



Apex Design Patterns

UPDATED May 2013

Apex allows you to build just about any custom solution on the Force.com platform. But what are the common design patterns and associated best practices for Apex development, and how can you leverage these patterns and best practices to avoid reinventing the wheel?

This article describes how to leverage common design patterns to optimize your code and ensure reusability, maintainability and performance. *This is a wiki version of the Dreamforce 2012 presentation 'Apex Design Patterns (<http://www.youtube.com/watch?v=J372XmYds-A>)'.*

Contents

- [1 Common Design Patterns](#)
- [2 Singleton](#)
 - [2.1 Problem](#)
 - [2.2 Unified Modeling Language](#)
 - [2.3 Implementation](#)
 - [2.4 Conclusion](#)
- [3 Strategy](#)
 - [3.1 Problem](#)
 - [3.2 Unified Modeling Language](#)
 - [3.3 Implementation](#)
 - [3.4 Conclusion](#)
- [4 Decorator](#)
 - [4.1 Problem](#)
 - [4.2 Unified Modeling Language](#)
 - [4.3 Implementation](#)
 - [4.4 Conclusion](#)
- [5 Facade](#)
 - [5.1 Problem](#)
 - [5.2 Unified Modeling Language](#)
 - [5.3 Implementation](#)
 - [5.4 Conclusion](#)
- [6 Composite](#)
 - [6.1 Problem](#)
 - [6.2 Unified Modeling Language](#)
 - [6.3 Implementation](#)
 - [6.3.1 Examples/Usage](#)
 - [6.4 Conclusion](#)
- [7 Bulk State Transition](#)
 - [7.1 Problem](#)
 - [7.2 Unified Modeling Language](#)
 - [7.3 Implementation](#)
 - [7.4 Conclusion](#)
- [8 About the Authors](#)

Common Design Patterns

The following are a list of design patterns, some of which are standard object-oriented patterns in a Force.com context, and some of which are specific Force.com patterns.

- **Singleton** - minimizing object instantiation for improved performance and to mitigate impact of governor limits
- **Strategy** - defining a family of algorithms, encapsulating each one and making them interchangeable and selectable at runtime
- **Decorator** - extending the functionality of an sObject in Apex
- **Facade** - simplifying the execution of classes with complex interfaces (e.g. web service callouts)
- **Composite** - treating a group of objects in a similar manner to a single instance of that object
- **Bulk State Transition** - efficiently tracking the change of a field value in a trigger and executing functionality based on this change

Singleton

The Singleton pattern attempts to solve the issue of repeatedly using an object instance, but only wishing to instantiate it once within a single transaction context. Common uses for this pattern include:

- Global variables - whilst Apex does not support global variables across execution contexts, this pattern allows you to create an object instance that will only ever be instantiated once within an execution context.
- Limiting Impact of Governor Limits - certain system objects and methods, such as Apex Describes, are subject to governor limits. The Singleton pattern allows repeated reference to these without breaching governor limits.
- As an implementation to other patterns - other design patterns, such as Facade, are often implemented as Singletons.

However, it's most common use is to create an object instance that's instantiated only once for the lifetime of that execution context.

Problem

Developers often write inefficient code that can cause repeated instantiation of objects. This can result in inefficient, poorly performing code, and potentially the breaching of governor limits. This most commonly occurs in triggers, as they can operate against a set of records.

The following code shows an example of repeated code invocation that can result in a breach of governor limits:

The Trigger

```
1 trigger AccountTrigger on Account (before insert, before update) {
2     for(Account record : Trigger.new){
3         AccountFooRecordType rt = new AccountFooRecordType();
4         ....
5     }
6 }
```

The Class

```
1 public class AccountFooRecordType {
2     public String id {get;private set;}
3     public AccountFooRecordType(){
4         // This could breach the governor limits on describes
5         // if a trigger is executed in bulk
6         id = Account.sObjectType.getDescribe()
```

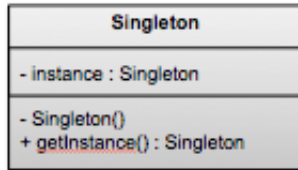
```

7 |         .getRecordTypeInfoByName().get('Foo').getRecordTypeId();
8 |     }
9 | }

```

The trigger will cause a repeated execution of the sObject `getDescribe()` method, resulting in a breach of the total number of describes governor limit if the trigger operates against more than 100 records.

Unified Modeling Language



(</page/File:Singleton.png>)

Implementation

In order to implement a Singleton pattern in Apex, the class must instantiate only a single instance and be globally accessible. It is implemented by:

- Creating a class with a method that creates a new instance of the class if it doesn't already exist
- If it already exists, then simply return a reference to that object

The following code sample demonstrates an implementation of the Singleton pattern for returning a record type describe within a trigger:

The Trigger

```

1 | trigger AccountTrigger on Account (before insert, before update) {
2 |     for(Account record : Trigger.new){
3 |         // Instantiate the record type using the singleton class
4 |         AccountFooRecordType rt = AccountFooRecordType.getInstance();
5 |         ....
6 |     }
7 | }

```

The Singleton Class

```

01 | public class AccountFooRecordType {
02 |     // private static variable referencing the class
03 |     private static AccountFooRecordType instance = null;
04 |     public String id {get;private set;} // the id of the record type
05 |
06 |     // The constructor is private and initializes the id of the record
07 |     type
08 |     private AccountFooRecordType(){
09 |         id = Account.sObjectType.getDescribe()
10 |             .getRecordTypeInfoByName().get('Foo').getRecordTypeId();
11 |     }
12 |     // a static method that returns the instance of the record type
13 |     public static AccountFooRecordType getInstance(){
14 |         // lazy load the record type - only initialize if it doesn't
15 |         already exist
16 |         if(instance == null) instance = new AccountFooRecordType();
17 |         return instance;
18 |     }
19 | }

```

The above code demonstrates the following:

- The getInstance() static method will only instantiate an instance of the class if it doesn't already exist in a **lazy-initialization** manner
- The constructor and the instance variable for the class is private to make sure that it cannot be instantiated outside of the getInstance() method
- The class defines a private, static instance of itself that can only be referenced via the getInstance() static method
- The ID member stores the record ID of the record type and is initialized in the constructor

This allows the trigger to obtain a reference to the record type without breaching governor limits.

The following code sample shows how to use **eager-initialization** so that a new instance is always created when the class is instantiated.

The Singleton Class - Eager Initialization variant

```
01 public class AccountFooRecordType {
02     // a static, final variable that initializes an instance of the
    class
03     // as it's final, it will only be initialized once
04     private static final AccountFooRecordType instance = new
AccountFooRecordType();
05     public String id {get;private set;}
06     private AccountFooRecordType(){
07         id = Account.sObjectType.getDescribe()
08             .getRecordTypeInfoByName().get('Foo').getRecordTypeId();
09     }
10     public static AccountFooRecordType getInstance(){
11         // eager load of the class
12         return instance;
13     }
14 }
```

The instance of the class is instantiated as a final, static variable, which means that only one instance ever exists. This method is typically used if the cost of creating the instance is small.

Conclusion

The Singleton design pattern allows Apex code to repeatedly reference an object instance in an optimal manner, whilst mitigating the impact of governor limits.

Strategy

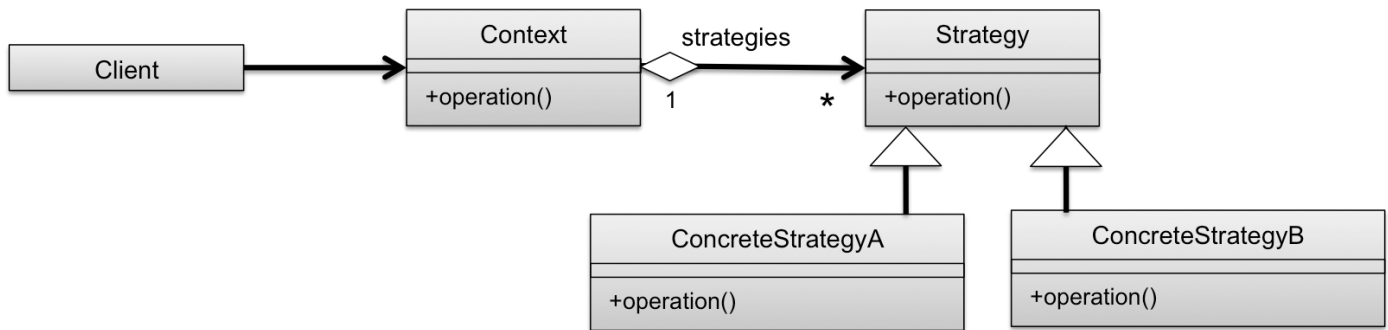
The Strategy pattern (aka the policy pattern) attempts to solve the issue where you need to provide multiple solutions for the same problem so that one can be selected at runtime.

Problem

You need to provide a geographical-based search engine solution where the implementing code can choose the search provider at runtime.

```
1 geocode('moscone center')
2 //=> 37.7845935, -122.3994262
```

Unified Modeling Language



(/page/File:Apex_Design_Patterns_Strategy.png)

Implementation

In order to implement a Strategy pattern in Apex, you need to define a family of algorithms, encapsulate each one, make them interchangeable, and selectable at runtime. It is implemented by:

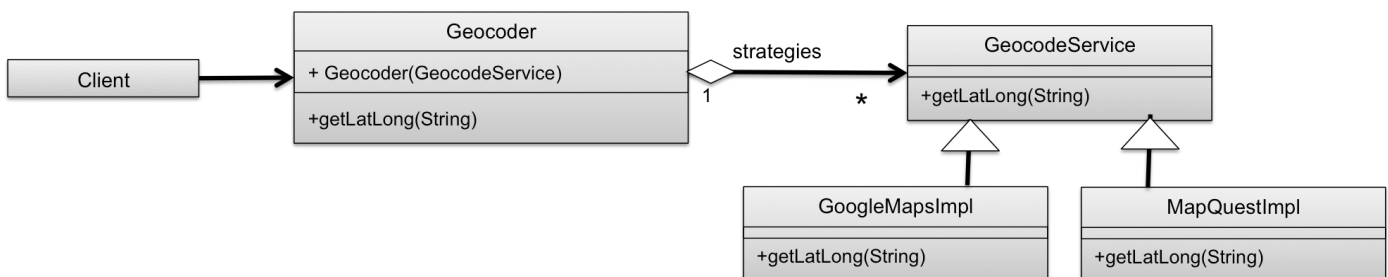
- Creating an interface class (the Strategy) with methods that will be implemented by other classes
- Creating a class for each concrete Strategy that implements the methods defined in the Strategy interface

The following code demonstrates an implementation of the Strategy pattern for the underlying GeocodeService strategy and implementations.

```

01 public interface GeocodeService{
02     List<Double> getLatLong(String address);
03 }
04
05 public class GoogleMapsImpl implements GeocodeService{
06     public List<Double> getLatLong(String address){
07         // Web service callout
08         return new List<Double>{0,0};
09     }
10 }
11
12 public class MapQuestImpl implements GeocodeService{
13     public List<Double> getLatLong(String address){
14         // Web service callout
15         return new List<Double>{1,1};
16     }
17 }
  
```

So the specific Unified Modeling Language (UML) for this implementation is



(/page/File:Apex_Design_Patterns_Strategy_Geocoder_Example.png)

where:

- Context => Geocoder
- operation() => getLatLong()
- Strategy => GeocodeService

- ConcreteStrategyA => GoogleMapsImpl
- ConcreteStrategyB => MapQuestImpl

Now that we have our core strategy interface and implementations, we need to make use of them. There are a couple of ways you might be tempted to do this, however, in order to decouple as much as possible, consider the following:

The Class

```

01 public class Geocoder {
02     public class NameException extends Exception{}
03
04     public static final Map<String,GeocodeService> strategies;
05
06     static{
07         // Retrieve comma delimited list of strategies from a Custom
08         Setting
09         GlobalVariable__c gv =
10         GlobalVariable__c.getInstance('strategies');
11
12         // Populate a List Collection of strategy names e.g.,
13         googleMaps, mapQuest etc
14         List<String> strategyNames = new List<String>();
15         if(gv != null && gv.value__c != null) strategyNames =
16         gv.value__c.split(',');
17
18         // Populate a map of strategy names to concrete implementations
19         // using Type.forName for each strategy string
20         strategies = new Map<String,GeocodeService>();
21         for(String name : strategyNames){
22             try{strategies.put(name, (GeocodeService)Type.forName(name
23 + 'impl').newInstance());}
24             catch(Exception e){continue;} //skip bad name silently
25         }
26     }
27
28     private GeocodeService strategy;
29
30     public Geocoder(String name){
31         if(!strategies.containsKey(name)) throw new
32         NameException(name);
33         strategy = strategies.get(name);
34     }
35
36     public List<Double> getLatLong(String address){
37         return strategy.getLatLong(address);
38     }
39 }

```

Calling googleMaps

```

1 Geocoder geocoder = new Geocoder('googlemaps');
2 System.debug(geocoder.getLatLong('moscone center'));
3 //=> 13:56:36.029 (29225000)|USER_DEBUG|[29]|DEBUG|(0.0, 0.0)

```

Calling MapQuest

```

1 Geocoder geocoder = new Geocoder('MapQuest');

```

```

2 | System.debug(geocoder.getLatLong('moscone center'));
3 | //=> 13:56:36.129 (29225000)|USER_DEBUG|[29]|DEBUG|(0.0, 0.0)

```

As you can see above, the calling code has a choice of implementations and the only change is the string passed to the Strategy interface.

Conclusion

The Strategy design pattern uses aggregation instead of inheritance, allowing better decoupling between the behavior and the class that uses the behavior. This allows the behavior to be changed without breaking the classes that use it, and the classes can switch between behaviors by changing the specific implementation used without requiring any significant code changes.

Decorator

The Decorator pattern attempts to solve the issue where you need temporary fields for processing (typically in Visualforce) but do not need to add these fields to the sObject.

Common uses for this pattern include:

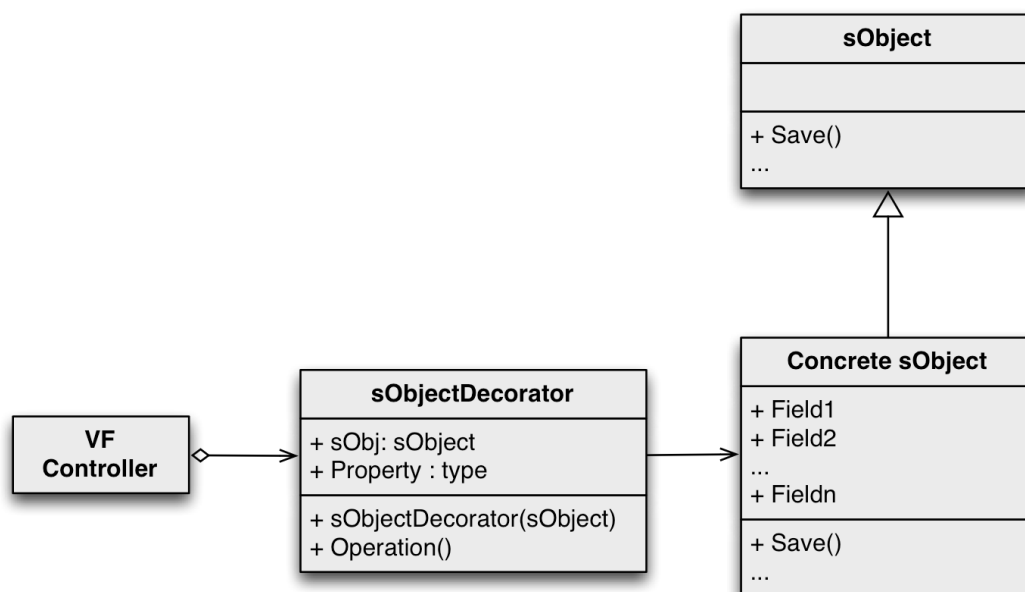
- Selection Checkbox – a list of records that the user selects and applies behavior to; the selection checkbox is not saved
- Calculated fields – a complex read-only value that cannot be easily done in a formula field (e.g. calculation of a check digit)
- Proxy fields – a field, that when updated, converts to a different value on the record (e.g. temperature figures presented to the user in C, but stored as F)

This pattern is primarily for Visualforce use cases.

Problem

You need to obtain or display temporary information on a Visualforce page that is not needed beyond the context of the interaction.

Unified Modeling Language



(/page/File:Apex_Design_Patterns_Decorator.png)

Implementation

In order to implement the Decorator pattern in Apex, we need to be aware that this is not a true OO implementation, but the intent is to add behavior at runtime rather than via inheritance at compile time. It is implemented by:

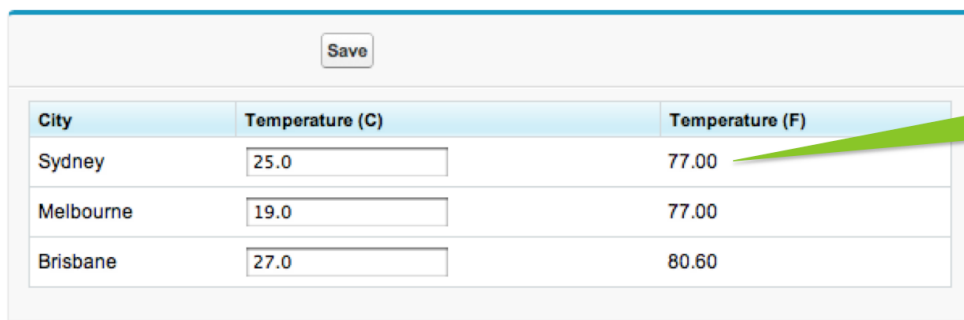
- Understanding the existing sObject superclass – the actual sObject class.
- Understanding the existing concrete sObject – our underlying sObject (e.g. Account, Opportunity) – this class is not extensible in Apex
- Creating an "sObject Decorator" class that wraps the sObject with a pointer to the concrete sObject, with additional operations and properties extending the behavior of the concrete sObject at runtime
- Creating a Visualforce controller/class that acts as a client to the decorator

Note that this is not a “true” implementation of the OO Decorator pattern:

- The decorator class does not implement an interface to sObject (you can’t do it)
- The sObject pointer in the class is public to simplify access to its behaviors (unless you want to recreate all the sObject methods in the class)
- It does have the intent of decorator, which is to add functionality at runtime

In our example scenario, we have a Weather sObject with City__c and TempInFahrenheit__c fields, and our Visualforce requirements are:

- the stored temperature in Fahrenheit should be displayed in Celsius
- when the user enters the temperature in Celsius it is stored as Fahrenheit



City	Temperature (C)	Temperature (F)
Sydney	25.0	77.00
Melbourne	19.0	77.00
Brisbane	27.0	80.60

Bi-directional Display and Calculation

(/page/File:Decorator_Weather_Example.png)

The Code - Decorated sObject Class

```
01 public class DecoratedWeather {
02
03     public Weather__c weather { get; private set; }
04
05     public DecoratedWeather (Weather__c weather) {
06         this.weather = weather;
07     }
08
09     public Decimal tempInCelcius {
10         get {
11             if (weather != null && tempInCelcius == null )
12                 tempInCelcius = new
Temperature().FtoC(weather.TempInFahrenheit__c);
13
14             return tempInCelcius;
15         }
16         set {
17             if (weather != null && value != null )
18                 weather.TempInFahrenheit__c= new Temperature().CtoF(value);
19
20             tempInCelcius = value;
21         }
22     }
```


The above code demonstrates how the decorator class extends or wraps the Weather sObject with new functionality.

- Reference the weather sObject as a public property, but make it a private set. (This is different to the true decorator pattern, which makes this private and delegates behavior instead)
- Constructor – is passed in the weather object
- Property for tempInCelcius
- Getter – converts the temp from F to C and returns it (assuming we have another class to convert temperature)
- Setter – uses the value entered to convert from C to F and a side effect to store into the weather.tempInF. The last line makes sure that the value is stored back on the property.

The Code - Custom Controller

```

01 public class Weather_Controller {
02
03     public List<DecoratedWeather> listOfWeather {
04
05         get {
06             if (listOfWeather == null) {
07                 listOfWeather = new List<DecoratedWeather>();
08
09                 for (Weather__c weather : [select name, temperature__c from
Weather__c]) {
10                     listOfWeather.add(new DecoratedWeather(weather));
11                 }
12             }
13             return listOfWeather;
14         }
15
16         private set;
17     }
18 }

```

The above code demonstrates how the Weather Controller is the client

- Has a property that is a list of DecoratedWeather (our wrapper class)
- Has a getter that lazy initializes the list by selecting from the Weather__c table
- Private setter – only this class can set it

The Code - Visualforce Page

```

01 <apex:page controller="weather_controller">
02     <!-- VF page to render the weather records with Ajax that only
03     rerenders
04     the page on change of the temperature
05     -->
06     <apex:form id="theForm">
07         <apex:pageBlock id="pageBlock">
08             <apex:pageBlockTable value="{!listOfWeather}" var="weather">
09                 <apex:column value="{!weather.weather.name}"/>
10                 <apex:column headerValue="Temperature (C)">
11                     <apex:actionRegion >
12                         <apex:inputText value="{!weather.tempInCelcius}">
13                             <apex:actionSupport event="onchange"
reRender="pageBlock"/>

```

```
14         </apex:inputText>
15     </apex:actionRegion>
16 </apex:column>
17     <apex:column headerValue="Temperature (F)"
18         value="{!weather.weather.Temperature__c}"
19         id="tempInF"/>
20 </apex:pageBlockTable>
21 </apex:pageBlock>
22 </apex:form>
23 </apex:page>
```

The above code demonstrates that the Visualforce page only needs to implement a little AJAX action support to rerender the page if the temperature changes. There is no real client side logic and no other fancy tricks, just a simple rerender. The getter and setter in the decorator class takes care of all the work.

Conclusion

So what we have is an example of how to extend sObject functionality in Apex with behavior at runtime, rather than through inheritance using a pseudo decorator pattern. Most of you will have seen this done with selection checkboxes, but it's something that can be applied with other use-cases as well.

Facade

The primary purpose of the Facade pattern is to provide a simpler interface to a complex class. This avoids repeated code and increases maintainability. Common uses include:

- Simplifying the execution of an Apex Web Service Proxy class
- Simplifying the execution of custom Apex classes with complex interfaces
- Providing a single interface to execute methods in multiple classes (e.g. multiple web service callouts)

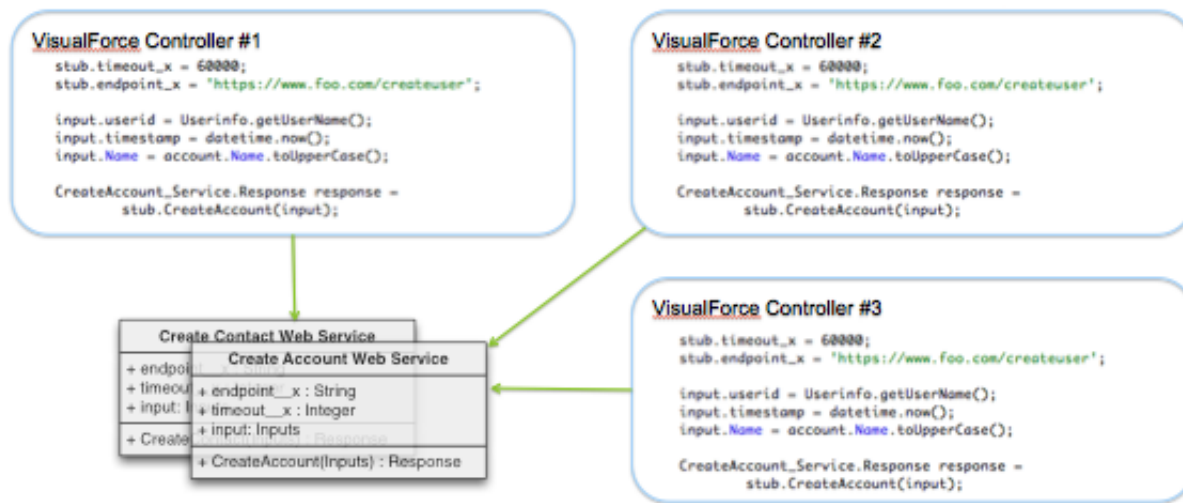
This pattern effectively abstracts one or more complex classes, simplifying their execution for the rest of the application.

Problem

Often times, the execution of a particular class method requires multiple lines of code or is complex in nature. If the same code is repeated multiple times across different parts of the application, this degrades maintainability.

In Force.com, one of the biggest examples of this is the execution of Web Service callouts. The generated Apex code often times requires complex code, such as setting timeout values, setting the target host, as well as setup of the various inputs and parsing of the callout results.

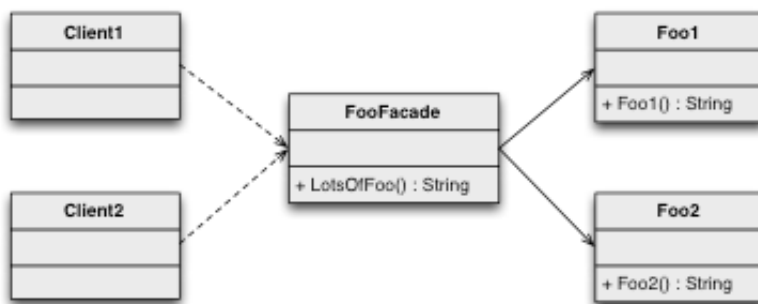
The following demonstrates the issue of repeated code when executing Web Service callouts, especially if the business process requires the execution of multiple web services.



(/page/File:Facade2.png)

In the above example, multiple clients attempt to create an account and contact in a target system. This results in repeated code to setup two web service proxies, degrading maintainability.

Unified Modeling Language



(/page/File:Facade.png)

```

1 public FooFacade {
2     public String LotsOfFoo() {
3         Foo1 f1 = new Foo1();
4         Foo2 f2 = new Foo2();
5
6         return f1.Foo1() + f2.Foo2();
7     }
8 }

```

Implementation

To implement a facade class, simply create another class that abstracts the implementation of the complex class(es). This facade class usually contains a simpler interface and in some cases, orchestrates the execution of multiple complex classes.

The purpose of the facade class is to simplify the execution of one or more complex classes with a simpler interface, increasing maintainability.

The following code sample shows a facade class that orchestrates the creation of an account and contact in an external system by calling multiple web services:

Facade Class

```

01 public class CreateCustomerFacade {
02
03     // Inner class representing the response from the web service
04     public class CreateCustomerResponse {

```

Join over 2.5 million Salesforce developers. [Subscribe Now!](#) [HIDE](#)

```

05         public String accountNumber;
06         public String contactNumber;
07
08         public CreateCustomerResponse(String accountNumber, String
contactNumber) {
09             this.accountNumber = accountNumber;
10             this.contactNumber = contactNumber;
11         }
12     }
13
14     // Method to call the two external web services
15     public String CreateCustomerExternal(String Name, String LastName,
String FirstName) {
16         // Setup the web service proxy class to create an account in
the target system
17         CreateAccount_Service.CreateAccount stubCA = new
CreateAccount_Service.CreateAccount();
18         CreateAccount_Service.Inputs inputCA = new
CreateAccount_Service.Inputs();
19
20         // Set timeout and endpoint
21         stubCA.timeout_x = 60000;
22         stubCA.endpoint_x = 'https://www.foo.com/ca
(https://www.foo.com/ca)';
23
24         // Default other values
25         inputCA.userid = Userinfo.getUserName();
26         inputCA.timestamp = datetime.now();
27         inputCA.Name = name.toUpperCase();
28
29         // Call the web service and retrieve the account number
30         String accountNumber = inputCA.CreateAccount(input);
31
32         /* REPEAT FOR CONTACT - Code not shown */
33
34         // Generate and return a response
35         return new CreateCustomerResponse (accountNumber,
contactNumber);
36     }
37 }

```

In the above facade class:

- The CreateCustomerExternal method wraps the execution of the two web service calls to create an account and contact in the external system. Repeated code, such as setting timeout values and endpoints, user IDs, timestamps, etc., is encapsulated within this method rather than repeated elsewhere.
- The CreateCustomerResponse inner class contains the account and contact numbers generated by the external system.

The facade class is then called in other parts of the application as follows:

```

01 public class FooController{
02
03     public Account account { get; set; }
04     public Contact contact { get; set; }
05     ...
06     public void CallCreateCustomerWS() {
07         CreateCustomerFacade ca = new CreateCustomerFacade();

```

```
08  
09         CreateCustomerFacade.CreateCustomerResponse resp =  
ca.CreateCustomerExternal(account.name, contact.LastName,  
contact.FirstName);  
10  
11         account.accountNumber = resp.accountNumber;  
12         contact.contactNumber__c = resp.contactNumber;  
13     }  
14 }
```

Conclusion

The Facade design pattern increases the maintainability of Apex code by simplifying the execution of one or complex classes with a facade class.

Composite

The Composite Design Pattern allows for representation of expressions, such as;

- 1 AND 2
- 1 OR (2 AND 3)
- (1 AND 2) OR ((3 OR 4) AND 5)

Problem

Our developer has a requirement to create a custom version of the Create/Edit List View screen. However, he is really struggling with how to represent an expression (outlined in red). The difficulty is due to the level of recursion involved.

Filter By Additional Fields (Optional):

Field	Operator	Value
1. Date	less than	9/18/2012
2. Owner Alias	equals	rvanhoo
3. Date	less than	8/18/2012
4. Owner Alias	equals	dthong
5. Days	equals	3

[Add Row](#) [Remove Row](#)

[Clear Filter Logic](#)

Filter Logic:

(1 AND 2) OR ((3 OR 4) AND 5)

[Tips](#) ?

Example: If you wanted to filter to key deals for your company, where key deals are deals over \$1,000,000 that are closing in the next 45 days, or deals owned by a VP, you would set up your filters as follows

Advanced Filters:

Field	Operator	Value
1. Amount	greater than	1000000
2. Closed Date	equals	NEXT 45 DAYS
3. Owner Role	starts with	VP
4. --None--	equa	
5. --None--		

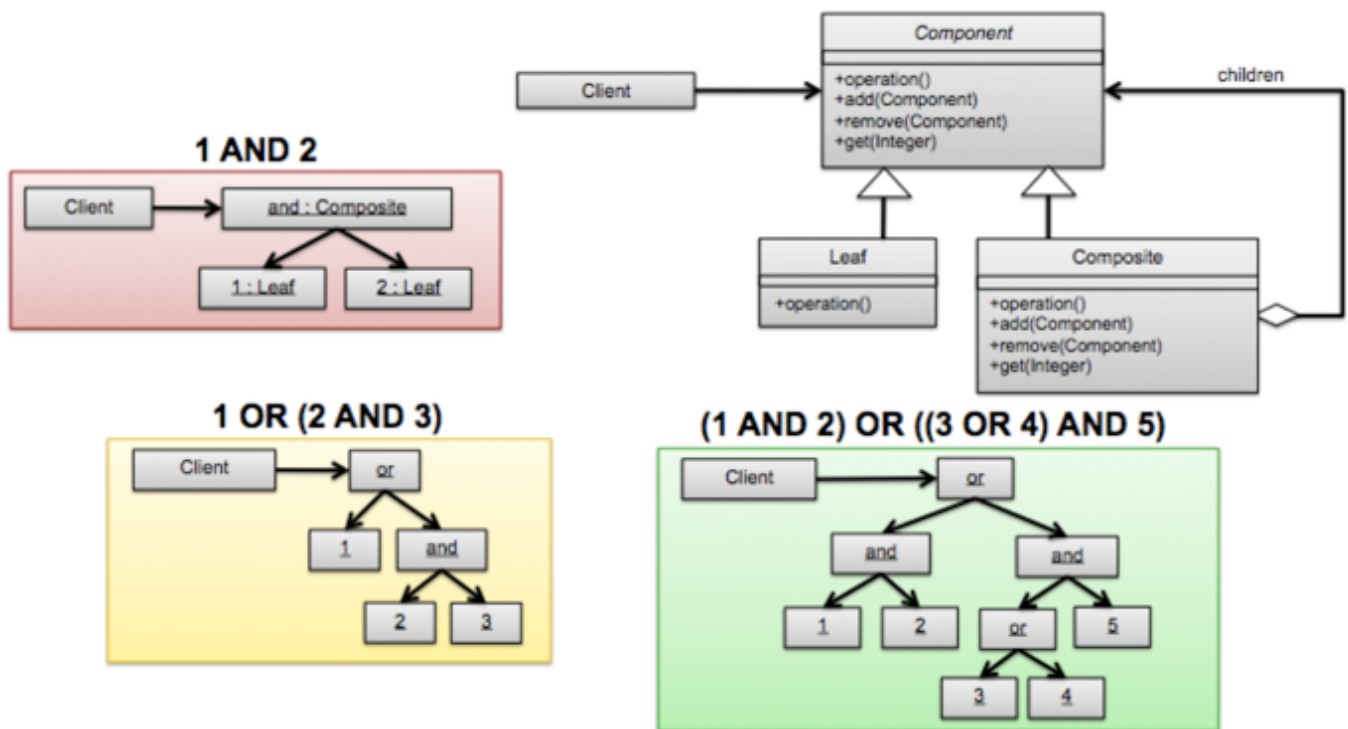
Advanced Filter Conditions:
(1 AND 2) OR 3



</page/File:ExpressionScreen.jpg>

Note: This discussion does not cover developing the actual screen or parsing the expression - it is limited to representing an expression in Apex.

Unified Modeling Language



(/page/File:Compositeuml.jpg)

Implementation

To implement the Composite Design Pattern, deploy the following interface and classes in your environment.

Expression Interface

```

1 public interface Expression {
2     Expression add(Expression expr);
3     Expression set(String name, Boolean value);
4     Boolean evaluate();
5 }

```

Composite class

```

01 public abstract class Composite implements Expression{
02     public List<Expression> children {get; private set;}
03     public Composite(){ this.children = new List<Expression>(); }
04     public Expression add(Expression expr){
05         children.add(expr); return this;
06     }
07     public Expression set(String name, Boolean value){
08         for(Expression expr : children) expr.set(name,value);
09         return this;
10     }
11     public abstract Boolean evaluate();
12     public Boolean hasChildren{get{ return !children.isEmpty(); }}
13 }

```

AndComposite Class

```

1 public class AndComposite extends Composite{
2     public override Boolean evaluate(){
3         for(Expression expr : children) if(!expr.evaluate()) return

```

```

    false;
4     return true;
5     }
6 }

```

OrComposite class

```

1 public class OrComposite extends Composite{
2     public override Boolean evaluate(){
3         for(Expression expr : children) if(expr.evaluate()) return
4         true;
5         return false;
6     }
7 }

```

Variable class

```

01 public class Variable implements Expression{
02     public String name {get;private set;}
03     public Boolean value {get;private set;}
04     public Variable(String name){ this.name = name; }
05     public Expression add(Expression expr){ return this; }
06     public Expression set(String name, Boolean value){
07         if(this.name != null && this.name.equalsIgnoreCase(name))
08             this.value = value;
09         return this;
10     }
11     public Boolean evaluate(){ return value; }
12 }

```

Examples/Usage

The examples below illustrate how to use the Expression interface.

Example: 1 AND 2

```

1 //1 AND 2
2 Expression expr = new AndComposite();
3 expr.add(new Variable('1'));
4 expr.add(new Variable('2'));

```

Example: 1 OR (2 AND 3)

```

1 //1 OR (2 AND 3)
2 Expression expr = new OrComposite();
3 expr.add(new Variable('1'));
4 Expression expr2 = new AndComposite();
5 expr.add(expr2);
6 expr2.add(new Variable('2'));
7 expr2.add(new Variable('3'));

```

Example: Method Chaining

```

1 //no need for expr2 var if using "method chaining"
2 //last line of add method: return this;
3 Expression expr = (new OrComposite())
4     .add(new Variable('1'))

```



```

5         .add((new AndComposite()))
6         .add(new Variable('2'))
7         .add(new Variable('3'))
8     );

```

Example: 1 OR (2 AND 3)

```

01 //1 OR (2 AND 3)
02 Expression expr = (new OrComposite())
03     .add(new Variable('1'))
04     .add((new AndComposite())
05         .add(new Variable('2'))
06         .add(new Variable('3'))
07     )
08     .set('1', false)
09     .set('2', true)
10     .set('3', false);
11
12 System.debug(expr.evaluate());
13 //FALSE OR (TRUE AND FALSE) => FALSE
14
15 expr.set('3', true);
16
17 System.debug(expr.evaluate());
18 //FALSE OR (TRUE AND TRUE) => TRUE

```

Conclusion

The Composite design pattern can be used to represent an expression in Apex regardless of expression complexity, whilst mitigating the impact of governor limits that can result from recursions.

Bulk State Transition

The Bulk State Transition design pattern is a general pattern used for performing bulk actions in Apex based on the change of state of one or more records.

Problem

Our developer has written a trigger to create an order upon the close of an opportunity, however, he has noted that:

- It always creates a new order every time the closed opportunity is updated
- When loading via Data Loader, not all closed opportunities result in an order

The Offending Code

```

1 trigger OpptyTrigger on Opportunity (after insert, after update) {
2
3     if (trigger.new[0].isClosed) {
4         Order__c order = new Order__c();
5         ...
6         insert order
7     }
8 }

```

The above code has more than a few issues. Namely;

- Occurs regardless of prior state - What if a closed opportunity was updated?

- No bulk handling - What if more than one opportunity was closed?
- Poor reusability - What if there are other places that we needed to create orders from an opportunity?

Our developer has rewritten the offending code, this time using a trigger and class.

Attempt 2: Trigger

```

1 trigger OpptyTrigger on Opportunity (after insert, after update) {
2
3     new OrderClass().CreateOrder(trigger.new);
4
5 }
```

Attempt 2: Class

```

01 public class OrderClass {
02
03     public void CreateOrder(List<Opportunity> opptyList) {
04         for (Opportunity oppty : opptyList) {
05             if (oppty.isClosed) {
06                 Order__c order = new Order__c();
07                 ...
08                 insert order;
09             }
10         }
11     }
12 }
13 }
```

While his second attempt improves reusability, there are still the issues:

- Occurs regardless of prior state
- No bulk handling

Our developer makes yet another attempt at addressing these concerns.

Attempt 3: Trigger

```

1 trigger OpptyTrigger on Opportunity (before insert, before update) {
2     if (trigger.isInsert) {
3         new OrderClass().CreateOrder(trigger.newMap, null);
4     }
5     else
6         new OrderClass().CreateOrder(trigger.newMap, trigger.oldMap);
7 }
```

Attempt 3: Class

```

01 public class OrderClass {
02
03     public void CreateOrder(Map<Opportunity> opptyMapNew
04                             Map<Opportunity> opptyMapOld) {
05         List<Order__c> orderList = new List<Order__c>();
06         for (Opportunity oppty : opptyMapNew.values()) {
07             if (oppty.isClosed && (opptyMapOld == null ||
08                                     !opptyMapOld.get(oppty.id).isClosed)) {
```

```

09         Order__c order = new Order__c();
10         ...
11         orderList.add(order);
12     }
13 }
14 insert orderList ;
15 }
16 }

```

This last attempt is slightly better. At least it now handles state transition (i.e. only if the opportunity is closed). Also, it's now bulkified – separate list to keep track of orders, and a single bulk insert. However, the order class is highly coupled and is very hard to reuse outside of the trigger context. You can pass in null into the second argument, but that relies on you knowing the inner workings of the class.

Unified Modeling Language



(/page/File:BulkTransition_uml.jpg)

In the Bulk State Transition design pattern:

- The trigger checks for eligible records that have changed state
- Calls a utility class to perform the work, only passing in the eligible records
- The utility class has a method that does the work in bulk

Implementation

We will implement the Bulk State Transition pattern utilizing two components:

- A trigger that which creates a filtered list of eligible records that have changed state
- A utility class that performs the logic for the list of eligible records in bulk

OpptyTrigger

```

01 trigger OpptyTrigger on Opportunity (after insert, after update) {
02     if (trigger.isAfter && (trigger.isInsert || trigger.isUpdate)) {
03         List<Opportunity> closedOpptyList = new List<Opportunity>();
04
05         //loop through opportunities and check if closed
06         for (Opportunity oppty : trigger.new) {
07             /* Note: for insert, check current state,
08              * for update, check current state and prior state */
09             if (oppty.isClosed && (trigger.isInsert ||
10                 (trigger.isUpdate &&
11                     !trigger.oldMap.get(oppty.id).isClosed)))
12                 closedOpptyList.add(oppty);
13         }
14
15         //send eligible records to OrderClass
16         if (!closedOpptyList.isEmpty())
17             new OrderClass().CreateOrderFromOpptys(closedOpptyList);
18     }

```

Above, we have the trigger, *OpptyTrigger*:

- Checks for the type of trigger (IsBefore, IsAfter, etc.) and the DML type (IsInsert, IsUpdate, etc.).
- Instantiates a list, *closedOpptyList*, that keeps track of our eligible records.
- Loops through the trigger.new.
 - If the oppty is eligible (IsClosed), we add it to the list of eligible records. The important thing here is that for inserts, no check is made for the state change (since if the criteria is met, it's always treated as a state change) and if it's an update, we check the oldMap to obtain the prior record's value. Only if it's changed will we make it eligible.
- It is important that the trigger checks for eligibility only and has no logic.

OrderClass

```

01 public class OrderClass {
02     public void CreateOrdersFromOpptys(List<Opportunity> opptyList) {
03         List<Order__c> orderList = new List<Order__c>();
04         for (Opportunity oppty : opptyList) {
05             Order__c order = new Order__c();
06             //more logic here
07             orderList.add(order);
08         }
09         insert orderList ;
10     }
11 }

```

Above we have the utility class, *OrderClass*:

- OrderClass is passed the list of eligible opportunities from our *OpptyTrigger*.
- We loop through the list of opportunities, creating a new Order object for each, and adding them to a list of orders.
- We then insert the list of Orders in one operation to ensure the operation is bulk safe.

Conclusion

With the above example, we've demonstrated implementing the Bulk State Transition design pattern as a means to perform bulk actions in Apex based on the change of state in one or more records.

About the Authors

[Dennis Thong \(/page/Dennis_Thong\)](/page/Dennis_Thong) is a New York-based Director of Technical Architecture in Salesforce Services. An industry veteran with 13 years in building and architecting CRM solutions, he specializes in all things platform related, including integrating with, securing, and customizing Salesforce.com. If he's not working for a customer, he's trying to find the best coffee in the city.

[Manu Erwin \(/page/Manu_Erwin\)](/page/Manu_Erwin) is a Cloud Computing professional with experience of implementing varied, full lifecycle CRM solutions. With over 5 years at Salesforce.com, 7 years working with the product, 6 years technical support management experience, and 11 years global consulting experience (Deloitte AU & US, Salesforce EMEA & APAC). [Read more... \(/page/Manu_Erwin\)](#)

[Eric Santiago \(/page/Eric_Santiago\)](/page/Eric_Santiago) has been involved with the Salesforce platform since 2005. As a Technical Architect with the Salesforce Services team, he has a broad expertise in all aspects of the platform particularly Apex/VisualForce development, Sites, and Portals. Eric

loves adventure travel and has visited 28 countries on 5 continents. He is based in New York City.

Technical Library

[Documentation \(/index.php/Documentation\)](#)
[Tools \(/index.php/Tools\)](#)
[Integration \(/index.php/Integration\)](#)
[App Logic \(/index.php/App_Logic\)](#)
[User Interface \(/index.php/User_Interface\)](#)
[Database \(/index.php/Database\)](#)
[Security \(/index.php/Security\)](#)
[Web Sites \(/index.php/Websites\)](#)
[Mobile \(/mobile\)](#)
[App Distribution \(/index.php/App_Distribution\)](#)
[Newsletter \(/index.php/News\)](#)
[Release Information \(http://developer.force.com/releases/\)](http://developer.force.com/releases/)

RSS Feeds

[Featured Content \(http://feeds.feedburner.com/force/featuredcontent\)](http://feeds.feedburner.com/force/featuredcontent)
[Blog \(http://feeds.feedburner.com/SforceBlog\)](http://feeds.feedburner.com/SforceBlog)
[Discussion Boards \(http://developer.salesforce.com/forums/ForumsRSS\)](http://developer.salesforce.com/forums/ForumsRSS)

Get started

[Salesforce App Cloud \(/platform\)](#)
[Force.com \(/gettingstarted\)](#)
[Heroku \(https://devcenter.heroku.com/start\)](https://devcenter.heroku.com/start)

Salesforce Dev Centers

[Heroku Dev Center \(http://devcenter.heroku.com/\)](http://devcenter.heroku.com/)
[Code @ ExactTarget \(http://code.exacttarget.com/\)](http://code.exacttarget.com/)
[Desk.com </developers> \(http://dev.desk.com/\)](http://dev.desk.com/)
[Pardot Developer Site \(http://developer.pardot.com/\)](http://developer.pardot.com/)

Developer resources

[Mobile Services \(/mobile\)](#)
[Force.com Docs](#)
[Force.com Downloads \(/page/Tools\)](#)
[Heroku Docs](#)
[Heroku Downloads \(http://toolbelt.heroku.com/\)](http://toolbelt.heroku.com/)
[Learn Salesforce with Trailhead \(/trailhead?\)](#)

utm_campaign=trailhead&utm_source=website&utm_medium=dsc_footer)

Community

[Developer Forums \(/forums\)](#)

[Salesforce Developer Events \(/calendar\)](#)

[Webinars \(/content/type/Webinar\)](#)

Learn more

[Salesforce AppExchange \(/appexchange\)](#)

[Salesforce Platform Overview \(//www.salesforce.com/platform/overview/\)](#)

[Salesforce.com Help Portal \(//help.salesforce.com/\)](#)

© Copyright 2000-2016 salesforce.com, inc. All rights reserved. Various trademarks held by their respective owners.

Salesforce.com, inc. The Landmark @ One Market, Suite 300, San Francisco, CA 94105, United States

[Privacy Statement \(//www.salesforce.com/company/privacy.jsp\)](#)

[Security Statement \(//www.salesforce.com/company/security.jsp\)](#)

[Terms of Use \(/files/tos/Developerforce_TOU_20101119.pdf\)](#)

[\[+\]Feedback](#)

[About Us \(http://www.salesforce.com/company/\)](#)

Language: English



[\(https://www.facebook.com/salesforcedevs\)](https://www.facebook.com/salesforcedevs)



[\(https://twitter.com/#!/salesforcedevs\)](https://twitter.com/#!/salesforcedevs)



[\(https://plus.google.com/118327959233932983591\)](https://plus.google.com/118327959233932983591)



[\(https://www.linkedin.com/groups/Developer-Force-Forcecom-Community-3774731\)](https://www.linkedin.com/groups/Developer-Force-Forcecom-Community-3774731)



[\(https://www.youtube.com/user/DeveloperForce\)](https://www.youtube.com/user/DeveloperForce)