**Naming Conventions**

* All objects created will use CamelCase notation.
* All object names will be alphanumeric only.

**Triggers**

* Triggers named as object then Trigger, e.g. AccountTrigger.
* No logic in trigger bodies.  Instead, Trigger.new, old, newMap or oldMap will be passed to static methods.  Only logic allowed is if different static methods must be called for different events by checking for isInsert, isUpdate, etc.
* All triggers should be "bulkified."  If there is a strong use case for not bulkifying, then Trigger.size should be used to ensure the batch size is always 1.
* Avoid recursion by using a static Boolean to be set the first time the trigger is run and surrounded by a conditional checking that Boolean's value.
* This is also true for any code using SOQL, SOSL and DML statements but the governor limits are much more stringent in triggers than elsewhere.  Use the Limit.getLimit methods to ensure code exits gracefully when a limit is encountered.
* Proposed: Where possible, avoid calling multiple methods from the trigger directly as this can result in an increased quantity of script statements by iterating multiple times over the same trigger record set.
* One trigger per object

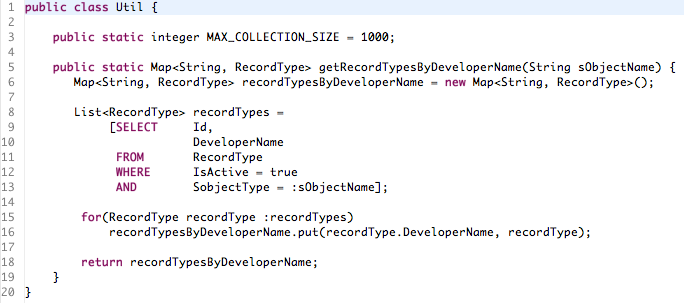
**VisualForce Controller Extensions**

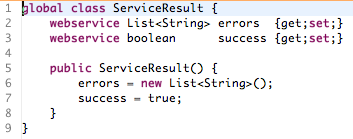
* Always create controller extensions unless there is a valid use case not to, e.g. overriding the standard search with a custom search screen where any object can be searched.
* Controller names always begin with an uppercase letter.
* Classes named by page and object extended and end with ContExt or Cont for controllers - E.g. CopyContactRolesOppContExt
* Use the transient keyword to declare instance variables that cannot be saved, and should not be transmitted as part of the view state for a Visualforce page. Declaring variables as transient reduces view state size. A common use case for the transient keyword is a field on a Visualforce page that is needed only for the duration of a page request, but should not be part of the page's view state and would use too many system resources to be recomputed many times during a request.
* Controllers are to be slim/lean, in that no logic - other than *some* pertaining to the implementation of the view - should be contained therein. Controllers are kept slim because their main responsibility is to initiate interactions *among classes in the model.*

**Classes**

* Methods - Method names always begin with a lower case letter.  All methods must have a descriptive name of what it actually does, e.g. copyAccountContactRolesToOpp.
* Don't litter code with system.debug in production code. If you really care about monitoring errors and actioning them then log a record in an error logging object after an unhandled error (as a bonus use workflow rule on the error logging object to send an email to the admin).
* Properties - Property names always begin with a lower case letter.  Any properties used as proxies in order to provide VF inputField components with all their validation glory will be prepended with proxy followed by the object being proxied, e.g. proxyAccount.  Always use the get/set syntax instead of creating get and set methods.
* Variables - Variable names always begin with a lower case letter.  Constants should be in all upper case (use final keyword)
* Enumerations - Enumeration names always begin with a capital letter and must be descriptive
* Declarations - Within any class, class level variables, properties and methods should be grouped together according to scope visibility...
* Private variables
* Public variables
* Public static variables
* Constructors
* Private properties
* Public properties
* Public static properties
* Private methods
* Public methods
* Public static methods
* All classes will be declared with the "with sharing" clause unless there is a valid use case not to, e.g. searching for duplicate records
* Web service classes created from WSDLs will have the name begin with the system that generated the WSDL and end with WebService, and if the WSDL also defines schemas, those classes' names will end with Schema
* Hardcoded IDs are not acceptable in any code.  Use custom settings or database queries instead.
* No empty catch blocks should be written.

**Util Class:** Store generic functionality in the Util class as shown below

**ServiceResult Class:** All Web Service returns are best wrapped in a generic result object as show below



Web Services must always return success or failure notification and hence

* cannot be void and
* must have robust error handling

**Web Service and Future methods:** Web Service and Future methods should not explicitly implement business logic; otherwise, testing them is difficult.

### Field naming

* Names of non-constant fields (reference types, or non-final primitive types) should use the infixCaps style.
* Start with a lower-case letter, and capitalize the first letter of any subsequent word in the name, as well as any letters that are part of an acronym. All other characters in the name are lower-case.
* Do not use underscores to separate words. The names should be nouns or noun phrases. Examples:

boolean resizable;

char recordDelimiter;

* Names of fields being used as **constants** should be all upper-case, with underscores separating words. The following are considered to be constants:
  + All static final primitive types (Remember that all interface fields are inherently static final).
  + All static final object reference types that are never followed by "." (dot).
  + All static final arrays that are never followed by "[" (dot).

Examples: MIN\_VALUE, MAX\_BUFFER\_SIZE, OPTIONS\_FILE\_NAME

* One-character field names should be avoided except for temporary and looping variables. In these cases, use:
  + b for a byte
  + c for a char
  + d for a double
  + e for an Exception object
  + f for a float
  + g for a Graphics object
  + i, j, k, m, n for integers
  + p, q, r, s for String, StringBuffer, or char[] objects
  + An exception is where a strong convention for the one-character name exists, such as x and y for screen coordinates.
* Avoid variable l (“el”) because it is hard to distinguish it from 1 (“one”) on some printers and displays.

**VisualForce Pages**

* All Visual Force Pages must have a "title" attribute specified like <apex:page title="Purpose of the page">.
* All interactive Visual Force Pages will have an <apex:pageMessages/> section for presentation of page errors.
* All interactive Visual Force Pages will have an <apex:sectionHeader title="" subtitle=""/> section header.
* Where possible, JavaScript code will reside in static resource files and be referenced by the page.
* Where styling is not available in the Salesforce.com standard, custom CSS should be added and properly commented for reference.
* Check for and handle null arguments when using QueryString arguments.
* When displaying data in Visualforce page, always wrap your EL statement in visualforce tags instead of just print out EL statement directly to avoid XSS attack (E.g.: <apex:outputText value=”{!EL}”/>).

**Code Comments**

All application code files (classes & triggers) will contain header comment blocks.  These comment blocks will describe the function of the code and list the original and all subsequent requirement changes.  Each requirement change should be detailed with information on when the code was changed, who wrote it (include company name), the project plan or case under which the code change was made and a description of the requirement that is being addressed.

**Dynamic SOQL & SOSL**

SOQL & SOSL Injection - To prevent SOQL / SOSL injection, use the escapeSingleQuotes method. This method adds the escape character (\) to all single quotation marks in a string that is passed in from a user. The method ensures that all single quotation marks are treated as enclosing strings, instead of database commands.

\* Embedded SOQL

\* Paramaterized Where-clauses

\* String.escapeSingleQuotes

**Mindful use of Constants**

Inline constant values, whether they be for testing purposes (I need *10* test records, etc) or for "configuration" values in your classes (cars have *4* tires), lead to brittle code that doesn't lend itself to being modified. Not only will hard-coding values lead to higher memory usage if these values need to be repeated several times, but any change to the value needs to be made in several places in the code that may be hard to find. It's in this light that constants are most useful - they provide a centralized place where one can quickly modify these essential values. The best approach to implementing these constants (scope, location, etc) depends on the context in which they're used and what purpose they serve.

Possible uses for constants:

* Generating Test Data (# of records, fake email address, etc)
* Values to validate against (Has this loop executed *X* times? Does this car have *Y* tires?)
* Enabling/Disabling of certain pieces of functionality and classes

In the first case, the value may be realized when one needs to quickly scale up tests (testing one record -> testing batch-safeness), or when validation rules are created against the field for which the constant contains a value; If an email field must end in "shire.com", for example, and we have that value stored in a constant in our test, then that's the only place requiring change. These kinds of constants are mostly stored in either the testing classes themselves, or in a TestingUtils class.

    In the second case - values to validate against in our code - the benefit sometimes comes from the sharing of these constants by both the constant containing class itself *and* any client classes that may interact with it. A good example of this would be a static constant that describes the number of wheels a car must contain. When a Car class is passed to a method on a Service Center class that will rotate tires, for instance, the Service Center might check said constant to ascertain how many wheels must be replaced. If the concept of non-four-wheeled vehicles are ever introduced to the solution, different vehicle classes can each have their own wheel # constant, and having utilized this approach will provide a clean point of extension for the newly required functionality.

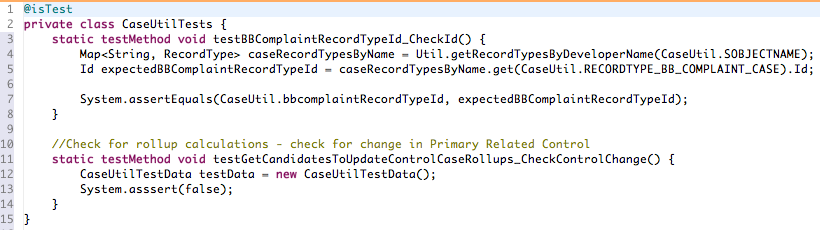
**Unit Testing**

* All test methods will reside in a separate class from the class in which the method being tested resides.
* These classes will be suffixed with the word Test followed by the name of the class being tested, e.g. OpportunityServicesTest.
* These classes will all use the @isTest annotation.
* Each test method must have an assertion.
* Use Test.startTest and Test.stopTest when testing governor limits.

Example:

|  |
| --- |
| static testMethod void verifyAccountDescriptionsWhereOverwritten(){        // Perform our data preparation.  List<Account> accounts = new List<Account>{};  for(Integer i = 0; i < 200; i++){  Account a = new Account(Name = 'Test Account ' + i);  accounts.add(a);  }  // Start the test, this changes governor limit context to  // that of trigger rather than test.  test.startTest();  // Insert the Account records that cause the trigger to execute.  insert accounts;  // Stop the test, this changes limit context back to test from trigger.  test.stopTest();  // Query the database for the newly inserted records.  List<Account> insertedAccounts = [SELECT Name, Description  FROM Account  WHERE Id IN :accounts];  // Assert that the Description fields contains the proper value now.  for(Account a : insertedAccounts){  System.assertEquals(  'This Account is probably left over from testing. It should probably be deleted.',  a.Description);  }  } |
|  |

* Only use isTest(SeeAllData=true) on class methods in exceptional cases where there are sObjects that don't allow DML operations.
* Test method names should use the following pattern  
  test{Method Name}\_{Purpose of the test}  
  A basic test class would look like below



Note in line 12 how the TestData class is instantiated for later use and failing assert.

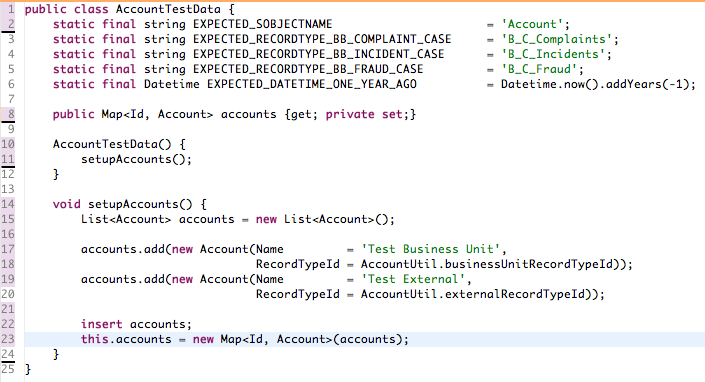
* **Creating test data classes**

**Naming:** Test Data class names should use the following pattern

{sObject Name without underscores}TestData

Example: AccountTestData, TaskTestData etc

A typical test data class would look like below:



Note the use of *private set;.* Important to use it to prevent test data modification once created.

At least 10 records must be created for test data.

**Unit Test Methods Best Practices**

Purpose of standard: Explain best practices for creating unit test methods.

A unit test method name should consist of three parts:

1. What you are testing
   1. This can be a specific method name such as “constructor” or “toString” or it can be behaviour such as “contactFieldsAutomaticallyPopulated”.
2. What data are you passing in this test?
   1. Examples include specific data types such as “nullParam” or “validString” or it can be a description of a set of data such as “validListOfAccounts” or “emptyList”.
3. What is the expected behaviour?
   1. Examples include return types such as “returnsString”, “returnsNull” or “returnsList”. It can also be a description of the result such as “throwsDmlException”.

Things to keep in mind as you’re writing your tests:

* Test method should not create the data within the method. Use a “TestDataUtility” or “TestUtils” or “TestServices” class as a test data factory.
* ALWAYS use seeAllData = false. There are a few very specific scenarios where you need to see all data but for the most part you should always be creating the data used by your tests.
* Test methods need to be “skinny”. Each method should only test one method or behaviour. Each method should also only perform a positive or negative test. There should never be multiple tests in one method.
  + For example, if you want to test the constructor with a NULL parameter and then a valid parameter this should be broken out into 2 methods:
* constructor\_nullParam\_throwsException
* constructor\_validString\_returnsNothing
* Every test method needs to have at least one assert() statement.
* Every assert statement needs to have an error description so that if the assert fails, the developer has a clear message as to why.

**Example names for unit test methods**

// Tests constructor

// We are passing a NULL parameter

**static testmethod void** constructor\_nullParam()

// Tests constructor

// We are passing a parameter of correct type, but with nothing filled out, e.g. ‘’ for a string or new Account() for an Account

**static testmethod void** constructor\_emptyParam()

// Tests constructor

// We are passing a “Default Record flag” param set to true

**static testmethod void** constructor\_createDefaultRecordTrue()

// Tests constructor

// We are passing a “Default Record flag” param set to false

**static testmethod void** constructor\_createDefaultRecordFalse()

// Tests a method named returnToRecord which returns page to record

// We are passing a NULL parameter

// We expect the method to return NULL

**static testmethod void** returnToRecord\_nullParam\_returnsNull()

// Tests a method named returnToRecord which returns page to record

// We are passing a valid parameter

// We expect the method to return a PageReference

**static testmethod void** returnToRecord\_validParam\_returnsPR()

// Tests a method named displayExceptionAsMessage which displays an error to the user

// We are passing a NULL params

// Has a void return type

**static testmethod void** displayExceptionAsMessage\_nullParams\_returnsNothing()

// Tests a method named displayExceptionAsMessage which displays an error to the user

// We are passing valid params

// Has a void return type

**static testmethod void** displayExceptionAsMessage\_validParams\_customException\_returnsNothing()

// Tests a controller method which displays an error to the user

// We are passing valid params. We expect a DMLException to be thrown

// Has a void return type

**static testmethod void** displayExceptionAsMessage\_validParams\_DmlException\_returnsNothing()

**Exception Handling**

* All DML operations (or database calls, as necessary) will be performed in a try/catch block or the method will be annotated @throws DMLException.
* Where multiple sequential DML operations make up one atomic unit, a Savepoint must be used to create a single transaction. Instantiate the Savepoint outside the try block so it remains in scope within the catch. In case of any errors within the transaction, rollback the database to the last valid Savepoint.
* Since all interactive VisualForce pages will have a pageMesssage component, all exceptions should be caught and displayed to the user.
* Additionally, all error exceptions, as opposed to informational exceptions or validation exceptions should be logged to a custom object.
* If a controller constructor calls methods that log exceptions to said custom object, VisualForce will throw an error because DML statements are not allowed in controller constructors.  In these occasions, an initPage() method will house this code and will be called on page load using the action attribute on the <apex:page> definition.  Methods called this way fire after the constructor and therefore do allow DML statements.