, but the changes are not well-known in advance.

Forces

- **Polymorphism:** The rules for creating an entity may vary according to its type or according to rules that apply to its data.
- **Customizability:** There are many customers using the product, each customer may modify its own AOM model.
- **Flexibility:** The behavior required in a variation point cannot be always known beforehand. The more flexible and unrestricted we allow a variation point to be, the more chances it will support new unpredictable requirements.
- **Adaptability:** The AOM model changes over time. The variations are required to withstand model changes.
- **Extensibility:** Over time, new locations in the system may be recognized as apt to changes. The development effort required to introduce a variation point in a new location should be taken into account.
- **Rapid Change:** The system may be required to adapt to changes very quickly. Some changes to the business rules are important to make available as soon as possible. If certain business requirements are not made available quickly, then it is possible that the relevance of the change will be lost and the time to market will be missed.
- **Manageability:** The varying behavior should be subject to the same manageability concerns as the rest of the system, such as logging, performance tracing, validation (e.g., type safety), and security auditing.
- **Ease of use:** The implementation of a new behavior isn't necessarily done by the same team that created the variation point (e.g., a development team creates variation points and a customer team implements them). A good solution would allow the variations to be made in an easy and simple manner, making the customer team's work more efficient and less error-prone.

- **Scoping:** The variations need a well-defined scope, which controls what they can and can't do, and which parts of the system are accessible to them and can be affected by them. The definition of a scope can serve as a communication tool between teams, and moreover can be enforced by the implementation. (Note that there is trade-off between scoping and flexibility.)
- **Reuse:** The same behavior may be required in several variation points. A solution that allows reuse is preferable to a solution that forces duplication of behavior.
- **Complexity:** A solution that allows for complex scripts, e.g. with inheritance, reuse between scripts, etc. will increase the risk of complexity explosion. Measures like limiting the size of the script should be considered.
- **Testability:** The solution should support testing the hook point implementation before launching it to production.

Trade-offs

- **Hook point implementation:** There are three possible implementation strategies for a hook-point (that is, techniques for implementing the customized behavior).
 - **Script (dynamic language, e.g., Java Script):** Expressive, has access to the entire Java model, dangerous when behavior becomes complex, has performance overheads, hard to read by non-programmers.
 - **Code (imperative language, e.g., Java):** Scalable to complex behavior, performance can be optimized, can't be read by non-programmers.
 - **Rules (e.g., list of condition/action pairs):** (can this be related to the rule-object pattern?). Rules are more restrictive in their expressiveness. Rules can be read by all stakeholders and can be authored by non-programmers.

It is hard to predict which implementation method is adequate for each hook-point as the use cases are unpredictable. Different use-cases for the same hook point have different trade-offs.

Example

Consider a system that calculates the prices of various products and services. The prices can include a discount, depending on global events (such as an end-of-season sale) and per-customer properties (such as membership discounts etc.).

The component responsible for discount calculation is abstracted by the `IDiscounter` interface. This interface contains a single method, `modifyPrice`, which receives a customer object and the original price, and returns the price after a possible modification. `IDiscounter` has several implementations (e.g. `FixedPercentDiscounter`).

The requirements for types of discounts change frequently. We're looking for a solution that would perform the generic discount calculation logic, and afterwards allow the AOM operator to define additional discount calculation. The additional calculation would receive the customer object, and the price as it was after the generic calculation step.

Solution

Define hook-points at well-defined places in the business logic where new behavior can be dynamically invoked. Figure 2 shows the main classes involved in the look-up and invocation of dynamic hooks.

The Dynamic Hooks framework supports various types of hooks such as scripting hooks, Java hooks and Rules (see Rule pattern TBD).

Each hook point is defined by a contract, such as an interface. Each interface that can be implemented by a hook point is registered with the Dynamic Hooks framework. The same interface can have several implementations, not necessarily all of them dynamic.

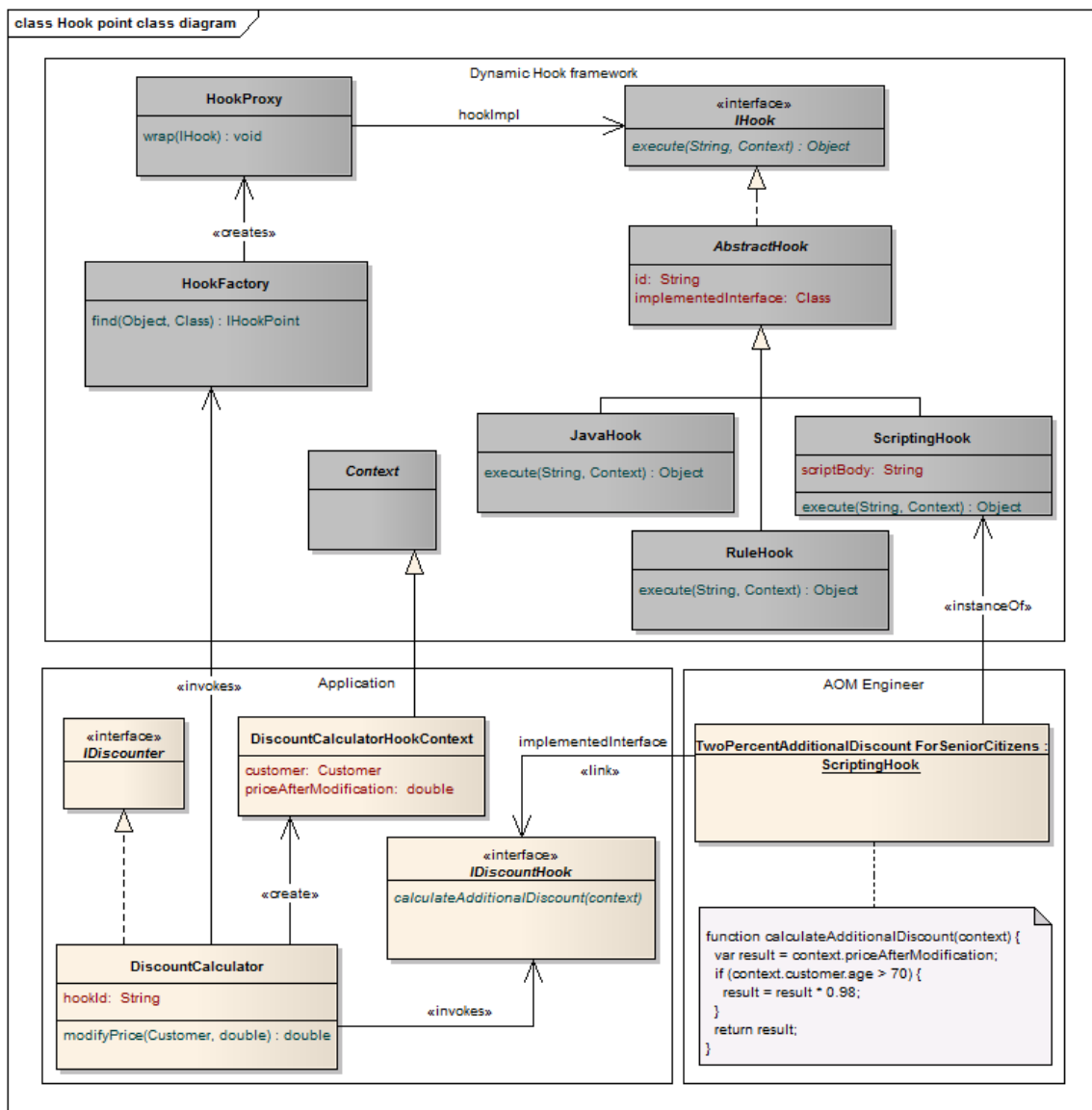


Figure 2 - Hook point class diagram

The Dynamic Hooks framework makes it easier to define new hook points. To define a new hook point (see DiscountCalculator in Figure 2):

1. Define the runtime context of the hook. The runtime context is available to the hook at runtime. It is defined as subclass of the framework's Context class.

- Define a property for each entity or data element you want to make available to the hook as well.
2. Define the interface the hook should implement.
 3. Keep track of the actual hook point implementation that is glued to each instance of your class (this is shown as `hookIdproperty` of `DiscountCalculator` in Figure 2.)
 4. In the right place of the flow (just after the price was modified in the example) call the hook allowing it to make its effect. This is done by:
 - a. Invoking the hook factory to find the hook point implementation.
 - b. Instantiating the context (`DiscountCalculatorHookContext` in the example) and populating it with contextual data.
 - c. Casting the result to the required interface (`IDiscountHook` in the example).
 - d. Invoking the hook and making the effect (returning the returned modified price in the example).

Description of how the hook point implementation is made available to the AOM engineer in the GUI was left out of this paper.

We also left out the details of Java hooks implementation and Rules implementation.

The AOM Engineer can choose the implementation technology for each use case making all the tradeoffs analysis on a per case basis. In the example (Figure 2) the chosen implementation is JavaScript, which provides an additional 2 percent discount for senior citizens. The script uses the context object to check if the customer is senior and modifies the price (which is also available on the context).

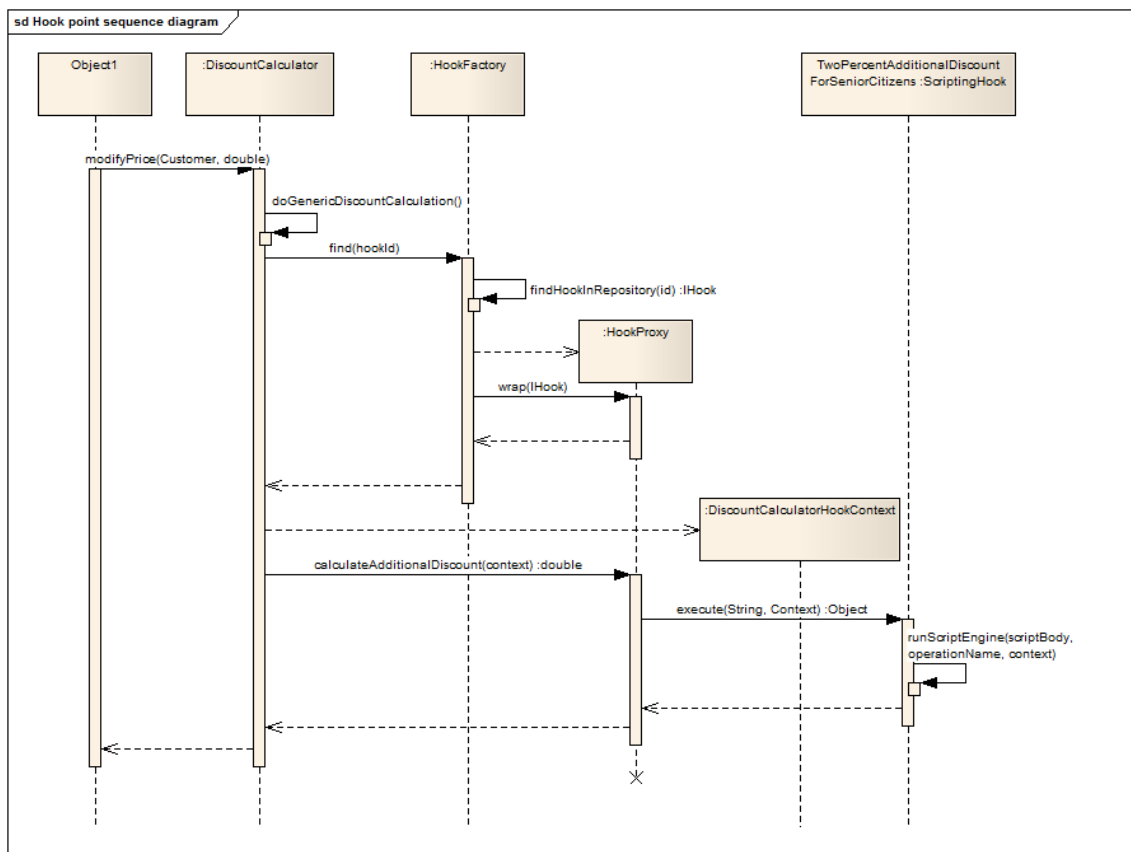


Figure 3 - Hook point sequence diagram

Client code requests from the HookFactory to supply an instance of the hook, by passing the hook's ID. The factory can have several lookup and instantiation strategies for different types of hook points (e.g. precompiled hooks, scripting language hooks, etc.). It returns an object that implements the required interface, possibly through a proxy object that hides away invocation details (such as when invoking a scripting language from a different host language).

Alternatively, the wiring between a client and a hook implementation can be done with Dependency Injection or similar mechanisms.

Since hook points provide high level of expressiveness and flexibility they should be defined within a clear scope and have a controlled context and API. This can be achieved also by using transaction-like isolation for the hook invocation.

Part1: Lookup; Part2: Invocation; Part3: Sequencing (control language) Part4: Return

Guarding clauses vs Name Value pair lookups. You can put a guarding clause at the top of the invocation to decide if this hook point is for this case or not...or, you can have a way of looking up the hook point for this case through a name value pair lookup in a db (e.g., services in OSGi).

List of rules to apply and the order to apply them via config data usually in a database.

Different from Strategy in that the Strategies need to be known ahead of time. These hook points can be implemented like a dynamic strategy that can evolve to a rule language.

Implementation

The following Java code illustrates how a hook point is implemented (some methods omitted for brevity):

```
public class Driver {

    public static void main(String[] args) {
        IHook hook = new ScriptingHook("TwoPercentDiscount", // ID
            "function calculateAdditionalDiscount(context) {" + // Script body
            "    var result = context.priceAfterModification;" +
            "    if (context.customer.age > 70) {" +
            "        result = result * 0.98;" +
            "    }" +
            "    return result;" +
            "}", IDiscountHook.class // Implemented interface
        );
        // Register the newly created hook with HookFactory

        DiscountCalculator calculator = new DiscountCalculator(hook.getId());
        Customer customer = retrieveCustomer();
        double result = calculator.modifyPrice(customer, 8);
        System.out.println(result);
    }
}

public class DiscountCalculator {

    public double modifyPrice(Customer customer, double originalPrice) {
        double modifiedPrice = doGenericDiscountCalculation(customer, originalPrice);

        IDiscountHook hook = (IDiscountHook) HookFactory.getInstance().find(this.hookId);
        DiscountCalculatorHookContext context = createAndPopulateContext(customer,
            modifiedPrice);
        return hook.calculateAdditionalDiscount(context);
    }

    private DiscountCalculatorHookContext createAndPopulateContext(Customer customer,
        double originalPrice) {
        DiscountCalculatorHookContext context = new DiscountCalculatorHookContext();
        context.priceAfterModification = originalPrice;
        context.customer = customer;
        return context;
    }
}

public class DiscountCalculatorHookContext extends Context {
    public Customer customer;
    public double priceAfterModification;
}

public interface IDiscountHook {
    public double calculateAdditionalDiscount(DiscountCalculatorHookContext context);
}

public class HookFactory {

    public Object find(String hookId) {
        IHook dynamicHookObject = findHookInRepository(hookId);
        HookProxy proxy = new HookProxy();
        proxy.wrap(dynamicHookObject);
        return proxy.createProxy();
    }
}

public class HookProxy implements java.lang.reflect.InvocationHandler {

    private IHook hookObject;

    public void wrap(IHook hookObject) {
        this.hookObject = hookObject;
    }
}
```

```

public Object createProxy() {
    return Proxy.newProxyInstance(getClass().getClassLoader(), new Class[] {
hookObject.getImplementedInterface() }, this);
}

@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    return hookObject.execute(method.getName(), (Context) args[0]);
}
}

public class ScriptingHook extends AbstractHook {

    @Override
    public Object execute(String operationName, Context context) {
        // Exception handling omitted
        javax.script.ScriptEngine engine =
new javax.script.ScriptEngineManager().getEngineByName("JavaScript");
        engine.eval(this.scriptBody);
        return ((Invocable) engine).invokeFunction(operationName, context);
    }
}

```

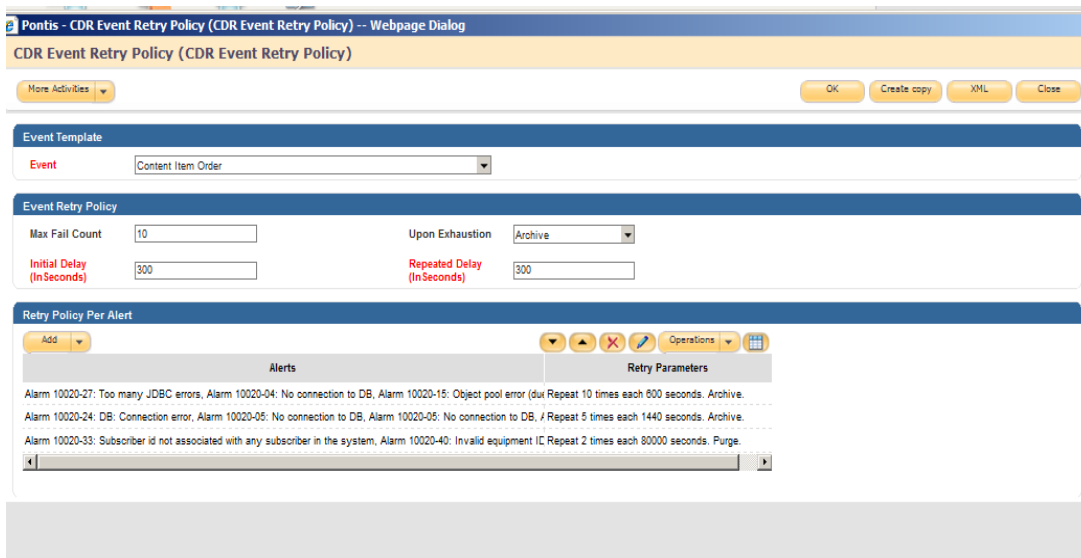


Figure 4 - Example of rule

Implementation notes:

- There are several techniques for invoking a scripting language from a separate host language. The Java Development Kit (starting from version 6) includes a scripting engine with built-in support for JavaScript.
- Java hooks can be implemented as:
 - o Regular jars.
 - o OSGi bundles – this helps manage their deployment dynamically with no downtime.
 - o On the fly Java compilation (like JSP) – This allows on site Java hooks. You can use the Java Compiler API or a number of open-source tools [ref TBD] to compile java JIT.

Rules – Rules can be implemented with expression-languages such as Spring EL (ref TBD), JSF EL (ref TBD) and XPath.

Variations

Dependency Injection

The pattern can be simplified when the application is developed using a Dependency Injection framework (e.g., Spring framework).

In the implementation above, the Hook Factory class is used to instantiate the hook objects and to wrap them in a proxy. With Dependency Injection, this class can be removed, and the hook objects can be instantiated by the Dependency Injection framework instead.

This way, the variation point (DiscountCalculator in our example) doesn't have to perform lookup via an identifier; instead, it defines a data member, and the Dependency Injection container assigns value to it.

Consequences

- ✓ **High Availability:** Changes to the architecture can be made without having to completely rebuild a system before releasing it into production.
- ✓ **More Flexible:** The AOM engineer can choose, on a per case basis, whether to use Scripting, Java code or Rules (see Figure 1). E.g. when performance is critical, a Java implementation will be bound to the hook, and where maintainability is the major concern the AOM engineer will opt for rules. When the rules are not expressive enough Java-Script can be used.
- ✓ **Model continuum:** The pattern can be applied on an interface for which there are several implementations in the application (e.g. IDiscounter in our example). This unites the hook with the application model providing the AOM engineer with the entire set of options shown in Figure 1.
- ✓ **Easy to add hooks:** When a new hook is needed in the system, the core team can add it easily.
- ✗ **Higher Complexity:** The dynamic hook code is more complex as hook methods and evolution code will become part of the system. There is now another level of indirection for debugging and testing.
- ✗ **Testability** Testing of the system can be more complex because you not only have to provide core tests for the core AOM architecture, but it additional one off tests much be provided for all the hookpoints to make sure they work properly in conjunction with the core AOM system.
- ✗ **Performance overhead:** using scripting language can impact performance. Using Dependency Injection or caching mechanisms can eliminate the performance overhead related to lookup and binding.
- ✗ **Interface Dependency:** The dynamic hooks must implement interfaces defined by the application layer. This increases the number of dependencies between components, and puts the dynamic hooks at risk when the interfaces are changed. This is particularly important for hooks that don't support type safety [REF safe scripts TBD]. The Evolution Resilient Script pattern [REF upgrade TBD] addresses this issue and improves type safety.

Related Patterns

This pattern is not dependant on other AOM patterns. It can be applied on any Object Oriented application to add dynamic behavior definition.

Strategy is similar in the fact that a hookpoint is like a dynamic strategy that can be plugged in at runtime. Strategies and hookpoints both have a common API. The main difference is that a Strategy is a class hierarchy of pluggable algorithms that are defined within the code while a hookpoint can be more than just a class hierarchy and also can be dynamically hooked in at runtime with new code.

Evolution Resilient Script (Safe Script) is related...we need to say how

Known Uses

Pontis Ltd. (www.pontis.com) is a provider of Online Marketing solutions for Communication Service Providers. Pontis' Marketing Delivery Platform allows for on-site customization and model evolution by non-programmers. The system is developed using ModelTalk [HLPS09] based on AOM patterns. Pontis' system is deployed in over 20 customer sites including Tier I Telcos. A typical customer system handles tens of millions of transactions a day exhibiting Telco-Grade performance and robustness.

Invoicing and Import system done for one of the Refactory clients.

An AOM system developed for the Illinois Department of Public Health used dynamic hook points.

Acknowledgements

We would like to thank our shepherd Kiran Kumar Reddy for his valuable comments and feedback during the AsianPLoP 2011 Shepherding process.

References

- [AOM] Adaptive Object-Models. <http://www.adaptiveobjectmodel.com>
- [And98] Anderson, F. *A Collection of History Patterns*. Proceedings of the 6th Pattern Language of Programs Conference (PLoP 1998), Monticello, Illinois, USA, 1998.
- [Ars00] Arsanjani, A. *Rule Object Pattern Language*. Proceedings of the 8th Pattern Language of Programs Conference (PLoP 2000). Technical Report WUCS-00-29, Dept. of Computer Science, Washington University Department of Computer Science. (2000).
- [BMR+96] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern Oriented Software Architecture, Volume 1: A System of Patterns*. Wiley & Sons. 1996.
- [BR98] Bäumer, D. , D. Riehle. *Product Trader*. Pattern Languages of Program Design 3. Edited by Robert Martin, Dirk Riehle, and Frank Buschmann. Addison-Wesley, 1998.
- [CW86] Caudill, P., Wirfs-Brock A. "A Third Generation Smalltalk-80 Implementation.", p. 119-130, OOPSLA '86 Conference Proceedings, Portland Oregon, September 29-October 2, 1986.
- [FCW08] Ferreira, H. S., Correia, F. F., and Welicki, L. 2008. *Patterns for data and metadata evolution in adaptive object-models*. Proceedings of the 15th Conference on Pattern Languages of Programs (Nashville, Tennessee, October 18 - 20, 2008). PLoP '08, vol. 477. ACM, New York, NY, 1-9.
- [Fow97] Fowler, M. *Analysis Patterns: Reusable Object Models*. Addison-Wesley. 1997.
- [Fow02] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley. 2002.
- [FPR01] Fontura, M., Pree, W., Rump, B. *The UML Profile for Framework Architectures*. Addison-Wesley. 2001.
- [FY98] Foote B, J. Yoder. *Metadata and Active Object Models*. Proceedings of Plop98. Technical Report #wucs-98-25, Dept. of Computer Science, Washington University Department of Computer Science, October 1998.
- [GHJ+95] Gamma, E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.
- [HNS+10] Atzmon Hen-Tov, Lena Nikolaev, Lior Schachter, Joseph W. Yoder, Rebecca Wirfs-Brock. *Adaptive Object-Model Evolution Patterns*, SugarLoafPLoP 2010. To appear.
- [HLPS09] Atzmon Hen-Tov, David H. Lorenz, Assaf Pinhasi, Lior Schachter: *ModelTalk: When Everything Is a Domain-Specific Language*, IEEE Software, vol. 26, no. 4, pp. 39-46, July/Aug. 2009.
- [Jon99] Jones, S. *A Framework Recipe*. Building Application Frameworks: Object-Oriented Foundations of Framework Design. Edited by Fayed,

- M., Johnson, R., Schmidt, D. John Wiley & Sons. 1999.
- [JW98] Johnson, R., R. Wolf. *Type Object*. Pattern Languages of Program Design 3. Addison-Wesley, 1998.
- [KJ04] Kircher, M.; P. Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.
- [KSS05] Krishna, A., D.C. Schmidt, M. Stal. *Context Object: A Design Pattern for Efficient Middleware Request Processing*. 13th Pattern Language of Programs Conference (PLoP 2005), Monticello, Illinois, USA, 2005.
- [Mar02] Martin, R. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.
- [RFBO01] Riehle, D., Fraleigh S., Bucka-Lassen D., Omorogbe N. *The Architecture of a UML Virtual Machine*. Proceedings of the 2001 Conference on Object-Oriented Program Systems, Languages and Applications (OOPSLA '01), October 2001.
- [RTJ05] Riehle D., M. Tilman, and R. Johnson. *Dynamic Object Model*. *Pattern Languages of Program Design 5*. Edited by Dragos Manolescu, Markus Völter, James Noble. Reading, MA: Addison-Wesley, 2005.
- [RY01] Revault, N, J. Yoder. *Adaptive Object-Models and Metamodeling Techniques Workshop Results*. Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001). Budapest, Hungary. 2001.
- [WYWJ07] Welicki, L.; J. Yoder; R. Wirfs-Brock; R. Johnson. *Towards a Pattern Language for Adaptive Object-Models*. Companion of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2007), Montreal, Canada, 2007.
- [WYW09] Welicki, L.; J. Yoder; R. Wirfs-Brock. *Adaptive Object-Model Builder*. 16th Pattern Language of Programs Conference (PLoP 2009), Chicago, Illinois, USA, 2009.
- [WYW07] Welicki, L, J. Yoder, R. Wirfs-Brock. *Rendering Patterns for Adaptive Object Models*. 14th Pattern Language of Programs Conference (PLoP 2007), Monticello, Illinois, USA, 2007
- [YBJ01] Yoder, J.; F. Balaguer; R. Johnson. *Architecture and Design of Adaptive Object-Models*. Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA 2001), Tampa, Florida, USA, 2001.
- [YJ02] Yoder, J.; R. Johnson. *The Adaptive Object-Model Architectural Style*. IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance (WICSA 2002), Montréal, Québec, Canada, 2002.
- [YR00] Yoder, J.; R. Razavi. *Metadata and Adaptive Object-Models*. ECOOP Workshops (ECOOP 2000), Cannes, France, 2000.