SALESFORCE UNIFIED TRIGGER FRAMEWORK

Before we delve into the concepts of our framework we should talk about the relevance of triggers with the advent of Visual Flows. Are triggers yesterday news and will eventually dissipate into the technology history pages. We believe not for a few key reasons:

1. Most companies have already invested heavily into building code in triggers and to switch to migrating to Flows would be expensive. Coupled with the fact that companies often have committed roadmaps of development, it is rare that such a migration would be considered higher priority.

2. Companies' decision process is usually governed by ROI. The relative ROI of delivering new products and capabilities on their roadmap will normally be higher than migrating triggers to Flows, even after factoring in the time saved in maintaining triggers.

3. More complex triggers are very difficult to replicate in a Flow

4. Building unit test code is not required for Flows, whereas it is required for a trigger. We believe it is important to unit test such core operational functionality of triggers and Flows and the temptation to bypass building unit tests for Flows, because they are not enforced by Salesforce would be too great to ignore for many companies, which in our belief builds technical debt into the system.

5. It is unlikely Salesforce will mandate Flows to be used instead of triggers as this will affect too many customers which means triggers are unlikely to be deprecated

The aim of this framework is to establish a consistent structure for Salesforce triggers to achieve the following:

1. Control the order of execution of code in triggers so we know what events will take place in sequence

2. To provide a more manageable and intuitive design framework so that defect fixing is quicker and logical separation of business rules are ensured

3. Improve the efficiency of triggers so that processing is performed faster which is one of the most processor intensive operations

4. Able to configure and control individual trigger operations for faster and more controlled processing that leads to a better user experience and fewer defects

5. Build monitoring into the fabric of the trigger so that defect fixing has a faster turn around

There are many well published trigger frameworks, and all have merits in that they improve the manageability of code, correctly orders execution, however as with everything, good technologists will continually evolve a model.

We were impressed with Tony Scotts http://developer.force.com/cookbook/recipe/trigger-pattern-for-tidy-streamlined-bulkified-triggers pattern as it simplifies the trigger.

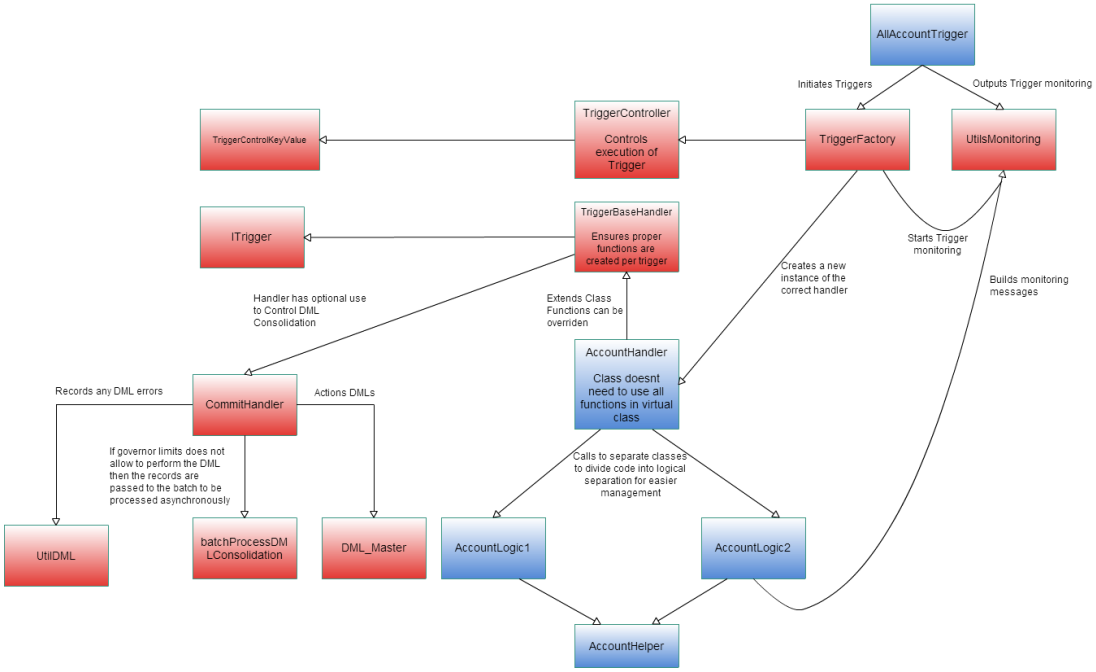
Independently to Hari Krishnan https://krishhari.wordpress.com/tag/apex-trigger-design-pattern/ We too noticed some room for improvement because Tony's framework would require continual adaption of the TriggerFactory class for every new trigger that is developed.

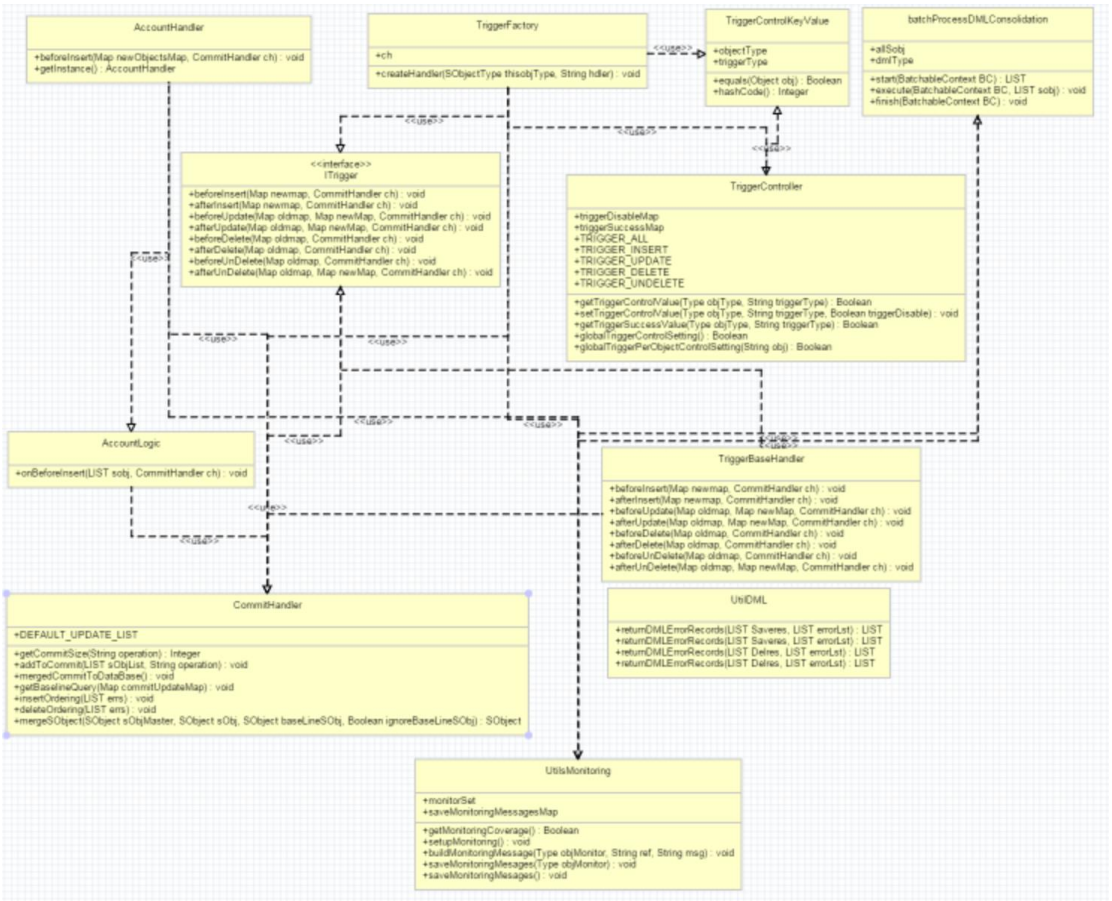
The solution that we came up with was similar to the 2nd framework above.

However, we were concerned that all frameworks to date have only been designed to solve the old-age problem of code manageability and order of execution, but we always incorporated far more into our frameworks, notably the following additional features:

- Trigger Control
- Code Monitoring
- DML Consolidation

Here is an outline to the structure of our framework





The red blocks represent the core framework, the blue blocks are specific for the trigger you are building and where the logic of the trigger resides.

1. Trigger Control

DMLs are very processor intensive and conserving this precious resource in a multitenant environment is ever more so important.

So, we built into the framework a capability using a Custom Setting to activate / deactivate per trigger for any User, Profile, or the entire Organization; or with a separate Custom Setting too be able to

activate / deactivate ALL triggers for any User, Profile, or the entire Organization. This was great in situations where companies require migrating data from 1 system to another and you want to safeguard that no unwanted actions occur that you didn't intend through the execution of code in the trigger.

However there are also many situations such as in unit tests where we create test data, to avoid running through all the code in triggers which is not necessary unless we are testing specifically the triggers to disable the triggers using the Custom Settings above will require running a DML and since we are trying to avoid DMLs because they are expensive to run we need another mechanism to bypass the code in the trigger, therefore we introduce a static variable to do this work.

We also need to be able to unit test the framework, to test the trigger control has been built correctly and remains operational. To test this part of the framework we don't want to test the outcomes from running each individual part of the trigger as the outcomes will be different per trigger, instead we just need to test that the code passed through the track of code we expect. For this purpose, a map is used to control execution. You will see this in action later.

2. Monitoring

Monitoring is activated using a hierarchy Custom Setting, which can be scheduled to activate during a specific time range for a user, Profile, or entire Organization.

The TriggerFactory first checks if monitoring has been activated. Then monitoring statements are held in memory and are grouped under a user defined identifier.

The standard monitoring capabilities in Salesforce has improved considerably recently allowing debug monitoring on individual users to be turned on at set times, at set levels of logging levels, for specific types of monitoring such as Apex logging. However, there are some gaps in standard monitoring capabilities in Salesforce, notably being:

1. Specific targeted monitoring messages only to be recorded making it much easier to output information that is required and allows the developer to focus the areas of interest to them.

2. To turn on/off monitoring for specific classes, triggers and even sub divide classes, triggers into segmented areas for monitoring.

3. For some applications you cannot simply re-run the operation to debug the defect because either the defect was time dependent, or the scenario was difficult to replicate, or for example for financial reasons you cannot replay the transaction because it will create financial irregularities or impacts to external systems.

4. Grouping messages under set categories, such as monitoring output of a specific record.

5. Allow the potential of performing actions and reports generated off the recording of the monitoring messages.

3. DML Consolidation

Let's explain this concept using an example.

An Account trigger calls 2 separate functions in a logic class that inserts a number of Contacts, and updates several Opportunities. The pseudo code is below:

```
Function A(){
    Insert contacts1;
    Update Opportunities1;
}

Function B(){
   Insert contacts2;
   Update Opportunities2;
   Update contacts3;
}
```

As you can see there are 5 separate DML statements above

To improve the efficiency of the code the DML Consolidation framework will hold in memory contacts1, contacts2, contacts3, Opportunities1 and Opportunities2

It groups the same sObject types together along with the type of DMLs

So you will have 3 groups in total:

1. Insert Contacts
2. Update Opportunities
3. Update Contacts

So instead of having 5 DML statements you will now only have 3. Of course on inception of the trigger you will not know how many DMLs you will require until the analysis by the framework has run.

For this reason, it is unavoidable to run the DMLs through a for loop, which is deemed bad practice normally.

Im not suggesting that the rulebook regarding DMLs in for loops to be rewritten but rather to be amended as the framework is clearly correct. We have our placed some guardrails in the framework to prevent governor limit breakages.

Whilst building the Logic classes where the actual work of the trigger is actioned, the developer has the choice to pass the reference of the CommitHandler or not to. Only if you require DML Consolidation do you need to pass the reference.

Of course you still have the option to process DMLs as you wish and not to utilise the DML Consolidation framework, this will not break any other part of the framework.

The DML Consolidation framework processes the DMLs at the end of the trigger in the afterinsert, afterupdate, afterdelete or afterundelete. First a check is done to ensure that the DML won't break governor limits before processing, if it doesn't it attempts to process the records, but if any errors occur, the framework checks if governor limits still allows errors to be recorded in the separate custom object. If the first check identifies that governor limits will be broken the processing is passed instead to a batch class to process the DMLs.

A key difference between this framework and Hari Krishnas framework is his framework creates several handler classes for each type of trigger beforeinsert, beforeupdate, beforedelete, beforeundelete, afterinsert, afterupdate, afterdelete, or afterundelete. However we segment by Business area instead for 2 main reasons:

1. Creating a potential of 8 handler classes where the work is done; then each trigger will have potentially 11 classes that you have to create, plus each of the classes that do the work ought to have a unit test class. And so, in total you could have 19 classes per trigger. For companies this can result in having hundreds of extra classes for your organisation which will be difficult to manage.

2. A business owner doesn't raise requests to change a beforeinsert trigger. They raise requests to change a business area. So, segmenting your classes into where the work is conducted by business area makes your triggers more manageable and understandable.

Let's start build the framework.
To begin we need to set up some of the base classes.

The following code may not appear indented correctly but do not be concerned once you copy the code into your apex classes the code will be properly indented.

ITrigger Interface

```
public interface ITrigger{
	void beforeInsert(SObject[] newObj, CommitHandler ch);

	void afterInsert(SObject[] newObj, Map<id,SObject> newmap, CommitHandler ch);

	void beforeUpdate(SObject[] oldObj, SObject[] newObj, Map<id,SObject> oldmap, Map<id,SObject> newMap, CommitHandler ch);

	void afterUpdate(SObject[] oldObj, SObject[] newObj, Map<id,SObject> oldmap, Map<id,SObject> newMap, CommitHandler ch);

	void beforeDelete(SObject[] oldObj, Map<id,SObject> oldmap, CommitHandler ch);

	void afterDelete(SObject[] oldObj, Map<id,SObject> oldmap, CommitHandler ch);

	void beforeUnDelete(SObject[] oldObj, Map<id,SObject> oldmap, CommitHandler ch);

	void afterUnDelete(SObject[] oldObj, SObject[] newObj, Map<id,SObject> oldmap, Map<id,SObject> newMap, CommitHandler ch);

}
```

Base handler that implements ITrigger

```
public virtual class TriggerBaseHandler implements ITrigger{

	public virtual void beforeInsert(SObject[] newObj, CommitHandler ch){

	}
	public virtual void afterInsert(SObject[] newObj, Map<id,SObject> newmap, CommitHandler ch){

	}
	public virtual void beforeUpdate(SObject[] oldObj, SObject[] newObj, Map<id,SObject> oldmap, Map<id,SObject> newMap, CommitHandler ch){
```

```
        }
        public virtual void afterUpdate(SObject[] oldObj, SObject[] newObj,
Map<id,SObject> oldmap, Map<id,SObject> newMap, CommitHandler ch){

        }
        public virtual void beforeDelete(SObject[] oldObj, Map<id,SObject> oldmap,
CommitHandler ch){

        }
        public virtual void afterDelete(SObject[] oldObj, Map<id,SObject> oldmap,
CommitHandler ch){

        }
        public virtual void beforeUnDelete(SObject[] oldObj, Map<id,SObject> oldmap,
CommitHandler ch){

        }
        public virtual void afterUnDelete(SObject[] oldObj, SObject[] newObj,
Map<id,SObject> oldmap, Map<id,SObject> newMap, CommitHandler ch){

        }
}
```

TriggerControlKeyValue used in the Trigger Control of the framework

```
public class TriggerControlKeyValue {

public system.type objectType;
public string triggerType;

        public TriggerControlKeyValue(system.type thisObjectType, string
thisTriggerType){
                objectType = thisObjectType;
                triggerType = thisTriggerType;
        }

        public boolean equals(object obj){
                if (obj instanceof TriggerControlKeyValue){
                        TriggerControlKeyValue t = (TriggerControlKeyValue)obj;
                        return (objectType.equals(t.objectType) &&
triggerType.equals(t.triggerType));
                }
                return false;
        }

        public integer hashCode(){
                return system.hashCode(objectType) * system.hashCode(triggerType);
        }
}
```

Along with TriggerControlKeyValue the TriggerController stores and retrieves values for each trigger

```
public class TriggerController {

	public static map<TriggerControlKeyValue, boolean> triggerDisableMap = new
map<TriggerControlKeyValue, boolean>();
	public static map<TriggerControlKeyValue, boolean> triggerSuccessMap = new
map<TriggerControlKeyValue, boolean>();

	public static final String TRIGGER_NONE = 'NONE';
	public static final String TRIGGER_ALL = 'ALL';
	public static final String TRIGGER_INSERT = 'INSERT';
	public static final String TRIGGER_UPDATE = 'UPDATE';
	public static final String TRIGGER_DELETE = 'DELETE';
	public static final String TRIGGER_UNDELETE = 'UNDELETE';

	public static Boolean getTriggerControlValue(System.Type objType, String
triggerType){
		TriggerControlKeyValue tkv = new TriggerControlKeyValue(objType
,triggerType);
		Boolean triggerDisable = false;
		if (triggerDisableMap != null && triggerDisableMap.containskey(tkv))
			triggerDisable = triggerDisableMap.get(tkv);

		return triggerDisable;
	}

	public static void setTriggerControlValue(System.Type objType, String
triggerType, Boolean triggerDisable){
		TriggerControlKeyValue tkv = new TriggerControlKeyValue(objType
,triggerType);

		for (TriggerControlKeyValue eachtk : triggerDisableMap.keyset()){
			if (eachtk == tkv){
				tkv = eachtk;
				break;
			}
		}
		triggerDisableMap.put(tkv, triggerDisable);
	}

	public static Boolean getTriggerSuccessValue(System.Type objType, String
triggerType){
		TriggerControlKeyValue tkv = new TriggerControlKeyValue(objType
,triggerType);
		Boolean triggerSuccess = false;


		for (TriggerControlKeyValue eachtk : triggerSuccessMap.keyset()){
			if (eachtk == tkv){
				triggerSuccess = triggerSuccessMap.get(eachtk);
				break;
```

```
                        }
                }

                return triggerSuccess;
        }

public static boolean globalTriggerControlSetting(){
        return (((Triggers_Off__c.getOrgDefaults() != null) ?
Triggers_Off__c.getOrgDefaults().value__c : false) ||
Triggers_Off__c.getInstance(UserInfo.getUserId()).value__c  ||
Triggers_Off__c.getInstance(UserInfo.getProfileId()).value__c) ;
}

public static boolean globalTriggerPerObjectControlSetting(String obj){
        if (obj != null && obj != '') {
                if (!obj.endswith('__c')) obj += '__c';
                        boolean s = false;
                        if (Trigger_Per_Object__c.getOrgDefaults() != null) s =
(boolean)Trigger_Per_Object__c.getOrgDefaults().get(obj);

                        boolean t = false;
                        if (Trigger_Per_Object__c.getInstance(UserInfo.getUserId()) !=
null) t =  (boolean)Trigger_Per_Object__c.getInstance(UserInfo.getUserId()).get(obj);

                        boolean u = false;
                        if (Trigger_Per_Object__c.getInstance(UserInfo.getProfileId()) !=
null) u =  (boolean)Trigger_Per_Object__c.getInstance(UserInfo.getProfileId()).get(obj);

                        if  (s == null) s = false;
                        if  (t == null) t = false;
                        if  (u == null) u = false;

                        return (s || t || u);
        }else
                return false;
}

}
```

First of all we need to create 2 Hierarchical custom settings:

Triggers_Off__c

This custom settings needs to have the following fields:

Field Name	Data Type
value	Boolean

Trigger_Per_Object__c

This custom settings needs to have the following fields:

Field Name	Data Type
Account	Boolean
Contact	Boolean

For each additional trigger you create you will need to create an additional field in this custom setting for that trigger, and you will need to add a new else if statement to the globalTriggerControlSetting() function in the TriggerController class. An example of this will be shown next for the Account and Contact objects.

The 2 custom settings above can be used to bypass the triggers typically when you are making a deployment to Production, or you are performing a data migration. The custom settings will allow you to disable individual triggers or all triggers per User, per Profile or the entire system.

The CommitHandler provides DML Consolidation part of the framework

```
public with sharing class CommitHandler extends DML_Master{


        public enum Operation {INSERT_OPERATION, UPDATE_OPERATION,
DELETE_OPERATION}
        public Integer DEFAULT_UPDATE_LIST = 0;
        private List<sObject> commitInsertList;
        private List<sObject> commitUpdateList;
        private List<sObject> commitDeleteList;
        private Map<Schema.SObjectType, String> qryFieldBaselineMap = new
Map<Schema.SObjectType, String>();
        private Map<Schema.SObjectType, ID[]> qryWhereClauseBaselineMap = new
Map<Schema.SObjectType, ID[]>();
        private Map<Id,sObject> baseLineSObjs;

        public CommitHandler(){
                this.commitInsertList = new List<sObject>();
                this.commitUpdateList = new List<sObject>();
                this.commitDeleteList = new List<sObject>();
        }


        public void mergedCommitToDataBase(){
                list<Debug_Error__c> errors = new list<Debug_Error__c>();
```

```
                //INSERTS
                insertOrdering(errors);

                if (!commitUpdateList.isempty()){
                        // UPDATE LOGIC
                        //builds a map of queries per object for the baseline
                        Map<Schema.SObjectType, List<sObject>> commitUpdateMap =
new Map<Schema.SObjectType, List<sObject>>();
                        Schema.SObjectType sObjType;
                        Boolean objectUpdatesMultipleTimes = false;
                        Map<Id,sObject> tmpCommitUpdateMapByID = new
Map<Id,sObject>();

                        //groups all updates by object Type
                        for(sObject sObj : commitUpdateList){
                           sObjType = sObj.getSObjectType();

                           // if sObjType exists
                           if(!commitUpdateMap.containsKey(sObjType))
                                commitUpdateMap.put(sObjType, new List<sObject>());

                           if(tmpCommitUpdateMapByID.containsKey(sObj.id)){
                                objectUpdatesMultipleTimes = true;
                           }

                           tmpCommitUpdateMapByID.put(sObj.Id, sObj);
                           commitUpdateMap.get(sObjType).add(sObj);
                        }


                        //baseline is only required when at least 2 separate updates dmls
are to be made on the same object
                        //because only then can a record be potentially be changed back
to the original value
                        if (objectUpdatesMultipleTimes)
getBaselineQuery(commitUpdateMap);
                                                                // UPDATE - MERGE
                                Map<Id,sObject> commitUpdateMapByID = new
Map<Id,sObject>();
                                for(sObject sObj : commitUpdateList){

                                        if(commitUpdateMapByID.containsKey(sObj.id) ){

                                                if (baseLineSObjs != null &&
baseLineSObjs.containsKey(sObj.id))
//sObj = CURRENT OBJECT
//commitUpdateMapByID = PREVIOUS OBJECT

        commitUpdateMapByID.put(sObj.Id,mergeSObject(sObj,
commitUpdateMapByID.get(sObj.id), baseLineSObjs.get(sObj.id), false));
                                                else

        commitUpdateMapByID.put(sObj.Id, sObj);
                                        }else{
                                                commitUpdateMapByID.put(sObj.Id,
sObj);
```

```
                                        }
                                }

                                Map<Schema.SObjectType, List<sObject>>
commitReorganiseUpdateMap = new Map<Schema.SObjectType, List<sObject>>();

                                for(sObject sObj : commitUpdateMapByID.values()){

                                        sObjType = sObj.getSObjectType();


        if(commitReorganiseUpdateMap.containsKey(sObjType)){


        commitReorganiseUpdateMap.get(sObjType).add(sObj);
                                        }else{


        commitReorganiseUpdateMap.put(sObjType, new List<sObject>{sObj});
                                        }
                                }

// UPDATE
                                for(Schema.SObjectType sObjectType :
commitReorganiseUpdateMap.keySet()){

        updateObjects(commitReorganiseUpdateMap.get(sObjectType), true,
defaultBatchQuantity, errors, false, TRIGGER_NONE);
                                }
                }

                //DELETES
                deleteOrdering(errors);


                if (!errors.isempty()){
                        insertObjects(errors, true, defaultBatchQuantity, null, false,
TRIGGER_NONE);
                }
        }


        //Helper functions for mergedCommitToDataBase()
        public void getBaselineQuery(Map<Schema.SObjectType, List<sObject>>
commitUpdateMap){
                Schema.DescribeFieldResult fd;
                for (Schema.SObjectType eachType : commitUpdateMap.keyset()){
                        sObject[] objList = commitUpdateMap.get(eachType);
                        for(sObject sObj : objList){
                                String qryFields = '';
                                if
(qryFieldBaselineMap.containskey(sObj.getSobjectType()))
                                        qryFields =
qryFieldBaselineMap.get(sObj.getSobjectType()) + ',';

                                Schema.SObjectType sobjectType =
sObj.getSObjectType();
```

```
                                Map<String, Schema.SObjectField> fMap =
Schema.getGlobalDescribe().get(String.valueOf(sobjectType).toLowerCase()).getDescribe(
).Fields.getMap();
                                        for(Schema.SObjectField ft : fMap.values()){

                                    fd = ft.getDescribe();

                                        Boolean sObjQueried = false;
                                                String fieldName = '';
                                        if(fd.isUpdateable()){
                                                fieldName = fd.getName();

                                                try{ String sObjMasterValue =
String.valueOf(sObj.get(fieldName)); sObjQueried = true; }catch(Exception e){
sObjQueried = false; }
                                        }

                                        if(sObjQueried && !qryFields.contains(fieldName)){
                                                //this builds up a list of field names that need to
be included in the baseline soql
                                                qryFields += fieldName + ',';
                                        }
                                        }
                                        qryFields = qryFields.removeEnd(',');

                                        qryFieldBaselineMap.put(sObj.getSobjectType(),
qryFields);

                                        if
(!qryWhereClauseBaselineMap.containskey(sObj.getSobjectType()))
qryWhereClauseBaselineMap.put(sObj.getSobjectType(), new ID[]{});

        qryWhereClauseBaselineMap.get(sObj.getSobjectType()).add(sObj.id);
                                }
                        }

                        baseLineSObjs = new Map<Id,sObject>();

                        for (Schema.SObjectType eachType : qryFieldBaselineMap.keyset()){
                                Schema.DescribeSObjectResult r = eachType.getDescribe();
                                String qry = 'Select ' + qryFieldBaselineMap.get(eachType) + '
From ' + r.getName();
                                ID[] qryIds;
                                if (qryWhereClauseBaselineMap.containskey(eachType)) {
                                        qryIds = qryWhereClauseBaselineMap.get(eachType);
                                        qry += ' Where Id In :qryIds';
                                }
                                baseLineSObjs.putall(Database.query(qry));
                        }
        }

        public void insertOrdering(Debug_Error__c[] errs){
                //groups the orders per Sobject type because you cannot insert different
Sobject types at the same time eg: Account and Contact
                if (!commitInsertList.isempty()){
                        Map<Schema.SObjectType, List<sObject>> commitInsertMap =
new Map<Schema.SObjectType, List<sObject>>();
```

```
                        // Loop variables init
                        Schema.SObjectType sObjType;

                        // INSERT PREPARE - order all the commitInsertList sObjs
                        for(sObject sObj : commitInsertList){

                                sObjType = sObj.getSObjectType();

                                if(commitInsertMap.containsKey(sObjType)){

                                        commitInsertMap.get(sObjType).add(sObj);
                                }else{

                                        commitInsertMap.put(sObjType, new
List<sObject>{sObj});
                                }
                        }

                        // INSERT
                        for(Schema.SObjectType sObjectType :
commitInsertMap.keySet()){
                                insertObjects(commitInsertMap.get(sObjectType), true,
defaultBatchQuantity, errs, false, TRIGGER_NONE);
                        }
                }
        }

        public void deleteOrdering(Debug_Error__c[] errs){
                if (!commitDeleteList.isempty()){
                        Map<Schema.SObjectType, List<sObject>> commitDeleteMap =
new Map<Schema.SObjectType, List<sObject>>();
                        Schema.SObjectType sObjType;

                        // DELETE PREPARE - order all the commitDeleteList sObjs
                        for(sObject sObj : commitDeleteList){

                                if(commitDeleteMap.containsKey(sObjType)){

                                        commitDeleteMap.get(sObjType).add(sObj);
                                }else{

                                        commitDeleteMap.put(sObjType, new
List<sObject>{sObj});
                                }
                        }

                        // DELETE
                        for(Schema.SObjectType sObjectType :
commitDeleteMap.keySet()){
                                deleteObjects(commitDeleteMap.get(sObjectType), true,
defaultBatchQuantity, errs, false, TRIGGER_NONE);
                        }
                }
        }

        public sObject mergeSObject(sObject sObjMaster, sObject sObj, sObject
baseLineSObj, Boolean ignoreBaseLineSObj){
```

```
                //sObjMaster = Record to be changed
                //sObj = Previous changed record held in memory
                //baseLineSObj = original state of record already saved before trigger
starts

                //If the field being changed exists in sObjMaster, then regardless if it
exists in sObj or not the value in sObjMaster is used and overwrites in sObj
                //If the field does not exist in sObjMaster but does in sObj and the
baseline has a value and is different to the baseline then sObjMaster is updated with the
value in sObj
                //If the field does not exist in sObjMaster and not in baseline but does in
sObj then sObjMaster is updated with the value in sObj

                Schema.SObjectType sobjectType = sObjMaster.getSObjectType();
        Map<String, Schema.SObjectField> fMap =
Schema.getGlobalDescribe().get(String.valueOf(sobjectType).toLowerCase()).getDescribe(
).Fields.getMap();

                // Loop variables init
        String sObjMasterValue;
        Boolean sObjMasterQueried;
        String baseLineSObjValue;
        Boolean baseLineSObjQueried;
           String sObjValue;
           Boolean sObjQueried;
           String fieldName;
           Schema.DescribeFieldResult fd;

                for(Schema.SObjectField ft : fMap.values()){

           fd = ft.getDescribe();


           if(fd.isUpdateable()){
                fieldName = fd.getName();
                 try{ sObjMasterValue = String.valueOf(sObjMaster.get(fieldName));
sObjMasterQueried = (sObjMasterValue != null);}catch(Exception e){ sObjMasterQueried
= false; sObjMasterValue = null;
}
                try{ sObjValue = String.valueOf(sObj.get(fieldName)); sObjQueried =
(sObjValue != null);}catch(Exception e){ sObjQueried  = false; sObjValue = null; }
                try{ baseLineSObjValue = String.valueOf(baseLineSObj.get(fieldName));
baseLineSObjQueried = (baseLineSObjValue != null);}catch(Exception e){
baseLineSObjQueried= false; baseLineSObjValue = null;
}

                                        if (!sObjMasterQueried && sObjQueried){
                                                if (baseLineSObjQueried){
                                                        if(equals(sObjValue,
baseLineSObjValue, fd.getType().Name()) == false)

           updateField(sObjMaster,fieldName, sObjValue, fd.getType().Name());
                                                }
                                                else

           updateField(sObjMaster,fieldName, sObjValue, fd.getType().Name());
                                        }
```

```
        }
                }

                return sObjMaster;
        }

        private static void updateField(sObject sObj, String fieldName, String
newfieldValue, String fieldType){

                if(fieldType == 'DATE'){
                        sObj.put(fieldName, Date.valueOf(newfieldValue));
                }else if(fieldType == 'DATETIME'){
                        sObj.put(fieldName, Datetime.valueOf(newfieldValue));
                }else if(fieldType == 'DECIMAL'){

                        sObj.put(fieldName, Decimal.valueOf(newfieldValue));
                }else if(fieldType == 'INTEGER'){
                        sObj.put(fieldName, Integer.valueOf(newfieldValue));
                }else if(fieldType == 'LONG'){
                        sObj.put(fieldName, Long.valueOf(newfieldValue));
                }else if(fieldType == 'DOUBLE'){
                        sObj.put(fieldName, Double.valueOf(newfieldValue));
                }else if(fieldType == 'BOOLEAN'){
                        sObj.put(fieldName, (newfieldValue.toUpperCase() == 'TRUE'));
                }else{// String, picklist
                        sObj.put(fieldName, newfieldValue);
                }
        }

        private static Boolean equals(String fieldValue1, String fieldValue2, String
fieldType){

                if(fieldValue1 == null && fieldValue2 == null){
                        return true;
                }else if(fieldValue1 != null && fieldValue2 == null || fieldValue1 ==
null && fieldValue2 != null){
                        return false;
                }else if(fieldType == 'DATE'){
                        return Date.valueOf(fieldValue1) == Date.valueOf(fieldValue2);
                }else if(fieldType == 'DATE'){
                        return Date.valueOf(fieldValue1) == Date.valueOf(fieldValue2);
                }else if(fieldType == 'DATETIME'){
                        return Datetime.valueOf(fieldValue1) ==
Datetime.valueOf(fieldValue2);
                }else if(fieldType == 'DECIMAL'){
                        return Decimal.valueOf(fieldValue1) ==
Decimal.valueOf(fieldValue2);
                }else if(fieldType == 'INTEGER'){
                        return Integer.valueOf(fieldValue1) ==
Integer.valueOf(fieldValue2);
                }else if(fieldType == 'LONG'){
                        return Long.valueOf(fieldValue1) == Long.valueOf(fieldValue2);
                }else if(fieldType == 'DOUBLE'){
                        return Double.valueOf(fieldValue1) ==
Double.valueOf(fieldValue2);
                }else if(fieldType == 'BOOLEAN'){
```

```
                return fieldValue1.toUpperCase() == fieldValue2.toUpperCase();
            }else{// String, picklist
                return fieldValue1 == fieldValue2;
            }
        }
}
```

UtilDML is used to provide Monitoring part of the framework

```
public class UtilDML {

    public static list<Debug_Error__c>
returnDMLErrorRecords(Database.Saveresult[] Saveres, String[] errorLst){
            list<Debug_Error__c> errors = new list<Debug_Error__c>();

    for(Database.SaveResult sr : Saveres) {
        if (sr.isSuccess()) {
            // Operation was successful, so get the ID of the record that was processed
                }
        else {
            // Operation failed, so get all errors
            for(Database.Error err : sr.getErrors()) {
                Debug_Error__c e = new Debug_Error__c();
                e.Error__c = err.getStatusCode() + ': ' + err.getMessage();
                if (errorLst != null) errorLst.add(e.Error__c);//this is optional
                errors.add(e);
            }
        }
    }

    return errors;
        }

    public static list<Debug_Error__c>
returnDMLErrorRecords(Database.Upsertresult[] Saveres, String[] errorLst){
            list<Debug_Error__c> errors = new list<Debug_Error__c>();

    for(Database.Upsertresult sr : Saveres) {
        if (sr.isSuccess()) {
            // Operation was successful, so get the ID of the record that was processed
            System.debug('Successfully inserted record: ' + sr.getId());
        }
        else {
            // Operation failed, so get all errors
            for(Database.Error err : sr.getErrors()) {
                Debug_Error__c e = new Debug_Error__c();
                e.Error__c = err.getStatusCode() + ': ' + err.getMessage();
                if (errorLst != null) errorLst.add(e.Error__c);//this is optional
                errors.add(e);
            }
        }
```

```
    }

    return errors;
      }

      public static list<Debug_Error__c>
returnDMLErrorRecords(Database.Deleteresult[] Delres, String[] errorLst){
            list<Debug_Error__c> errors = new list<Debug_Error__c>();

    for(Database.Deleteresult sr : Delres) {
      if (sr.isSuccess()) {
        // Operation was successful, so get the ID of the record that was processed
        System.debug('Successfully inserted record: ' + sr.getId());
      }
      else {
        // Operation failed, so get all errors
        for(Database.Error err : sr.getErrors()) {
          Debug_Error__c e = new Debug_Error__c();
          e.Error__c = err.getStatusCode() + ': ' + err.getMessage();
          if (errorLst != null) errorLst.add(e.Error__c);//this is optional
          errors.add(e);
        }
      }
    }

    return errors;
      }

      public static list<Debug_Error__c>
returnDMLErrorRecords(Database.Undeleteresult[] Delres, String[] errorLst){
            list<Debug_Error__c> errors = new list<Debug_Error__c>();

    for(Database.Undeleteresult sr : Delres) {
      if (sr.isSuccess()) {
        // Operation was successful, so get the ID of the record that was processed
        System.debug('Successfully inserted record: ' + sr.getId());
      }
      else {
        // Operation failed, so get all errors
        for(Database.Error err : sr.getErrors()) {
          Debug_Error__c e = new Debug_Error__c();
          e.Error__c = err.getStatusCode() + ': ' + err.getMessage();
          if (errorLst != null) errorLst.add(e.Error__c);//this is optional
          errors.add(e);
        }
      }
    }

    return errors;
      }


}
```

The UtilDML class requires 2 objects (you can name the objects differently if you want but you will need change the code where they are referenced)

DebugParent__c

Field Name	Data Type
Run_Category__c	Text(100)

Debug_Error__c

Field Name	Data Type
DebugParent__c	Lookup to Debug_Parent__c object
Error__c	Text(1000)

UtilsMonitoring is also used to provide the Monitoring area of the framework

This class requires 2 Custom Settings

Monitoring__c (hierarchy)

Field Name	Data Type
Active__c	Boolean
Monitor_Datetime_From__c	Datetime
Monitor_Datetime_To__c	Datetime

MonitoringCoverage__c (hierarchy)

Field Name	Data Type
value__c	Boolean

```
public with sharing class UtilsMonitoring {
//Monitoring variables
public static Set< String > monitorSet;
public static Map<System.Type, Map<String, list<String>>>
saveMonitoringMessagesMap;

	public static Boolean getMonitoringCoverage(){
		//if this is set to true then monitoring will output everything held in
saveMonitoringMessagesMap otherwise it will revert to whatever is set to the default
section to monitor
		//eg UtilsMonitoring.saveMonitoringMesages(Account.class); will only save
messgaes generated whereby the key is Account.class, but if the CS is True it will just
save everything in saveMonitoringMessagesMap
		return (((MonitoringCoverage__c.getOrgDefaults() != null) ?
MonitoringCoverage__c.getOrgDefaults().value__c : false) ||
MonitoringCoverage__c.getInstance(UserInfo.getUserId()).value__c  ||
MonitoringCoverage__c.getInstance(UserInfo.getProfileId()).value__c) ;
	}

	public static void setupMonitoring(){
		//builds monitorSet of Monitoring that has been activated by the CS

		if (monitorSet == null || monitorSet.isempty()){
			Map<String, Monitoring__c> mon = Monitoring__c.getAll();
			if (!mon.isempty()){
				monitorSet = new Set< String >();
				for (Monitoring__c thisMon: mon.Values()){
					if ((thisMon.Active__c &&
thisMon.Monitor_Datetime_From__c == null && thisMon.Monitor_Datetime_To__c ==
null) || (thisMon.Active__c && thisMon.Monitor_Datetime_From__c >= Datetime.Now()
&& thisMon.Monitor_Datetime_To__c <= Datetime.Now())){
						if (thisMon.Name != null)
monitorSet.add(thisMon.Name);
					}
				}

				if (!monitorSet.isempty() && saveMonitoringMessagesMap
== null){
					saveMonitoringMessagesMap = new
Map<System.Type, Map<String, list<String>>>();
				}
			}
		}
	}

	public static void buildMonitoringMessage(System.Type objMonitor, String ref,
String msg){
	//if getMonitoringCoverage() return true all monitoring messages are saved,
otherwise if the Type is Active specified in setupMonitoring()

		if (getMonitoringCoverage() || (monitorSet != null &&
!monitorSet.isempty() && monitorSet.contains(ref))){
			if (saveMonitoringMessagesMap != null &&
saveMonitoringMessagesMap.containskey(objMonitor)){
```

```
                                    if
(saveMonitoringMessagesMap.get(objMonitor).containskey(ref))

        (saveMonitoringMessagesMap.get(objMonitor).get(ref)).add(msg);
                                    else{

        saveMonitoringMessagesMap.get(objMonitor).put(ref, new list<String>{msg});
                                    }

                                }
                                else{
                                    saveMonitoringMessagesMap.put(objMonitor, new
Map<String, list<String>>{ref => new list<String>{msg}});

                                }
                            }
                        }

        public static void saveMonitoringMesages(System.Type objMonitor){
                    //saves only a particular type of monitoring, so you can save only debug
log coming from Trigger using Account.class, if you want to output only from a class called
XYZ you will specify XYZ.class instead


                if (saveMonitoringMessagesMap != null &&
saveMonitoringMessagesMap.containskey(objMonitor)){
                            if (Limits.getDmlStatements() <=
(Limits.getLimitDmlStatements() -2)){

                                    Map<String, list<String>> saveMsgs =
saveMonitoringMessagesMap.get(objMonitor);
                                    if (saveMsgs.keyset().size() < Limits.getLimitDmlRows()){

                                            DebugParent__c[] newDbgParents = new
DebugParent__c[]{};
                                            for (String kys : saveMsgs.keyset())
                                                    newDbgParents.add(new
DebugParent__c(Run_Category__c = kys));

                                            insert newDbgParents;

                                            Map<String,id> debugParentMap = new
Map<String,id>();
                                            for (DebugParent__c ErrParent : newDbgParents)

        debugParentMap.put(ErrParent.Run_Category__c, ErrParent.id);

                                        Debug_Error__c[] allErrs = new Debug_Error__c[]{};
                                    for (String ky : saveMsgs.keyset()){
                                        for (String msg : saveMsgs.get(ky)){
                                            if (allErrs.size() < Limits.getLimitDmlRows())

                                                allErrs.add(new
Debug_Error__c(Error__c=msg, Debug_Parent__c = (debugParentMap.get(ky) != null) ?
debugParentMap.get(ky) : null));
                                        }
                                    }
```

```
                                if (!allErrs.isempty())
                                    Database.Saveresult[] res = Database.insert( allErrs ,
false);

                                saveMonitoringMessagesMap = new Map<System.Type,
Map<String, list<String>>>();
                                    }
                            }

                    }
            }

            public static void saveMonitoringMesages(){
                    //no monitor type passed to function so this will output all monitoring
messages that has been accepted in buildMonitoringMessage() using
getMonitoringCoverage()  and monitorSet


                if (saveMonitoringMessagesMap != null){
                        if (Limits.getDmlStatements() <=
(Limits.getLimitDmlStatements() -2)){
                                integer imap = 0;
                                Map<integer, DebugParent__c> newDbgParents = new
Map<integer, DebugParent__c>();
                                Map<integer, Debug_Error__c[]> allErrs = new
Map<integer, Debug_Error__c[]>();

                                for (System.Type eachType :
saveMonitoringMessagesMap.keyset()){
                                        Map<String, list<String>> saveMsgs =
saveMonitoringMessagesMap.get(eachType);
                                        for (String ky : saveMsgs.keyset()){
                                                newDbgParents.put(imap, new
DebugParent__c(Run_Category__c = ky));

                                        for (String msg : saveMsgs.get(ky)){
                                                if (allErrs.size() < Limits.getLimitDmlRows()) {

                                                        if (!allErrs.containskey(imap))
                                                                allErrs.put(imap, new
Debug_Error__c[]{});

                                                        allErrs.get(imap).add(new
Debug_Error__c(Error__c=msg));
                                                }
                                        }
                                        ++imap;
                                }
                                }

                                insert newDbgParents.values();

                                if (!allErrs.isempty()){
                                        Debug_Error__c[] totalErrors = new
Debug_Error__c[]{};
                                        for (integer thisIMap : allErrs.keyset()){
```

```
                                                DebugParent__c dP =
newDbgParents.get(thisIMap);
                                                for (Debug_Error__c er :
allErrs.get(thisIMap)){
                                                        er.Debug_Parent__c = dP.id;
                                                        totalErrors.add(er);
                                                }
                                        }

                                    if (Limits.getDmlStatements() <=
(Limits.getLimitDmlStatements() - 1))
                                        Database.Saveresult[] res = Database.insert(
totalErrors , false);
                                        else{
                                                if (!System.isBatch() &&
!System.isFuture()) Database.executebatch(new
batchProcessDMLConsolidation(totalErrors, 'INSERT'));
                                        }
                                }

                            saveMonitoringMessagesMap = new Map<System.Type,
Map<String, list<String>>>();
                        }
                }
        }
}
```

batchProcessDMLConsolidation provides a generic batch for the DML_Master class to utilise

```
global class batchProcessDMLConsolidation implements Database.Batchable<sObject>
{
        global Sobject[] allSobj;
        global String dmlType;

        global batchProcessDMLConsolidation(Sobject[] thisSobj, String thisdmlType){
                allSobj = thisSobj;
                dmlType = thisdmlType;
        }

        global list<Sobject> start(Database.BatchableContext BC) {
                return allSobj;
        }

        global void execute(Database.BatchableContext BC, list<Sobject> sobj) {
                Database.saveresult[] Saveres;
                if (dmlType == 'INSERT')
                        Saveres = Database.insert(sobj,false);
                else if (dmlType == 'UPDATE')
                        Saveres = Database.update(sobj,false);
                else if (dmlType == 'DELETE')
                        Database.Deleteresult[] Delres = Database.delete(sobj,false);
                else{
```

```
        }
        }

        global void finish(Database.BatchableContext BC) {
        }
}
```

DML_Master provides many features to make DMLs safe. This class will be heavily utilised throughout your system as a helper class.

```
public with sharing abstract class DML_Master {
//the functions below will check governor limits and if they are going to be exceeded it will automatically check if the user has allowed the records to be processed by batch
//if processByBatch=true and the context is currently not in a batch or a future the records will be processed by a batch and the user can specify a batch quantity
//otherwise if limits are not going to be exceeded and the user has passed Debug_Error__c[] to the function then the records will be processed by Saveresult
//any errors are returned by the function or if saveErrors=True then the erors are saved to the database directly from the function
//only merge and convertlead are not included but these are not used that often and merge will require at least 2 separate objects to be passed which doesnt really work for the structure of this framework

private enum dmlType {INSERT_OBJECT,UPDATE_OBJECT,UPSERT_OBJECT,DELETE_OBJECT,UNDELETE_OBJECT}
public static integer defaultBatchQuantity = 100;
public static final String TRIGGER_NONE = 'None';
public static final String TRIGGER_ALL = 'ALL';
public static final String TRIGGER_INSERT = 'INSERT';
public static final String TRIGGER_UPDATE = 'UPDATE';
public static final String TRIGGER_DELETE = 'DELETE';
public static final String TRIGGER_UNDELETE = 'UNDELETE';
public static Debug_Error__c[] masterErrors;

public static boolean insertObjects(Sobject[] sobj){
        //this is the default insert function which allows the user to pass fewer arguments
        masterErrors = new Debug_Error__c[]{};
        return genericDML(sobj, true, defaultBatchQuantity, masterErrors, true, dmlType.INSERT_OBJECT, 'Insert Objects', TRIGGER_NONE);
}

public static boolean insertObjects(Sobject[] sobj, Boolean processByBatch, Integer batchQuantity, Debug_Error__c[] errors, Boolean saveErrors, String trigOff){
        return genericDML(sobj, processByBatch, batchQuantity, errors, saveErrors, dmlType.INSERT_OBJECT, 'Insert Objects', trigOff);
}

public static boolean updateObjects(Sobject[] sobj){
        masterErrors = new Debug_Error__c[]{};
```

```
        return genericDML(sobj, true, defaultBatchQuantity, masterErrors, true,
dmlType.UPDATE_OBJECT, 'Update Objects', TRIGGER_NONE);
}

public static boolean updateObjects(Sobject[] sobj, Boolean processByBatch, Integer
batchQuantity, Debug_Error__c[] errors, Boolean saveErrors, String trigOff){
        return genericDML(sobj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.UPDATE_OBJECT, 'Update Objects', trigOff);
}

public static boolean deleteObjects(Sobject[] sobj){
        masterErrors = new Debug_Error__c[]{};
        return genericDML(sobj, true, defaultBatchQuantity, masterErrors, true,
dmlType.DELETE_OBJECT, 'Delete Objects', TRIGGER_NONE);
}

public static boolean deleteObjects(Sobject[] sobj, Boolean processByBatch, Integer
batchQuantity, Debug_Error__c[] errors, Boolean saveErrors, String trigOff){
        return genericDML(sobj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.DELETE_OBJECT, 'Delete Objects', trigOff);
}

public static boolean undeleteObjects(Sobject[] sobj){
        masterErrors = new Debug_Error__c[]{};
        return genericDML(sobj, true, defaultBatchQuantity, masterErrors, true,
dmlType.UNDELETE_OBJECT, 'UnDelete Objects', TRIGGER_NONE);
}

public static boolean undeleteObjects(Sobject[] sobj, Boolean processByBatch, Integer
batchQuantity, Debug_Error__c[] errors, Boolean saveErrors, String trigOff){
        return genericDML(sobj, processByBatch, batchQuantity, errors, saveErrors,
dmlType.UNDELETE_OBJECT, 'UnDelete Objects', trigOff);
}

public static boolean genericDML(Sobject[] sobj, Boolean processByBatch, Integer
batchQuantity, Debug_Error__c[] errors, Boolean saveErrors, dmlType thisDMLType,
String setErrorMsg, String trigOff){
        try{
                UtilsMonitoring.setupMonitoring();
                Boolean processMonitoringMsg = false;

                Type convertedType =
Type.forname(String.valueOf(sobj[0].getSobjectType()));
                if (trigOff != TRIGGER_NONE){
                        //this allows you to bypass triggers to improve efficiency of
processing
                        if (trigOff == TRIGGER_ALL)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_NONE, true);
                        else if (trigOff == TRIGGER_INSERT)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_INSERT, true);
                        else if (trigOff == TRIGGER_UPDATE)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_UPDATE, true);
                        else if (trigOff == TRIGGER_DELETE)
```

```
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_DELETE, true);
                        else if (trigOff == TRIGGER_UNDELETE)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_UNDELETE, true);
                }

                if (Limits.getDmlStatements() <= (Limits.getLimitDmlStatements() - 1)){
                        if (errors != null){
                                Database.saveresult[] res;
                                if (thisDMLType == dmlType.INSERT_OBJECT) {
                                        res = Database.insert(sobj,false);
                                        errors.addall(UtilDML.returnDMLErrorRecords(res,
null));
                                }
                                else if (thisDMLType == dmlType.UPDATE_OBJECT){
                                        res = Database.update(sobj,false);
                                        errors.addall(UtilDML.returnDMLErrorRecords(res,
null));
                                }
                                else if (thisDMLType == dmlType.UPSERT_OBJECT){
                                        return false;//NOT ALLOWED ON GENERIC
SOBJECT
                                }
                                else if (thisDMLType == dmlType.DELETE_OBJECT){
                                        Database.Deleteresult[] Upsres =
Database.delete(sobj,false);

        errors.addall(UtilDML.returnDMLErrorRecords(Upsres, null));
                                }
                                else if (thisDMLType == dmlType.UNDELETE_OBJECT){
                                        Database.Undeleteresult[] Upsres =
Database.undelete(sobj,false);

        errors.addall(UtilDML.returnDMLErrorRecords(Upsres, null));
                                }
                        }
                        else{
                                if (thisDMLType == dmlType.INSERT_OBJECT)
                                        insert sobj;
                                else if (thisDMLType == dmlType.UPDATE_OBJECT)
                                        update sobj;
                                else if (thisDMLType == dmlType.UPSERT_OBJECT)
                                        return false;//NOT ALLOWED ON GENERIC
SOBJECT
                                else if (thisDMLType == dmlType.DELETE_OBJECT)
                                        delete sobj;
                                else if (thisDMLType == dmlType.UNDELETE_OBJECT)
                                        undelete sobj;
                        }
                }
                else{
                        if (processByBatch){
                                if (!System.isBatch() && !System.isFuture()) {
                                        if (thisDMLType == dmlType.INSERT_OBJECT)
```

```
                                                Database.executebatch(new
batchProcessDMLConsolidation(sobj, 'INSERT'), (batchQuantity != null) ? batchQuantity :
200);
                                        else if (thisDMLType ==
dmlType.UPDATE_OBJECT)
                                                Database.executebatch(new
batchProcessDMLConsolidation(sobj, 'UPDATE'), (batchQuantity != null) ? batchQuantity :
200);
                                        else if (thisDMLType ==
dmlType.UPSERT_OBJECT)//NOT ALLOWED ON GENERIC SOBJECT
                                                return false;
                                        else if (thisDMLType ==
dmlType.DELETE_OBJECT)
                                                Database.executebatch(new
batchProcessDMLConsolidation(sobj, 'DELETE'), (batchQuantity != null) ? batchQuantity :
200);
                                        else if (thisDMLType ==
dmlType.UNDELETE_OBJECT)
                                                Database.executebatch(new
batchProcessDMLConsolidation(sobj, 'UNDELETE'), (batchQuantity != null) ?
batchQuantity : 200);
                                }else{

        UtilsMonitoring.buildMonitoringMessage(DML_Master.class, setErrorMsg, 'Could
not save records. Processing by batch was not allowed as context is already in batch or a
future.');
                                        processMonitoringMsg = true;

                                }
                        }
                        else{

        UtilsMonitoring.buildMonitoringMessage(DML_Master.class, setErrorMsg, 'Could
not save records. Processing by batch was not allowed.');
                                processMonitoringMsg = true;
                        }
                }

                try{
                        if (saveErrors)
                                insert errors;
                }
                catch(Exception ex){
                        processMonitoringMsg = true;
                        UtilsMonitoring.buildMonitoringMessage(DML_Master.class,
setErrorMsg, 'Could not save error messages.' + ex.getMessage());
                }


                if (trigOff != TRIGGER_NONE){
                        //this resets trigger control
                        if (trigOff == TRIGGER_ALL)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_NONE, false);
                        else if (trigOff == TRIGGER_INSERT)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_INSERT, false);
```

```
                        else if (trigOff == TRIGGER_UPDATE)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_UPDATE, false);
                        else if (trigOff == TRIGGER_DELETE)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_DELETE, false);
                        else if (trigOff == TRIGGER_UNDELETE)
                                TriggerController.setTriggerControlValue(convertedType,
TriggerController.TRIGGER_UNDELETE, false);
                }


                if (processMonitoringMsg){
                        UtilsMonitoring.saveMonitoringMesages(DML_Master.class);
                        return false;
                }
                else
                        return true;
        }
        catch(Exception ex){
                UtilsMonitoring.buildMonitoringMessage(DML_Master.class, setErrorMsg,
setErrorMsg + ' failed ' + ex.getMessage());
                if (saveErrors)
UtilsMonitoring.saveMonitoringMesages(DML_Master.class);

                return false;
        }
}

}
```

Finally, the TriggerFactory glues all of the framework base classes together by using the CommitHandler, UtilsMonitoring, TriggerController and the handler class which we will cover next

```
public with sharing class TriggerFactory {

public class TriggerException extends Exception{}
public static CommitHandler ch;

public static void createHandler(Schema.sObjectType thisobjType, String hdler){
        Schema.DescribeSObjectResult d = thisobjType.getDescribe();

        String thisType = d.getName();
        ITrigger handler = getHandler(hdler);

        if (handler == null)
                throw new TriggerException('No Trigger Handler registered for Object
Type: ' + thisType);

   //setup monitoring
```

```
    UtilsMonitoring.setupMonitoring();

          execute(handler, thisType);

}

private static void execute(ITrigger handler, String objType){

          boolean notriggerSetting;
          boolean noTriggersPerObject;
          try{
              notriggerSetting = TriggerController.globalTriggerControlSetting();
              noTriggersPerObject =
TriggerController.globalTriggerPerObjectControlSetting(objType);
          }
          catch (Exception ex){system.debug('error in trigger controller ' + ex); }

          Type soType = Type.forName(objType);


    if (!notriggerSetting && !noTriggersPerObject &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_ALL)) {

                    if (Trigger.isBefore){
                              if (Trigger.isUpdate &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_UPDATE)){
                                        handler.beforeUpdate(Trigger.old, Trigger.new,
Trigger.oldMap, Trigger.newMap, ch);
                                        TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_UPDATE), true);

                              }
                              else if (Trigger.isDelete &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_DELETE)){
                                        handler.beforeDelete(Trigger.old, Trigger.oldMap,
ch);
                                        TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_DELETE), true);
                              }
                              else if (Trigger.isInsert &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_INSERT)){

                                        handler.beforeInsert(Trigger.new, ch);

                                        TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_INSERT), true);
                              }
                              else if (Trigger.isUnDelete &&
!TriggerController.getTriggerControlValue(soType,
TriggerController.TRIGGER_UNDELETE)){
                                        handler.beforeUnDelete(Trigger.old,
Trigger.oldMap, ch);
                                        TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_UNDELETE), true);
                              }

                    }
```

```
                else{

                        if (Trigger.isUpdate &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_UPDATE)){
                                handler.afterUpdate(Trigger.old, Trigger.new,
Trigger.oldMap, Trigger.newMap, ch);
                                TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_UPDATE), true);
                        }
                        else if (Trigger.isDelete &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_DELETE)){
                                handler.afterDelete(Trigger.old, Trigger.oldMap,
ch);
                                TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_DELETE), true);
                        }
                        else if (Trigger.isInsert &&
!TriggerController.getTriggerControlValue(soType, TriggerController.TRIGGER_INSERT)){
                                handler.afterInsert(Trigger.new,
Trigger.newMap, ch);
                                TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_INSERT), true);
                        }
                        else if (Trigger.isUnDelete &&
!TriggerController.getTriggerControlValue(soType,
TriggerController.TRIGGER_UNDELETE)){
                                handler.afterUnDelete(Trigger.old, Trigger.new,
Trigger.oldMap, Trigger.newMap, ch);
                                TriggerController.triggerSuccessMap.put(new
TriggerControlKeyValue(soType, TriggerController.TRIGGER_UNDELETE), true);
                        }
                }

                ch.mergedCommitToDataBase();
   }

}

private static ITrigger getHandler(String hdler){
        ch = new CommitHandler();

        Type typ = Type.forName(hdler);
        return (ITrigger)typ.newInstance();

}

}
```

That makes up all of the base classes for the framework. Now we will create classes for a specific Trigger that are used in the framework.

Lets say we have an object called Financial_Reconcile__c and we want to do something when records are inserted and updated.

Note you dont need to include all the functions stated in the interface of the TriggerBasehandler

```
public with sharing class FinancialReconcileHandler extends TriggerBaseHandler{

        //the handler passes to each Logic class to perform whatever actions this allows the handler to have multiple logic classes for different purposes and sobj that this class doesnt grow exponentially
    private FinancialReconcileLogic fLogic;

    public FinancialReconcileHandler() {
        //Note can create multiple instances of different classes if required to segregate functionality into appropriate classes
        this.fLogic = new FinancialReconcile Logic();
    }

    private static FinancialReconcileHandler instance;

    public static FinancialReconcileHandler getInstance() {
        if (instance == null) instance = new FinancialReconcileHandler();
        return instance;
    }

public override void afterInsert(Sobject[] newObjects, Map<id,SObject> newmap,
CommitHandler ch){

                    fLogic.onAfterInsert((Financial_Reconcile__c[])newObjects,
newmap.keyset(), ch
}

        public override void afterUpdate(Sobject[] oldnewObjects, Sobject[]
newObjects, Map<Id, Sobject> oldObjectsMap, Map<Id, Sobject> newObjectsMap,
CommitHandler ch){
                    fLogic.onAfterUpdate((Financial_Reconcile__c[])newObjects,
newObjectsMap.keyset(), ch)
        }

}
```

The trigger itself is very lightweight

```
trigger AllFinancialReconcileTrigger on Financial_Reconcile__c (after delete, after
insert, after undelete,
after update, before delete, before insert, before update) {

        TriggerFactory.createHandler(Financial_Reconcile__c.sObjectType,
'FinancialReconcileHandler');
        if (UtilsMonitoring.getMonitoringCoverage())
                UtilsMonitoring.saveMonitoringMesages();
        else
                UtilsMonitoring.saveMonitoringMesages(Financial_Reconcile__c.class);

}
```

Note: If you decide you do not need to output debug messages you do not need to include the last 4 lines of code