

BMC Corporate Directory Manager (CDM) Workflow Design Practices

Reusable concepts for large CDM projects

Srikanth Shenoy

ObjectSource Tech Pvt Ltd.

Contacting ObjectSource

You can access the ObjectSource Tech. Web site at <http://www.objectsource.com>. From this Web site, you can obtain information about the company, its products, corporate offices, special events, and career opportunities.

United States, Canada and India

About the author

Srikanth Shenoy is the founder & CEO of ObjectSource (<http://www.objectsource.com>) with aim of Innovation Driven Objective Outsourcing Delivery

He provides guidance, strategy, architecture and design and implementation of entire IT systems, particularly:

- 1) Applications, Middleware Systems and Portals
- 2) Service Oriented Integration and SOA Migration

He has in-depth experience in architecture, development, integration and deployment of complex enterprise-wide business systems.

In his current position he is focusing on developing several innovative products for Healthcare, Manufacturing, Finance, Transportation and Construction industry using a combination of technologies such as Java, Java EE, Spring, .NET, REST GWT etc.

He also leads a team of highly qualified and motivated staff for outsourced project deliveries.

Previously he was Enterprise Architect at Sun Microsystems helping customers in the areas of Java EE, SOA, Identity and Access Management. focusing on architecture and design of end-to-end SOA systems integrated with Identity and Access Management.

Before that was a Senior Identity Management Consultant at BMC Software.

Prior to that he was solutions architect at JPMorgan Chase and ObjectSeek and has architected several scalable turnkey systems using J2EE, CORBA using commercial and open source application servers and frameworks and has integrated them with various Identity Management products from several vendors.

He is the author of a best selling book on Struts - the leading web development framework in the J2EE world until recently. He has also authored several technical articles related to Java and software development for leading industry forums, websites and journals.

Index

Index	3
1 Workflow Patterns Scope and Prerequisites	4
1.1 Scope	4
1.2 Pre-requisites	4
2 Patterns – Big Picture	5
3 Patterns Catalog	6
3.1 Entity Object DAO	6
3.2 Entity Object	8
3.3 Domain Object.....	10
3.4 Domain Object State.....	13
3.5 Domain Object Assembler	14
3.6 Workflow Facade.....	15
3.7 Workflow Façade Folder Helper	16
3.8 View Helper	19

1 Workflow Patterns Scope and Prerequisites

This section covers the scope of the CDM workflow design patterns document. Understanding of Gof Patterns (<http://www.amazon.com/gp/product/0201633612/>) and Core J2EE Patterns (<http://www.amazon.com/gp/product/0131422464/>) helps in enhancing the understanding of CDM workflow design patterns, but not necessary. At the end of this document, you will know the concepts necessary for designing large workflow with lot of customization so that the design is elegant and the implementation is easily maintainable in the long run.

1.1 Scope

Small to medium workflow implementations do not derive much benefit from using these documented CDM workflow design patterns. Small to medium workflow implementations have very little in terms of custom Java code, very little customizations. A characteristic of these projects is that a majority of effort will be spent in the workflow studio. Any minor customizations can be done by using computed attributes in Resources, variables or creating computed resources, links, pivots etc.

Then there are large projects (large not in terms of just the number of workflows, but in terms of complexity and customization) where having certain amount of reusable Java code is beneficial along with custom attributes and computed resources.

This document is based on my experiences at a reasonably large project where I started the workflow implementation and as it was growing, I started realizing that the Java code needed significant refactoring. The refactoring led me to identify the patterns that would work best in these scenarios. Usage of these patterns will help in

- 1) Make development faster – because it is easier to determine where a piece of code goes, thus promoting reuse of code and when bugs occur, it is easier to drill down to the exact spot of bug.
- 2) Field tested good programming practices that recur as solutions everytime
- 3) Reducing the errors that can happen
- 4) Provide a common terminology
- 5) In the distant future may be useful for pattern based custom object generation via eclipse plugin to make workflow development even faster.

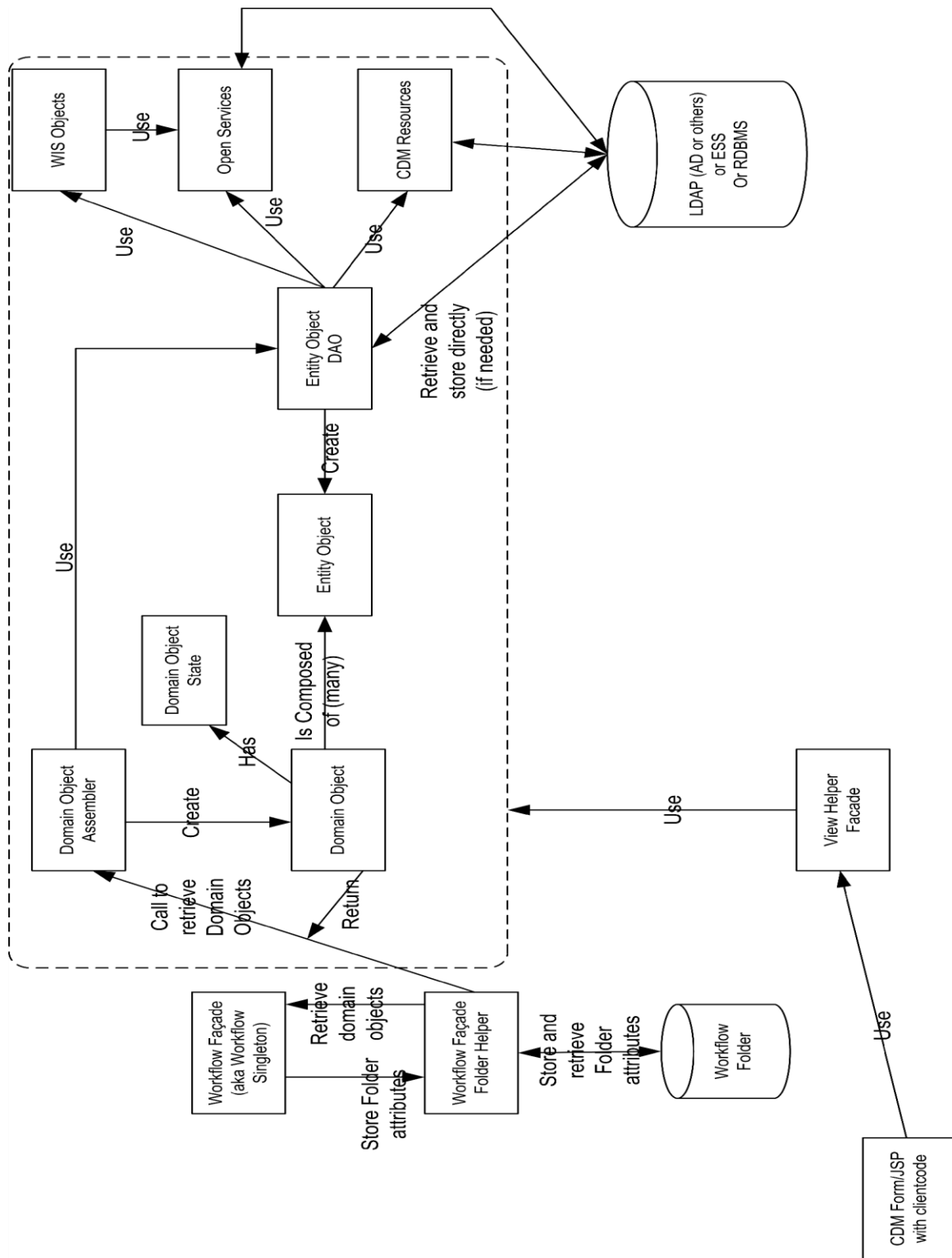
Note that each of the patterns themselves, alone, don't seem to be much and are even common sense. But there is a lot of value when all of them are used in conjunction. It results in a great design. All patterns elaborated here are field tested.

1.2 Pre-requisites

No pre-requisites, although knowledge of OO, CRC, Refactoring, Gof Patterns and J2EE Patterns is helpful.

2 Patterns – Big Picture

The following is the complete set of patterns that I found useful as I was implementing CDM workflow.



3 Patterns Catalog

The cataloging format used is similar to Gof. Each pattern is identified by a name, then its intent, aka, and motivation for the pattern and advantages. You will have keep referring to the previous figure to find its place in the big picture.

3.1 Entity Object DAO

Intent:

To provide a standard object to contain all sorts of similar search. (DAO stands for Data Access Object)

Also Known As:

N/A

Motivation:

Consider a search done using OSPerson as follows:

```
OSCriteria[] criterias = new OSCriteria[1];
criterias[0] = new OSCriteria("user_id",personId,OSCriteria.EQUALS, OSCriteria.NONE);
OSFilter filter = new OSFilter(criterias);
OSPerson person = null;
OSSearchResultItem[] items = null;

OSPerson person = new OSPerson(this.contextId);
OSSearchResultItem[] items = person.customSearch(100, RETURN_FIELDS, filter);
if (items == null || items.length == 0 || items[0] == null)
{
    throw new ObjectNotFoundException("Cannot find record for person id =" + personId...);
}
OSSearchResultItem item = items[0];

String userId = item.getAttributeValue("user_id");
if (userId == null || userId.trim().length() == 0)
{
    throw new InvalidObjectException(null, "OSPerson", "user id is null or empty string in
returned resultitem for criteria " + criterias[0].toString(), ErrorCategory.DATA_ERROR);
}
...
...
```

This sort of search is typical and is often invoked by workflow singleton.

If the code like this was to be put in a single class (called EssPersonDAO) and be invoked from a method named (say findByPersonId), then

- 1) Purpose of the method becomes evident because of the a meaningfully named method for searching on a first class object is more readily understandable instead of just being a method in a workflow singleton doing a random search on some resource.
- 2) Method becomes more reusable as it being advertised on a first class object and would be used from other workflow singletons

Format:

```
public class EssPersonDAO implements IDataAccessObject
{
    private String contextId;

    private static final String ID = "user_id";
    private static final String USER_NAME = "user_name";
    private static final String __99__PE_lastFourSSN = "__99__PE_lastFourSSN";
    private static final String __99__PE_position = "__99__PE_position";
    private static final String __99__PE_departmentName = "__99__PE_departmentName";
    private static final String __99__PE_supervisorEmpID = "__99__PE_supervisorEmpID";

    private static final String[] RETURN_FIELDS = {
        ID, USER_NAME, __99__PE_lastFourSSN,
        __99__PE_position, __99__PE_departmentName,
        __99__PE_supervisorEmpID,
    };

    public EssPersonDAO (String aContextId)
    {
        this.contextId = aContextId;
    }

    public EssPerson findById(String personId) throws FinderException
    {
        OSCriteria[] criterias = new OSCriteria[1];
        criterias[0] = new OSCriteria("user_id", personId, OSCriteria.EQUALS,
        OSCriteria.NONE);
        OSFilter filter = new OSFilter(criterias);
        OSPerson person = null;
        OSSearchResultItem[] items = null;

        OSPerson person = new OSPerson(this.contextId);
        OSSearchResultItem[] items = person.customSearch(100, RETURN_FIELDS,
        filter);
        if (items == null || items.length == 0 || items[0] == null)
        {
            throw new ...
        }
        OSSearchResultItem item = items[0];

        String userId = item.getAttributeValue("user_id");
        if (userId == null || userId.trim().length() == 0)
        {
            throw new ...
        }
        ...
    }
}
```

```

        EssPerson p = new EssPerson(userId, ....);

        return p;
    }
}

```

Advantages:

- 1) Purpose of the method becomes evident because of the a meaningfully named method for searching on a first class object is more readily understandable instead of just being a method in a workflow singleton doing a random search on some resource.

Example:

Consider looking up a person's windows account id from ess. To do this you can use OSAccount and create a search filter and search for rss_type = Win2000, rss_name = ".." ... etc...

Instead this whole method can be put inside a AccountDAO class that has a method called findWindowsAccount(String personId). The input parameter for this method is just the personId since the rest of the details (rss_name, rss_type etc..) required for this search can be maintained internal to the AccountDAO. This makes the caller's life much easier...

- 2) Method becomes more reusable as it being advertised on a first class object and would be used from other workflow singletons
- 3) When the data fetching mechanism and datastore field names are placed in a single place, changes to the data store (schema for example) will be contained within the DAO and not percolate through the entire codebase.
- 4) Since data fetching mechanism is encapsulated in a single class, we now can enforce additional checks in this one place. For instance, if user_id was expected, but was missing, you can throw custom exceptions. If the datastore schema changes it will only impact the DAO and not elsewhere.
- 5) All additional business specific checks on data can be put in this single place in DAO

3.2 Entity Object

Intent:

To provide typed access to a object from LDAP, ESS or RDBMS

Also Known As:

N/A

Motivation:

Consider the search done in the DAO in 4.1.

When the search is done in the DAO, the results are returned as SearchResultItem object, from which the field is retrieved using a name provided as java.lang.String. If an external caller were to obtain the data by calling findByPersonId, then the return value would be a pair of name value pairs. This is not type safe and the name needs to be known by the caller.

Since the data retrieval is anyway encapsulated in the DAO, we can create a first class object for the return value as seen in the DAO code in 4.1 The findByPersonId is returning a object called EssPerson.

Format:

The EssPerson class looks like follows. Notice that the entity object

- a) Implements serializable interface
- b) All parameters are supplied in constructor.
- c) There are getter methods, but no setters. This makes the Entity Object immutable, which is good because it prevents some other code from unknowingly tampering the data.

```
public class EssPerson implements Serializable
{
    private String personId;
    private String personName;
    private String lastFourDigitsOfSSN;
    private String position;
    private String departmentName;
    private String supervisorEmpID;

    public EssPerson(String aPersonId, String aPersonName, String ..... )
    {
        this.personId = aPersonId;
        this.personName = aPersonName;
        ...
    }

    public String getPersonId()
    {
        return this.personId;
    }

    //Other getter methods go here
}
```

Advantages:

- 1) The return value from a search is typed. Hence any problems show up as compile time errors instead of runtime errors or worse as bugs in production.
- 2) The caller of the Entity Object need not know the name of field. Instead they can call getters. Mistake in call will manifest as compilation error and fixed quickly.

Other Advantages and Implementation Notes:

- 1) You will notice that the Entity Object is always created by Entity Object DAO. In other words, you will never have to write a custom search outside the DAO. Any search you write is a candidate method for the corresponding DAO.
- 2) The DAO constructor is always takes the CDM context id. This is pretty much standard.

- 3) In the above example if a caller (workflow singletons or anywhere else) needs the Person information for a given id, then they just call new EssPersonDAO(contextId).findByPersonId(personId);
- 4) In the starting stages of IdM projects, corrupt data is a big issue. The DAO can throw exceptions if it finds problems with data.
- 5) The DAO can also catch data store specific exceptions, log them and then convert them to meaningful business exceptions (if business exceptions are modeled as runtime exceptions, the caller need not even trap any exceptions- making it clean code). For example if the data store under consideration is ESS and if ESS throws ESSServiceException or SQLException (if using direct JDBC), then the DAO can catch those exceptions, log them and rethrow meaningful business exceptions.

3.3 Domain Object

Intent:

To provide an object meaningful to the business logic, encapsulate the core business logic and make callers transparent to the data stores to be linked to create a object relevant to the business logic.

Also Known As:

Business Logic Object

Motivation:

Imagine that a person starts a workflow for changing the current application access. When this happens a typical requirement is also to submit the request to his/her manager for approval, [which in turn requires the manager's dn - to be recipient of the next task].

This information is typically spread across multiple places.

- 1) Get the employee id for the currently logged in person using the DN. This may involve looking up in ESS using OSAccount object in WIS. You can create a entity object dao - RssAccountDAO to wrap the OSAccount [and a finder method – findByWindowsDN(...)] and get a typed object [For instance an entity object called RssAccount] out of it.
- 2) From the employee id you will need to get more detail on the employee from ESS Person record and/or HR.
- 3) Most of the times, the person id is generally the employee id. Using the person id from the RssAccount object, you may have to go to the HR (Oracle, Peoplesoft - rdbms tables) to get the supervisor employee id.
- 4) Then come back to RSS User table to get windows DN for the supervisor

All of these can be pretty involved to repeat again and again for every workflow. So instead of putting these tasks behind a façade and calling the façade from workflow singleton, I found it much useful to construct a domain object and use the resulting domain object in workflow singleton.

This derives inspiration from Domain Driven Programming (where the heart of logic is captured within domain object and uses CRC), instead of task driven programming which is so much used in the name of OO today. For more info on domain driven programming – refer to Domain Driven Design by Eric Evans (<http://www.amazon.com/gp/product/0321125215/>).

Format:

In this case, we can use a façade object (which I call with a fancy name – Domain Object Assembler) to construct a Domain Object by implementing the logic that I covered above for a person's data. To address the scenario above, I construct a domain object called EnterprisePerson as follows.

```
public class EnterprisePerson implements Serializable
{
    private EssPerson essPerson;
    private RssAccount windowsAccount;
    private EnterprisePerson supervisor;

    private String[] subordinatelds;

    public EssPerson(EssPerson p, RssAccount rssAccount.....)
    {
    }

    public String getPersonId()
    {
        return essPerson.personId;
    }

    public String getWindowsDN()
    {
        return windowsAccount.dn;
    }

    //Other getter methods go here

    public EnterprisePerson getSupervisor()
    {
        return supervisor;
    }

    ...
}
```

Note that:

- 1) The domain object can have setters. There is no need for it to be immutable.
- 2) But I like it to keep immutable and create a Updateable Domain Object on demand [I wont cover it here.. but the idea is to create a sub class with setter methods that internally uses cglib (<http://cglib.sourceforge.net/>) to track which setters were called and accordingly generate the update instructions to cdm, wis etc.... This is part of my vision and have not implemented. Didn't see value in doing this from scratch for the client.]
- 3) You will see more value of this pattern in the next example below.

Consider a new hire workflow with the following rules:

- 1) Normally HR puts new hire records in HR database. A nightly batch job diff's ESS with HR and inserts new records into ESS. The new record in ESS is only the Enterprise User record.

- 2) Sometimes HR does not have record and the employee/contractor is already in. In this case, the user is not defined in ESS at all.
- 3) CDM new hire workflow is used to request AD account, mailbox and some profiles and out of profile apps.
- 4) The requestor is unaware of the fact if there is an ESS record or not. They always use the same workflow
- 5) Mails have to be sent out to the manager, requestor, app approver and custodian at appropriate stages. The messages have to be context sensitive.

These requirements sound pretty typical. Let's see how Domain Object helps here. Consider a Domain Object called NewHire

```
public class NewHire {

    // these values come straight from what was entered on the screen
    // These are also stored in workflow folders, but put here by Workflow Façade Helper

    private String firstNameEntered;
    private String lastNameEntered;

    //These are object representations for selected persons on new hire screens

    private EnterprisePerson existingRecord;

    private EnterprisePerson supervisorSelected;

    private EnterprisePerson hrRepSelected;

    ..
    ..

    private NewHireStatus newHireStatus;

    private boolean userExistsInESS(..) { }

    public void sendMailToRequestor ..(..) { }

    public void sendMailToCustodian(...) { }

}
```

When the initiator submits the new hire form, the workflow singleton can now create the NewHire object and let the NewHireObject take care of handling the intricacies of business logic of

- a) when the user should be created
- b) when the AD account should be created and linked to ESS person
- c) What happens when a matching user is found in ESS
- d) Is the user a duplicate
- e) Should the record be merged with existing record in ESS
- f) What mail should be sent to initiator, what are the contents of that email

Things like these are very much the core of the workflow logic and do not belong in Workflow Singleton, or elsewhere. They deserve their own type of object capturing the domain logic with the domain data inside Domain Object.

Now the job in workflow singleton becomes very easy. All that needs to be done in singleton is create the NewHire object, then call save on that new hire object and finally call sendMailTo.... on the same object. In the absence of the domain objects, the singleton will be bogged down by organization of the workflow tasks and execution of logic workflow tasks.

This is much similar to a organization structures. Managers don't perform details. They delegate, but organize things. A workflow singleton is like a manager. It should organize things, delegate and control them and execute them one after another. But the details of the trenches are better left to Domain objects – the worker bees in this case.

3.4 Domain Object State

Intent:

To provide the context of the domain object.

Also Known As:

State

Motivation:

In a typical workflow, emails have to be sent at different points in the workflow. The content of the emails should be context sensitive. The Domain Object State maintains the state of the Domain Object and provides the context sensitive email contents based on the state.

In the Domain Object NewHire, you saw a reference to a class called NewHireStatus. This is the Domain Object State object. The definition of the NewHireStatus is as follows

Format:

```
public class NewHireStatus
{

//Create a state object for all possible states
public NewHireStatus NON_EXISTENT_USER = new NewHireStatus(msg1, msg2, msg3.....);

public NewHireStatus USER_WITH_ESS_RECORD = new NewHireStatus(msg1, msg2...);

public NewHireStatus USER_WITH_ESS_AND_AD_RECORD = new NewHireStatus(....)

//Constructor is private
private NewHireStatus(....) { ....}

}
```

When the workflow singleton creates the NewHire object, it immediately uses the Entity ObjectDAOs to fetch the data from the data store and make a decision regarding current status. This status becomes the Domain Object State instance variable in NewHire object. When the mail has to be sent to the appropriate people at any time, the methods on NewHire is used as follows

```

public void sendMailToInitiator()
{

String messageBody = newhireStatus.getMessage1();
Wfmail.notifyInitiator(.....)
}

```

Notice that the contents of the mail don't have to be built case by case. Instead identify the state and store them statically within the state object. Just use that from the state object when the time comes to send mail. This will save your objects from being cluttered with if elses while creating context sensitive messages in workflow.

Advantages and Implementation Notes:

- 1) The state object is useful not only when sending context sensitive emails. It can also be used to statically capture all context sensitive data.
- 2) Since all possible runtime states are captured statically during design within the State object, possibility of errors decrease drastically
- 3) Code is less cluttered and becomes more elegant and readable and maintainable

3.5 Domain Object Assembler

Intent:

To construct the domain object.

Also Known As:

Domain Object DAO

Motivation:

In section 4.3, we saw that there is a logic behind creation of domain object. This logic is mostly about how to compose the domain object itself. This logic is of the format of "Use this Entity Object DAO and get this entity object." etc... This is done for several Entity DAO before the Domain Object is completely composed. This Domain Object composition (or assembling) logic is captured in the Domain Object Assembler.

Domain Object Assembler takes two forms.

One that is explicitly called by callers to construct the Domain Object. These are mostly of the format:

```
new EnterprisePersonAssembler(contextId).constructEnterprisePerson(initiatorDn)
```

or they can be internal to the Domain Object itself. In this flavor they play a helper role during composition. They are not directly called by the callers. Instead the caller says

```
new EnterprisePerson(contextId, initiatorDn);
```

The domain object assembler is used internally to query relevant Entity Object DAOs to populate the Domain Object.

Format:

Flavor 1:

```
public class EnterprisePersonAssembler
{
    public EnterprisePersonAssembler(String contextId) {...}

    public EnterprisePerson constructEnterprisePerson(String dn)
    {....
        ..
        ..
        return new EnterprisePerson(essPerson, windowsAccount..);
    }
}
```

Flavor 2:

```
public class EnterprisePerson implements Serializable
{
    private EssPerson essPerson;
    private RssAccount windowsAccount;
    private EnterprisePerson supervisor;

    private String[] subordinatelds;

    public EssPerson(String dn)
    {
        EssPersonHelper.getthis(...)
        EssPersonHelper.getThat(..)
    }
}
```

Notice the difference in the EnterprisePerson constructor in the two flavors of assembler used.

3.6 Workflow Facade

Intent:

To provide a façade behind which all fine grained access and logic can be hidden

Also Known As:

This is same as what is otherwise called Workflow Singleton. (Calling it a singleton is misnomer. Just put a log statement in the constructor and see how many times it is called even within the context of a single task execution of a workflow – especially in the workflow init task !!!! I made the mistake in the very beginning of adding instance variables within the singleton

assuming that they will be retained as is in the workflow instance and found that they were losing their value every time.)

Motivation:

The new name is just the indication of what this class is really meant to be. Nothing special here.

More details on combined usage of Workflow Façade and Workflow Façade Folder Helper are available below

3.7 Workflow Façade Folder Helper

Intent:

To isolate the access to folder and hide the folder attribute values from propagating thru the system and instead only expose first class domain objects or entity objects to its callers (workflow facades).

Also Known As:

N/A

Motivation:

Consider how a folder value is retrieved in a workflow singleton.

```
String cn = (String) Folder.getAttributeValue("wfkView", "aADUsers0", "cn");
```

There are two problems here. But first the obvious one. The names are hard coded. The simplest way to address it is to declare the folder entry and attribute names as public static final Strings. But that just solves the tip of the iceberg.

Lets say in a particular task of a workflow, you put in a value and later down the line try to retrieve that folder value. You'd probably do it as follows:

```
public class MyWorkflowSingleton extends WfCustomSingleton
{
    public static final String x = "...";
    public static final String y = "...";
    public static final String z = "...";

    public void handleTask1(WfFolder folder)
    {
        //.. some logic goes here
        folder.setAttributeValue(x, y z, "blah");
        folder.write();
    }

    public void handleTaskX(WfFolder folder)
    {
```

```

        //.. some logic goes here
        folder.getAttributeValue(x, y z);
    }
}

```

Now consider what happens if task1 was within a branch and another task in the parallel branch never set this value. When you try to retrieve the value later in task X, you will get a exception. This is just one example.

Another manifestation of direct usage of folder.get(...) and folder.set(...) is task driven programming which clutters the logic everywhere.

We can use this interesting pattern called Workflow Façade Folder Helper to move from task driven programming to domain driven programming,

Below is an example of Workflow Façade Folder Helper

Format:

```

public class MySingletonFolderHelper
{
    public static final String x = "...";
    public static final String y = "...";
    public static final String z = "...";

    public static NewHire extractNewHireFromFolder(WfFolder folder)
    {
        String firstName = getFirstNameFromFolder(folder);
        String lastName = getLastNameFromFolder(folder);
        String reqDN = getRequestorDNFromFolder(folder);
        String managerId = getmanagerIdFromFolder(folder);
        ..
        NewHireObjectAssembler assembler = new NewHireObjectAssembler(...);
        assembler.setInitiator(reqDN);
        assembler.setManagerId(managerId);
        ..
        ..
        NewHire newHire = assembler.assemble();
        newHire.initialize();
        return newHire;
    }

    private static String getFirstNameFromFolder(WfFolder folder)
    {
        return (String) folder.getAttributeValue(x, y z);
    }

    private static String getLastNameFromFolder(WfFolder folder)
    {
        return (String) folder.getAttributeValue(a, b c);
    }

    public static EnterprisePerson getNewHireManagerFromFolder(WfFolder folder)

```

```

    {
        String managerId = getManagerIdFromFolder(folder);
        return new EnterprisePerson(managerId);
    }
}

```

Workflow Façade Folder Helper is both powerful and interesting. What's so powerful about this seemingly simple pattern is that

- 1) It COMPLETELY encapsulates folder access within that particular workflow.
- 2) It never provides methods for getting a single folder attribute.
- 3) All methods for accessing folder attributes are private
- 4) All public methods actually return a meaningful Domain Object or entity object

Advantages:

- 1) Since folder attribute access is completely encapsulated within the helper class, typos will be eliminated
- 2) More importantly, since the folder helper never allows a single folder attribute access, rather it has public methods only for returning entity object or domain objects, the code in workflow façade is always more meaningful. If the workflow singleton is accessing single folder attributes, its probably engaging in fine grained task driven programming. By using the coarse grained access, it is always dealing with business related objects – which is a very good programming practice.
- 3) Eliminates assembly line programming – whereby a value is dumped in folder in a particular task and later down the line that folder value is pulled out to do if else based coding in workflow singleton.
- 4) Since the folder access is limited to the helper class and is NEVER passed to the domain objects or entity objects, they are not “polluted” by workflow related objects. Hence those domain objects/entity objects can be unit tested outside of CDM, outside of workflow. This is particularly helpful if reaching a particular domain object logic for testing in the workflow requires several approvals to be done first.
- 5) Notice the method getRecipientForTask2 below. When the recipient for a task is determined by what happens in the previous task, the usual approach is to have a automated task that does all calculations and dumps those recipient values in the folder. Instead they can now easily use the elegant methods within the workflow singleton to determine the recipients next in line.

Implementation Notes:

- 1) The following is example of how Workflow Singleton works with this helper object to create and store domain objects and entity objects:

```

public class MyWorkflowSingleton extends WfCustomSingleton
{
    public void handleTask1(WfFolder folder)
    {
        NewHire h = MySingletonFolderHelper. extractNewHireFromFolder(folder)
        if (h.getStatus()...) { }
    }
}

```

```

public String getRecipientForTask2(WfFolder folder)
{
    EnterprisePerson manager =
        MySingletonFolderHelper.getNewHireManagerFromFolder(folder);
    return manager.getDN();
}
}

```

- 2) With the Façade Helper you are always thinking at a higher level of abstraction that at low level folder attributes.

3.8 View Helper

Intent:

To hide the access to domain objects and entity objects within a Visual Inline Client code in CDM Form/ JSP (if visual inline client code is absolutely necessary) behind a single class.

Also Known As:

N/A

Motivation:

The visual inline client code needs access to domain objects and entity objects (most likely they will since our approach explained above has decreased a lot of unnecessary attributes from being stored in folders).

In this case consider what happens when the domain and entity objects and their DAOs are refactored/renamed/methods changed/ signatures changed (I am assuming you are using a IDE like Eclipse with support for refactoring). The references in the cdm classes will also change, but the JSPS will remain unchanged and hence they will throw runtime compile errors.

To eliminate this and also to prevent logic getting percolated into the view, the entire access is hidden as much as possible behind the view helper class. So refactoring (a constant part of good programming) does not become a nightmare for you.

Implications:

The biggest implication of using the domain objects, or any other Java objects is this:

In CDM, any Java class used in Form/JSP needs to be located in devloader/lib (or in any directory specified by devloader.classpath file). This means that an entire dependency graph of classes is created that needs to be moved out of standard CDM project folders such libraries, triggers etc...