DAF II: DIGITIZATION ASSETS FACTORY

A DIGITIZATION WORKFLOW MANAGEMENT SYSTEM FOR MASSIVE DIGITIZATION PROJECTS

SYSTEM DOCUMENTATION

VERSION: DRAFT

4 SEPTEMBER, 2007

[AUTHORS: FADI EDWARD, MOHAMMED ABUOUDA, MOHAMED YAKOUT]

DRAFT

CONTENTS

DRAFT

DRAFT

1. INTRODUCTION

When an end user accesses images, PDF, audio, video, or any other multimedia document through the Internet, this means that the primary purpose of the entire digitization effort is met. From start to finish, digital multimedia documents production is a manufacturing and delivery process which should be likened to an assembly line. To date, the digitization process in many libraries has concentrated on the input (scanning) side, and fulfillment via the Internet and Content Management Software. What has not received sufficient attention is automating, tracking, and managing the entire workflow with particular emphasis on what happens between scanning and delivery.

BA accepted this challenge as a part of its Digital Assets Repository (DAR) project to achieve a truly device-independent, integrated and automated workflow. The result was the Digitization Workflow Management System, which is a highly reliable digitization workflow management system that can be customized for large and challenging digitization projects, or be used out-of-the-box. In either case, the BA's workflow system improves productivity and therefore reduces both the cost of production and the time it takes to complete a project.

A Digitization Laboratory requires an efficient and highly integrated digitization system consisting of hardware, software and workflow management processes that can fully exploit the unique capabilities of the Digital Lab. Several experiences with large digitization projects taught us the need for a highly integrated system that manages the whole process of digitization with its phases, system users, exception handling, history tracking of actions, files movement, archiving, and integration with the LIS and the library digital repository. Such a system would need to be flexible enough to simultaneously manage multiple projects with a diversity of materials ,covering books, journals, newspapers, manuscripts, unbound materials, audio, video, and slides. The system would also need to seamlessly feed content to the libraries' digital repository to ensure the preservation of the content for years to come.

As any workflow, the digitization workflow is a description of a business process in sufficient details that it is able to be directly executed by a workflow management system. A digitization workflow is composed of a sequence of phases. The phases are undertaken by the digital lab resources, such as digital lab operators and devices (scanners or encoding servers). Several tools are used to execute elementary activities within a digitization phase, such as image processing and OCR suits.

Workflow management systems are used to configure and control structured business processes from which well-defined workflow models and instances can be derived. However, the proprietary process definition frameworks imposed make it difficult to support (i) dynamic evolution (i.e. modifying process definitions during execution) following unexpected or developmental changes in the business processes being modeled; and (ii) deviations from the prescribed process model at runtime. Without support for dynamic evolution, the occurrence of a process deviation requires either suspension of execution while the deviation is handled manually, or an entire process abort. However, since most processes are long and complex, neither manual intervention nor process termination are satisfactory solutions. Manual handling incurs an added penalty: the corrective actions undertaken are not added to the system history or transaction log, and so natural process evolution is not incorporated into future iterations of the

DRAFT

process. Other evolution issues include problems of migration, synchronization and version control.

From our experience, the digitization process is one of the business processes that is affected by the above limitations. These limitations make it hard to be commited to a rigid modeling structure due to the lack of flexibility inherent in a framework that, by definition, imposes rigidity. As a result, users are forced to work outside of the system, and/or constantly revise the static process model in order to successfully support their activities, thereby negating the efficiency gains sought by implementing a workflow solution in the first place. It is, therefore , desirable to extend the capabilities of workflow systems by developing an approach to dynamic flexibility based on natural work practices.

DAFv2 avoids the above limitations by providing the facility to design a rigid workflow and allow for the dynamic evolution and deviations. This helps in handling exceptions by forwarding the jobs to an appropriate phase, which is not in the rigidly defined sequence, without the manual intervention. For example when digitizing books, a printed book passes by scanning, processing, OCRing, archiving then encoding to PDF, while for a handwritten book, it could not go to the OCR phase and, hence, should be forwarded to the archiving then encoding phase. The jobs can also be redirected back to a previous phase due to a quality assurance decision.

DRAFT

2. SYSTEM DATA MODEL

To achieve a truly device-independent, integrated and automated digitization workflow, the system introduces a data model capable of defining different workflows for various types of objects. Each type of object can have its workflow defined by what we call *Phase Sequence.*

2.1 Abstract Data Model

The data model diagram as shown in Fig. 1., consists of six types of entities involved in managing the digitization workflow.

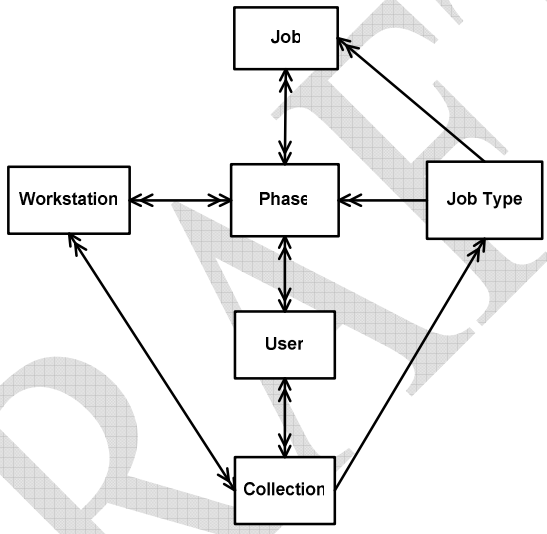


Fig. 1. DAFv2 System Data Model.

2.1.1 The Job

It is the main entity that represents the object being digitized. For example, a printed book for Naguib Mahfouz, photos for an event, a map of Alexandria, a music sheet for Omar Khayrat, a video film about the High Dam, etc. The Job can be one of any Job Type in the system. The Job should pass through the entire digitization Phases required for the Job Type. Each Job is identified by a unique ID and also can be identified by an External ID, which is the ID of the document in the external source as the ILS ID and/or Barcode. The Job has a priority and a life time in the workflow otherwise it will be reported as a Late Job.

2.1.2 The JobType

The *Job Type* entity represents all the types of materials that can be digitized. The *Job Type* can be a book, a map, an audio item, a video, or any other type of document that needs a special digitization workflow.

DRAFT

2.1.3 The Phase

The *Phase* entity represents a task or a unit of work that should be applied on a specific *Job Type* in its digitization workflow. Each *Job Type* has its own sequence of *Phases* defined apriori in a *Phase Sequence* to obtain a digitized version of the *Job*. Each *Job* passes through several *Phases* according to its *Job Type*. For example, for a Printed Book *Job Type*, the digitization *Phases* could be Scanning, Processing, OCRing, Archiving and PDF Encoding, while for a Map *Job Type*, the digitization *Phases* could be Scanning, Processing, and JPG Internet derivatives. The same *Phase* can be done on several *Workstations* in the system; *Workstations* can be assigned for each *Phase* according to their capabilities. Scanning is done on *Workstations* with book scanners, Processing is done on *Workstations* with image processing software and so on. A time period is attached to each *Phase* so that if it took longer time, the *Job* will be reported as *Late Job*. After finishing any *Phase*, the *Users* are able to provide information about the *Phase*. This information is divided into *Phase* specific information, general comments, and file level information. This information will help the next operator who is working on the next *Phase* ,whether it is a new or a previous *Phase.*

2.1.4 The User

The *User* entity represents the system *Users* or the Digital Lab operators. Several roles can be defined for the *Users* to manage their access on the *Jobs*. The *User* can perform several types of *Phases* of a specific *Job Type*. The *User* can also be assigned to work on some *Collections* in the system.

2.1.5 The Collection

The *Collection* entity represents logical grouping for the *Jobs*. It may represent a digitization project or a private collection. A *Collection* may contain several documents of different *Job Types*. A group of *Users* can be assigned to work on a *Collection*. Several *Workstations* in the system can be allocated for a *Collection.*

2.1.6 The Workstation

The *Workstation* entity represents the computer where the execution of the *Phases* is performed. Several *Phases* can be done on one *Workstation* and a *Workstation* can be allocated for several *Collections.*

2.2 Entity Relationship Diagram (ERD) <MA>

2.3 Entity T S Diagram (ETS) <MA>

DRAFT

2.4 Data Flow Diagram <MA>

2.5 Class Diagram <MA>

DRAFT

3. SYSTEM MODULES

Fig. 2. shows a representation for the architecture of the DAFv2. The system provides all the services through five main modules: *Check-In, Phase Manager, Reporting, Archiving and Administration* modules. All modules provide the services after passing through an *Authentication and Authorization Handler*. The *XML Phases Definition Handler* accepts the requests related to applying the necessary checks and actions before and after performing a digitization *Phase*. The *File Handler* is used mainly by the *XML Phases Definition Handler* to manage the files checks, copying and movements. All the system configurations, parameterizations and transactions are stored in a database and managed through the Database Handler.

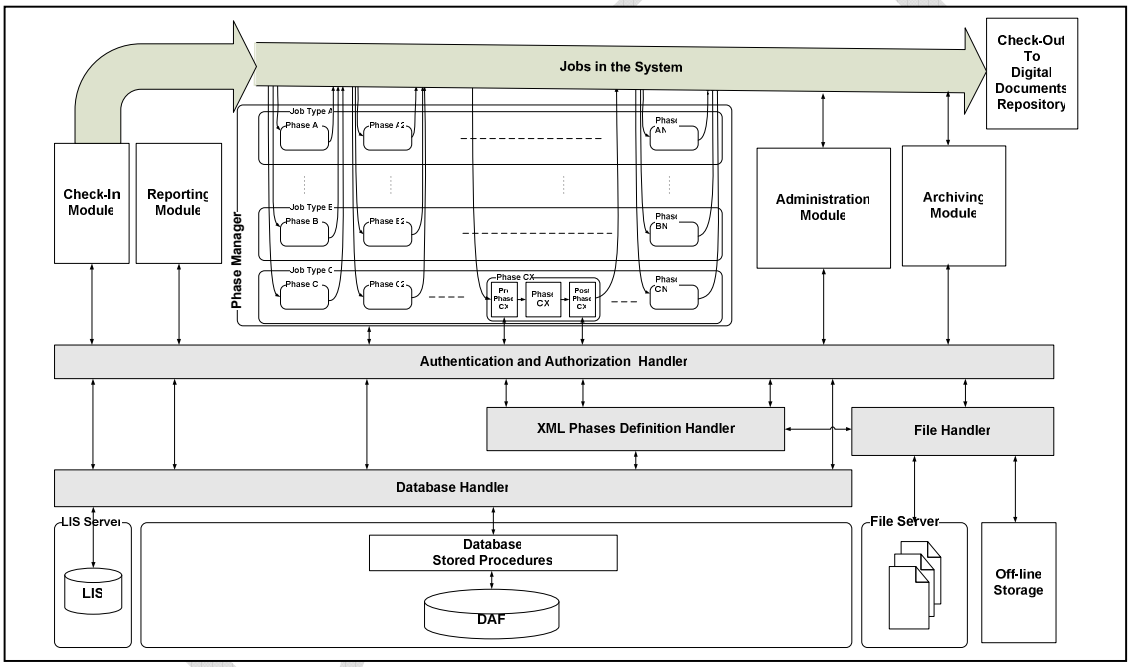


Fig. 2 System Architecture

3.1 Check-In Module

The Check-In Module is responsible for creating a Job in the system and fires it to start. Although the Check-in Module determines the first Phase of a Job depending on its Phase Sequence , the system allows for handling an exception and allows the Job to start from an intermediate Phase within the workflow as long as its prerequisites are met. For example, although a printed book Phase Sequence is defined to pass through the Phases Scanning, Processing, OCRing, Archiving, and Encoding, it is also possible to assign it to the Processing

Phase as long as the TIFF files are already in the working folder. This allows the accommodations of the system to receive scanned books from external sources.

DAFv2 check-in has been designed and built to allow for the integration with any metadata source, as Integrated Library System (ILS), document registry, MARC or MODS files. This flexible integration has been achieved by making the check-in module plug-in based as described in Fig. 4. The system allows each library to write its own check-in plug-in for its metadata source

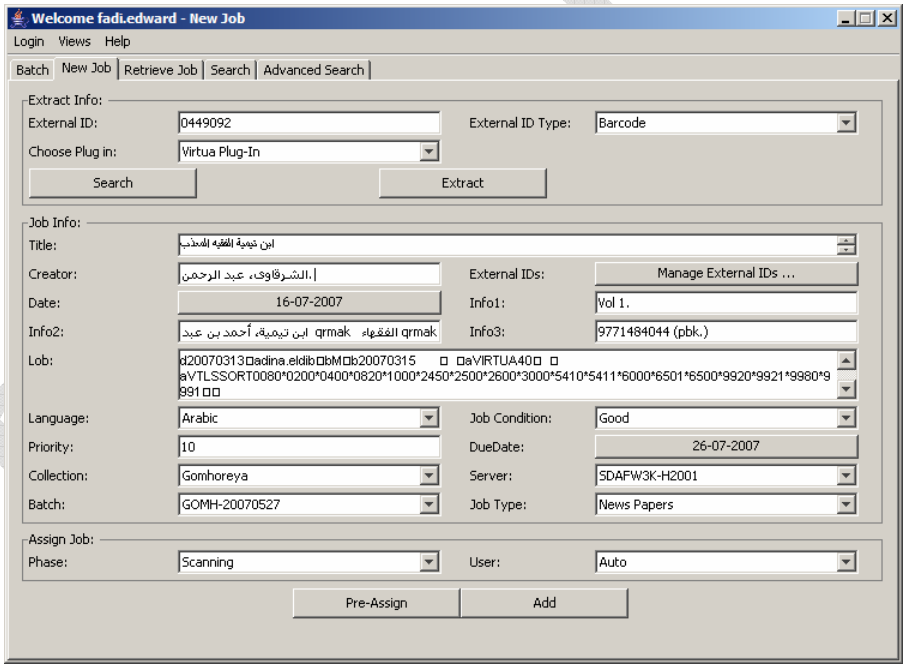
In Order to check-in a Job in a digitization system, four topics should be addressed:

1- The Job Metadata and how to rejoin it to its source after the digitization is complete,

2- Defining the Path the Job is going to take in the digitization workflow and any work done before that can be used to accelerate the digitization process.

3- The integration of Plug-ins into the client to extract the Information from several ILS systems.

4- Batches creation and modification.



3.1.1 Metadata of the Job

There is no Metadata Stored within the DAF system, instead all Metadata is stored in a separate Database and this Metadata is updated and retrieved through a Views and a Stored Procedures layer inside the Metadata DB. Such a design will allow for larger flexibility for storing the Metadata but with less performance.

The metadata maintained by the system is not a comprehensive description, as the system is not used for displaying or fetching Jobs based on the Metadata. In the mean time, the system allows the user to store all the metadata he/she wants to add within a Lob Object.

So basically, the system maintains:

a. The Title,

b. The Creator of the Job,

c. Three generic Information fields that can be used to store the volume or any other information according to the users choices,

d. The Language (all languages are added in a dynamic way through the Administration Manager),

e. A Lob Object.

The Title and Creator are stored mainly for display purposes. The Lob Object can be used to store a Marc21 record, a Dublin Core, a MODS Xml or any other sort of metadata that the user needs to store with the Job, and that he/she requires in the publishing or Check-Out phase. So the Lob Object is used to compensate any deficiencies that may occur due to insufficient Metadata fields.

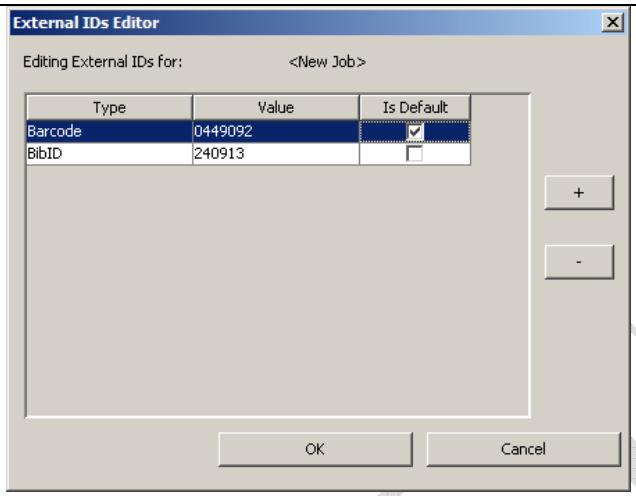
The Check-In Module maintains the relation with the ILS system (source of metadata) to be able to rejoin the Job after it has been checked out, through a dynamic list of External IDs. DAFv2 maintains a dynamic External IDs Types list defined by the user, and also maintains several External IDs for each Job according to the following criteria.

1- Any Job should have at least one External ID.

2- Each Job has only one default External ID.

3- The default External ID is unique among all Jobs External IDs with the same type (even the non-default ones).

DRAFT



A simple scenario concerning the External IDs is as follow:

1- Create an External ID Type equivalent to your ILS unique Identifier.

2- Implement the Check-In Plug-in that extracts from your ILS, and for the method getTypeIDPairs() make sure it returns the ILS unique key as one of the returned External IDs.

3.1.2 Job Behavior in the System

Since the Check-In Module is the main entry point of the system, there is some information that should be specified about the Job and how it will behave in the system. Some of these properties can be modified later, others cannot (in this version). These properties can be enumerated as follows:

1- Collection: For each Job you need to define its Collection, which is the logical grouping or the project, or the source this Job came from.

2- Batch: You need also to specify which Batch from within the selected Collection this Job came in.

3- Job Condition: Defining the Job Condition helps the operators to better handle the Job.

4- Job Type: The Job Type define the Path the Job is going to take in the system.

5- Priority: Each Job is assigned a Priority that will help the System decide its position in the priority queues within the System (In accordance with its due date and if it is Re-Done or not).

6- Server: DWMS allows User to have his/her Jobs on several servers, but each Job has to be fully contained on one server and cannot be divided. (In the future actions/versions we will allow Job move from one server to another using the Client).

DRAFT

7- Assign: You need to assign a Phase for the Job to start with (by default it is set to the first Phase in the Sequence of this Job Type). As well as a User that can performs this Phase (by default it is set to "Auto" user which would allow any free user to start it). This feature is added to allow 2 things: a- Ingesting the Job into the system in an advanced Phase (rather than the first Phase in the Sequence of the Job Type). b- Explicitly assign the Job to a specific user that may have extra skills to handle this Job or at least to increase its priority by assigning it to a user that is currently available.

8- Pre-Assign: A Pre-Assign feature was added (which User can ignore and not use) that allows the Check-In Operator to define who is the User/Operator who will work on this Job through all the Different Phases of the Sequence of this Job. This feature can be used to force special types of Jobs to be assigned to Operators with special skills needed by this Job.

3.1.3 Check-In Plug-In

DWMS check-in has been designed and built to allow for the integration with any metadata source as the Integrated Library System (ILS), document registry, MARC or MODS files. This flexible integration has been achieved by making the check-in module built as plug-in based. The system allows each library to write its own check-in plug-in for its metadata source.

Each new ILS System should implement its own Check-In plug-in by implementing the provided Check-In Interface (you can find the interface at `org.bibalex.daf.managers.checkin.ICheckInPlugin`).

This interface includes 2 types of methods; the first type is control methods, like `collect` that indicates whether a successful retrieval is done or not, or `search` that allows the plug-in to perform a search instead of an exact match. The second type is used by the system to collect the gathered information and assign them to their respective locations in the system, like `getTitle()`, `getCreator()`, `getLanguage()`, …. Please visit the Plug-In Java Doc page for more details about how to implement each function.

For each plug-in added, the User need to add the full Plug-in path within the NewJob section in a Node called "CheckInPlugIns" in its Caption, for several plug-ins add them with semi column separated as follow:

```
<Element Name="CheckInPlugIns"
Caption="org.bibalex.daf.virtuaplugin.VirtuaPlugIn;org.bibalex.da
f.digiarabplugin.DigiArabPlugIn;org.bibalex.daf.ameelPlugIn.Ameel
PlugIn;org.bibalex.daf.indianplugin.IndianPlugin;org.bibalex.daf.
ebarcodeplugin.EBarcodePlugIn;org.bibalex.daf.digitalplugin.Digit
alPlugIn"></Element>.
```

3.1.4 Batches Creation and Modification

Moreover, since the Jobs arrive at the digitization laboratory in batches, the check-in module contains a part to manage the arrived batches. The Batches Manager allows the Check-In Operator to Add/Delete/Modify Batches. For each Batch the User need to specify the following:

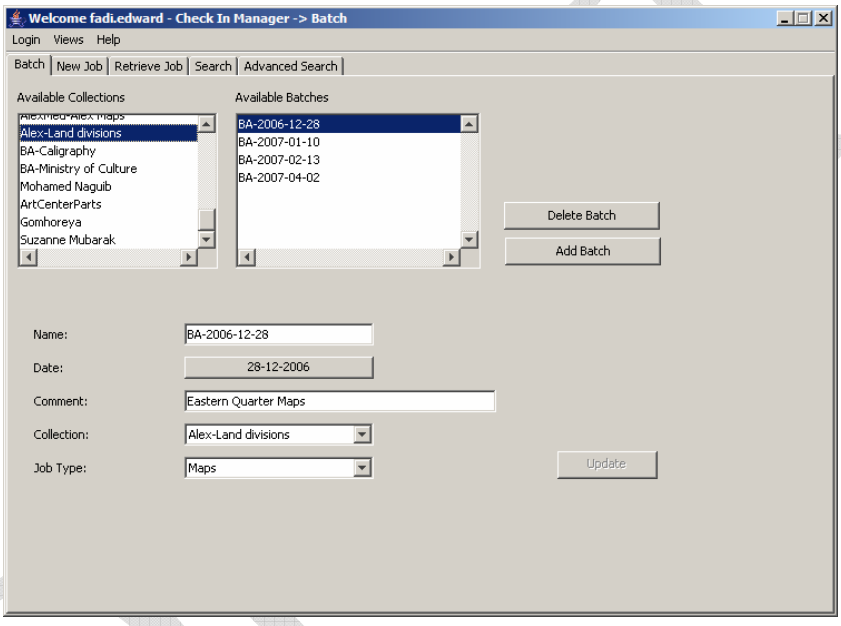
1- Name: which is the name identifying each Batch. It is recommended to add some kind of number or date that relates to the actual physical Batch in this field.

2- Date: The Batch date, the default value is the current date.

3- Comment: Any comment you need to add to the batch.

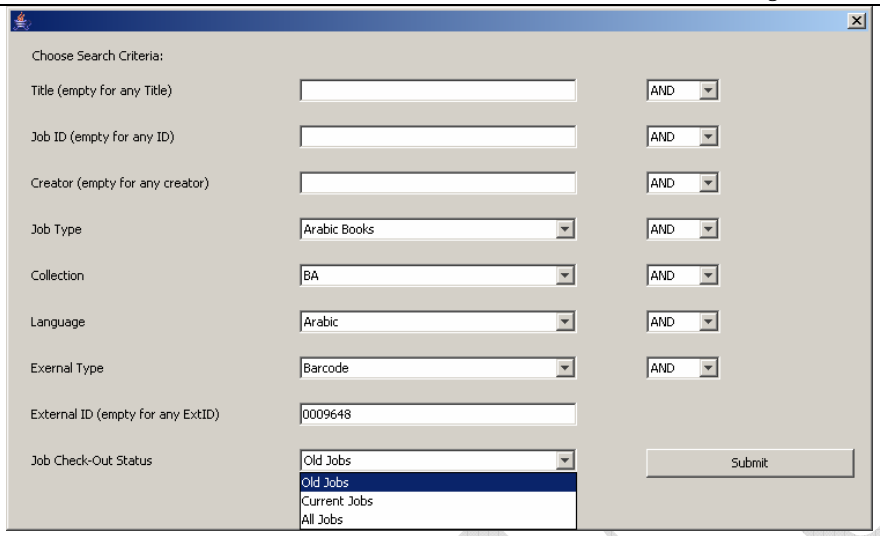
4- Collection: As mentioned earlier, each Batch belongs to only one Collection.

5- Job Type: A user can define a Job Type for each Batch that can be used by the Check-In Manager as a recommendation for all the Jobs in this Batch. The user, however, can still modify the Job Type for each Job in the Batch if necessary.



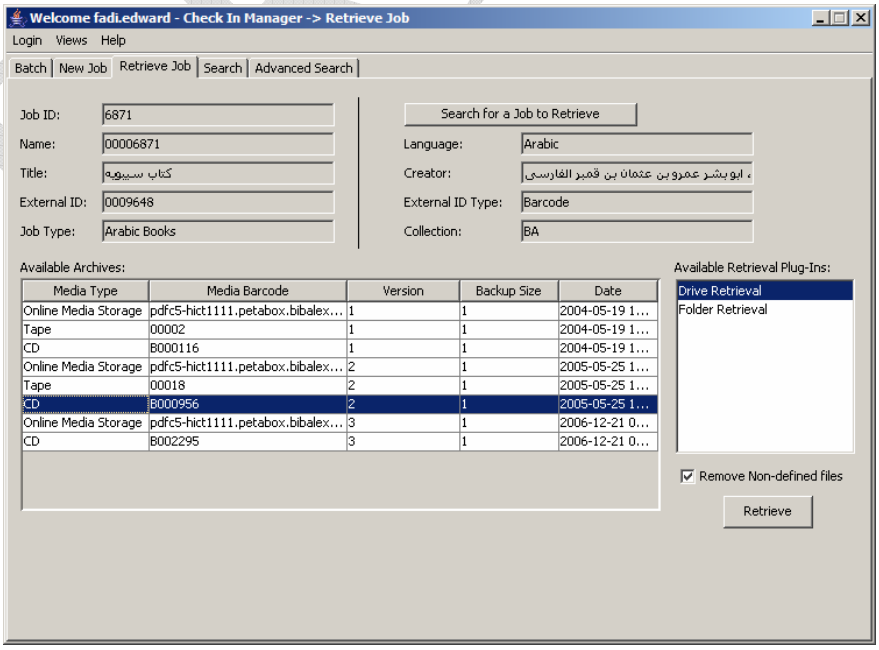
3.2 Retrieval Module

The retrieval module serves for the retrieval of the Jobs from the Archiving system, either for the Jobs that have been Checked-Out from the System and need more processing, or for the Jobs that are still in the system but it is prefered to work on their archived version rather than the one still in the System (in this case the Job folder on the server is overwritten by the version on the Storage).



Since the Archiving system allows for the archiving of more than one version of the Job on different Medias, the retrieval as well is able to retrieve any version of the Job from the selected Media. You will need to search for the Job, retrieve it, and then assign it to a Job Type / Phase / User.

To find the Job you want to retrieve, you can search for it (using a simple Search Screen similar to the Simple Search page. In addition you can Choose whether you are searching within the Checked-Out Jobs/Active Jobs or Any). All Versions/Medias are then listed so that the User can choose which version to retrieve from which Media.



The retrieval is implemented in a plug-in based way, each retrieval plug-in should implement the provided Retrieval Interface (you can find the interface at `org.bibalex.daf.managers.checkin.IRetrievalPlugin`).

This interface includes the method retrieve Job. This method takes the Job object to retrieve, the version, the media barcode, the `backupInfo` of the job and the `RetrievalEventNotifier` object to notify the GUI. Please visit the Plug-In Java Doc page for more details about how to implement each function.

For each plug-in added, the User need to add the Full Plug-in path within the `RetrievalManagerGUI` section in a Node called "`RetrievalPlugIns`" in its Caption, for several plug-ins add them with semi column separated as follows:

```
<Element Name="RetrievalPlugIns"
Caption="org.bibalex.daf.retrievalplugins.DriveRetrieval;org.bibalex.daf.retrievalplugins.FolderRetrieval;org.bibalex.daf.retrievalplugins.OnlineStorageRetrieval"></Element>
```

In addition to that each plug-in should have a section with the name of its class. This section contains the Media Types from which this plug-in can retrieve.

For example, the `FolderRetrieval` plug-in can retrieve jobs from two media types, tape and CD, so its section in the resource file is:

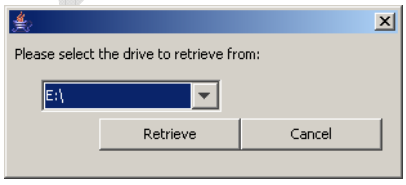
```
<Section Name="org.bibalex.daf.retrievalplugins.FolderRetrieval">
<Element Name="MediaTypes" Caption="CD;Tape"></Element>
</Section>
```

The system is shipped with 2 Plug-Ins implemented:

1. **Drive Retrieval:**

   If the selected Media Type is CD, the Drive Retrieval option appears in the "Available Retrieval Plug-ins" list .

   This Plug-in retrieves Jobs from the drive specified by the user. The initialize method of this plug-in launches a dialog box where the user selects the drive to retrieve the Job from.



   The retrieveJob method in this plug-in takes the Job-object-to-retrieve, the version, the media barcode, the `backupInfo` of the job and the `RetrievalEventNotifier` object to notify the GUI.

   The retrieve location can be:

   If the backupAs info has length, the retrieve location is:
   Drive:\\mediaBarcode\job.getPath()_vVersion
   Or *Drive:\job.getPath()_vVersion*

DRAFT

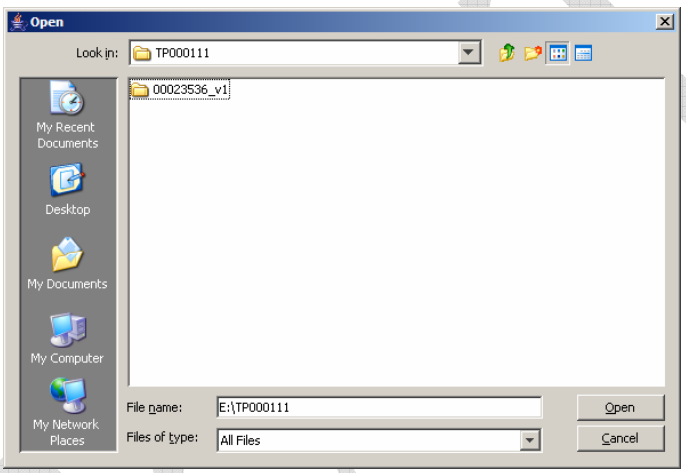
If the backupAs is empty, the retrieve location is:
Drive:\\mediaBarcode\job.getPath()_vVersion

If the Job folder exists on the selected drive with all the input parameters correct, it is fetched to your local working directory.

2. Folder retrieval:

If the selected Media Type is CD or Tape, the Folder Retrieval option appears in the "Available Retrieval Plug-ins" list .

This Plug-in retrieves Jobs from the folder specified by the user. The initialize method of this plug-in launches a dialog box where the user browses for the location of the job folder on the m/c.



The retrieveJob method in this plug-in takes the Job-object-to-retrieve, the version, the media barcode, the `backupInfo` of the job and the `RetrievalEventNotifier` object to notify the GUI.

The retrieve location can be:

If the backupAs info has length, the retrieve location is:
FolderPath\mediaBarcode\ job.getPath()_vVersion
Or
FolderPath:\job.getPath()_vVersion
If the backupAs is empty, the retrieve location is:
FolderPath:\ mediaBarcode\ job.getPath()_vVersion

If the Job folder exists on the given location with all the input parameters correct, it is fetched to your local working directory.

After the Job is fetched to your local working directory, you can inspect the files and modify them if needed. Then you should assign the Job to a Job Type (default value is the Job Current Job Type), a Phase to start in (the last Phase this Job has visited is selected by default), the User to perform this phase (the last user who performed this phase is selected by default). Phase

Checks are performed to make sure that the retrieved Job, with its current files and folder structure, matches the requirements needed by the Phase that the Job was assigned to.

Afterwards, the Job is automatically transferred to the server and assigned to the designated user in the pointed Phase and Job Type. In case an Active Job has been retrieved, the files on the server are overwritten and the current assignment as well.

The retrieval also support Command Line Retrieval.

3.3 Phases Manager Module

The Phase Manager provides the interface to the digitization laboratory operator. It allows the operator to request a new Job to work on, download the working files, and submit the Job back to the system to continue in its workflow. Moreover, the Phase Manager allows the operator to reject a Job after starting working on it. In this case, the operator will have to submit a rejection reason. Afterwards, the system will automatically assign the Job to the administrator to review the rejection reason and take the necessary actions. Also, the operator can redirect the Job to another Phase not in its default path; the redirection of Jobs is automatically confirmed if the operator has enough permission. Otherwise to redirect Jobs, the Jobs is automatically assigned to the administrator as pending until he/she accepts or denies this redirection. The redirection may be due to a problem in a previous Phase. For instance, while performing the OCR Phase, the operator might have discovered that there were some pages that need further Processing or that there were some pages missing that require scanning, thus they need to be returned to the Processing or Scanning Phase.

In order to simplify solving problems that happen in previous Phases of digitization, DWMS allows the operator to add information on the produced files' level. The information contains the files' numbers, the required Phase that should be revisited, and the source of the problem. This information is saved in the database. Once the Job is revisiting a previous Phase, the operator will be informed with the files that require reprocessing and the reasons. DWMS allows the administrator to define a list of reasons for each Phase to be revisited so that the operator can select the reasons from a drop-down list. This information is propagated for the next phases to take the necessary actions on the newly produced files. For example, suppose that in the OCR Phase files 20 to 25 are missing and require Re-Scanning. The Job will be redirected to the Scanning Phase with file info about the page numbers that require scanning. The Job will be forwarded to the Processing Phase with the files' level information indicating that files 20-25 require processing and so on.

TODO: We could add : Phase Manager Concepts:

From UI Perspective:

Phase Manager:

- The Phase Manger Module provides the interface to the digital laboratory user.

DRAFT

- The Phase Manager GUI shows the *Currently Working Jobs* list, The list displays the following information:
  ID: The ID of the Job

  Job Name: The title of the Job

  Phase: The phase to be applied on the Job

  Status: The status of the Job (i.e. started)

  Date: The date when the Job was started.

  Started Before: The number of times this Job was started before on the corresponding phase.

  Ext.ID: The default External ID value of this Job.

  Ext.ID Type: the Type of the above mentioned External ID.

- A Right Click on a Job in this list provides a context menu with 4 options: View Job History, View Metadata, View File Info, Edit File Info.

    - Job History shows detailed information about the history of this job. The history information is given by the User, Phase and Workstation for the job. Each row shows also the Job Status, Action Date and the File Count, which is number of files that have been added or modified during this Phase (and not the total pages count of the digitized media). This screen is very helpful in tracing the flow the job.

    - View Metadata displays the Job MetaData screen, which includes all the meta data information.

    - View File Info displays the 'TODO' File Info. It displays the type of the information whether *inst* or *info*, the phase to be revisited as well as the pages that have the problem, the reason for the redirect, the suggested user and the Done check box. For a Job to be done, all the Done check box entries in the File Info must be checked.

    - Edit File Info enables the user to edit the 'Next' File Info, add or remove Info. The added information has a type, whether inst or info, the phase to be revisited, the pages that have the problem, as well as the reason for the redirect and a comment on that redirection action.

DRAFT

On the bottom of the form, there exists buttons for the following actions:

1. Get Job:

- This action allows the user to select a job to work on. Get Job starts by launching a dialog that lists all the available jobs for the user. A job is available for the user to work on if:
    - The user can work on the current phase of this job
    - and, the user can work on the collection to which this job belongs
    - and the job is assigned to this user or to the 'Auto' user
- The list of Available Jobs displays information such as Job ID, Job Title, Next Phase to be done, User, External ID and External ID Type. Number of revisiting times for each job, with the corresponding phase, appears in the Revisiting Times column and it also shows whether the current workstation is valid for the corresponding job or not. In case the Job is not eligible to start on the current Workstation, it will still appear in the list but with the flag "is valid workstation" set to false.
- If the user is not allowed to select a job, the job to be started is by default the first job in the list which has the check box '*Is Valid Workstation*' checked.
- However, if the user has the right to select a job he/she must choose one of the jobs with box '*Is Valid Workstation*' check box checked otherwise he/she will get an indicative error message and the Job will not start.
- *Selecting a job makes it in the starting state.* In order to start a job the following actions take place:
    1) Check if the user can start the job(valid job and phase for this user on this workstation)and assign it to the current user, if needed(if assigned to 'Auto' user)
    2) Perform the pre-phase checks.
    3) Perform the pre-phase actions. The progress bar shows the process of transferring the job folder from the server to the local working directory.
    4) Propagate the previous status data from the previous *TransactionLog* entry
    5) Adjust the job status in the database.
- When the Job folder is fetched successfully from the local working directory, The *FileInfoViewer* screen appears. This screen lists any *FileLevelInfo* for this job to be done in this phase.
- The Job is now the first job in the '*Currently Working Jobs*' list with its Status column set to '*started*'.

DRAFT

2. Done Job:

Done Job button is used when the process of the phase is finished and the user wants to submit the job back to the system to continue in its workflow. '*Done Job*' puts the job in the finishing state. For a job to finish, the following actions take place:

1) Get necessary information to finish the job: phase, file handler, source and destination

2) Check if the user can finish the job from his current workstation or not.

3) If the job is being finished from the command line, mark all the FileLevelInfo entries for this phase as 'Done'

4) Determine whether the user can finish the job because of the FileLevelInfo or not. A job cannot be finished in the following 2 cases:
    a) There exist entries in the '*Next*' FileLeveloInfo file. In this case it should be redirected.
    b) There exist undone entries in the '*ToDo*' FileLeveloInfo file. The user should 'Done' entries.

5) Apply the post-phase checks.

6) Apply the post-phase actions.

7) Show the DataBase Info dialog to set database values such as in the OCR phase the user enters the Font Family, Learning Pages, Accuracies. And in the QA-PDF phase the DataBase Info dialog is used to decide whether the Book is Image On Text, Contents Matched, Wrong text and Image Pairing, Pages in right order and PDF Error.

8) The progress bar shows the process of transferring the job folder from the local working directory to the server.

9) Update the *'ToDo*' FileLevelInfo, Update the '*Next*' FileLevelInfo and form the status data lob to add to the database.

10) Insert Finish entry for this job in the database, and If the transaction completes successfully, delete the job folder from the local directory

DRAFT

When the job is done successfully, the 'Currently Working Jobs' table is re-loaded to exclude the finished job.

Special cases examples:

Backup phase:

The Phase Manager doesn't insert the Finish entry for the job in the database because this step is done in the reflection call for the backup phase. The same thing applies on updating the FileLevelInfo, it is updated in the reflection call and not in the Phase Manager.

3. Redirect Job:

A job could be redirected to a Phase other than the next phase in its default path. The redirection may be due to a problem in a previous Phase. For instance, while performing the *OCR* Phase, the operator may discover that there were some pages that need further Processing or that there were some missing pages that require scanning, thus they need to be returned to the *Processing* or *Scanning* Phase.

File Level Info:

It contains the type of the information whether *inst* or *info*, the phase to be revisited as well as the pages that have the problem, the reason for the redirect and a comment on that redirection action.

The user edit the FileLevelInfo in the 'Next' FileLevelInfo Editor for the job that requires redirection before pressing the 'Redirect Button'. 'Redirect Job' dialog appears to enable the user to set the Next Use and the Next Phase. The job enters the redirecting state and in order to redirect successfully the following actions take place:

1) Get necessary information to finish the job: phase, file handler, source and destination

2) Determine whether the job can be redirected to the given phase because of the FileLevelInfo or not. A Job can be redirected to the given phase if it satisfies the following conditions:
    a. All the *ToDo* entries that point to this phase are done.
    b. The given phase is equal to or less than the minimum phase in the *ToDo* and *Next* FileLevelInfo.

3) Applying post phase checks to redirect the job

DRAFT

4) Applying post phase actions to redirect the job

5) The progress bar shows the process of transferring the job folder from the local working directory to the server.

6) Update the FileLevelInfo and the *'Next*' FileLevelInfo.

7) Mix the *'Next*' FileLevelInfo and the updated '*ToDo*' FileLevelInfo to form the status data lob to add to the database.

8) Insert Redirect entry for this job in the database and if the transaction completes successfully, delete the job folder from the local directory

The redirect information is propagated for the next phases to take the necessary actions on the new produced *LevelInfo* files. For example, suppose in the *OCR* Phase files 20 to 25 are missing and require Re-Scanning, the Job will be redirected to the Scanning Phase with file info about the page numbers that require scanning. The Job will be forwarded to the *Processing* Phase with the files' level information indicating that files 20-25 require processing with the reason '*ORIGINATED from missing pages'* and so on.

4. Reject Job:

The user is allowed to reject a *Job* after he/she has started working on it. The rejection reason is added in the *FileLevelInfo*. When a user rejects a job, he/she sets the Next Phase and Next User in the *'Reject Job'* dialog.

After submitting the suggested Next Phase and Next User, the job becomes in a rejected state. A Job can be rejected from the given phase if it satisfies the following conditions:

1) Get necessary information to finish the job: phase, file handler, source and destination.

2) Determine whether the job can be rejected or not. A Job can be rejected to the given phase if the given phase is equal to or less than the minimum phase in the *ToDo* and *Next* FileLevelInfo.

3) Post phase checks are not applied in this case.

4) Apply post phase actions to redirect the job.

5) Update the FileLevelInfo and the '*Next*' FileLevelInfo.

6) Mix the *'Next*' FileLevelInfo and the updated '*ToDo*' FileLevelInfo to form the status data lob to be added to the database.

7) Insert Redirect entry for this job in the database, and If the transaction completes successfully, delete the job folder from the local directory .

5. Download:

The user can download the selected job (download files defined for the current phase) in the '*Currently Working Jobs*' list from the server to the local working directory. This is helpful for when the user wants to restore the original server files after making changes on the local files, or when he/she changes the machine after uploading the job to the server (refer to Upload section). The following steps are needed to download the job to the user's workstation:

1) Get necessary information to download the job: phase, file handler, source and destination.

2) Gets the phase definition used to download the job which has the following characteristics.
   - The pre-phase physical section doesn't have any repeated nodes, thus no folder is allowed twice.
   - The pre-phase doesn't contain any reflection or database sections.
   - The post-phase definition will have the "Name" and "NewName" values swapped.

3) Apply pre-phase actions to download the job (download the files required for the current phase).

4) Insert in the database download entry for this job and the number of files required to be transferred in order to download the job.

Note: In order to download the whole job folder, the 'Retrieve' button in the search screen can be used.

6. Upload:

The user can upload the selected job (upload files defined for the current phase) in the '*Currently Working Jobs'* list to the server from the local working directory. This is helpful for when the

user wants to change the workstation before finishing the job. A user can not upload a job that was not downloaded on his current machine. The following steps are needed to download the job to the user's workstation:

1) Get necessary information to upload the job: phase, file handler, source and destination.

2) Disable reflection call and database post-phase actions for the phase definition.

3) Apply post-phase actions to upload the job (upload the files required for the current phase).

4) Insert in the database the upload entry for this job and the number of files required to be transferred in order to upload it.

5) If the transaction completes successfully, delete the job folder from the local directory

Note: In order to download the whole job folder, the 'Overwrite' button in the search screen can be used.

7. Refresh List:

If changes occur to the 'Currently Working Jobs' list, the user should refresh this list. Refresh List, loads the Working Job Table into the SortableTable object from the database.

3.4 Administration Module

The Administration Module is responsible for the necessary system parameterization and settings. It allows the administrator to define and manage the Job Types with their Digitization workflow and Phases, the Roles of the Users, Workstations, and Collections. It also provides the facility to control the matrix covering the relation between Users, Workstations, Job Types, and Collections. For example, the BA collection contains two Job Types; J1 and J2. The collection will be handled on Workstations W1 and W2 by the Users U1 and U2. U1 will be working on the Job Type J1, while U2 will be working on Job Type J1 and J2.

Admin module carries out, mainly, the functionalities of (Add, Delete and Update) for system entities and settings which are Roles, Job Types, Phases, Users, Workstations, Collections and Collection Owners. Previous listing contains what we may introduce as "Major Entities" or most important ones. Actually, DAFv2 has other entities considered as "Minor Entities" which are Devices, External ID Types, Job Conditions, Languages, Operating Systems, and Media Types.

DRAFT

In order to clarify matters, major or minor entities both are required and necessary to build up DAFv2.

This module also links up system entities. For example, assigning users to work in special workstations for accomplishing special Collection's Jobs, requiring the administrator to make sure that all entities exist and are defined correctly, and then administrator can combine these entities together.

Organizing dependencies is one of the admin module tasks, it is some kind of logical grouping for entities. Technically, you can consider organizing dependencies as implementation of one to many entities relation, i.e.. According to DAFv2's database, relation between Job Type and Phase is one to many. If the administrator decides to create a new phase, he/she must be sure that a Job Type for this Phase is already found or he/she must create new Job Type. This could be applied to Collection Owner and Collection (one to many).

3.4.1 Commons In The Administration Module

Simple User and Developer can find common technique in the admin module interface and implementation; interface wise, user can detect that every tab of admin module has the following:

1- Sortable Object (Table or List) for loading data.

2- Add, Delete and Update buttons.

These four functionalities affect the code and database implementation by creating four major functions (load, add, delete and update) for every tab (Class) in admin module.

General speaking, this technique is followed throughout DAFv2.

Another common technique is Master-Details viewing, which is simply, loading all master data in a section and adjusting all details by selecting any of master records. These two sections are implemented in 2 basic panels named materPanel and detailsPanel.

3.4.2 Implementation

Handling every entity in DFAv2 such as User, Workstation, Collection or any other entity, comes through implementing a special class for this one. These classes are grouped under one package "**org.bibalex.daf.entities**". Developer can consider any entity class as the lowest level acts as coordinator between system Managers and Handlers. All entities are inherited from ***AdminModuleBase*** abstract class, which has the three basic functions (add, delete and update). Each entity implements these functions.

DAFv2 Managers are the GUI components, in other words, DAFv2 parts that are responsible of interfacing data, all managers are grouped under **"org.bibalex.daf.managers".** Administration module is one of these managers. System Handlers will be illustrated late on.

Managers have main entry class; this one inherits **JTabbedPane and it's** responsible of tabs objects instantiation, ordering these tabs and initializing ***ResourceManager***. Usually, the entry class name is ManagerGUI. In admin module the class name is "***AdminManagerGUI***".

In conclusion, Administration Module is one of system managers that interface and control system entities. System entities are represented in independent objects; they encapsulate all functionalities and actions and coordinates between system manager and handlers. Figure 3.3.1 shows this hierarchy.

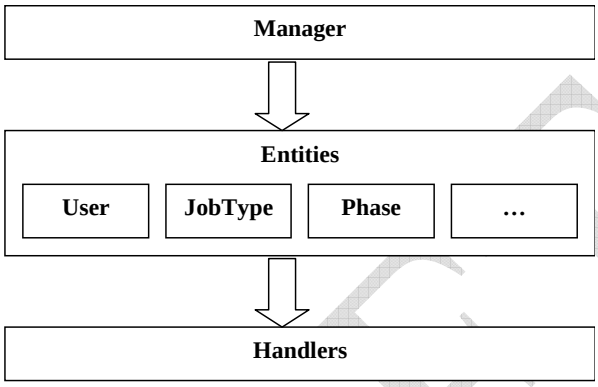


Fig. 3.3.1 DAFv2 Component Hierarchy

Starting from here, we will explain each tab of the administration modules, and focus on used entities, handlers and database actions. Also, we will provide scenarios, flow charts and affected database schema (portion of schema) if there is a need.

3.4.3 Roles

A Role is a grouping of Permissions, each User is assigned to a single Role. Any Permission can be found in more than one Role. To implement the Authorization system we had several choices, mainly 2 of them were more convenient than the others. The 1st one (not used) was to assign the Permissions to the Users directly using a many-to-many relation table. The 2nd one (used in our implementation) is to add Roles in the middle between the Users and the Permissions. Each Role is assigned a set of Permissions (many-to-many relation) and each User is assigned a single Role.

The 1st option was better in case each User in the system requires Permissions different from those of his/her colleagues. In this case we could of implemented a “Copy Permissions” feature or “Permissions Templates”so that the administrator can use it to reduce the overhead of creating a new User. The 2nd option is better in the maintenance of the Users’ permissions, but the Administrator will need to create a Role for each User in the case that all Users are different (require different Permissions). Moving from one solution to another will require minor

changes.

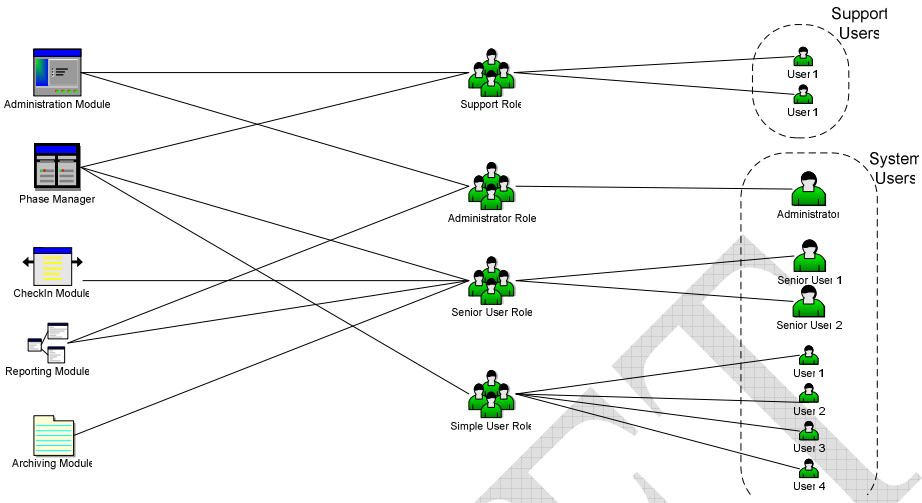
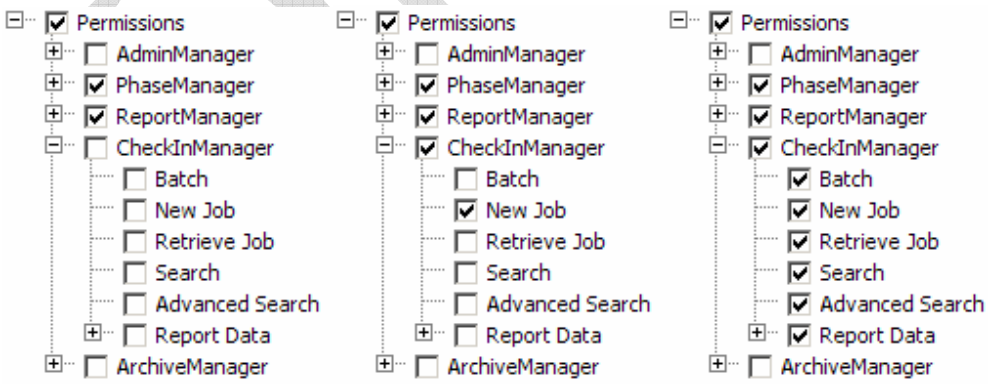


Fig. 3.4.3.2

It is recommended to grant the Users only the required permissions for their day to day use of the system, even if they are highly trusted Users. This will reduce the occurrence of accidental errors. To achieve that goal, the Administrator needs to carefully define his Roles and who is assigned to which Role.

Permission is implemented in a Tree; we have chosen such a design as it matches with our UI design. For example, we grouped all Checking-In functionality in the Check-In Module. From within the Check-In module, some Users can either create new Batches or ingest new Jobs. Imagine we have 3 categories of Users; first group is not allowed to make any kind of check-In operations, so they will not need to see the Check-In module at all. Another maybe allowed to only Ingest Jobs, but cannot create new Batches. The 3rd has full right on the Check-In module. So the 3 trees will be as follow:



The currently implemented permissions are based on the UI except for a few cases. The first level in the Tree are the views, the second level are the Tabs; these 2 level are fully implemented for all the Views and Tabs. The 3rd level is either Buttons or special kinds of Permissions. Not all buttons are mapped to permissions, only the critical ones that may cause harm to the system in

case of Bad Use. For example, with the Delete Job Button, the Job Information will be lost in the case of its accidental usage. We have some special permissions (non-UI related) like:

a. The View All in the Rates tab inside the Reporting module, such that a simple Operator can view only his rates while an administrator or an Advanced User can View all Users' rates.

b. Select Permission (Inside the Phase Manager): A User cannot select the Job he/she can work on, or he/she would only choose easy Jobs and leave the ones that need a lot of effort. An advanced User has the right to select the Job he/she wants to work on.

c. Reject and Redirect (Inside the Phase Manager): To prevent unneeded Job delays in the system, only advanced User can Reject or Redirect a Job to a Phase that is not in the normal path, while a normal User can only propose such an action, and it has to be either approved or denied by an Advanced or Senior User.

Permissions cannot be added from the UI, as they are attached to code according to each permission. So adding a new permission has to be done through the database directly, and an appropriate code is to be added to enforce the newly added permission. At system load, all permissions allowed to the logged-in User are loaded into a hash table () that is later on fetched for checking on Permissions. (Please review the Javadoc for the MainControllerGUI.Initialize() and the Javadoc for the User.loadPermissions()).

Wherever you need to check if the User has a specific Permission or not use:

```
if(
MainController.getMainController().getCurrentUser().hasPermission("CheckInManager$Report
Data$Delete"))
```

This code checks if the logged-in User has Permission to the Delete Button inside the Report Data within the CheckInManager or not. Please note that Tree levels are concatenated using "$".

3.4.3.1 Role Entity

This entity inherits ***AdminModuleBase*** and implements the three basic functions (add, delete and update).

Entity class' constructor has one parameter for specifying working ***Role***. It initializes database connection, queries the database with ***Role*** ID, and sets all class' variables with queried data. Thus, initializing ***Role*** entity loads all its data in memory and gives developer opportunity to work on this data offline (without database connection), unless the operation he/she uses needs database connection (add, delete or update). Achieving this goal needs implementing setter and getter functions for all fields.

Class' methods (neither setters nor getters) have been implemented based on following concepts:

1- Set the necessary variables (fields)

2- Prepare array list of these variables (argument list)

3- Pass store procedure name and argument list to database handler

4- Receive data (if exists) from database into OperationInfo object (see database handler)

Find the following code snippets

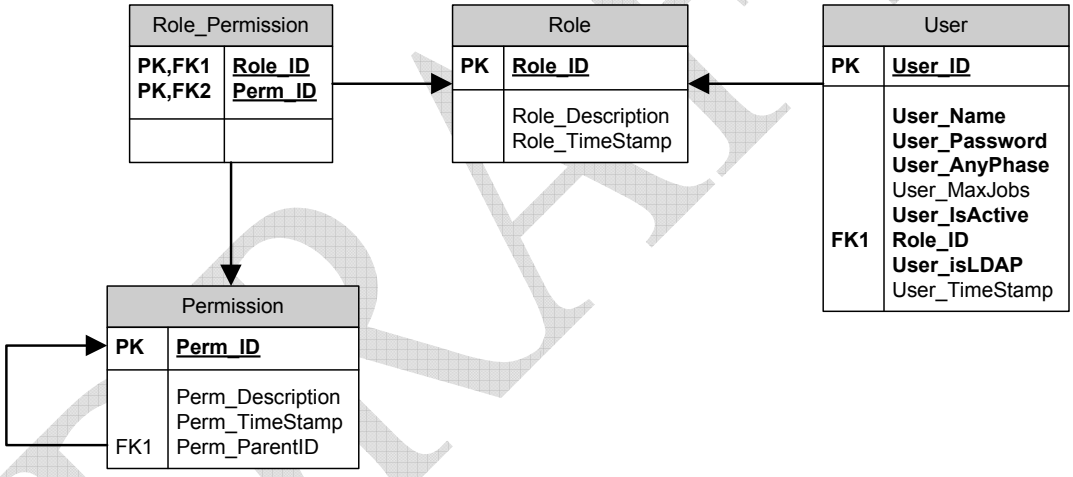
```
1.public  OperationInfo update(){
2.ArrayList<Object> argList = new ArrayList<Object> (1);
3.argList.add(new Integer(id));
4.argList.add(new String(roleDescription));
5.return conn.executeProcedure("Role_Update",argList);
6.}
```

Line 1: Declares the function signature which returns ***OperationInfo*** object.

Line 2: Prepares argument list

Lines 3-4: Sets argument list

Lines : Passes stored procedure name and argument list to database connection and return data into the ***OperationInfo.***



3.4.3.2 Role Relations

The Role table is part of 2 relations:

1- Role_Permission: each Role has a set of Permissions. The Permissions are static (cannot be modified by the User), but the Roles are dynamic (an Administrator can dynamically create a new Role from the RoleManagerGUI). Assigning Permissions to a specific Role are stored in the Role_Permission table (many-to-many relation between Role and Permision).
Also we need to mention that a Permission has an internal Parent Relation to Represent-the-Permissions-Tree (Perm_ParentID is a foreign key to Perm_ID).

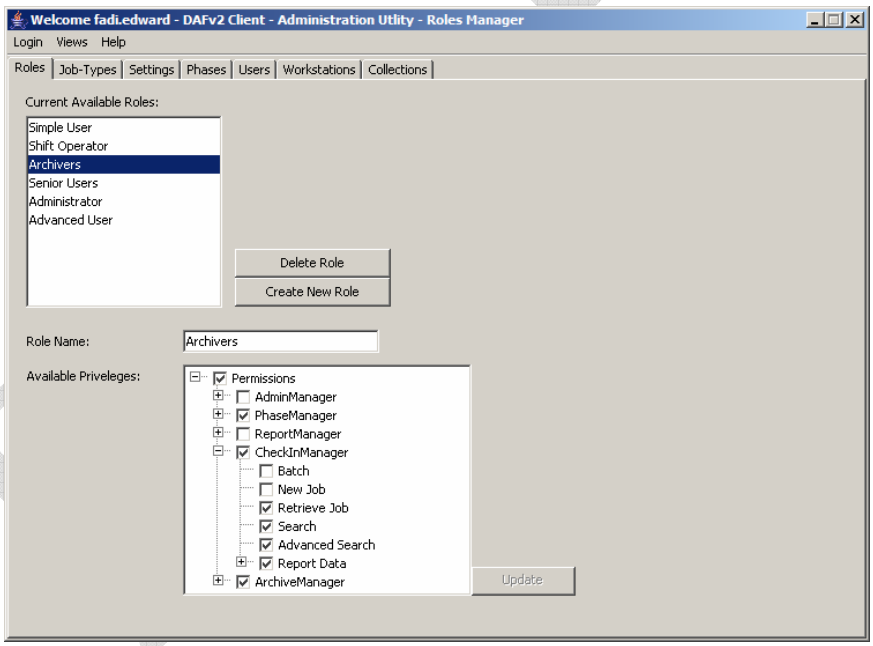
2- User: Each User is assigned to a single Role from which he/she inherits its permissions. This is a one-to-many relation maintained in the User table.

DRAFT

3.4.3.3 Role GUI

AdminManagerGUI class initializes object of "***RolesManagerGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel***. "***RolesManagerGUI"*** has 3 panels**:** jPanel, masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPanel. masterPanel is the container of sortable table that loads all Collections of DAFv2. detailsPanel contains a JTextField that displays and edits the name of the selected ***Role*** from the masterpanel->ListBox or when adding a new Collection. The detailspanel also consists of a Check Boxes Tree. The text values within the tree are fixed since permissions are not edited dynamically, while the Check Boxes state depends on the current selected Role. Checking a parent node checks all its children, but they can be later unchecked. Un-checking a parent node unchecks all child nodes. It is not allowed to check a child node while one of its parents is not checked.

When the user chooses the Roles tab, all Roles in DAFv2 will be loaded into the List Box. First row in this table will be selected and details data of the selected record will be adjusted in the details panel. Since the ***RolesManagerGUI*** inherits from the ***BaseManagerGUI***, the implementation of the ***reloadManager*** nullifies all controls and reloads them as tabs change. This helps in getting the last updated data on refreshing tab.



Changing any details data of selected row will enable the ***Update*** button, firing this button will save data into the database. Delete and Create buttons completing the 3 basic functionalities (add and delete).

According to the Role Entity relations, data from the RoleManagerGUI is mapped to 2 tables,

a. The Role Name is stored in the Role table (Role_Description).

b. The Attached Permissions are stored in the Role_Permission table; On each update all the Permissions for this Role are deleted and then re-added from scratch.

3.4.4 Job Type

DAFv2's ***Job Type*** is the logical grouping of ***Jobs***. ***Job*** terminology is used widely in DAFv2 system; it indicates for any object that need to be digitized, or any item in DAFv2. This includes Books, Images, Manuscripts, etc. Every ***Job*** must fall into 2 logical groups: ***Job Type*** and ***Collection.***

To answer the question why every ***Job*** falls into ***Job Type*** and ***Collection***, we have to define each of them separately. A ***Job Type*** is a group of ***Phases***, all ***Jobs*** belong to this type should pass over its ***Phases*** in order to undergo actions assigned to every ***Phase*** (see ***Phase***). For example, if DAFv2 has a ***Job Type*** called "Latin Books", this Job Type has 3 Phases Scanning, Processing and OCRing respectively. This means all ***Jobs*** belonging to "Latin Books" must pass over these ***Phases*** to accomplish their lifecycle. So, ***Job Type*** is a list of ordered actions distributed on one or more ***Phases***; these actions must be applied (in order) to finish ***Job*** lifecycle.

Collection of ***Jobs*** is a grouping of physical ***Jobs***, it has no actions. ***Jobs*** frequently come in ***Batches*** that are organized in to ***Collections.*** Figure 3.4.3.1 and 3.4.3.2 shows deference between ***Job Type*** and ***Collection.***

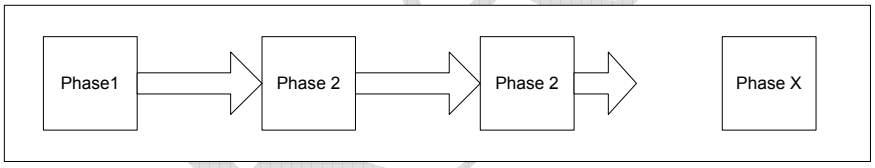


Fig. 3.4.3.1

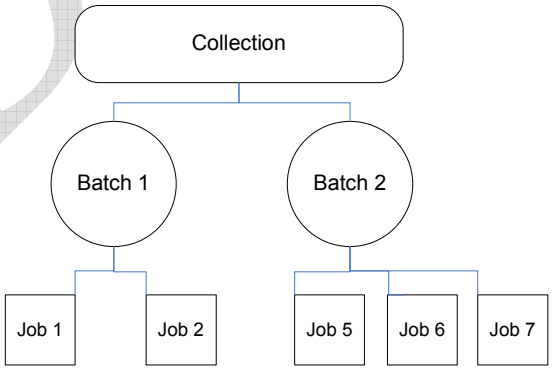


Fig. 3.4.3.2

Transactions on ***Job Type*** are performed through ***Job Type*** entity and saved into JobType table.

3.4.4.1 JobType Entity

This entity inherits ***AdminModuleBase*** and implements the three basic functions (add, delete and update).

Entity class' constructor has one parameter for specifying working ***Job Type***. It initializes database connection, queries the database with ***Job Type*** ID, and sets all class' variables with queried data. Thus, initializing ***Job Type*** entity loads all its data in memory and gives the developer the opportunity to work on this data offline (without database connection), unless he/she uses an operation that needs database connection (add, delete or update). Achieving this goal needs implementing setter and getter functions for all fields.

Class' methods (neither setters nor getters) have been implemented based on following concepts:

5- Set the necessary variables (fields)

6- Prepare an array list of these variables (argument list)

7- Pass store procedure name and argument list to database handler

8- Receive data (if exists) from database into OperationInfo object (see database handler)

Find the following code snippets

```
1. public OperationInfo update(){
2.      ArrayList<Object> argList = new ArrayList<Object> (5);
3.      argList.add(new Integer(id));
4.      argList.add(new String(JT_Name));
5.      argList.add(new String(JT_Description));
6.      argList.add(new Integer(JT_LifeTime));
7.      argList.add(new String(JT_NamingConvention));
8.      return conn.executeProcedure("JobType_Update",argList);
}
```

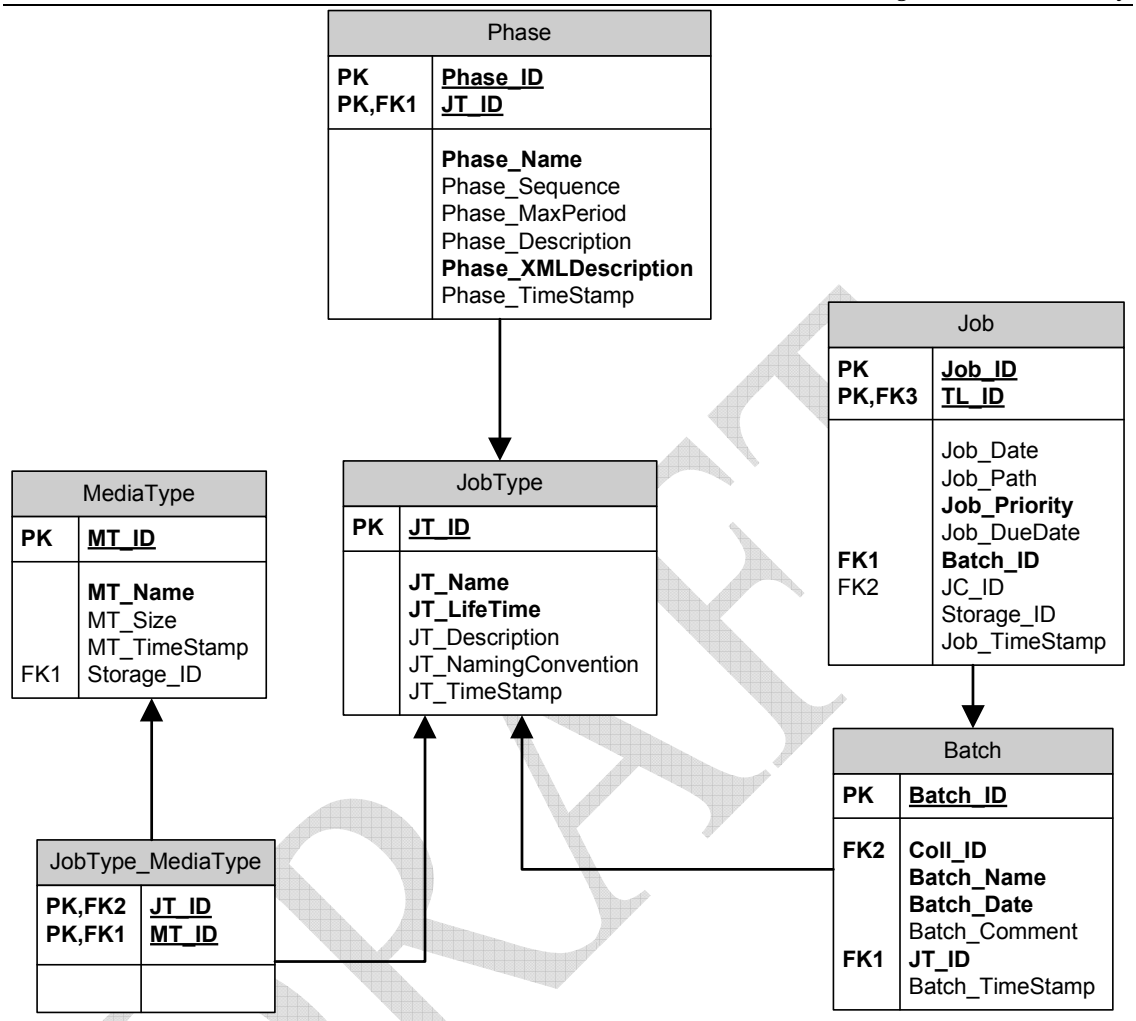
Line 1: Declares the function signature which returns ***OperationInfo*** object.

Line 2: Prepares argument list

Lines 3-7: Sets argument list

Lines 8: Passes stored procedure name and argument list to database connection and return data into the ***OperationInfo.***

DRAFT



3.4.4.2 JobType Relations

JobType table relates 4 tables (Phase, Job, Batch and MediaType)

Job Type relates Phase, Job and Batch as one to many relation, and relates MediaTypes as Many to Many relation (broken into JobType_MediaType table).

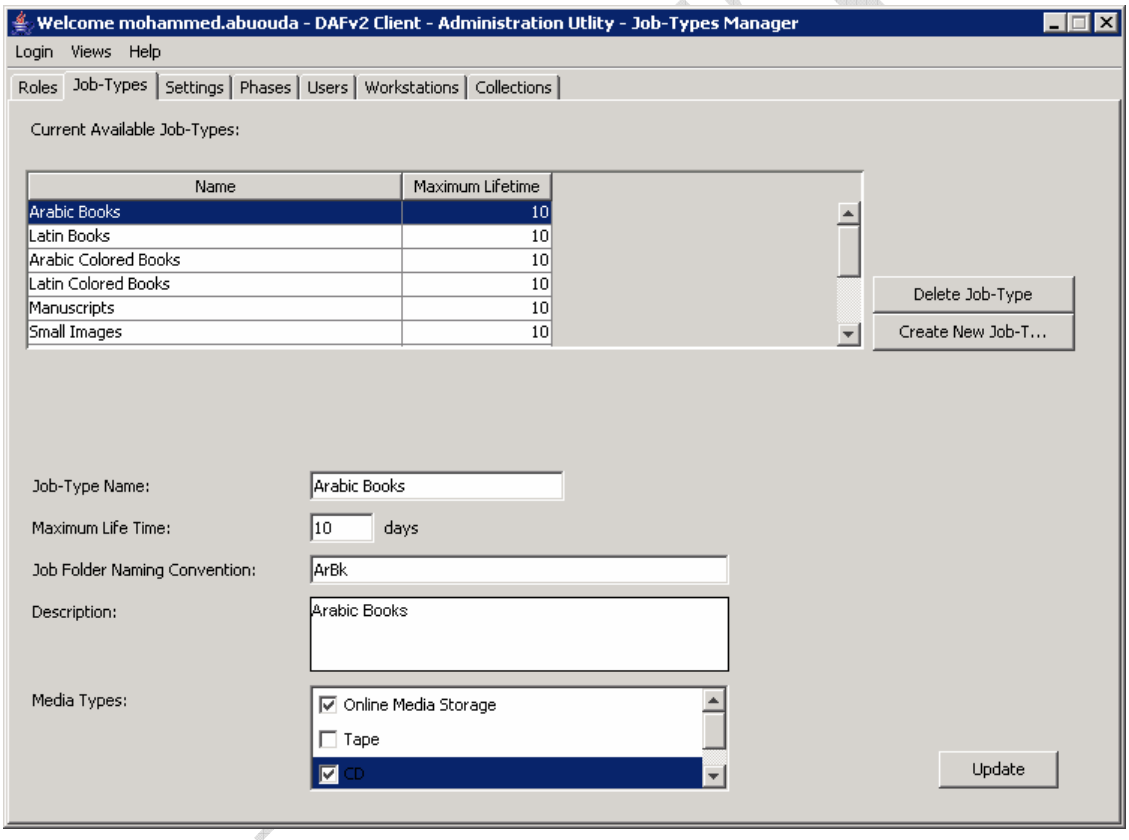
We have discussed the relation between Phases and Job. For Batch, it is similar to Jobs relation's; admin can declare that all Job in a Batch belong to a specific Job Type. Job Type's relation with Media Type affects the archiving module (see Archiving module). It is answers the question "Which Job Type should be archived on Media types?". For example, if the admin needs "Latin Books" to be archived on Tapes, then he/she should assign "Latin Book" to "Tape" media type.

DRAFT

3.4.4.3 JobType GUI

AdminManagerGUI class initializes object of "***JobTypesManagerGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel.*** "***JobTypesManagerGUI"*** has 3 panels**:** jPanel, masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPanel. masterPanel is the container of a sortable table that loads all Job Types of DAFv2. detailsPanel contains JButtons, JTextFields , JTextAreas, CheckBoxLists and JLabels.

On choosing Job Type tab, all Job Types in DAFv2 will be loaded into the sortable table, first row in this table will be selected and details data of the selected record will be adjusted in the details panel. According the base class ***"BaseManagerGUI",*** the abstract function ***reloadManager*** implemented in ***JobTypesManagerGUI***, this function nullify all controls and reload them on tabs change. This helps in getting the last updated data on refreshing tab.



Changing any of details data of a selected row will enable the ***Update*** button; firing this button will save data into the database. Delete and Create buttons complete the 3 basic functionalities (add and delete).

As illustrated in Figure 3.4.3.5, all data fields will be saved into ***JobType*** table, except the data coming from the Media Type check list, will be saved into ***JobType_MediType*** table (see Figure 3.4.3.4).

3.4.5 Phases

Phase is the guide of Job; it's the milestone in DAFv2 workflow. Checks and Actions are embedded into Phase. All Jobs must pass a pre-defined path of Phases to finish their lifecycle properly.

Phases are grouped by Job Types; each Job Type has many Phases (see Job Type). Job Type organizes Phases inside by sequencing them, in other words, all Phases in a Job Type have sequence value, here comes the way marinating workflow. Jobs start, normally, with the least Phase sequence and move automatically to the next sequence.

DAFv2 can maintain also abnormal workflow; consider the following example; a Job Type JT has 3 Phase X, Y and Z. Phases are ordered according to sequences as X, Y and Z. A user has a Job in Phase X and needs to move this one to Phase Z. This scenario will be applied, as such:

1- Job will finish in Phase X.

2- DAFv2 will check the validity of starting the Job in Phase Z.

3- If valid, DAFv2 performs the movement.

4- Job will finish in Phase Z.

DAFv2 will continue afterwards normally until another enforced intervention.

Previous scenario has special Handling with Backup Phase. Backup Phase is an obligatory Phase and must be the Phase with the largest Phase Sequence in its Job Type (the last Phase). Return to previous example, if Phase Z is the Backup Phase, DAFv2 will finish step 4 and do the following:

5- Get the normal flow Phase Sequence (will be Phase Y)

6- Redirect the Job automatically to this Phase

7- Job will continue in flow normally until other intervention

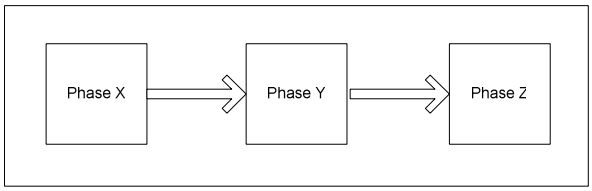
See figure 3.4.4.1.

The concept of changing the flow will be introduced in details in Phase Manager Module through explaining Redirect and Reject functionalities.

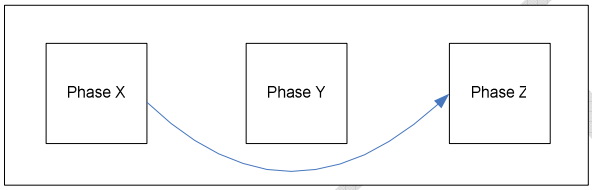
Usually Phases have sequence, but this is not always the case. Sometime Admin creates a Phase and prefers to inject it into the flow now so that he/she can create the Phase without assigning sequence value. This Phase is called "Available Phase" not "Sequence Phase". In terms of "Sequence Phase", the sequence of "Available Phase" must tend to infinity.

DRAFT

Normal Flow



Redirection to Phase Z



Redirection to Backup Phase

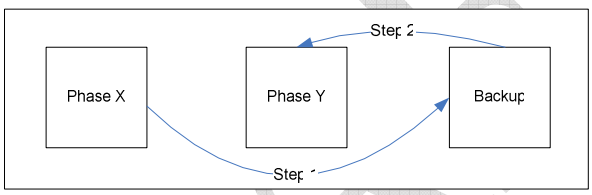


Fig. 3.4.4.1

Phases has significant role in applying checks and actions, all Phases have XML definition that holds checks and actions format. This XML has 3 sections: Pre-Phase, Post Phase and Reflection call. In Pre-Phase, Admin can write checks and actions in XML format to be applied before Job starting in this Phase. Same logic applies to Post-phase, except that the execution of checks and actions will be on finalizing the Job. Sometimes Admin needs to write direct Java code to be implemented either on Pre-Phase or Post-Phase, this can be applied by writing Java code and feeding the XML definition with Java package and method name.

3.4.5.1 Phase Entity

As with ***Job Type*** entity -and all entities- ***Phase*** entity inherits from ***AdminModuleBase*** and implements the three basic functions (add, delete and update). Entity class' constructor has two parameters for specifying working ***Phase*** and ***Job Type***. It initializes database connection, queries the database with ***Phase*** and ***Job Type*** IDs, and sets all class' variables with queried data.

Class' methods implementation is similar to ***Job Type*** Entity implementation. Phase has another class for handing the process of Parsing XML and applying checks, actions and reflection calls.

---- apply phase check methodology comes here

3.4.5.2 Phase Relations

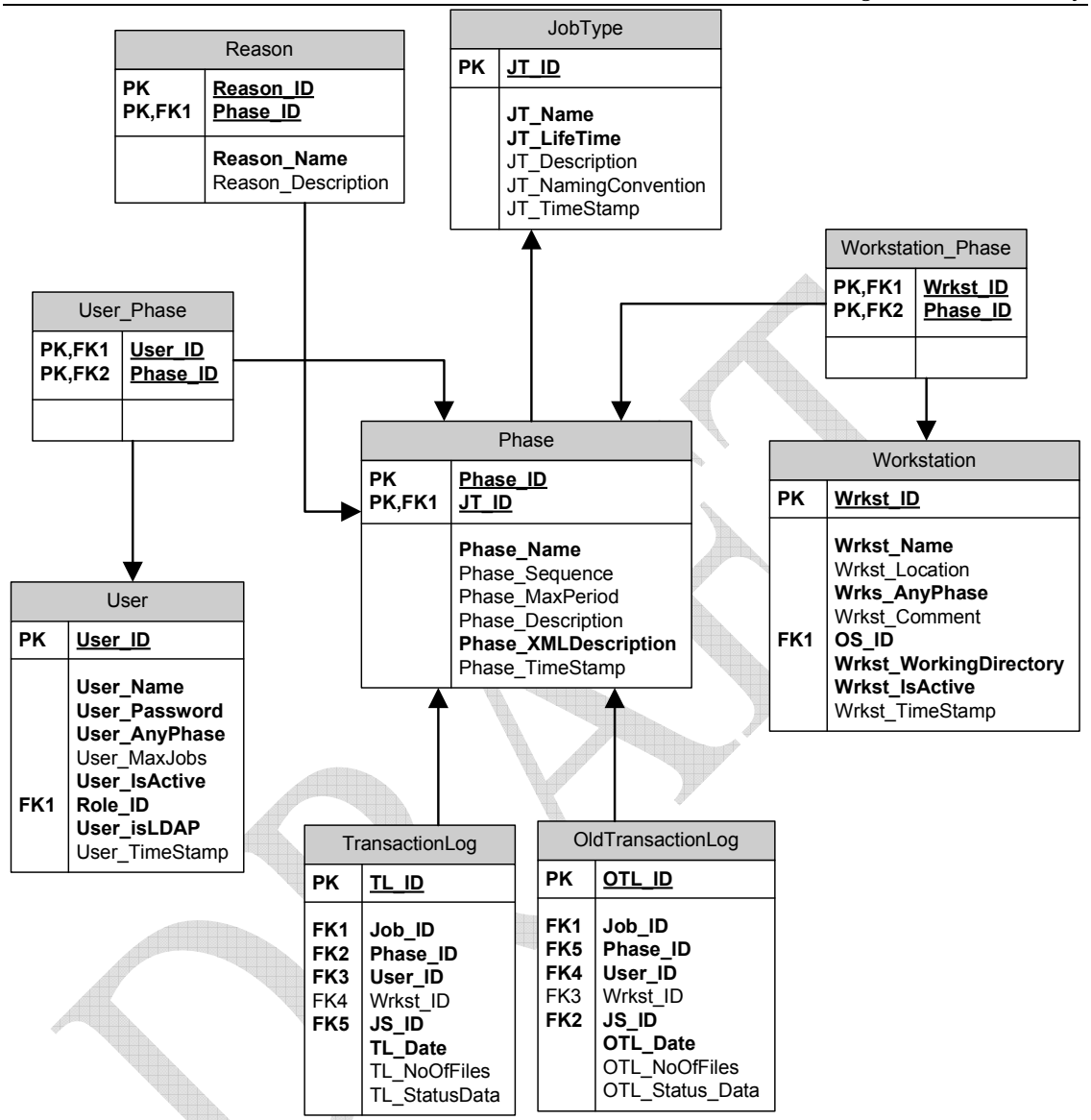
Phase table relates 6 tables (JobType, Use, Workstation, Reason, TransactionLog and OldTransactionLog)

Phase relates User and Workstation as Many to Many relation (broken into User_Phase and Workstation_Phase respectively). It relates Reason, TransactionLog and OldTransactionLog as One to Many relation, and it has One to Many relation with Job Type (Job Time has Many Phases).

Phase relation with User and Workstation helps in defining which User and Workstation can work on the Phase. For example, if User ABC should work on Phase P1 on Workstations W1 and W2, then Admin must set up this matrix. If any element of this matrix is not set, the work will not be accomplished due to the lack of configuration.

Reason table has a set of reasons or problems for every Phase. These reasons help in redirecting and rejecting Jobs (See Phase Manager module). TransactionLog and OldTransactionLog save all transactions for a Job and reference Phase ID in transactions.

DRAFT



3.4.5.3 Phase GUI

AdminManagerGUI class initializes object of "***PhasesManagerGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel.*** "***PhasesManagerGUI"*** has 3 panels**:** jPanel, masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPanel. masterPanel is the container of JComboBox and JList that loads all Job Types and their Phases of DAFv2. detailsPanel contains JButtons, JTextFields , JTextAreas and JLabels.

On choosing Phase tab, all Job Types in DAFv2 will be loaded into the ComboBox, and will load Available Phase and Sequence Phase into two separate Lists. Any selected Phase in Available Phase List will adjust the details panel. Since the ***PhasesManagerGUI*** inherits from the

BaseManagerGUI, the implementation of the reloadManager nullifies all controls and reloads them as tabs. This helps in getting the last updated data on refreshing tab.

Admin can move Phase from Sequence List to Available List using movement buttons. Also the user can change the sequence of the Phases by using the Up and Down buttons then clicking on the Update Sequence Button.

.

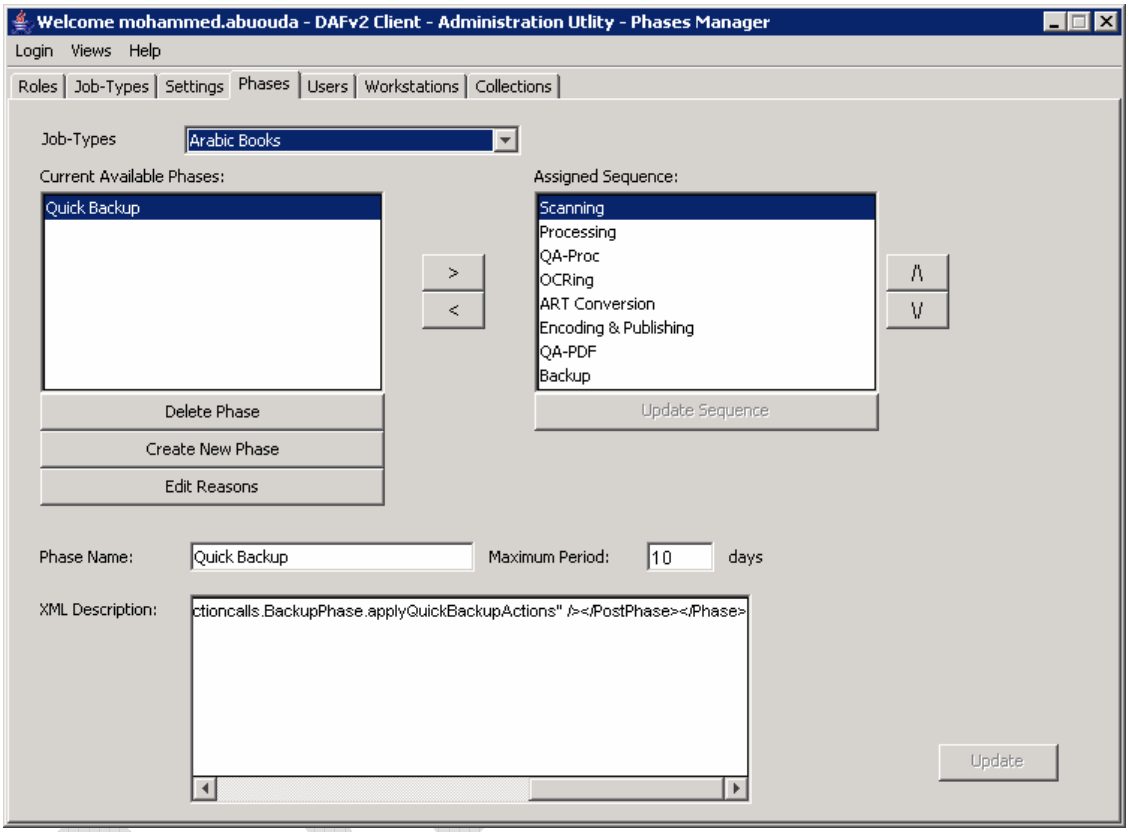


Fig. 3.4.4.3

Changing any of details data of a selected Phase will enable the ***Update*** button; firing this button will save data into the database. Delete and Create buttons complete the 3 basic functionalities (add and delete).

As illustrated in Figure **3.4.4.3**, all data fields will be saved into ***Phase*** table, except the data that comes from Reason GUI, which will be saved into ***Reason*** table (see Figure **3.4.4.2**).

For adding and editing Reasons for a selected Phase, Admin must use Edit Reason button. Reason GUI follows Job Type and Phase also, it has a master panel that loads all reasons for this phase and a details panel for details data. Admin can add, delete and update, using Create, Delete and Update buttons, see figure 3.4.4.4.

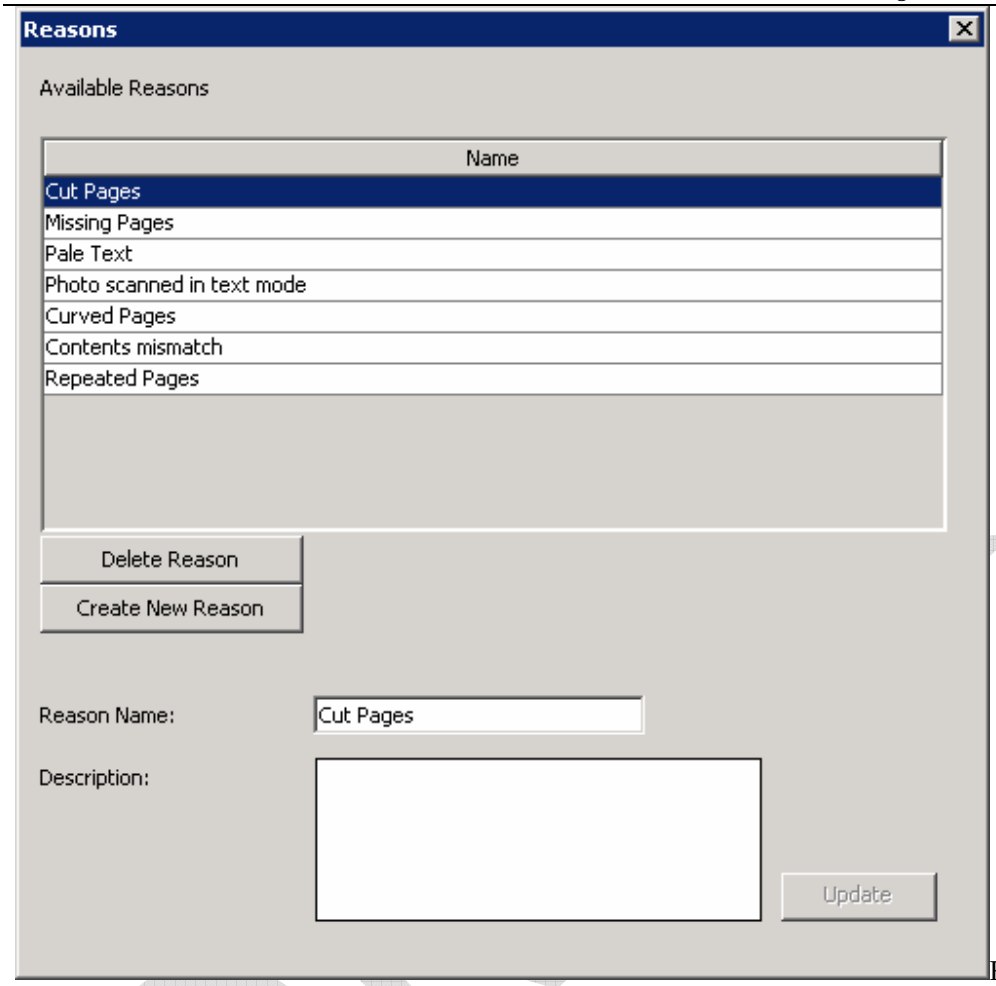


Fig. 3.4.4.4

3.4.6 User

User is the human resource in DAFv2. To login to DAFv2, user credentials must be recorded in the database. Users can be categorized into different system Roles (see Roles); every user must have one and only one role.

User data is used for authentication, authorization and validation; the first two functionalities are interesting in defining who is the logged in user and what this user should work on or be allowed to see. Validation User is interesting in checking if User can work on some Phases on a specific Workstation with Jobs from a special Collection.

Validation process requires the administrator to choose the Phases on which User can work, in addition to defining the maximum number of Jobs that the user can start at a time. Validation process is not limited to User only; it is extended to Collection and Workstation as will be discussed later.

Technically, you can consider User validation as a four dimensional matrix (User, Phase, Workstation and Collection). Admin must take care of filling up this matrix. For simplicity, DAFv2 has distributed matrix filling into 3 tabs: User, Workstation and Collection. In User tab, admin can link up User with Phases. In Workstation tab, admin can link up Workstation with Phase and, finally, admin can link up Collection with User and Workstation in Collection tab.

Each User in DAFv2 has a specific number of maximum Jobs that he/she can work on at a time. For example, if this number equals 2, User cannot work on more than those 2 Jobs at the same time. This condition with matrix condition must be satisfied in order to start a Job correctly.

DAFv2 schema has some relaxation fields that reduce the settings of validation matrix; a User who has ***All Phase*** field is enabled to work on all Phases if ***All Phase*** was set. These relaxation fields are applied to Workstation and Collection.

3.4.6.1 User Entity

User entity inherits from ***AdminModuleBase*** and implements the three basic functions (add, delete and update). Entity class' constructor has one parameter for specifying working ***User***. It initializes database connection, queries the database with ***User*** ID and sets all class' variables with queried data.

Class' methods implements basic entity functions (load, add, delete and update), in addition to authentication and authorization function.

User Authentication and Authorization

User authentication applies three methods in order authenticate a User

1- DAFv2 detects the logged in User; if this User Name is register in DAFv2 database then DAFv2 will authenticate this User.

2- If a User changes the logins through log in dialog; then DAFv2 will check if the User Name and Password are matched with a record in DAFv2 database. If this scenario is failed, DAFv2 tries the last method.

3- DAFv2 will check if User name is register in DAFv2 database and will compare the Password with User's domain password (for LDAP users only). Users can be declared as ***LDAP (Lightweight Directory Access Protocol)*** using User GUI.

DRAFT

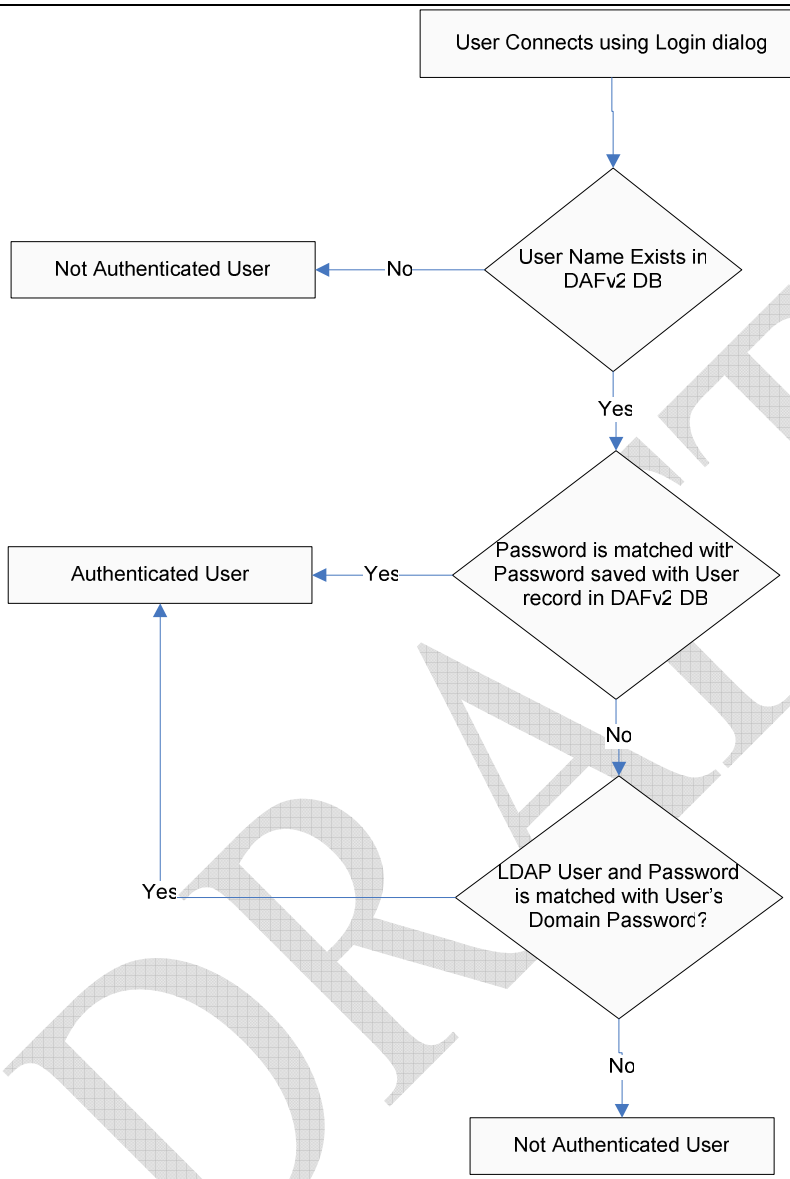


Figure 3.4.5.1

User authorization comes through implementing methods that check ***User's Role*** and this ***Role's Permissions*** (see Role).

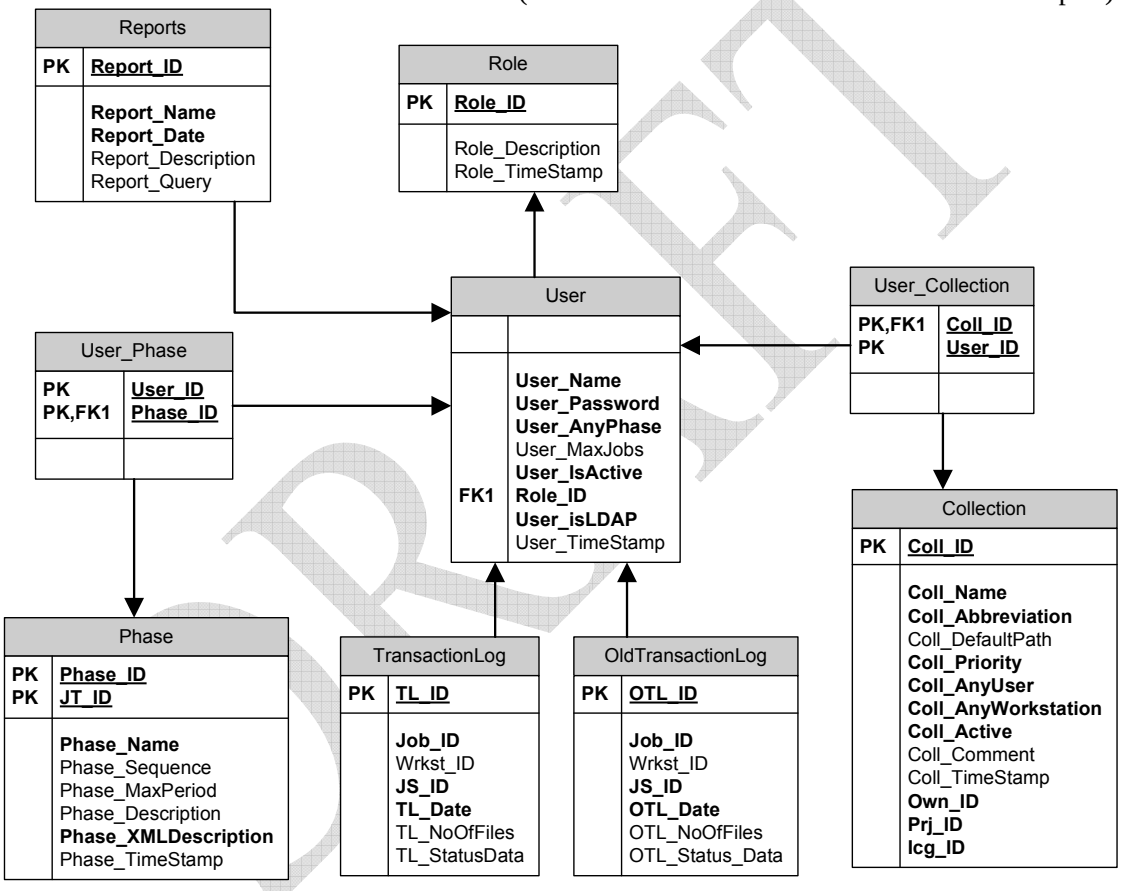
3.4.6.2 User Relations

User table relates 6 tables (Role, Reports, Phase, Collection, TransactionLog and OldTransactionLog)

User relates Phase and Collection as Many to Many relation (broken into User_Phase and User_Collection respectively). It relates Reports, TransactionLog and OldTransactionLog as One to Many relation, and it has One to Many relation with Role (Role has Many User).

User relation with Phase helps in defining which User can work on which Phase. For example, if User ABC should work on Phase P1 then Admin must set up this relation using User_Phase table. Relation with Collection defines User domain Collections.

As mentioned before, every User has one system's Role; this relation is maintained using Role_ID filed in User table. Reports, TransactionLog and OldTransactionLog are reference the User who did the action (made a transaction or a report).



3.4.6.3 User GUI

AdminManagerGUI class initializes object of "***UsersManagerGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel.*** "***UsersManagerGUI"*** has 3 panels**:** jPanel, masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPane. MasterPanel is the container of a sortable table that loads all Users of DAFv2. detailsPanel contains JButtons, JTextFields , JComboBoxs, JTrees, JLists, JCheckBoxs and JLabels.

On choosing User tab, all Users in DAFv2 will be loaded into the sortable table. First row in this table will be selected and details data of the selected record will be adjusted in the details panel.

According the base class ***"BaseManagerGUI",*** the abstract function ***reloadManager*** implemented in ***UsersManagerGUI***, this function nullifys all controls and reload them on tabs change. This helps in getting the last updated data on refreshing tab.

Admin can assign any of the available phases, introduced in Available Phases Tree and grouped by Job Type to User, Using movement buttons. Assigned Phases represent the Phases that User can work on. Admin can relax User_Phase relations by checking ***All Phases*** check box. Other controls are for choosing User Role, setting maximum Jobs' number , setting User Name and Password, setting User as LDAP or not and activating or deactivating User.

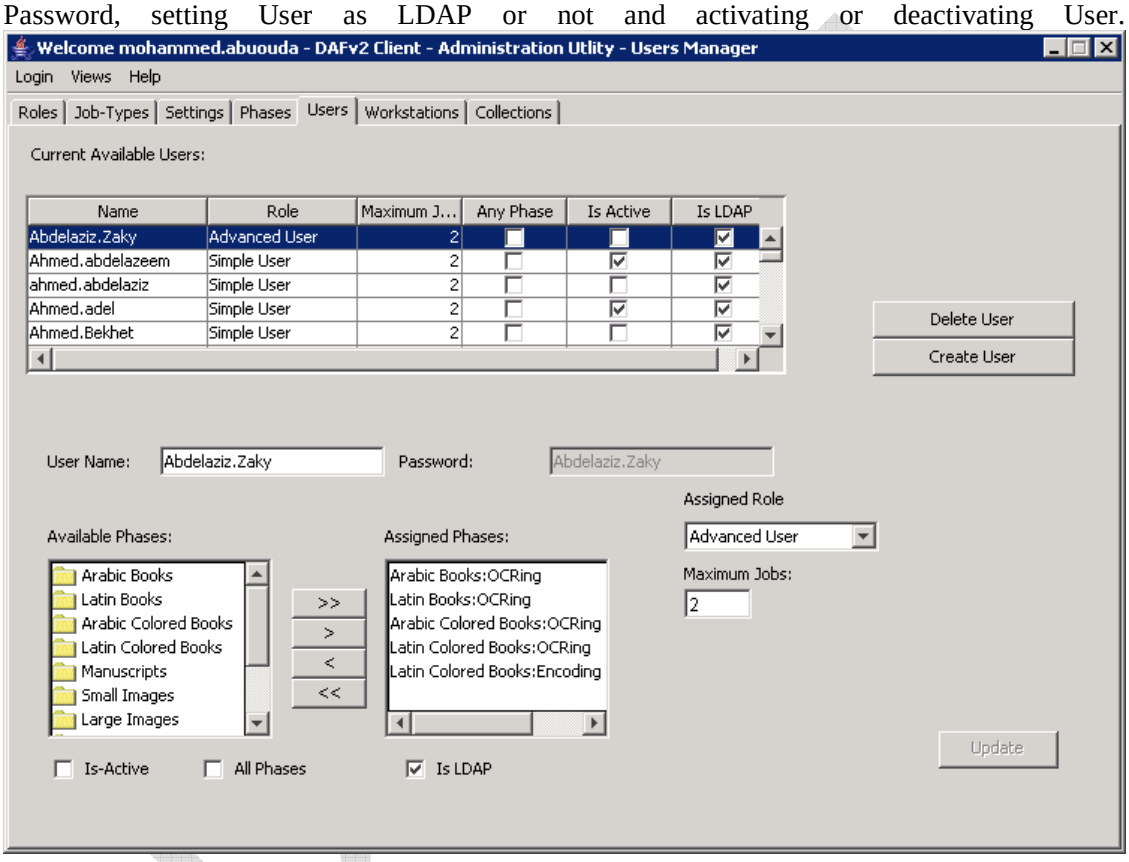


Fig. 3.4.5.3

Changing any of details data of a selected row will enable the ***Update*** button, firing this button will save data into the database. Delete and Create buttons complete the 3 basic functionalities (add and delete).

As illustrated in Figure 3.4.5.3, all data fields will be saved into ***User*** table, except the data that comes from Assigned Phases List, which will be saved into ***User_Phase*** table (see Figure 3.4.5.2).

3.4.7 Workstation

DAFv2 has two types of machines; Workstations and Storage servers. Workstations are machines used by User to accomplish Jobs. Storage server is used to maintain workflow (see Phase Manager Module).

Workstation is one edge of a 4D validation matrix; it relates to Phase and Collection. Therefore, admin must define Phases that the Workstation can work on and the Collection also. The concept of Phase relaxation is applied to Workstation as well as User, each Workstation has ***All Phases*** attribute carrying out this functionality.

Admin determine Phases that can be worked on a Workstation according to Workstation's attached Devices (Scanners, Plotters,…) or installed software (OCRing tool).

Workstation is the User arena; he/she can download, edit and submit Jobs from Workstation. So, each Workstation has a working folder attribute that specifies every User's working directory. Working directory value is preferred to be function of User, in other words, every User should has separate working directory on one Workstation in order to avoid overlapping. For example, consider a Workstation WS and Users U1 and U2. Working directory function for WS is "D:\User Data\$username$", trace this actions

1- U1 logged in WS

2- DAFv2 calculates U1 working directory on WS by replacing "$username$" placeholder with U1 name. This results in working directory "D:\User Data\U1"

3- U1 logged out WS

4- U2 logged in WS

5- DAFv2 calculates U2 working directory with same logic and gives the path "D:\User Data\U2".

This indicates that there will be a private and a separate working folder for every User. It should be mentioned that User Name filed in User table is unique, thus DAFv2 will not create one directory for two or more Users. The concept of separation is required in order to let every User be the sole controller on his/her work (this is not applicable on administrator; he/she has privileges to control every thing in DAFv2).

Workstation can be configured to be more that a working area, it can be an archiving tool (see Archiver Manager Module). Suppose that Jobs must be archived on Blu-Ray or Tape; based on this, Admin must declare that there should be Workstations thathave Blu-Ray or Tapes drivers in order to write data on Medias. This declaration is what we call Workstation-MediaType relation ; admin must identify Media Types capabilities on each Workstation to help Archiver Manager. If Archiver Manager works on a Workstation that is not configured to accept archiving media, the operation will fail.

DAFv2 accepts identifying Devices (Scanner, Printers…). Admin can declare that a Device X, Y or Z is attached to Workstation WS1.

DRAFT

Workstations in DAFv2 can be without any attached peripherals (Devices or Media Types Drivers), such Workstations function as working area for Users.

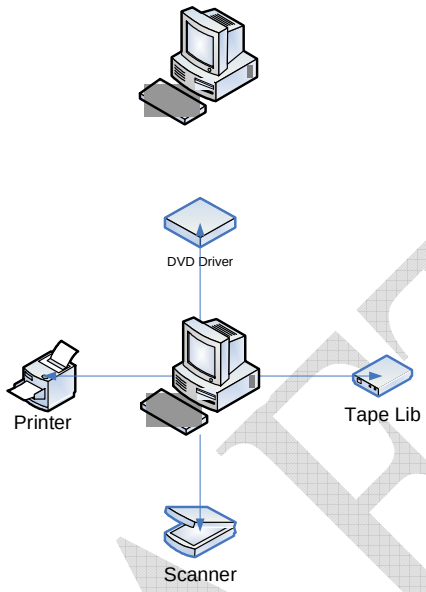


Fig. 3.4.6.1 Workstation can archive on Taps and DVD and have Scanner and Printer

3.4.7.1 Workstation Entity

Workstation entity inherits from ***AdminModuleBase*** and implements the three basic functions (add, delete and update). Entity class' constructor has one parameter for specifying working **Workstation**. It initializes database connection, queries the database with **Workstation** ID and sets all class' variables with queried data. Class' methods implements basic entity functions (load, add, delete and update).

Authenticate Workstation depends on Workstation IP or Workstation Name. Firstly, DAFv2 checks if logged in Workstation name is registered in the database or not. If not, DAFv2 tries to check if Workstation IP is recorded in the database or not. Success of any of the previous scenarios authenticates the Workstation.

3.4.7.2 Workstation Relation

Workstation table relates 7 tables (Phase, Collection, Device, MediaType, OperationSystem, TransactionLog and OldTransactionLog)

Workstation relates Phase, Collection, Device and MediaType as Many to Many relation (broken into Workstation_Phase, Workstation_Collection, Workstation_Device and Workstation_MediaType respectively). It relates TransactionLog and OldTransactionLog as One to Many relation, and it has One to Many relation with OperatingSystem (many workstations may have the same operating system).

DRAFT

Workstation relation with Phase helps in defining which Phases can work on which Workstations. Also the Workstation_Collection defines which Workstation can work on which Collections. Defining peripherals is done using Workstation_Device and Workstation_MediaType which declares attached devices and Media Types recording availability on a Workstation.

Relation relaxation is applied to Workstation as well as Phase, the filed Wrks_AnyPhase shortcuts linking Workstation with all Phases. Collection Relation has two relaxation fields (AnyUser and AnyWorkstaion), thus completing the concept of Relation relaxation that will be discussed in Collection section.

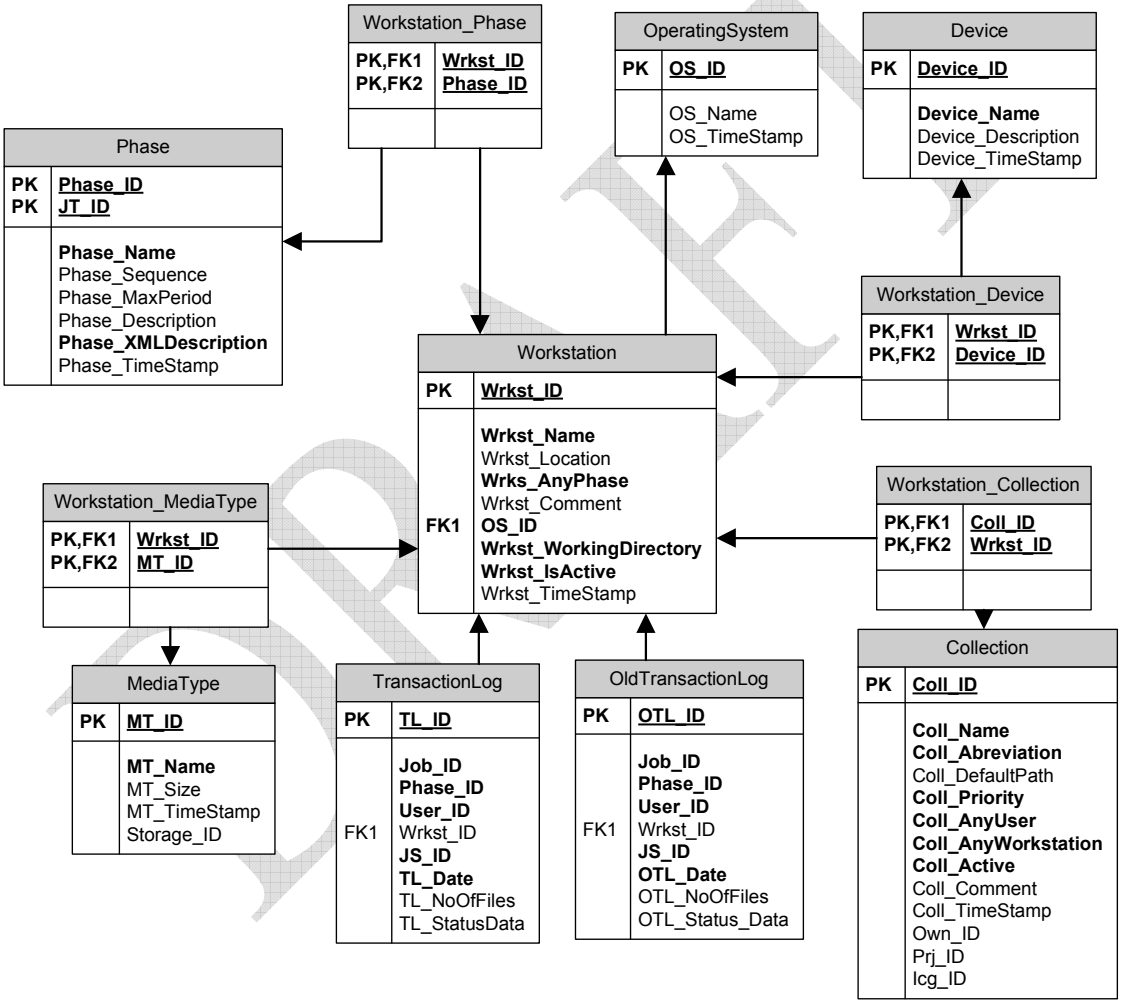


Fig 3.4.6.2

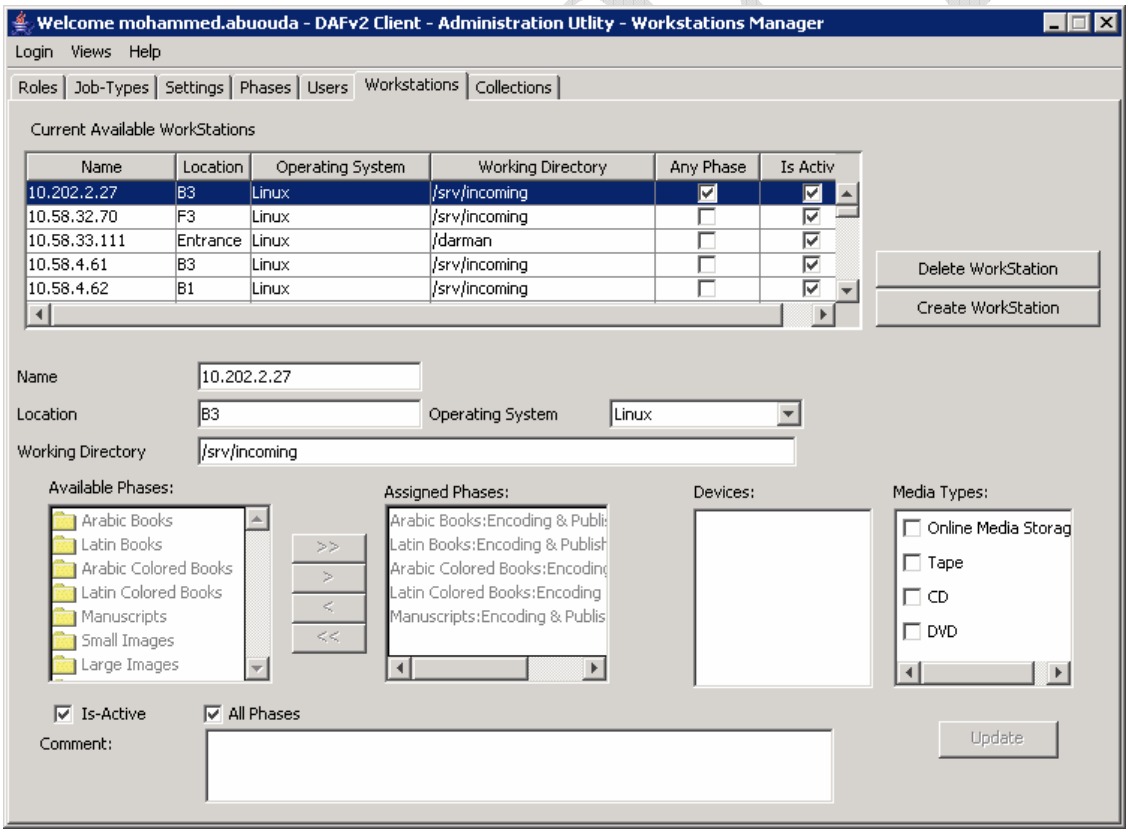
3.4.7.3 Workstation GUI

AdminManagerGUI class initializes object of "***WorkstationsManagerGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel.*** "***WorkstationsManagerGUI"*** has 3 panels**:** jPanel,

masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPane. MasterPanel is the container of a sortable table that loads all Users of DAFv2. detailsPanel contains JButtons, JTextFields , JComboBoxs, JTrees, JLists, JCheckBoxs, CheckBoxList and JLabels.

On choosing Workstation tab, all Workstations in DAFv2 will be loaded into the sortable table, first row in this table will be selected and details data of the selected record will be adjusted in the details panel. Since the ***WorkstationsManagerGUI*** inherits from the ***BaseManagerGUI*** , the implementation of the ***reloadManager*** nullifies all controls and reloads them as tabs change. This helps in getting the last updated data on refreshing tab.

Admin can assign any of the Available Phases, introduced in the Available Phases Tree and grouped by Job Type to Workstation, using movement buttons. Assigned Phases represent the Phases that a Workstation can work on. Admin can relax Workstation_Phase relation by checking ***All Phases*** check box. Other controls are for choosing Operating System, defining name or IP or working directory, and setting the attached Devices or Media Types.



3.4.8 Collections

DAFv2's ***Collections*** is the logical grouping of ***Jobs***. ***Job*** terminology is used widely in DAFv2 system; it indicates any object that needs to be digitized, or any item in DAFv2. This includes Books, Images, Manuscripts, etc. Every ***Job*** must fall into 2 logical groups: ***Job Type*** and ***Collection.*** A new ***Collection*** is to be introduced when a new project starts or an existing project has a new Owner (as the ***Collection*** has only one owner, this will be changed to allow multiple owners per collection).

Collection of ***Jobs*** is a grouping of physical ***Jobs***, it has no actions. ***Jobs*** frequently come in ***Batches*** which are organized in ***Collections.*** Figure 3.4.3.2 shows the ***Collection.***

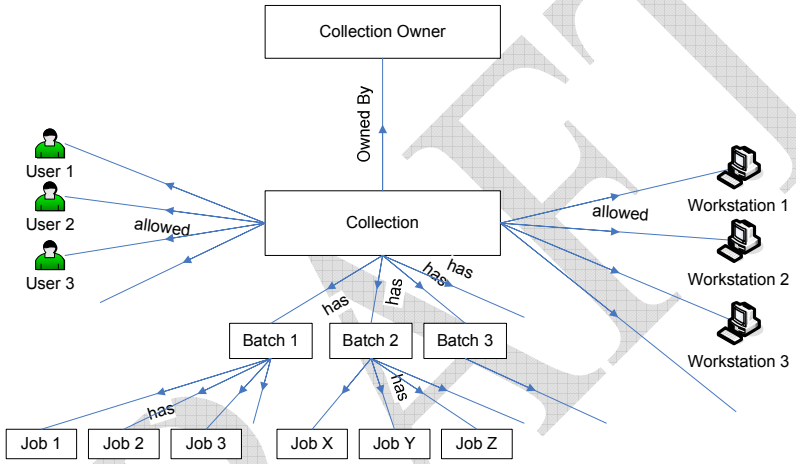


Fig. 3.4.3.2

3.4.8.1 Collection Entity

This entity inherits ***AdminModuleBase*** and implements the three basic functions (add, delete and update).

Entity class' constructor has one parameter for specifying working ***Collection***. It initializes database connection, queries the database with ***Collection*** ID and sets all class' variables with queried data. Thus, initializing ***Collection*** entity loads all its data in memory and gives the developer opportunity to work on this data offline (without database connection), unless he/she uses an operation that needs database connection (add, delete or update). Achieving this goal needs implementing setter and getter functions for all fields.

Class' methods (neither setters nor getters) have implemented based on following concepts:

9- Set the necessary variables (fields)

10- Prepare array list of these variables (argument list)

11- Pass store procedure name and argument list to database handler

12- Receive data (if exists) from database into OperationInfo object (see database handler)

Find the following code snippets

```
1. public  OperationInfo update(){
2.      ArrayList<Object> argList = new ArrayList<Object> (10);
3.      argList.add(new Integer(id));
4.      argList.add(new String(collName));
5.      argList.add(new String(collAbrevation));
6.      argList.add(new String(collDefaultPath));
7.      argList.add(new Integer(collPriority));
8.      argList.add(new Boolean(collAnyUser));
9.      argList.add(new Boolean(collAnyWorkstation));
10.     argList.add(new Boolean(collActive));
11.     argList.add(new String(collComment));
12.     argList.add(new Integer(collOwnID));
13.     return conn.executeProcedure("Collection_Update",argList);
14.}
```

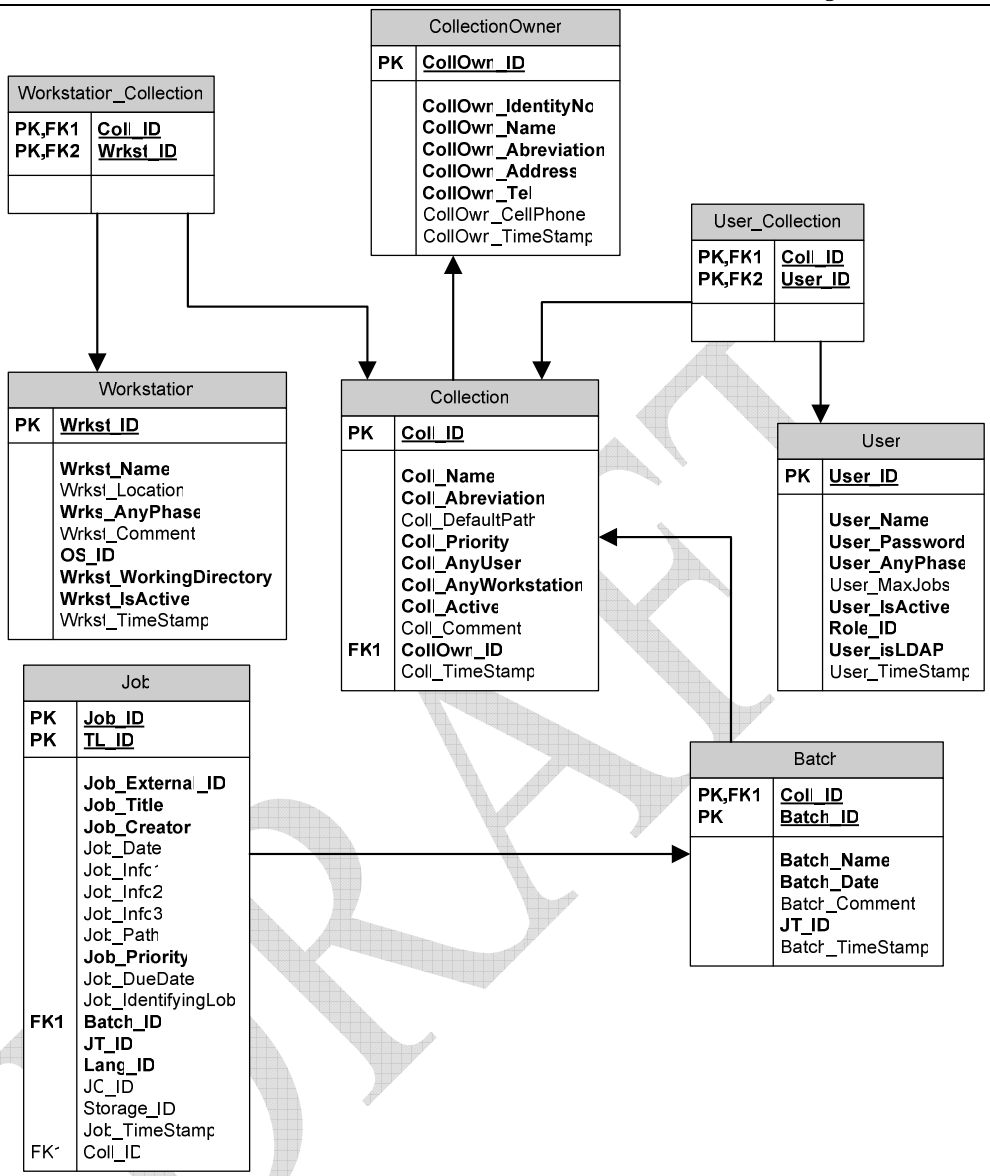
Line 1: Declares the function signature which returns ***OperationInfo*** object.

Line 2: Prepares argument list

Lines 3-12: Sets argument list

Lines 13: Passes stored procedure name and argument list to database connection and return data into the ***OperationInfo.***

DRAFT



3.4.8.2 Collection Relations

The Collection table is part of 6 relationships:

1- Owner: each Collection has an Owner (Single owner). The Collection-Owner relation does not affect the flow or the behavior of the Job in the system in any way. The Owner is a View in the Metadata DB and is not modified within the DAF system.

2- Project: each Collection has a Project (Single Project). The Collection- Project relation does not affect the flow or the behavior of the Job in the system in any way. The Project is a View in the Metadata DB and not modified within the DAF system.

3- In-Charge: each Collection has an In-Charge (Single In-Charge). The Collection- In-Charge relation does not affect the flow or the behavior of the Job in the system in any way. The In-Charge is a View in the Metadata DB and not modified within the DAF system.

4- User: Each collection is allowed only a set of Users to work on, due to special skills required for this collection or just for organizational purposes. To represent this relation another table is added (User_Collection) as this is a many to many relation. Jobs in Collections that are not legitimate to Users do not appear for these Users while retrieving a new Job. This relation can be relaxed by setting the Coll_AnyUser (one of the Collection table fields) flag to true. In this case any User will be able to operate on this Collection.

5- Workstation: Same as Users, each collection is allowed only a set of Workstations to work on, due to special devices needed for this collection or just for organizational purposes. To represent this relation another table is added (Workstation_Collection) as this is a many to many relation. Jobs in Collections that are not legitimate to Workstations appear in the Get Job screen without allowing the User to select them. If the User would like to work on one of these Jobs then he/she has to change the Workstation he/she is using. This relation can be relaxed by setting the Coll_AnyWorkstation (one of the Collection table fields) flag to true. In this case any Workstation can be used to work on Jobs from this Collection.

6- Batch: Jobs are joined to Collections using Batches, a Batch could be a logical grouping or a time grouping of Jobs within this Collection. A Batch can be contained in only one Collection and can have any number of Jobs. Please refer to the Batch Creation and Modification section in the Check-In module.

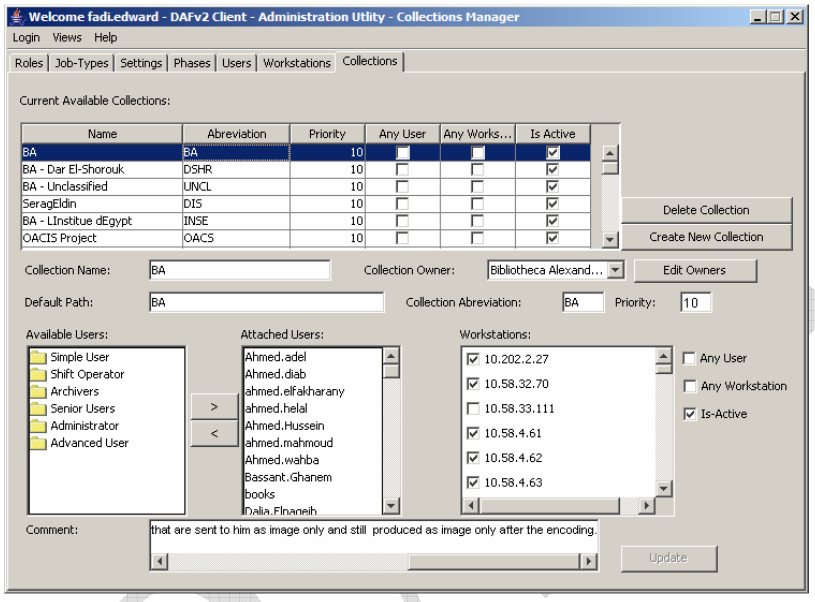
For example, we have a collection that should use only high quality equipments and very skilled users due to its historical importance. This Collection is Owned by a party that only lent the material to the Digitization Lab for a short amount of time. This Collection will be represented in the system by: 1- Adding the Borrower as the CollectionOwner, 2- Adding the very skilled Operators only to this Collection and setting the Coll_AnyUser to false. 3- Adding only the Workstations equipped with the latest equipments and setting the Coll_AnyWorkstation flag to false. 4-Creating Batches for this collection according to the actual physical Batches they come in. 5- And finally adding Jobs to the Batches of this Collection accordingly.

3.4.8.3 Collection GUI

AdminManagerGUI class initializes object of "***CollectionsManagerGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel.*** "***CollectionsManagerGUI"*** has 3 panels**:** jPanel, masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPanel. masterPanel is the container of a sortable table that loads all Collections of DAFv2. detailsPanel contains JButtons, JTextFields , JTextAreas, CheckBoxLists and JLabels that display and edit the details of the selected ***Collection*** from the masterpanel->SortableTable or when adding a new Collection.

DRAFT

When the user chooses the Collections tab, all Collections in DAFv2 will be loaded into the sortable table, first row in this table will be selected and details data of the selected record will be adjusted in the details panel. According the base class ***"BaseManagerGUI",*** the abstract function ***reloadManager*** implemented in ***CollectionsManagerGUI***, this function nullify all controls and reload them on tabs change. This helps in getting the last updated data on refreshing tab.



Changing any of the details data of selected row will enable the ***Update*** button, firing this button will save data into the database. Delete and Create buttons complete the 3 basic functionalities (add and delete).

According to the Collection Entity relations, data from the CollectionManagerGUI is mapped in to 3 tables,

a. The Collection Information like Collection Name, Collection Abbreviation, Priority, Any User (allows any user to work on this Collection regardless of the Attached Users), Any Workstation (allows any workstation to work on), Is Active (used to reduce lookup lists) and the Comment are stored in the Collection table as well as the Owner, Project and In-Charge relations.

b. The Attached Users are stored in the User_Collection table. On each update, all the Users for this collection are deleted and then re-added from scratch.

c. The Attached Workstations are stored in the Workstation_Collection table. On each update all the Workstations for this Collection are deleted and then re-added from scratch.

3.4.9 General Settings

DAFv2 has infrequent modifications in entities such as (Device, Storage, job Condition, , Operating System and Media Type). One can create or update these entities at one time. DAFv2 grouped these setting in single GUI in Admin Manager Module; they are located under "Settings" tab. Admin can choose the targeted entity to be change through Combo Box contains all these entities and apply whatever change he/she need.

3.4.9.1 Device Entity

As mentioned in Workstation section, Workstation may have attached devices, like scanners, or Printers. Admin can define or edit devices using Device entity. Find Device relation in figure 3.4.7.1.

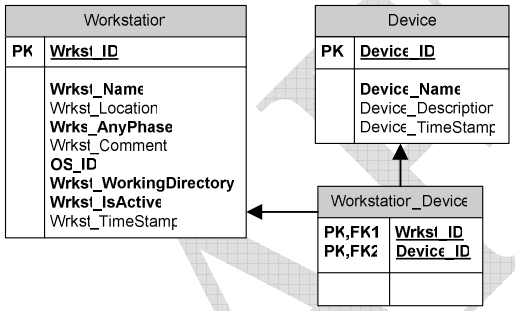


Fig. 3.4.7.1

3.4.9.2 JobCondition Entity

Every Job in DAFv2 has Condition state; it could be Good, Bad or Unknown. Find Job Condition in figure 3.4.7.2.

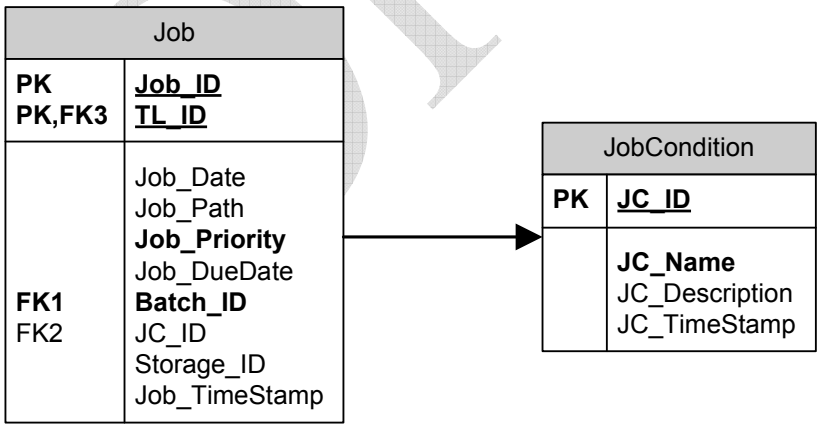


Fig. 3.4.7.2

3.4.9.3 OperatingSystem Entity

Workstation must have an Operation System; Admin can create, edit or delete an Operating System in DAFv2. Find Operaing System relation in figure 3.4.7.3.

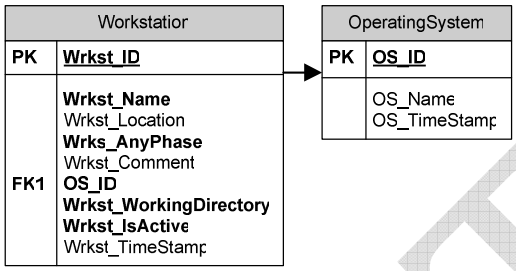


Fig. 3.4.7.3

3.4.9.4 Storage Entity

DAFv2 uses Storage during ordinary workflow to upload and download jobs from storage server to a User's workstation and vice versa. Storage is defined using IP, User Name and Password.

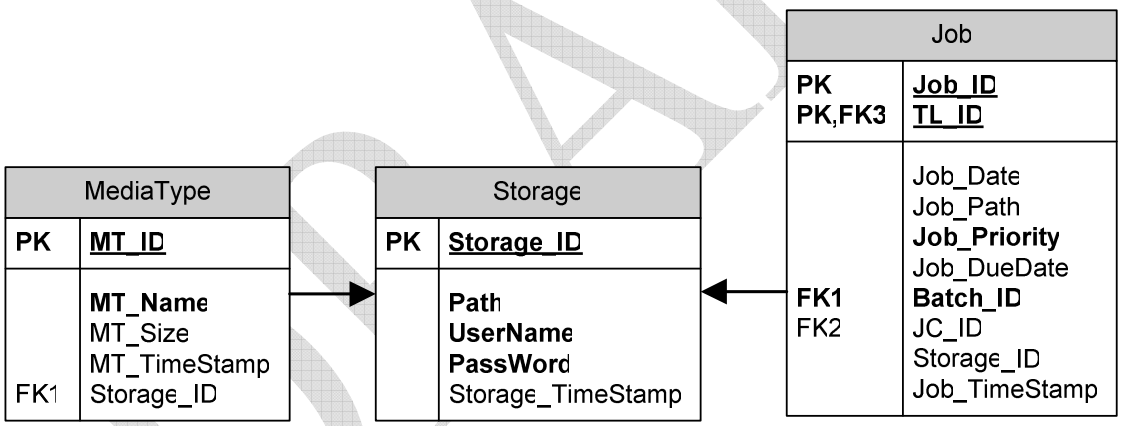


Fig. 3.4.7.3

3.4.9.5 MediaType Entity

To archive Jobs, they must be written physically on Media Types. Media Types examples are (CDs, Tapes, OnlineStorage, Blu-Ray or Professional Disc for DATA (PDD)). Admin decides which Media Type he/she may use according to the availability of the Media and Media Driver he/she has.

DRAFT

Supporting Online Storage is feature of DAFv2; it can be beneficial if the system has an Online Storage or very large scale storage. Archiving data into Online Storage does not need Drivers to write data, furthermore, it needs FTP service and User credentials to write data on.

Admin must configure FTP service on the Online Storage and fed up data to DAFv2, to make DAFv2 able to archive data on it. Find Media Type relation in figure 3.4.7.4.

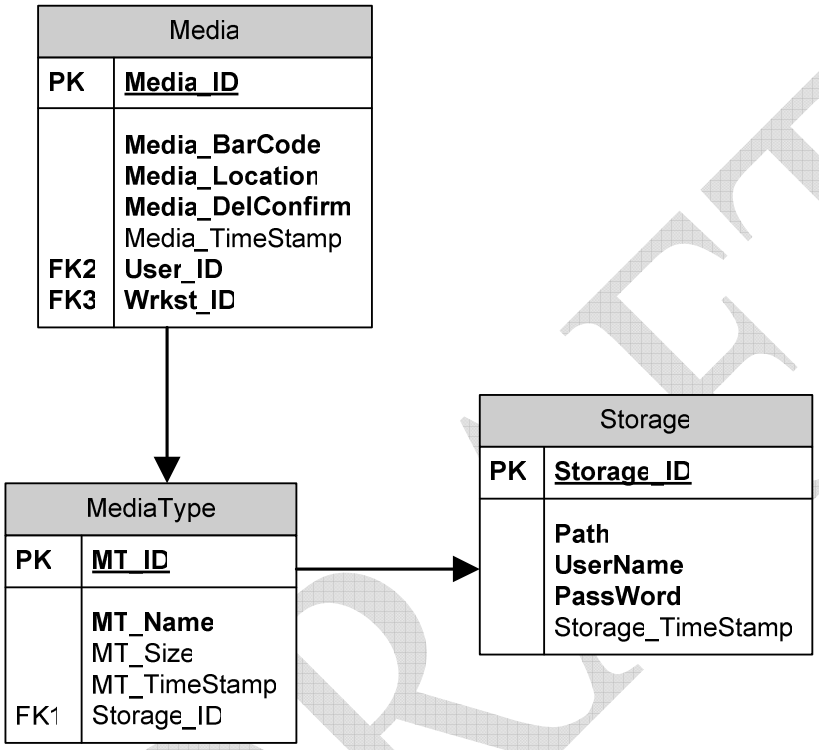


Fig. 3.4.7.4

3.4.9.6 Settings GUI

AdminManagerGUI class initializes object of "***GeneralSettingsGUI",*** which inherits ***"BaseManagerGUI",*** which inherits ***JPanel.*** "***GeneralSettingsGUI"*** has 3 panels**:** jPanel, masterPanel and detailsPanel. jPanel is the container of masterPanel and detailsPane. masterPanel is the container of a Setting JComboBox that loads all of DAFv2 settings. detailsPanel contains JButtons, JTextFields and JLabels.

On choosing Setting tab, all setting entities will be loaded into the JComboBox, any selected entities will load their data into JList, show the controls related to their entity, and disable irrelevant controls to the selected entity. For any selection in the JList, details data of the selected record will be adjusted in the details panel. According the base class ***"BaseManagerGUI",*** the abstract function ***reloadManager*** implemented in ***GeneralSettingsGUI***, this function nullify all controls and reload them on tabs change. This helps in getting the last updated data on refreshing tab. Find figures below.

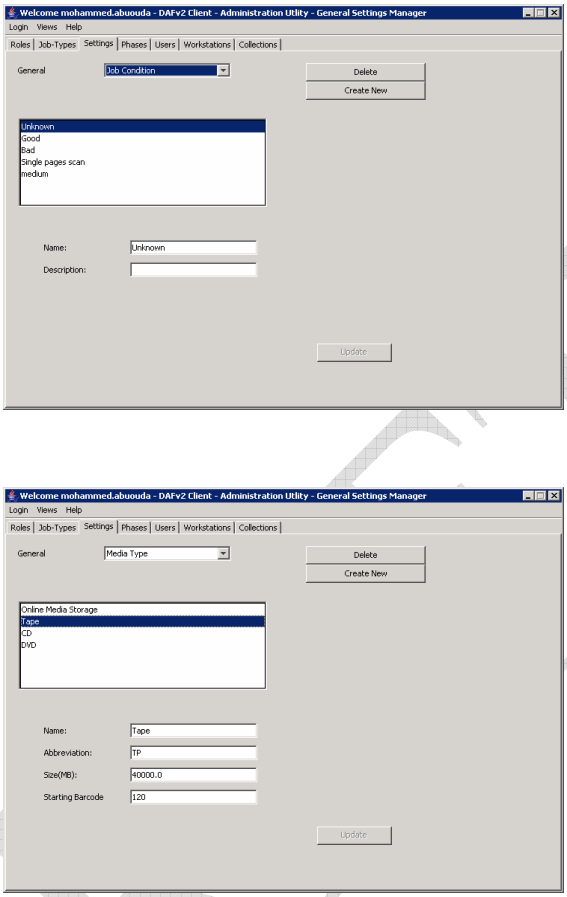


Fig 3.4.7.5

3.5 Reporting Module

Due to the large number of Jobs in DAFv2 workflow, a monitoring or Reporting tool becomes significant. Reporting module provides monitor for the reporter user to find a Job or define its state and measure User performance and rate.

DAFv2 has four types of reports with four different functionalities;

1. System Workflow report
2. Pending Jobs
3. Late Jobs
4. User rate

Each of the previous will be discussed in detail in the following subsections.

For flexibility's sake, the Reporting module gives the reporter opportunity to create his/her custom report through a report builder. Report builder gives only the administrator ability to write a direct query.

3.5.1 System Workflow

All Jobs in DAFv2 can be classified into Active or Inactive Jobs (revise CheckOut section, Archiving Module). System Workflow is concerned with the Active Jobs only (Jobs in TransactionLog table not OldTransactionLog).

According to DAFv2 state diagram (Fig 3.5.1), every Job can be labeled as a member of one of the following three groups;

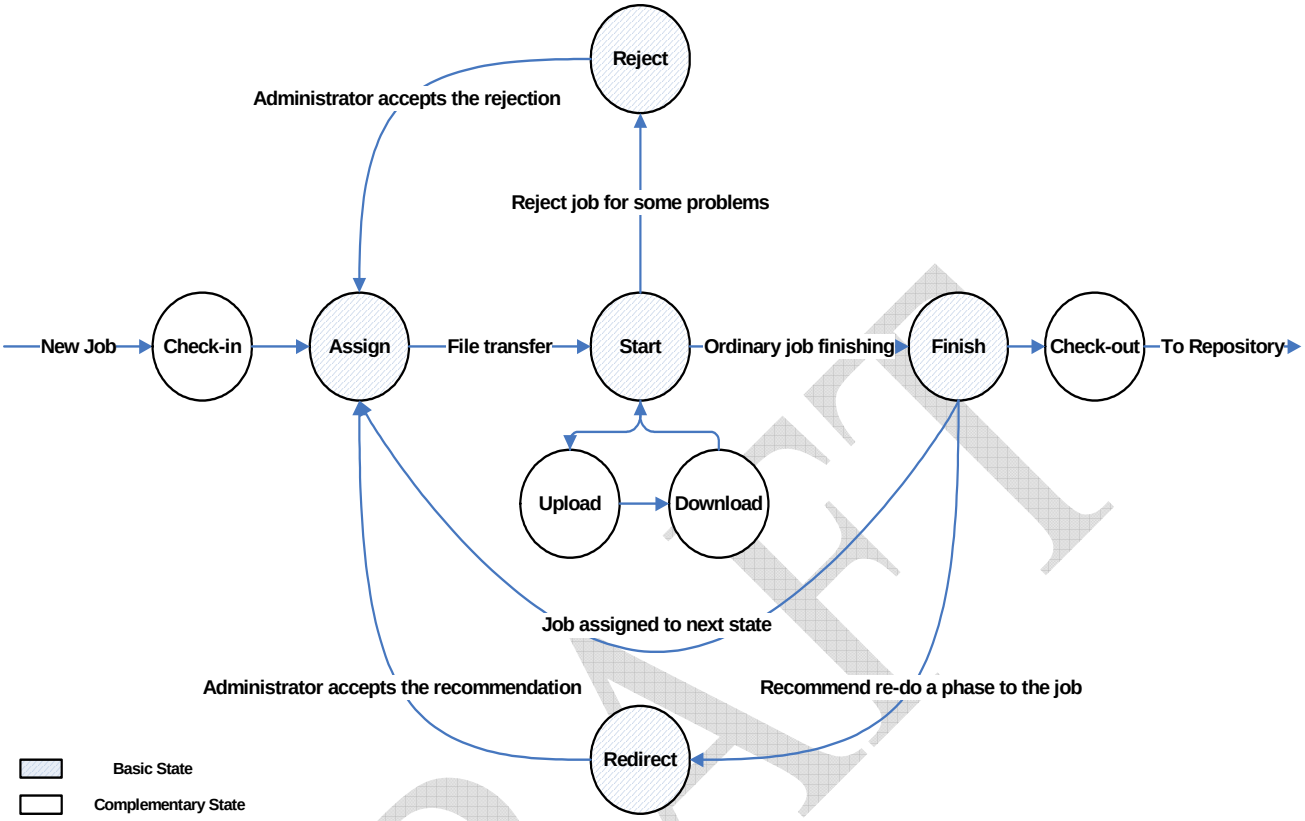
a. Pending
b. Processing
c. Finished

Pending Jobs group includes all Jobs that are in ***Assign***, ***Redirect*** or ***Reject*** state.

Processing Jobs group includes all Jobs in ***Start*** state.

Finished Jobs group include all Jobs in ***Finish*** state.

DRAFT



DRAFT

Fig. 3.5.1 State Diagram

There will not be any Job out of this classification, as the states (Assign, Redirect, reject, Start and Finish) are the ***DAFv2 Basic States.*** In other words, **Complementary States** can be reduced to ***Basic States***, i.e. if a Job in Download or Upload state, it can be considered as in Start state.

The System Workflow report provides mainly Jobs' states of a selected Job Type. It can go in depth by selecting Collection or Batch. Resultant data are distributed between selected Job Type's Phases.

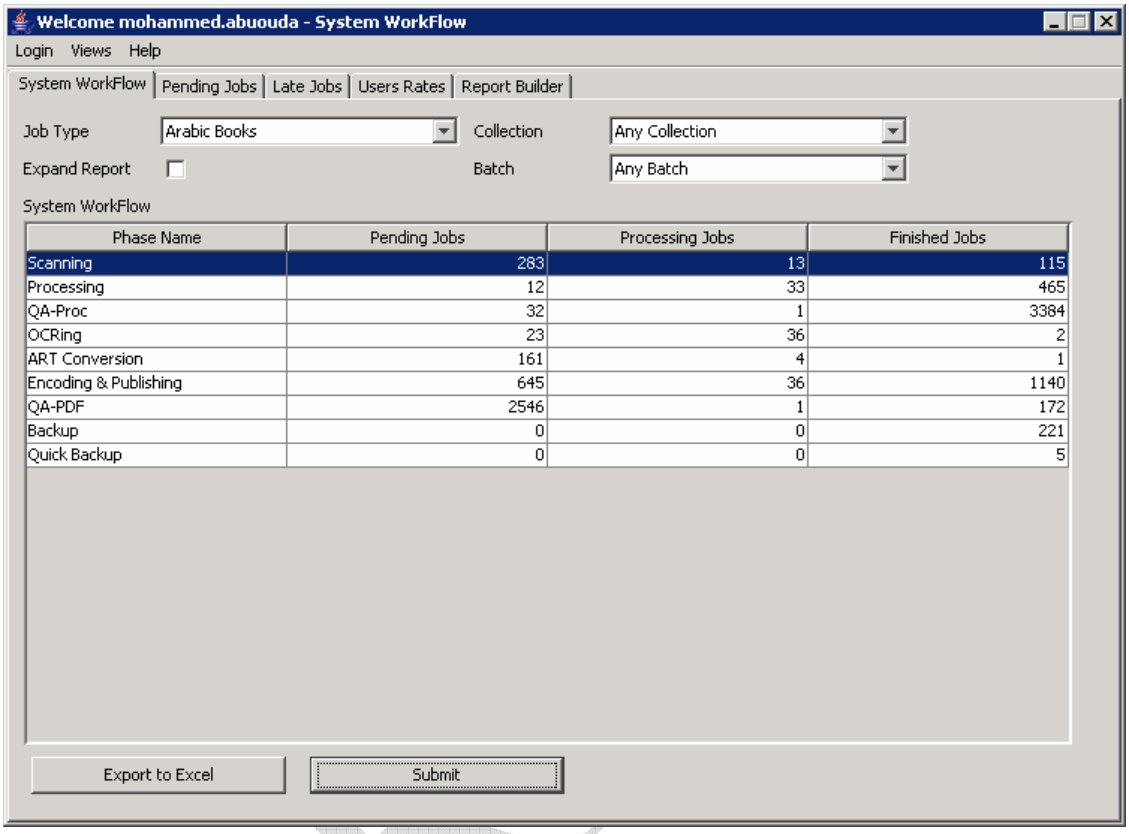
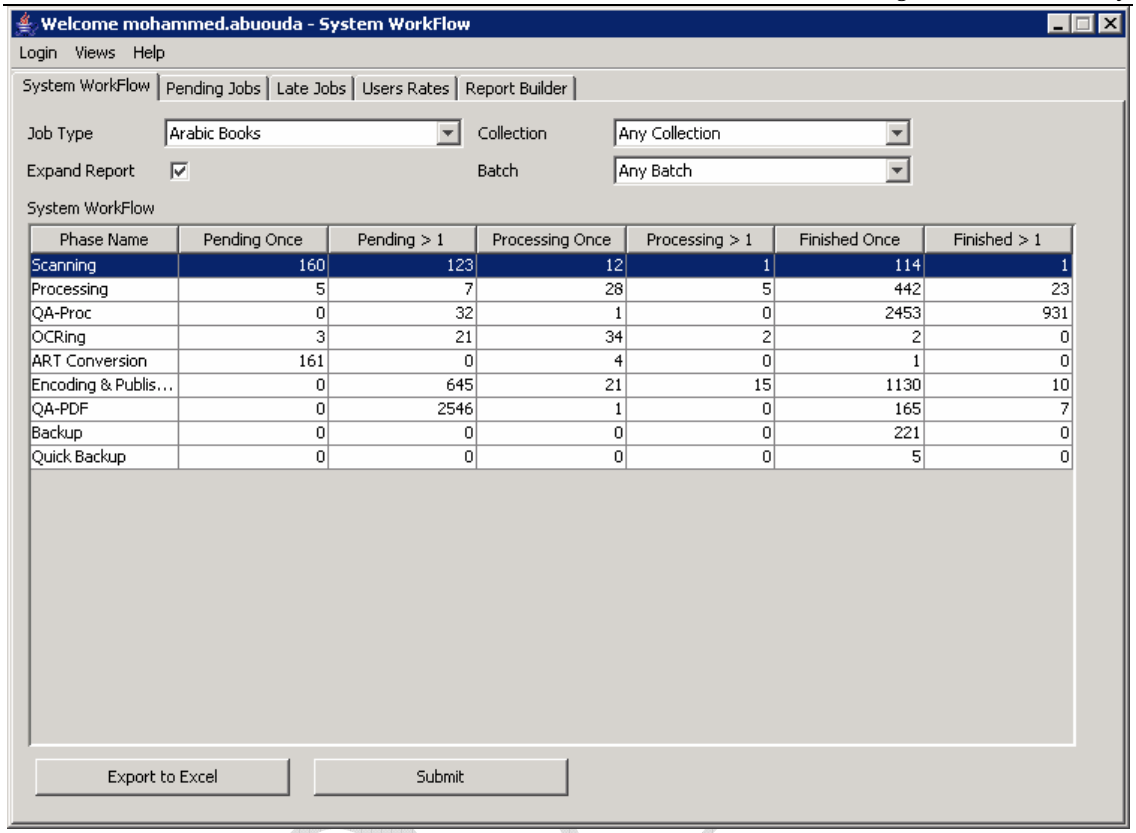


Fig. 3.5.2 Workflow Report

System workflow report can be expanded into 6 groups instead of 3;

a. Pending Jobs (first time)
b. Pending Jobs (more than once)
c. Processing Jobs (first time)
d. Processing Jobs (more than once)
e. Finished Jobs (first time)
f. Finished Jobs (more than once)

This classification helps the report to detect the Jobs that have been in pending, processing or finished groups for the first time and the Jobs that have passed over these groups before. This discrimination is done on Phase level. See figure 3.5.3



Phase Name	Pending Once	Pending > 1	Processing Once	Processing > 1	Finished Once	Finished > 1
Scanning	160	123	12	1	114	1
Processing	5	7	28	5	442	23
QA-Proc	0	32	1	0	2453	931
OCRing	3	21	34	2	2	0
ART Conversion	161	0	4	0	1	0
Encoding & Publis...	0	645	21	15	1130	10
QA-PDF	0	2546	1	0	165	7
Backup	0	0	0	0	221	0
Quick Backup	0	0	0	0	5	0

Fig. 3.5.3 Expanded Workflow Report

It should be mentioned that every cell in the Workflow report is clickable for producing a list of Jobs that form the Workflow cell's number.

3.5.2 Pending Jobs

Pending Jobs are the Redirected or Rejected Jobs. Redirect and Reject Job are two Basic States and functionalities in DAFv2. The difference between the two states is the following;

User redirects a Job If the Job is finished correctly and User has a recommendation for this Job to be started in a recommended Phase and with recommended User. He/She rejects a Job, however, if the Job has a problem preventing him/her from continuing work.

Redirect or Reject can be done through Phase Manager buttons. To control the process of redirection and rejection, administrator must have a look on the redirected or rejected Jobs' reasons. He/she also needs the ability to approve or deny the redirected or rejected Jobs.

The Pending Jobs' report loads all redirected and rejected Jobs associated with recommended Users and Phase, and gives the reporter the ability to approve or deny a Job through Approve and Deny buttons. It has a Job Download facility if the reporter needs to navigate the physical files of the pending Job; this may help in approving or denying a Job. See figure 3.5.4

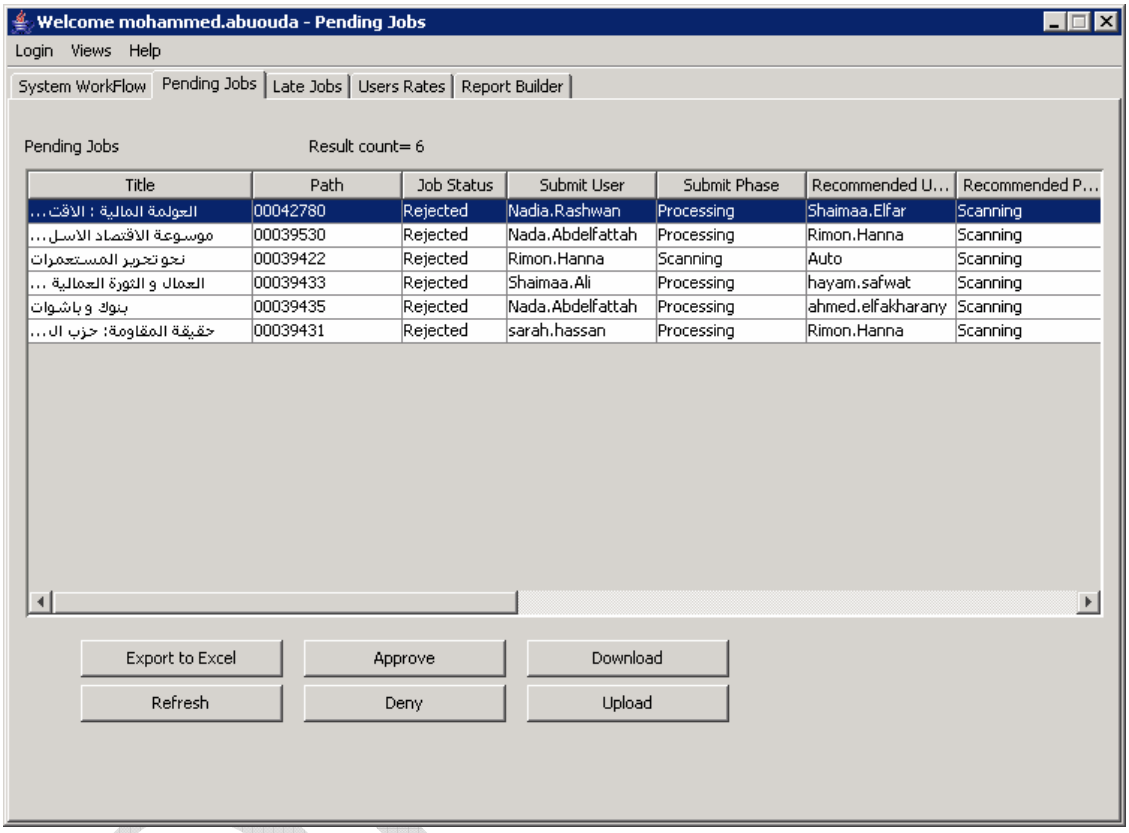
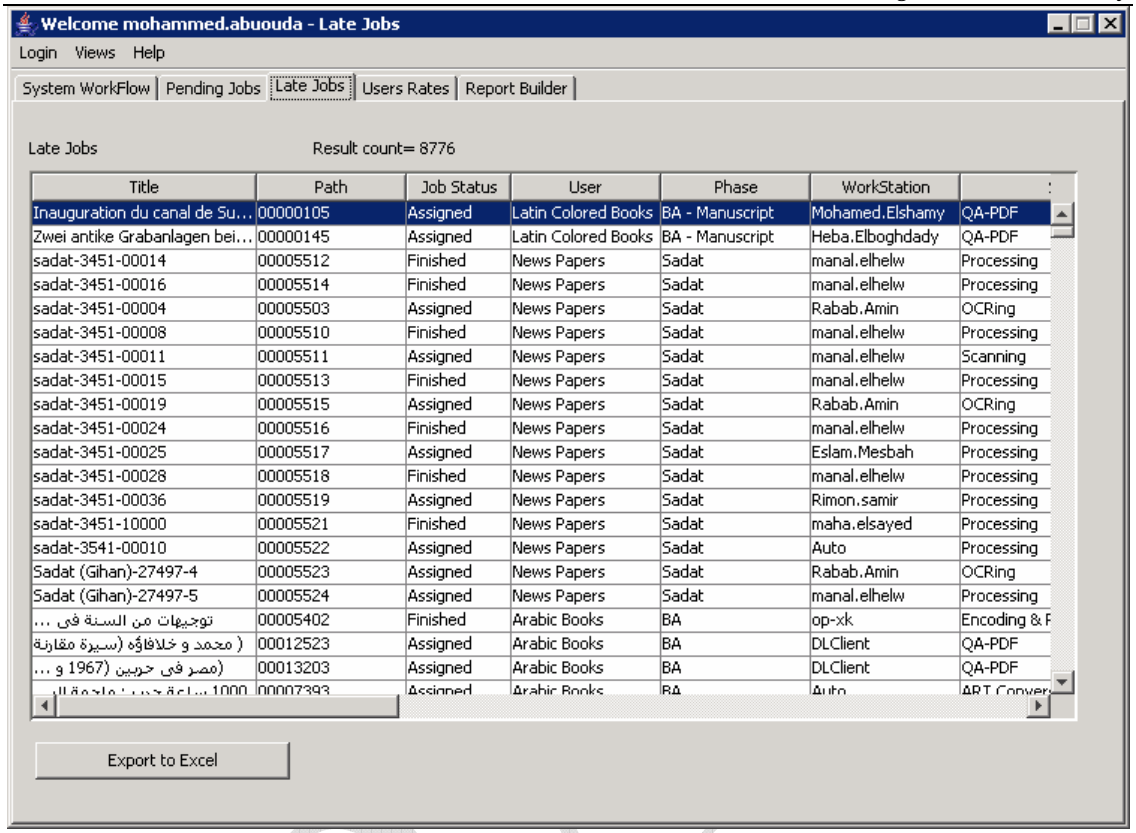


Fig. 3.5.4 Pending Jobs Report

Reporter should note that the definition of Pending Jobs in the Workflow's report differs from the one in the Pending Jobs' report. The first includes three states (Assign, Redirect and Reject). The second includes Redirected and Rejected Jobs only.

3.5.3 Late Jobs

The Late Jobs' report is a straightforward one; it simply shows the Jobs that are past their Due Date. Every Checked-In Job in DAFv2 has a Due Date value. A Job should not pass this value, otherwise it will be considered a late Job.



Title	Path	Job Status	User	Phase	WorkStation	
Inauguration du canal de Su...	00000105	Assigned	Latin Colored Books	BA - Manuscript	Mohamed.Elshamy	QA-PDF
Zwei antike Grabanlagen bei...	00000145	Assigned	Latin Colored Books	BA - Manuscript	Heba.Elboghdady	QA-PDF
sadat-3451-00014	00005512	Finished	News Papers	Sadat	manal.elhelw	Processing
sadat-3451-00016	00005514	Finished	News Papers	Sadat	manal.elhelw	Processing
sadat-3451-00004	00005503	Assigned	News Papers	Sadat	Rabab.Amin	OCRing
sadat-3451-00008	00005510	Finished	News Papers	Sadat	manal.elhelw	Processing
sadat-3451-00011	00005511	Assigned	News Papers	Sadat	manal.elhelw	Scanning
sadat-3451-00015	00005513	Finished	News Papers	Sadat	manal.elhelw	Processing
sadat-3451-00019	00005515	Assigned	News Papers	Sadat	Rabab.Amin	OCRing
sadat-3451-00024	00005516	Finished	News Papers	Sadat	manal.elhelw	Processing
sadat-3451-00025	00005517	Assigned	News Papers	Sadat	Eslam.Mesbah	Processing
sadat-3451-00028	00005518	Finished	News Papers	Sadat	manal.elhelw	Processing
sadat-3451-00036	00005519	Assigned	News Papers	Sadat	Rimon.samir	Processing
sadat-3451-10000	00005521	Finished	News Papers	Sadat	maha.elsayed	Processing
sadat-3541-00010	00005522	Assigned	News Papers	Sadat	Auto	Processing
Sadat (Gihan)-27497-4	00005523	Assigned	News Papers	Sadat	Rabab.Amin	OCRing
Sadat (Gihan)-27497-5	00005524	Assigned	News Papers	Sadat	manal.elhelw	Processing
توجيهات من السنة فى ...	00005402	Finished	Arabic Books	BA	op-xk	Encoding & F
محمد و خلافاؤه (سيرة مقارنة)	00012523	Assigned	Arabic Books	BA	DLClient	QA-PDF
(مصر فى حربين (1967 و ...	00013203	Assigned	Arabic Books	BA	DLClient	QA-PDF
...ساعة حرب: ملحمة ال... 1000	00007393	Assigned	Arabic Books	BA	Auto	ART Convers

Fig. 3.5.5 Late Jobs Report

Reporter can push late Jobs by changing Jobs' due date or by raising Job priority value (figure 3.5.6).

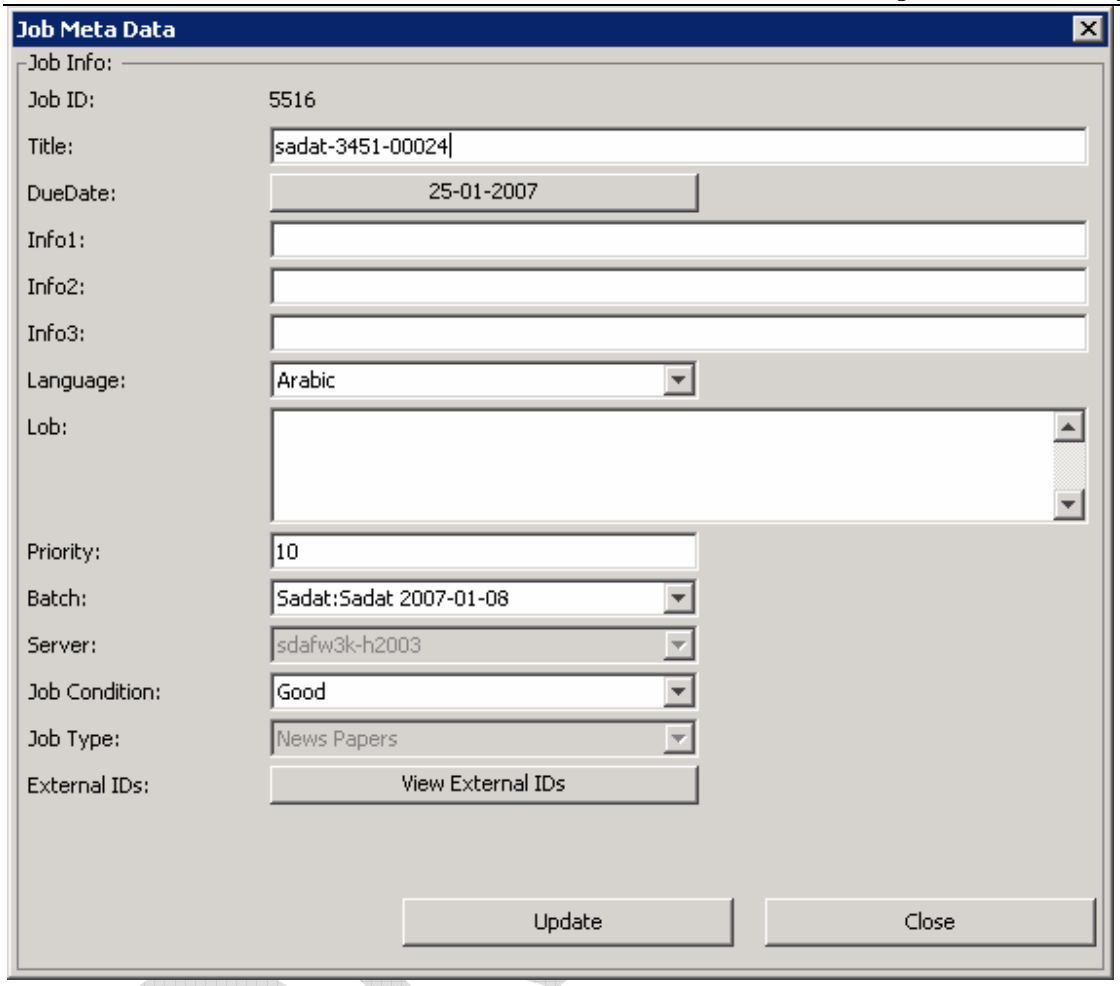


Fig. 3.5.6 Change due date or priority

3.5.4 User Rates

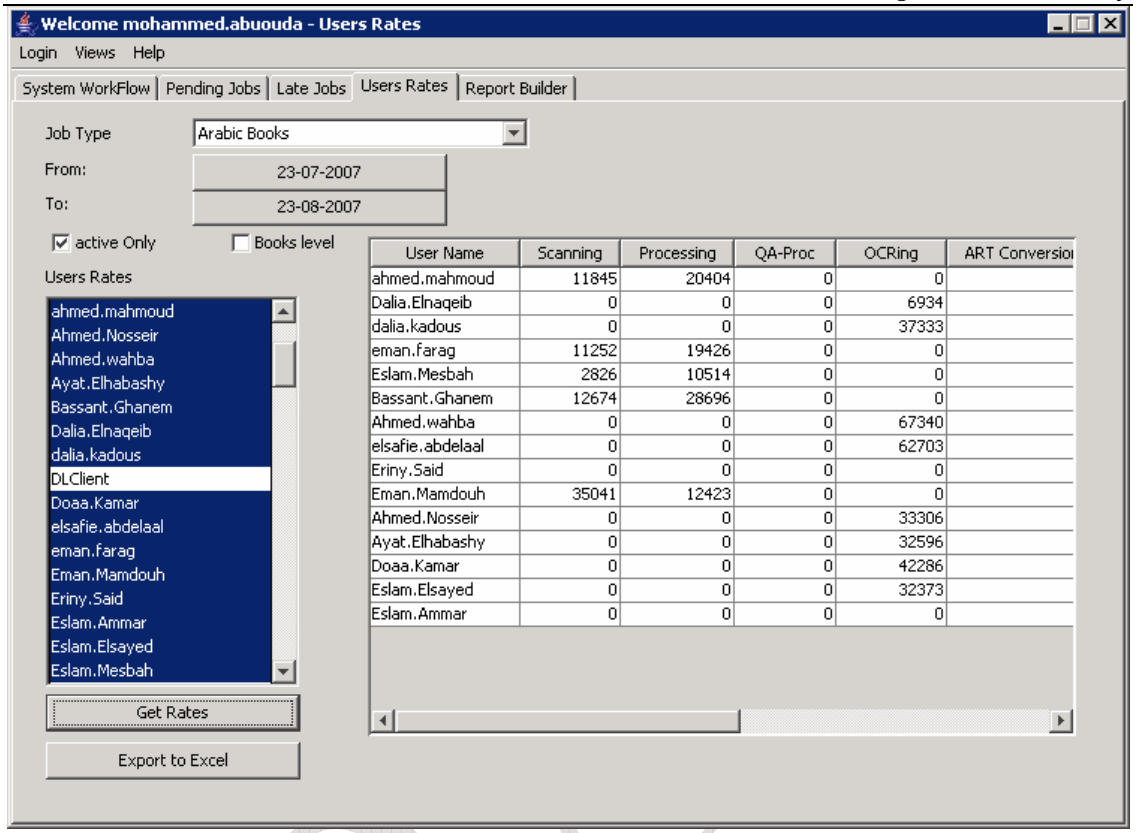


Fig. 3.5.7 User Rates (Pages Level)

DRAFT

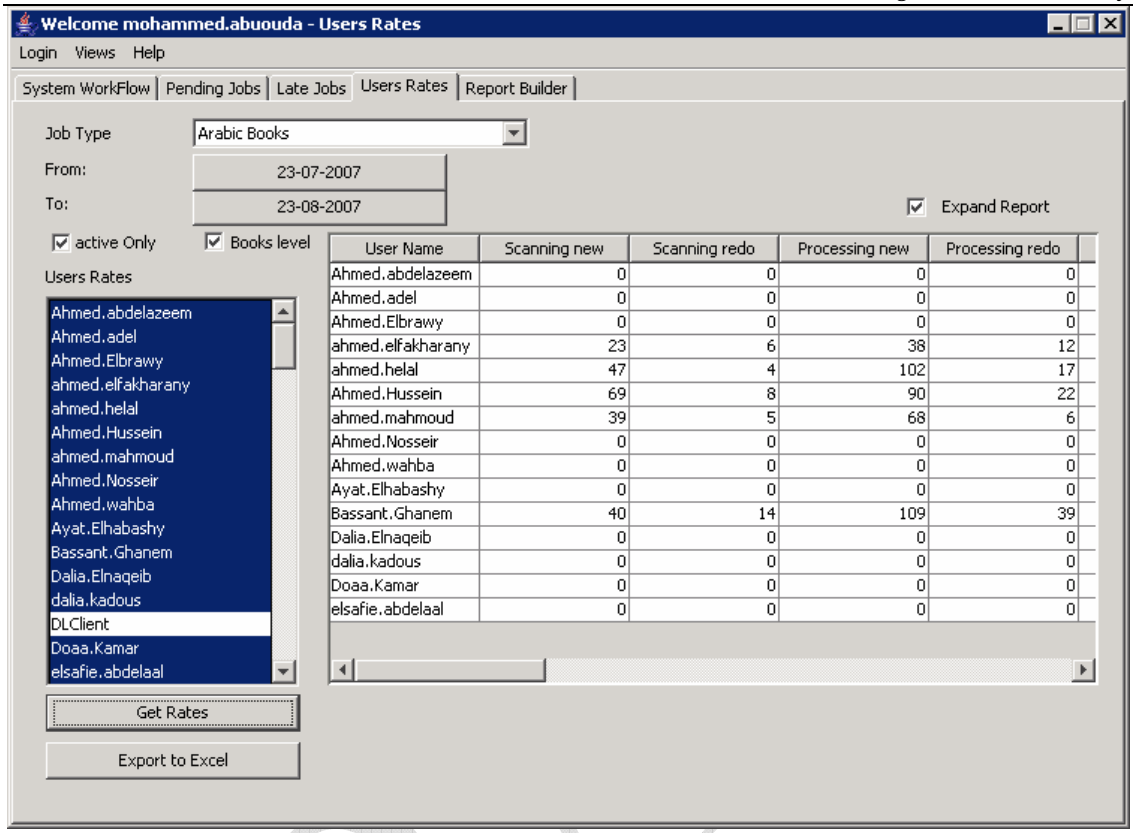


Fig. 3.5.8 User Rates (Books Level)

3.5.5 Report Builder

Reporting Manager gives the reporting User the ability to create custom reports through Report Builder. It is primitive since it could have sophisticated report building using all resources in DAFv2.

It covers mainly the most used resources (Collection, JobType, Language,… etc) and also provides searching on JobStatus, i.e. User can build a report for searching Finished Jobs.

Reporter User can choose his/her searching domain; Old Jobs, New Job or both. He/she can also define the attributes that should appear in the customized report. This attributes list is loaded from Attributes section in resource file; administrator can add, delete or update attributes in this section.

Report builder detects the administrator and makes him/her able to write a direct query to the database; this query is of Select type. Delete, Insert or Update queries are prevented by code checks.

Generated reports are saved into Report table as report name and query. They are loaded if the user chooses Report builder tab and double clicks any of them. If User chooses New Report button, he/she will be redirected to building report GUI

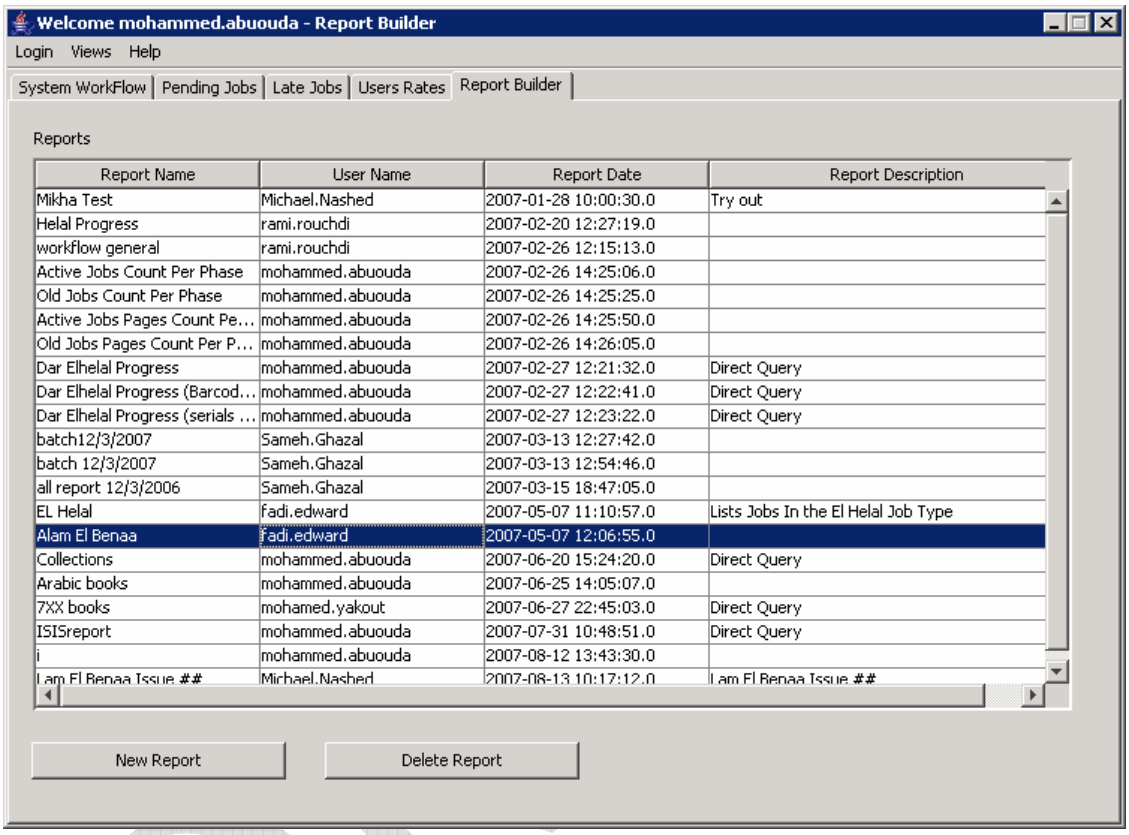


Fig. 3.5.9 Customized Reports

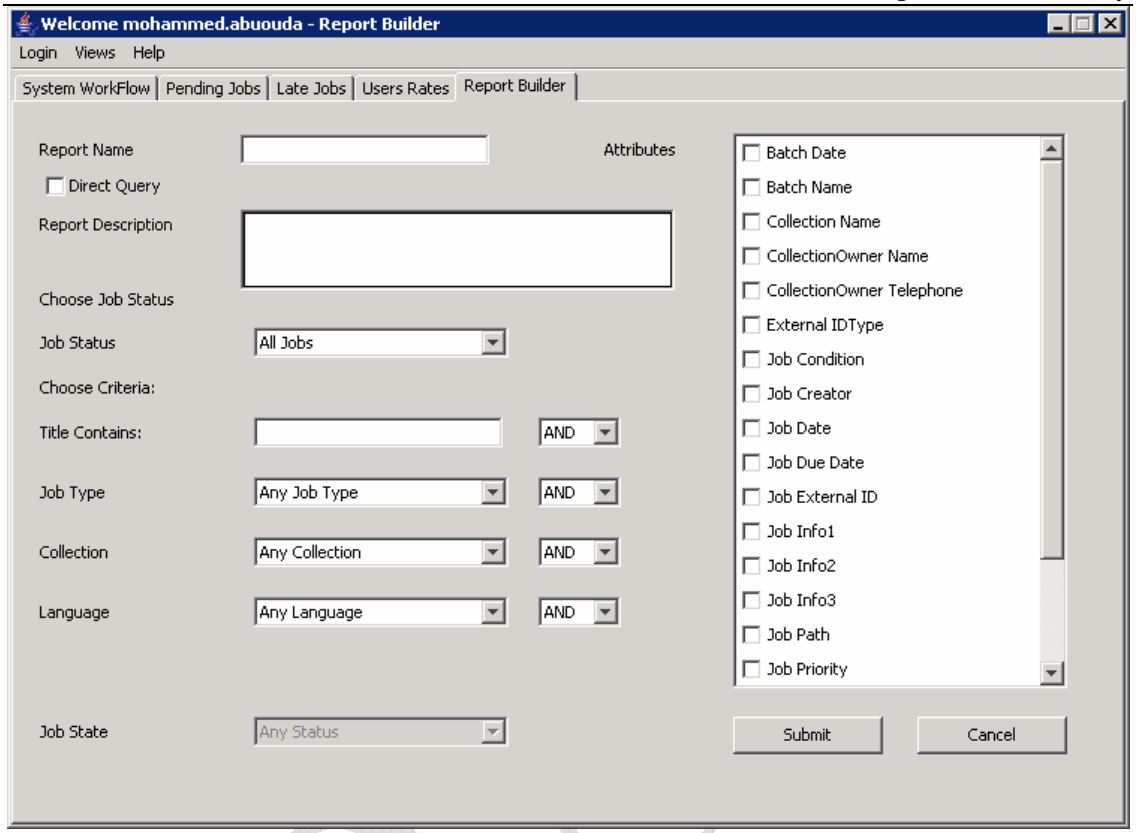


Fig. 3.5.10 Report Builder

3.6 Archiving Module

DAFv2 workflow uses Storage servers in downloading or uploading Jobs (discussed in Phase Manager). Ordinary workflow will eventually fill up all space in these servers. Archiving data is a solution for this problem; it also offers the opportunity to save data into external Medias and to retrieve data from this media in the future.

Logically, archiving solves the space problem by giving the archiver the ability to ***Check Out*** Jobs, which means removing the physical files of the checked out Jobs from Storage Server. DAFv2 prevents checking a Job out unless it was archived.

Technically, archiving process' input is the Backup Phase's output, in other words, Backup Phase is responsible of zipping and versioning the Job folder (Job can get more than one

version). Archiver seeks every backed up versions and those that have not been archived before and marks them as ready to archive.

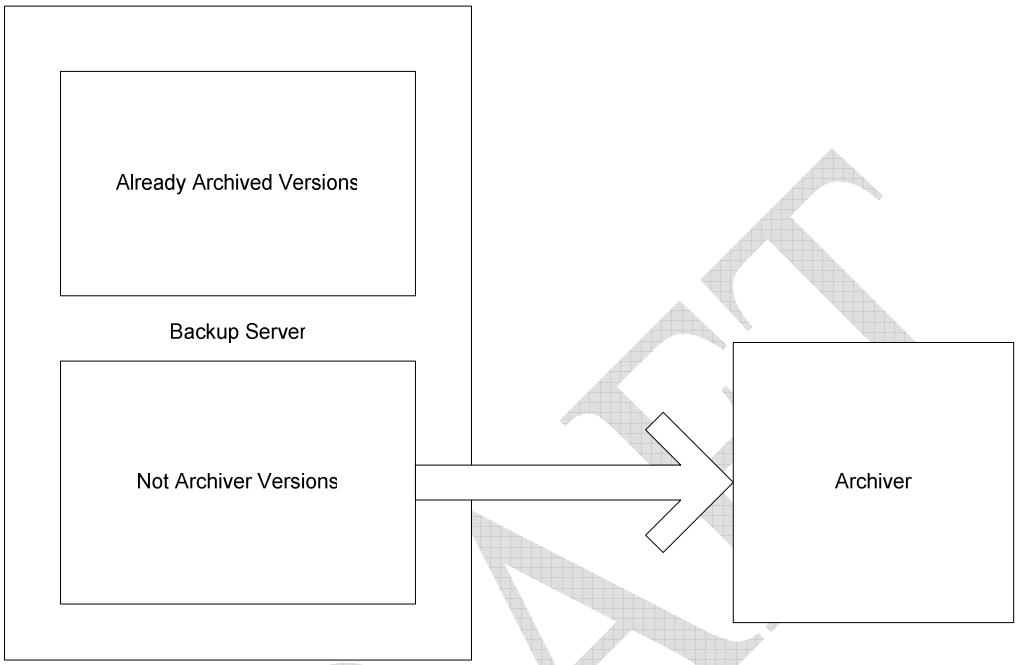


Figure 3.6.1

Versions' Separations (Already archived or Not archived) is done using database logic; it does not depend on physical files' separation; this will be discussed in details later.

Referring to Job Type and Media Type relation (Admin Manager Module), a Job Type can be defined to be archived on multiple Media Types, thus, all versions produced for every Job belonging to this Job Type must be archived on these Media Types.

This can be represented as a 2D matrix. Check the following example:

Jobs X and Y belong to Job Type JT1; JT1 is defined to be archived on Tapes and CDs. Backup Phase has ran twice on X lifecycle (2 different versions), and once on Y (only one version).The process now is handled by the archiver who will do the following;

1. Decide which versions have been archived (it will yield no results)
2. Start archiving X version 1, version 2 and Y version 1, on CDs.
3. Start archiving the 3 versions on Tapes.

The formed matrix is:

Job Version	CD	Tape
X_version1	√	√
X_version2	√	√
Y_version1	√	√

Figure 3.6.2

Archive processing needs human interaction; Archiving manager forms physical folders on logged on User's working directory and waits for the User's confirmation that he/she has written this folder to the media correctly through media driver.

Checking a Job out requires that the Job has a nonempty archiving matrix; all versions must be marked as archived on dedicated media.

Summarizing all, Archiving Manager helps archiving User by showing Jobs Versions that have not been archived before, forms Physical folder on User working directory and awaits User's confirmation to mark these versions as archived.

Archiving Manager has 3 modules: Archiver, Confirmation and Check Out module. We will go through each of them in the following sections.

3.6.1 Archiver

Archiver carries out the archiving process core. It decides which Jobs Versions need to be archived and on which Media, marks versions as archived, helps User to form media and creates media physical folder on working directory, waiting for the User to take action.

Understating Archiver functionalities requires some concepts clarification:

3.6.1.1 Media Type and Media relation

A Media is one Instance of Media Type; it s distinguished by Barcode, which is a unique value. For example, a CD Media type may has multiple CDs (Medias), each of them may have a different barcode.

DRAFT

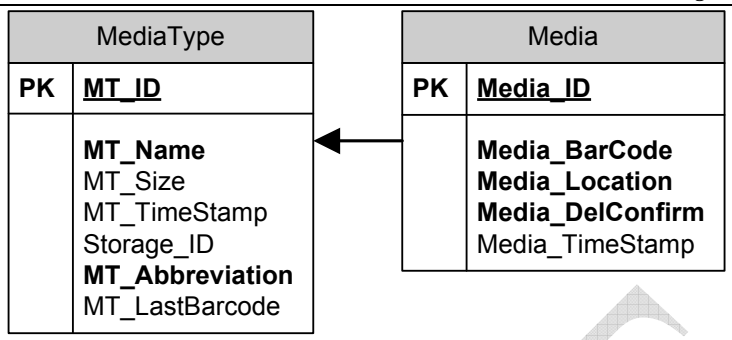


Figure 3.6.1.1

3.6.1.2 Barcoding

Generating a new barcode requires checking that the barcode is totally unique and taking concurrent connections in consideration. When Archiver issues a decision creating new Media, DAFv2 does the following

1. Defines the issued Media Type
2. Locks Media Type table (for concurrency)
3. Reads the last Barcode value that was saved into Media Type table (MT_LastBarcode attribute)
4. Creates a Media item with this barcode in Media table
5. Increments MT_LastBarcode value to be MT_LastBarcode+1
6. Unlocks Media Type table

Online Storage Barcoding is different; Online Storage depends on having huge storage, then there is no need to create more than one instance of Medias. Unless the Online Storage was changed, this is detected by changing Online Storage IP. The link between MediaType and Storage tables comes through Storage_ID value, which, utilized with Online Storage only, archive checks Storage IP or Name on issuing new Media. If this IP differs from previous Online Storage, then Archiver will create a new Media instance with these details; otherwise it will return the current Media of Online Storage. The following are the steps of Online Storage barcoding:

1. Check If Media Type =1 (Online Storage)
2. Get Barcode of last created Media instance of Online Storage's Meda Type
3. Get the current IP or machine Name of Online Storage through joining Storage and Media Type tables on Storage_ID link, where Media Type ID is online Storage (ID=1)
4. Compare Values from step 2 and 3; if both are equals then do not create new Media instance, instead create Media instance with data from step 3

It should be noted that Media barcode with Online Storage will be the Storage IP or machine name.

DRAFT

3.6.1.3 Data Source

As shown in figure 3.6.1, Archiver input is the output of Backup Phase. All backed up files must pass – one time at least- through the archiver. Archiver avoids Duplicates according to Choosing Criteria logic.

3.6.1.4 Choosing Criteria

Archiver depends totally on Backup Phase since it is its source and it fires up building Archiving matrix for every backed up Job versions. Once a Backup Phase finishs backing up a Job and forms a version of this Job, Archiver applies on this Job two actions

1. Defines which Media Types this version should be written on according to JobType– MediaType relation (revise Job Type – Admin Manager Module).
2. Build up the Archiving Matrix for the backed up Job (figure 3.6.2)

Once the archiving process is started, Archiver checks up the Archiving Matrix and avoids all selected boxes. It seeks unmarked entries and considers then as source.

The following example may help: a Job belongs to a Job Type that should be archived on CDs and Online Storage. Backup Phase produced two versions in the Job lifecycle. Archiving User decides to archive on CD. Archiver will do the following

1. Builds up an archiving matrix

Job Version	CD	Online Storage
version1		
version2		

2. It finds that version1 is not archived, so, archive it and mark its entry.

Job Version	CD	Online Storage
version1	√	
version2		

3. If Archiving User continues archiving on CDs, Archiver will exclude all marked entries and introduce unmarked ones. It will choose version2 and mark it.

Job Version	CD	Online Storage
version1	√	
version2	√	

4. Archiver will not find any unmarked entries for this Job to archive on CDs, its archiving matrix is full
5. Same process will be applied when archiving on Online Storage

DRAFT

Technically, Archiver chooses as follow

1. Selects all finished backup entries from TransactionLog table
2. Gets all Archived Entries on the specified Media Type
3. Excludes all finished backup entries from already archived ones
4. Introduces the excluded list as ready to archive list (versions list).
5. For every entry of this list, archiver inserts entry on Job_Media table if the selected version is archived on selected Media. This step makes the selected version belong to the list that comes from Step 2 and hence guarantees that this version will not be one of Step 3 list if the same Media was chosen.

Record inserted in Job_Media table refers to Backup entry (Job's Version) and Media ID (instance of Media Type). Backup entry is TransactionLog entry (for active Jobs) or OldTransactionLog entry (for old Jobs). See archiving schema in figure 3.6.1.2

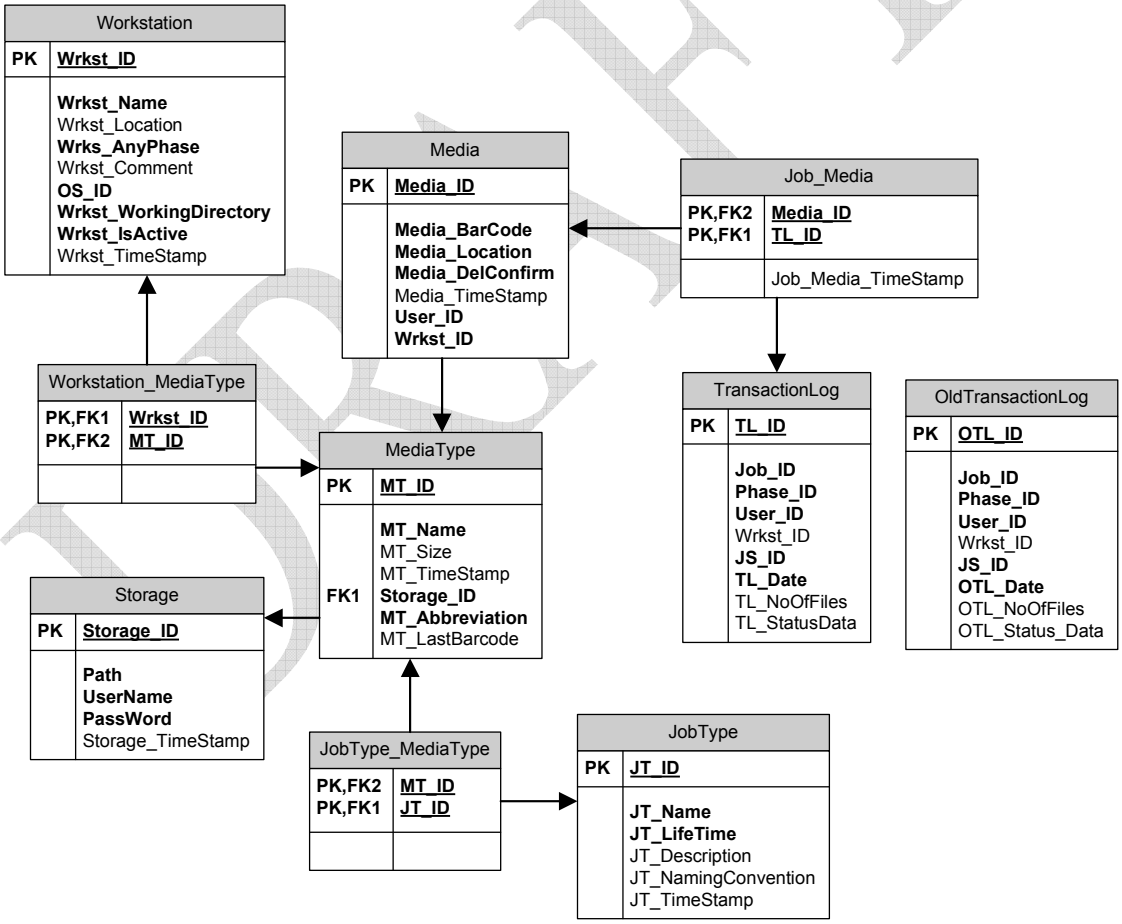


Figure 3.6.1.2

3.6.1.5 Concurrent Archiving processes

Archiving Module supports concurrent archiving processes; one or more can archive simultaneously. Archiver simply checks if the selected versions are currently in process with anther archiver or not, if there is a single intersected version, it notifies the User and refreshes the GUI since this anomaly can be done if the User does not refresh the ready to archive list.

For Example User U1 and U2 Open Archiver GUI for a while (and do not refresh it). U1 starts archiving and, after a small period of time, U2 starts archiving also on the not refreshed list and he/she chooses versions used by U1. DAFv2 applies checks on this list and finds that U2's list has currently intersected versions with the other User list (U1). DAFv2 will notify U2 to select another set of version and will reload Ready to Archive list.

3.6.1.6 Archiving Scenario

After making concepts and criteria clear, it is time to introduce the Archiving process or scenarios. First of all, Admin defines for each Job Type its archiving Media Type from Admin Manager Module. Using Archiver Manager (Figure 3.6.1.3), User chooses which Media Type he/she needs to work on and the Archiving process will begin:

1. Archiver will validate if the current Workstation can do the Archiving process for the selected Media Type or not (Workstation_MediaType relation). If yes, it will continue.If otherwise, it will halt.
2. Archiver loads all Job versions that can be archived on the selected Media Type (see Choosing Criteria)
3. User can select versions to be archived or even User First Fit option to facilitate selecting process
4. User may find help in Size progress bar. While it is blue, this means it's safe to form Media that does not pass its size (If size exceeds the media size, the archiving for all the selected Jobs will not be done).
5. Clicks on Archive button, Archiver will generate new Media instance (see Barcoding)
6. Archiver checks if all the selected versions to archive have not been archived before on the same media (except if the media is Online Storage).
7. All the selected versions are inserted into the database (`Job_Media table`) in one transaction.
8. Archiver will form Physical folder on working directory for the logged in User. This folder name will be the Media barcode.
9. Starts downloading all selected versions from the backup server into the formed folder
10. Every downloaded entry will be marked as archived on this Media temporary waiting for a confirmation (will be discussed later). And hence building up archiving matrix

3.6.1.7 First Fit Algorithm

DRAFT

Archiver facilitates selecting Jobs' versions according to their size, using First Fit algorithm, which iterates an all versions in Ready to Archive list and selects versions that have accumulative size less than total Media size.

Once it finds a version that has an accumulative size greater that total Media size, it neglects it and tries to find another one that has a position next to this version in Ready to Archive list.

3.6.1.8 Command Line Archiver

DAFv2 supports command line functions in order to automate processes using scripts and batches. The archiving process can be automated using ***Start_Archiving*** and ***Finish_Archiving*** commands.

Start_Archiving needs 2 arguments to start working; Media Type and Jobs versions' count to download, i.e. the command ***java –jar DAFv2 Start_Archiving 1 /8,*** will create instance of Media Type with ID 1 and download 8 versions on this Media. It can be also ***java –jar DAFv2 Start_Archiving 1 /all,*** which will download all versions on this Media. ***Start_Archiving*** forms the Media according to threshold value saved in the ***ResourceFile***; this value is a percentage of total Media Type's size. If Media Type size = 700 MB and threshold value = 80 %, ***Start_Archiving*** will discard Media forming until the total downloaded versions' size pass 560 MB (80%).

Finish_Archiving takes 3 arguments to finish (confirm) archiving process, i.e. ***java –jar DAFv2 Finish_Archiving CD000102 15 1*** will finish version 1 of Job 15 in Media with barcode CD000102.

DRAFT

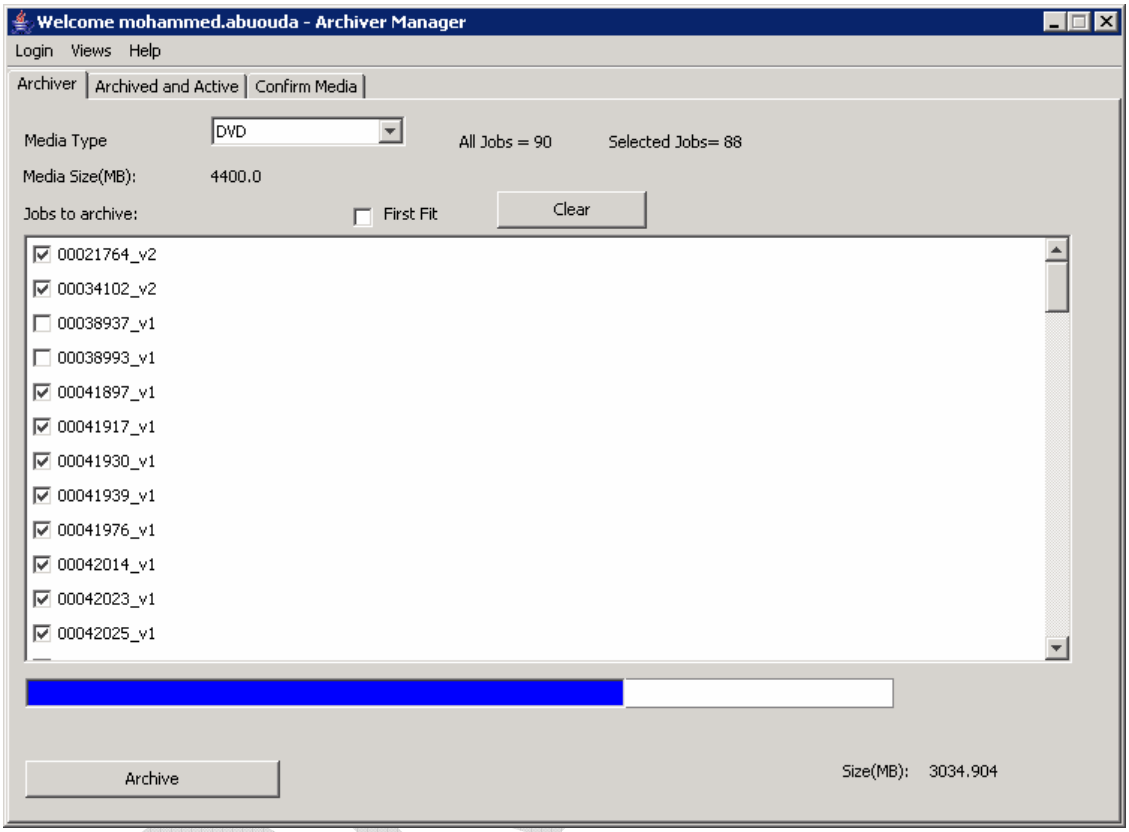

Welcome mohammed.abuouda - Archiver Manager
Login
Views
Help
Archiver
Archived and Active
Confirm Media
Media Type
DVD
All Jobs = 90
Selected Jobs= 88
Media Size(MB):
4400.0
Jobs to archive:
First Fit
Clear
00021764_v2
00034102_v2
00038937_v1
00038993_v1
00041897_v1
00041917_v1
00041930_v1
00041939_v1
00041976_v1
00042014_v1
00042023_v1
00042025_v1
Archive
Size(MB): 3034.904

Figure 3.6.1.3

DRAFT

3.6.2 Confirmation

Archiving process ends with forming Media folder on the working directory. On finishing Jobs versions downloading, first level of confirmation will be fired, which is "Confirm Jobs versions on Media". Figure 3.6.2.1

Confirm Jobs versions on Media is the first confirmation level (out of 2). It helps the archiving User in answering the question "Are these versions what I need to archive on this Media?". If the User confirms that the archived versions really chosen correctly to be archived on this Media type, then he/she must click on Confirm button, otherwise he/she can discard Media totally by clicking on Discard button.

Discarding Media deletes Media Physical folder from the working directory and unmarks all versions' entries (deletes all entries form Job_Media table related to the delete Media).

Burning or writing the formed Media folder on the Media needs human interaction; he/she should do the writing process outside DAFv2, then he/she must confirm that the Media folder is written successfully on the physical Media (CD, Tape, DVD, or otherwise). This is the second level of confirmation; it can be done through Confirm Media tab (Figure 3.6.2.2)

Archiving versions on Online Storage requires first level of confirmation; second level of confirmation is done implicitly. The versions are moved directly to Online Storage through FTP service. If this movement is done successfully without any failure, then the versions are confirmed on Online Storage; otherwise, rollback these versions. Thus, second level of confirmation is done implicitly.

DRAFT

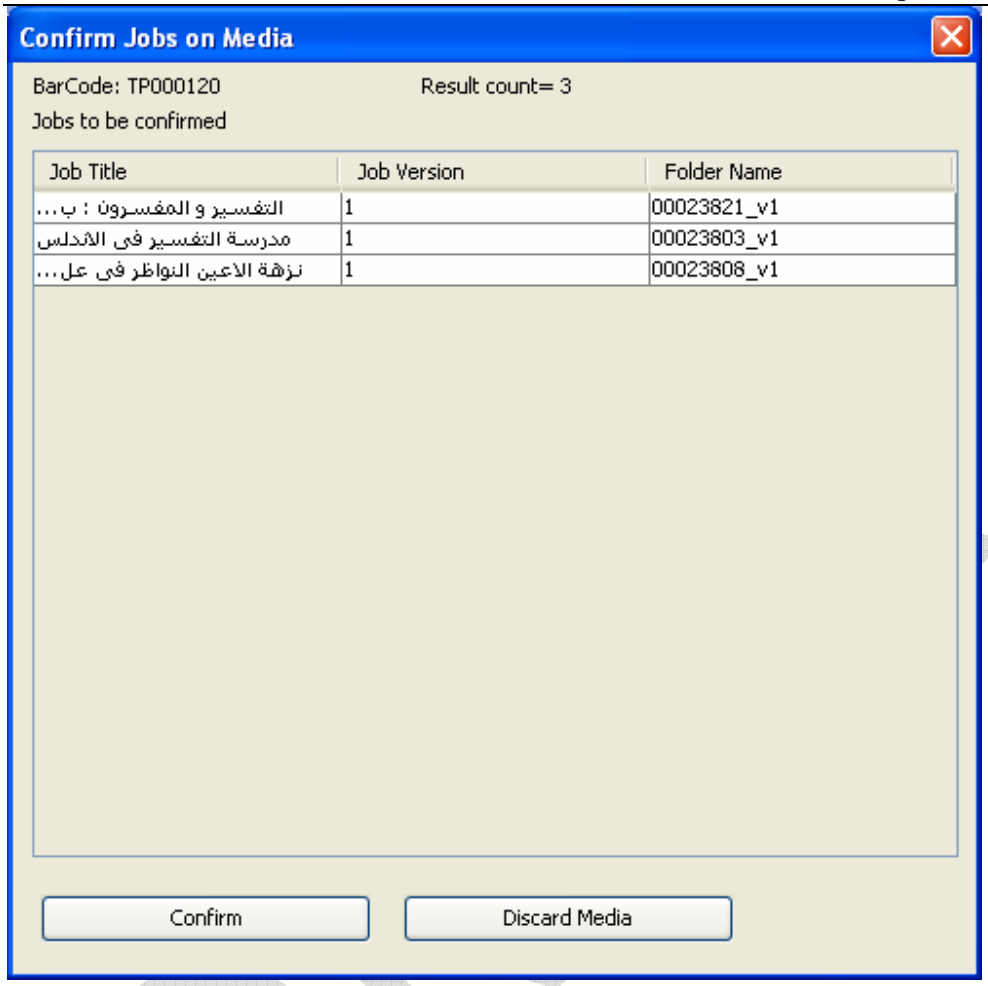

Confirm Jobs on Media
BarCode: TP000120
Result count= 3
Jobs to be confirmed
Job Title
Job Version
Folder Name
التفسير و المفسرون : ب...
1
00023821_v1
مدرسة التفسير فى الاندلس
1
00023803_v1
نزهة الاعين النواظر فى عل...
1
00023808_v1
Confirm
Discard Media

Figure 3.6.2.1

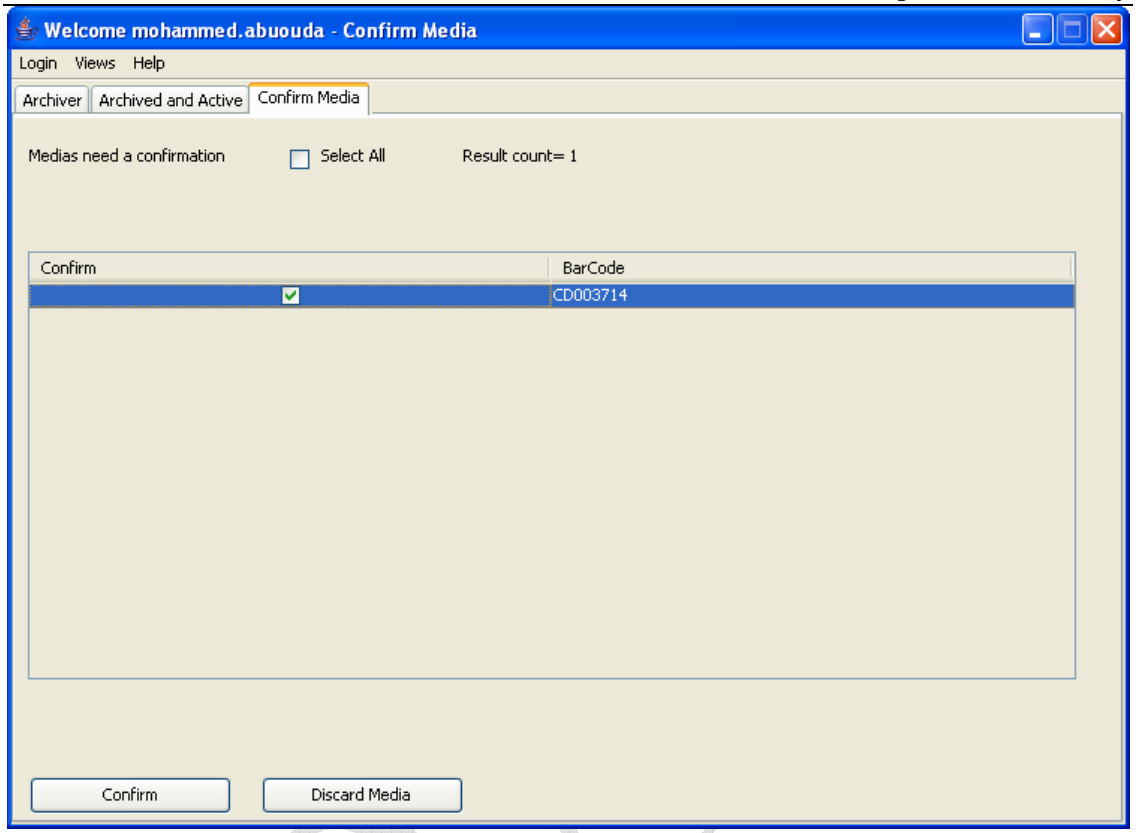


Figure 3.6.2.2

3.6.3 Check out

Job lifecycle ends with archiving all its versions on dedicated Media Types, in other words, if the Archiving matrix is full. Checking a Job out requires two conditions

a. That the Job be in inactive state
b. That the Job's Archiving matrix is full

First condition means that the Job is in a finished Backup phase state. Any other state will be considered as an active state.

Actions associated with Checking out a Job are

1. Moving Job's entries from TransactionLog table to OldTransactionLog table.
2. Deleting Job's folder from Storage Server
3. Deleting all Job's versions from Backup Server

Forced Check out

Jobs that have not satisfied the first condition in the Check Out process cannot ever be checked out, Jobs, however, that have not satisfied the second one can be. User can use Forced Check Out option if the Archiving matrix is not full, but this action is not preferable, since Job's versions are not archived correctly on all dedicated Media Types.

DRAFT

4. SYSTEM HANDLERS

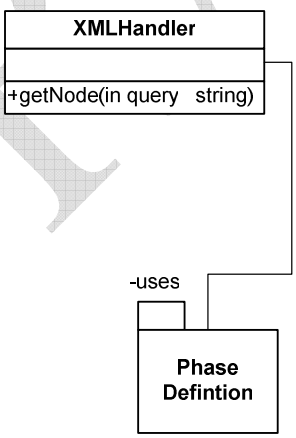
4.1 Authentication and Authorization Handler

All the provided interfaces and services of the system are accessible through an authentication and authorization handler. This handler is responsible for customizing the application interface for the logged in User. Moreover, it authorizes each action or request submitted by User.

4.2 XML Phases Handler

The XML Phases Definition Handler is responsible for interpreting and applying the XML definition of the Phases. Each Phase has its own XML Phases Definition, specifying the prerequisites and actions that need to be done before and after each Phase. The XML definition contains two main sections; Pre-Phase and Post-Phase. Each of these sections is composed of three subsections; Physical, Database and Reflection Call.

All XML operations are done using a simple XML Handler that parses XML files called `XmlHandler`. This class parses the XML files and adjusts the encoding if necessary. Please refer to the JavaDoc section for the XMLHandler class.



- Physical section: In the Pre-Phase section, the Physical section allows to describe the necessary folders and files structure required to start work in the Phase and which of them should be copied to the client's working folder to execute the Phase. For example, it is possible to say that the OCR Phase can not start unless there are OTIFF folder with TIFF files and PTIFF folder with TIFF files on the main file server. Only the PTIFF folder is required to be copied from the file server to the client's working folder to do the OCR Phase. In the Post-Phase section, the Physical section allows to describe the

necessary files and folders structure required to complete the Phase. It also defines which of the folders and files should return to the file server. For example, when finishing the Processing Phase there should be a PTIFF folder with a number of TIFF files equal to the ones in the OTIFF folder.

- Database section: It is usually used in the Post-Phase section. It allows to define the structure of database information that should be submitted after finishing the Phase. It contains listing and naming for the fields that should be filled by the operator during his work in the Phase. This fields are saved as XML text, with the Phase information in the transaction log for reference and query later using XPath.
- Reflection Call section: In this section, DWMS allows to specify the Java function that should be executed either in the Pre-Phase or Post-Phase. The function can start any process including files management, data entry, zipping, or encoding the files. In this section, it is possible to write the necessary code to ingest the objects in the digital document repository

The following is an example of an OCR Phase definition:

```
<Phase Name="Book Arabic OCR">
      <PrePhase>
            <Physical Mode="UnRestricted">
                  <Folder Name="OTIFF" Create="false"
              ToDestination="false" NewName="OTIFF"
              Mode="Restircted">
                        <File Name="OriginalFiles" Type="tif" Count="+"
              ToDestination="false" Compare=""/>
                  </Folder>
      .
      .
            </Physical>
      </PrePhase>
      <PostPhase>
            <Physical Mode="UnRestricted">
                  <Folder Name="TXT" Create="false"
              ToDestination="true" NewName="TXT"
              Mode="Restircted">
                        <File Name="" Type="frf" Count="1"
              ToDestination="true" Compare=""/>
                        <File Name="" Type="art" Count="1"
              ToDestination="true" Compare=""/>
                  </Folder>
            </Physical>
            <Database>
                  <Field Name="Font" DisplayName="Font Family: " />
                  <Field Name="LrnPage" DisplayName="Learn Page : "/>
      .
      .
            </Database>
            <ReflectionCall Method="packageName.doSomething" />
      </PostPhase>
</Phase>
```

In the diagram XYZ you will find the Objects into which the XML Phase definitions is mapped. This map is done using the XML Phases Handler. The `PhaseDef` object maps to the root of the

XML definition. It contains 2 `SemiPhaseDef`s one for the Pre-Phase and the other for the Post-Phase. Each `SemiPhaseDef` may contain any of the 3 sections mentioned above (`DatabaseDef`, `ReflectionDef` and `PhysicalDef`). A `DatabaseDef` is a set of `DatabaseElements`. Each `DatabaseElement` has a name, a display name, a default value, a mandatory status, and a list of predefined values to choose from (used in the UI as a dropdown list). The values are saved in the Database, in the StatusDef XML string in the DatabaseInfo section for the Done entry in the transaction log for the specified Phase.

The `ReflectionDef` contains the path for the ReflectionCall method to call while performing the Phase Actions. It should be a fully qualifying name (the namespace.classname.methodname). All reflection calls have one parameter only, which is the JobID.

The `PhysicalDef` section, describes the physical actions and checks to be done. Some of the attributes are used only for checks, others for actions. A `PhysicalDef` contain many `PhysicalElements` . Each `PhysicalElement` can either be afile (`FileElement`) or a folder (`FolderElement`).

A `FolderElement` may contain another `PhysicalElement`. The folder `FolderElement` has a name, which is current name in the source location, and a newName, which is his name in the destination location (If the 2 names are not the same then the folder is copied into a new name). The toDestination property indicates whether it should be copied or not. The mode property is either restricted (allows only defined files or folder within this folder) or unrestricted (allows at least the defined files and folders).

A `FileElement` describes a set of files with the same properties. Each `FileElement` could have a name to be able to compare with later on. The compare takes a value another `FileElement` name to compare with. The compare is done based on the file count only, it compares the first files from the server and the second one from the client Working Folder. This check was added to prevent intentional or accidental files loss. The type property maintains the extension of the file set each file set can have only one extension or type. The toDestination property indicates whether to copy the folder from source to destination or not (from server to client in the pre-phase and from client to server in the post-phase). And finally the count flag is used to make sure that the files has a count (* for any count, + 1 or more, and any other number for that specific number).

DRAFT

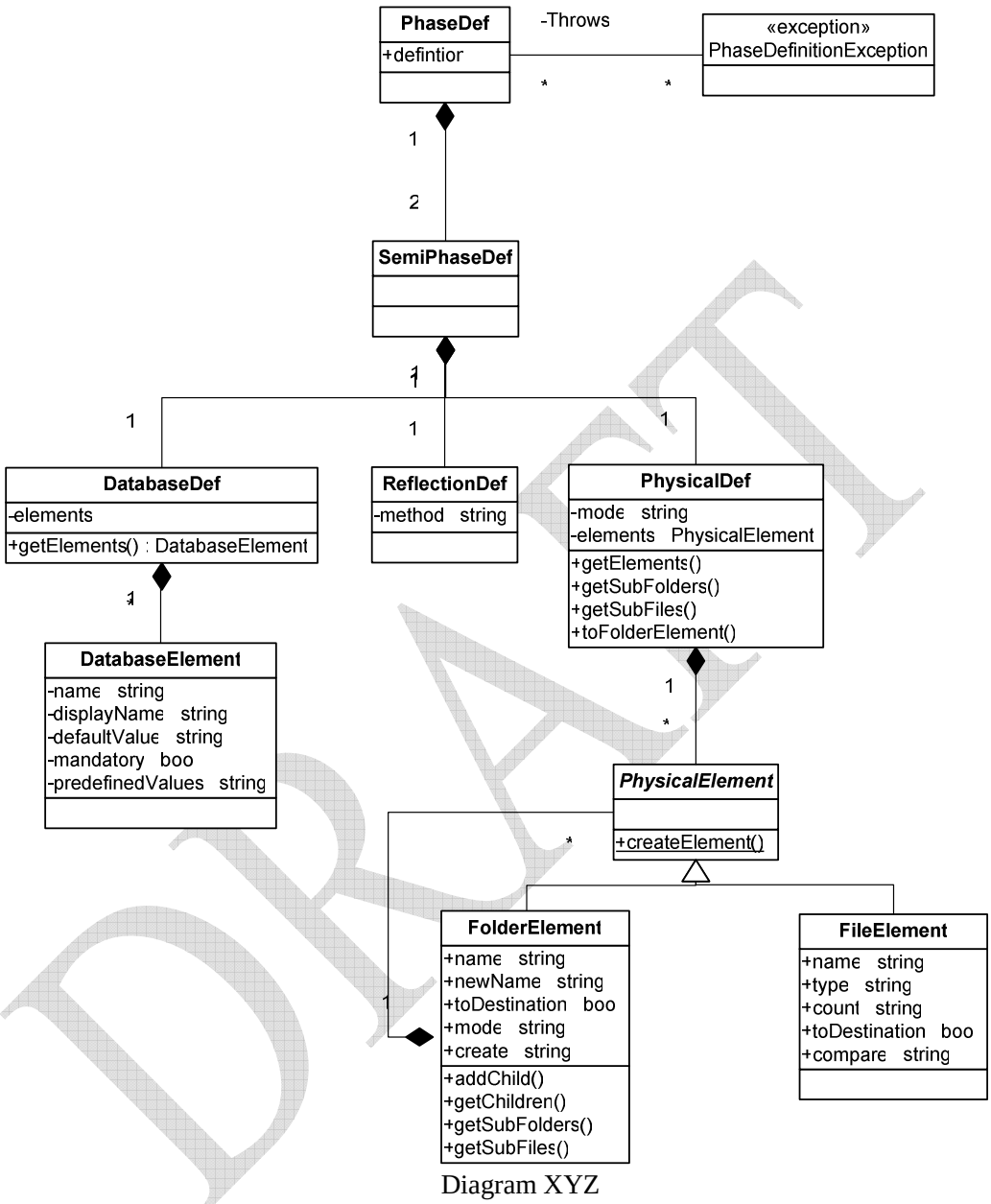
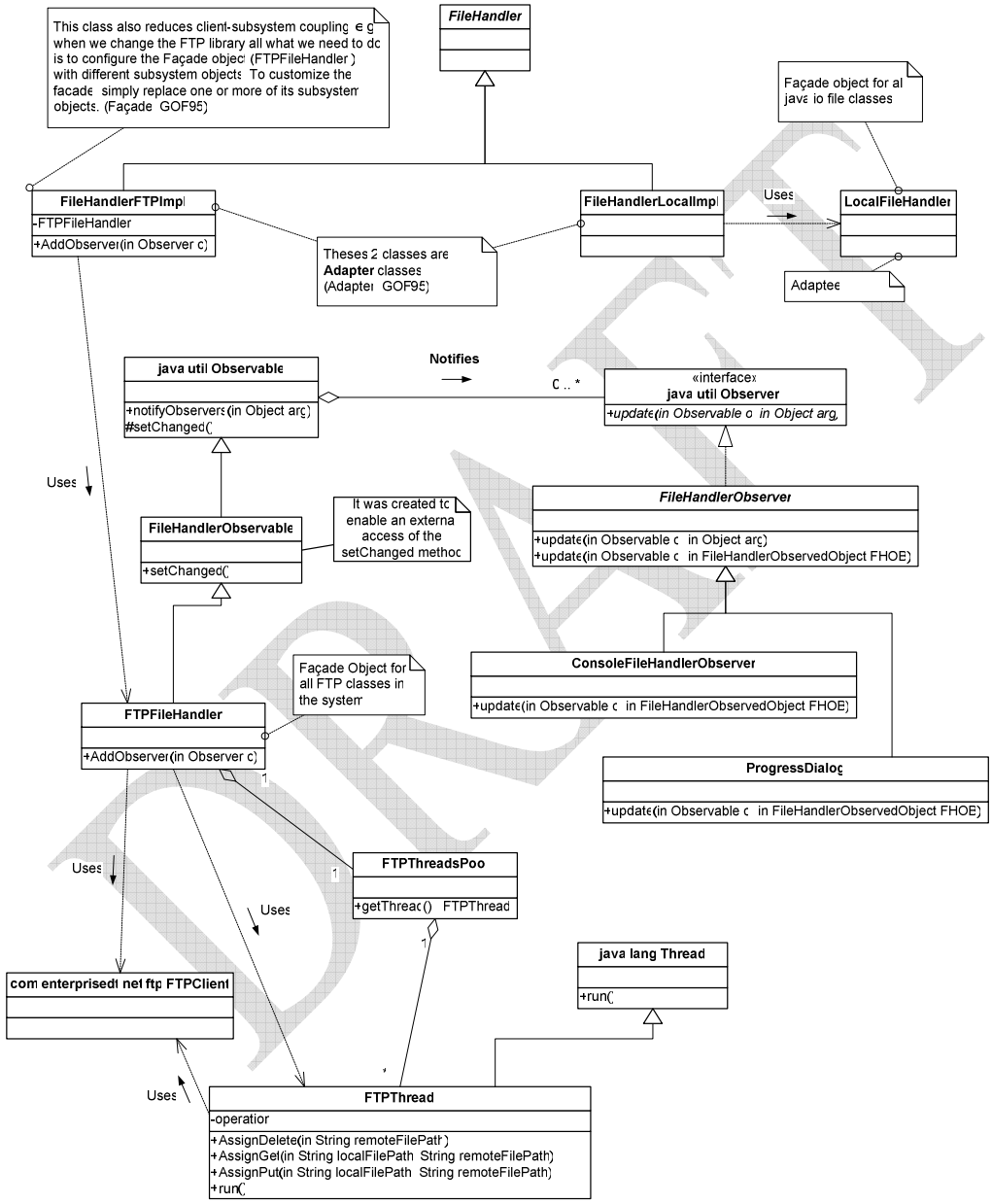


Diagram XYZ

4.3 Files and FTP Handler

The File Handler component is used by the XML Definition Handler to manage the file copying and movement. It is responsible for the necessary FTP handling with the file server and local file handling for the clients.

File handler is responsible for handling files and folders(e.g. downloading, uploading, creating, deleting and getting file and folder information). File handler handles requests belonging to the local file system and handles requests regarding remote machines using File Transfer Protocol(FTP) . The following class diagram illustrates the classes involved in this handler module and the relationships between them.

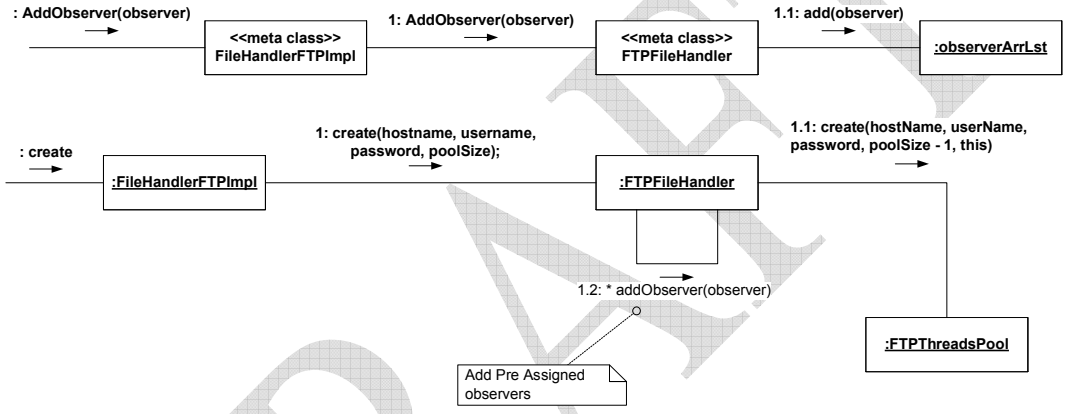


Remote File Handling

The previous class diagram illustrates the classes involved in implementing remote file handling. We used entrerprised.net.ftp FTP library. The design applies various design patterns which allows for the flexibility and usability of system classes. FileHandlerFTPImpl is an adapter class to adapt the interface provided by FTPFileHandler class with the one expected by the clients of FileHandler abstract class. FTPFileHandler is a Façade class for all FTP classes in the system. Most of the file handling operation are done via threads to allow multiple file manipulation.

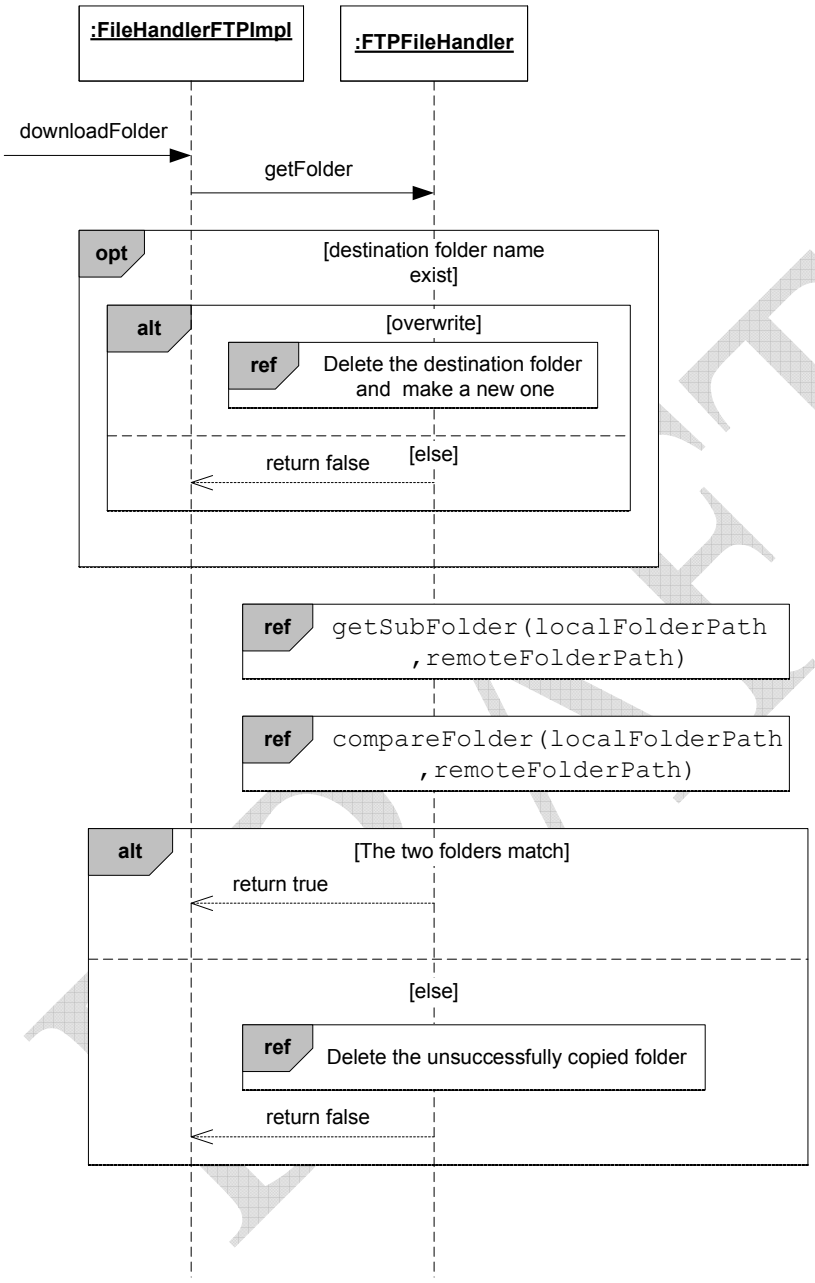
To allow monitoring of the status of file handling operation graphically or in consol (e.g. the name of transferred files, and a progress bar to indicate how much left), we apply an observer pattern, as we shown in the previous class diagrams.

The following communication diagram illustrates how we add the observers to the observable object, and the creation of thread pool that holds the threads that handle file operation.

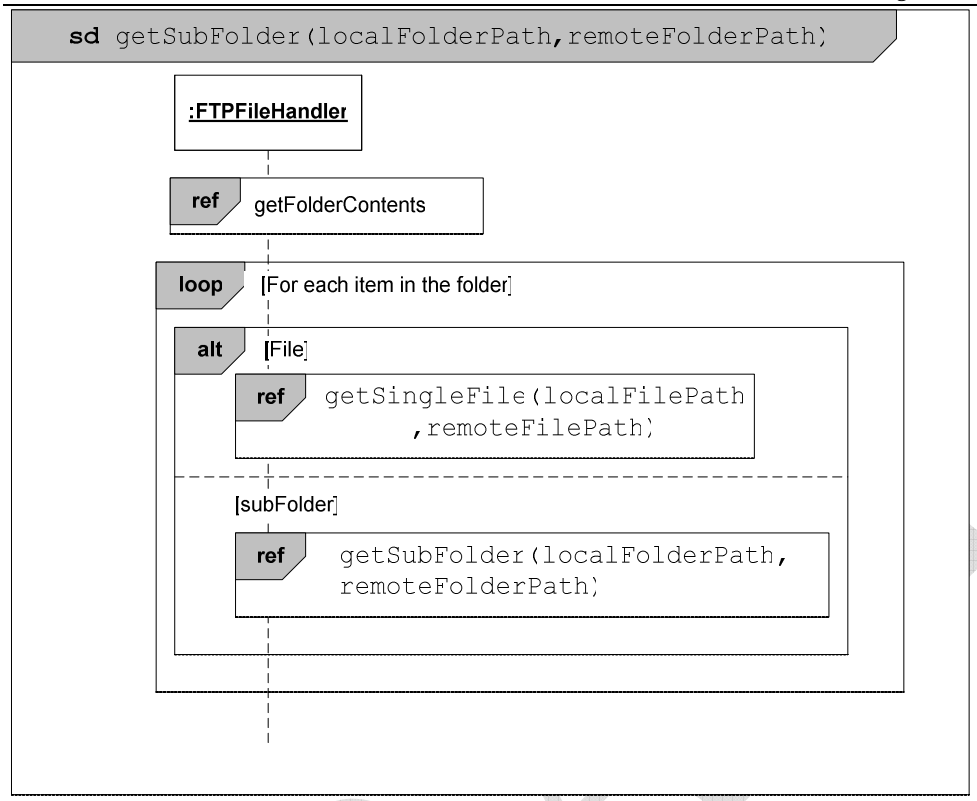


The following sequence diagram illustrates the scenario of downloading a folder from a remote machine. As depicted after finishing the download operation we compare the source folder with the copied one and delete the copied folder if the two folders do not match. We perform the comparison also in the case of uploading a folder.

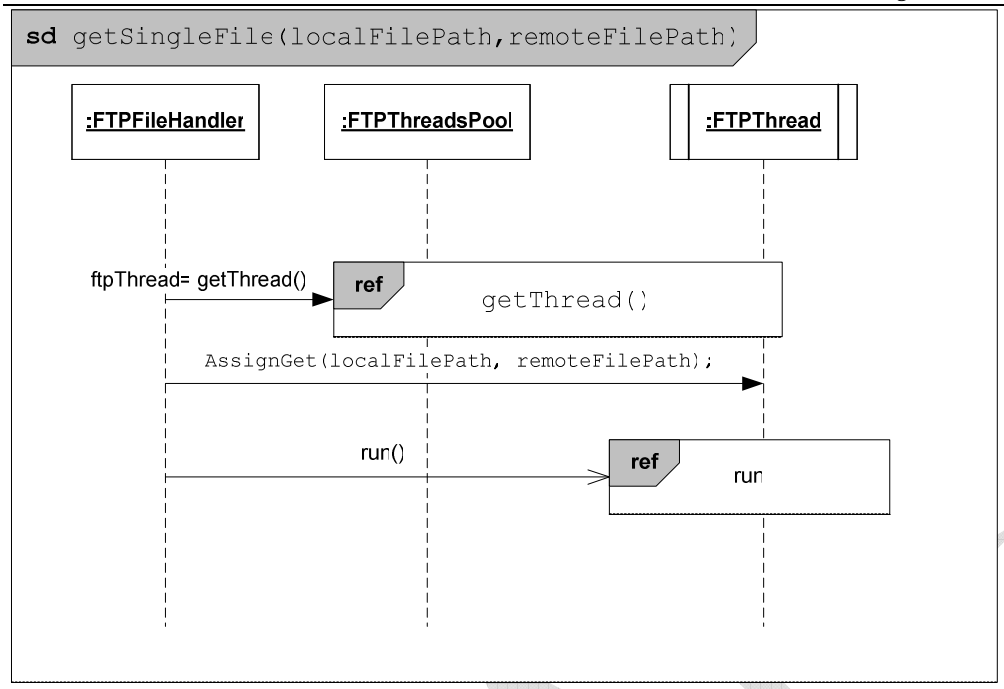
DRAFT



The following sequence diagram continue illustrating the download operation for a folder, which is referenced in the previous sequence diagram getSubFolder. As depicted, the downloaded folder is a recursive one, which means that if there are subfolders inside the source folder, they will be also copied.

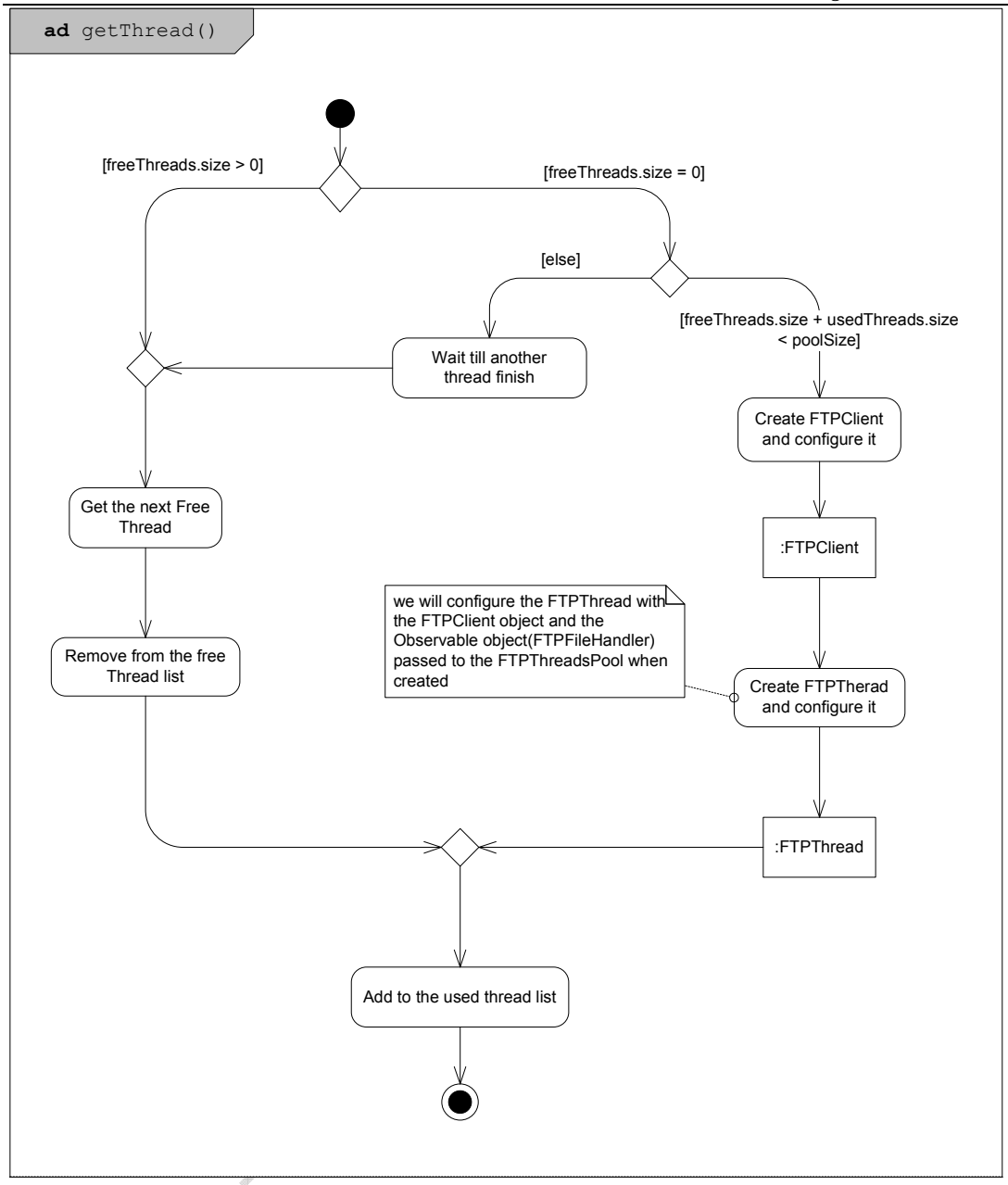


The following sequence diagram illustrates the scenario to download one file and how we use threads to fetch multiple files simultaneously, which is referenced in the previous sequence diagram getSingleFile. As depicted in the above class diagram, the FTPThread uses the FTP class library FTPClient to perform the file handling operation.

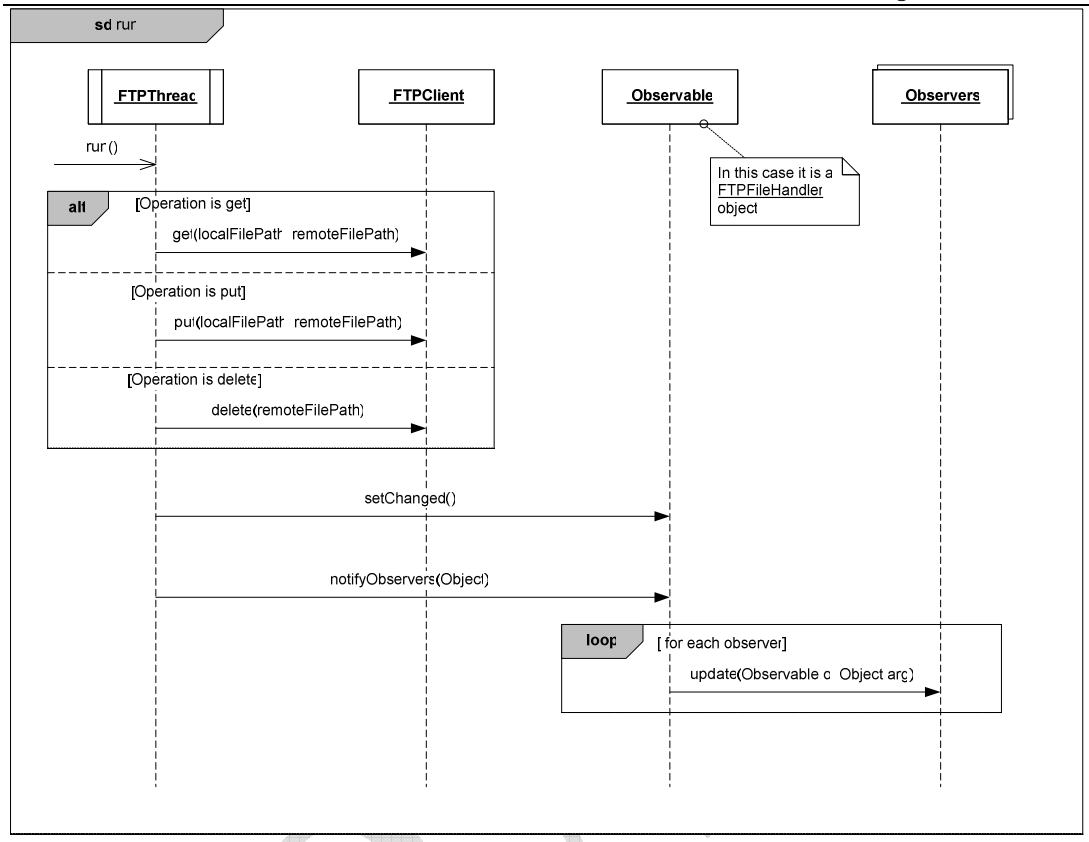


The following activity diagram shows how we get a thread from the thread pool. This diagram is referenced in the previous sequence diagram, getThread.

DRAFT



The following sequence diagram illustrates how the FTPThread handles file operations(download, upload, and delete) using the FTP class library FTPClient.
It also shows how the FTPThread changes the state of the observable object, and in turn notifies the observers' objects to update their states to reflect the status of file handling operations.



Local File Handling

The previous class diagram illustrates the classes involved in implementing local file handling. We used java.io file classes in the standard Java library. The design also applies two design patterns which allows for the flexibility and usability of system classes. FileHandlerLocalImpl is an adapter class to adapt the interface provided by LocalFileHandler class with the one expected by the clients of FileHandler abstract class. LocalFileHandler is a Façade class for all java.io file classes in the system. Unlike remote file handling, we did not used threads.

4.4 Database Handler

DAFv2 logic is implemented and saved into stored procedures and functions, there is not any single query passed to the database without calling Stored Procedure. Going after this technique was for security and code flexibility reasons.

Logic's security is applied into DAFv2 by embedding it inside database's stored procedures and functions and creating database users have no permissions except executing stored procedures and functions. Thus, users cannot hack the logic since they cannot insert, delete or update permissions.

This technique helps also in making DAFv2 flexible. If a DAFv2 developer decides to change some logic, he/she need not modify the Java code, instead, he/she will modify the effected stored procedures or functions and rebuild them. Definitely, this scenario is valid for code's segments that need nothing but database changes. In other words, developers need to modify and rebuild the Java code if the changes are related to GUI (for example).

Once the developer decides to get or set values in the database, he/she must do the following;

a. Decide which stored procedure he/she will use
b. Prepare arguments' list for this procedure
c. Get database connection
d. Pass storage procedure and arguments list to the method that execute the store procedures.
e. Receive the result data, if it exists.

It should be mentioned that these steps are encapsulated inside DAFv2 Entities (Revise administrator Module). Some functionalities, however, are not covered in System Entities. In such cases, applying preceding steps will be the solution.

4.4.1 Singletoon Connection

DAFv2 has at most a single live connection all over system modules, this is to reduce connection numbers and server management overhead. Singleton connection are implemented by declaring the connection constructor as a private one and creating a static method for getting the default connection. This function checks if the connection is alive or not; if not, it creates a new connection instance and considers it as the default one (Review JavaDoc file, class org.bibalex.daf.handlers.dbhandler.DBConnection).

4.4.2 OperationInfo Object

Executing a select statement in a procedure or function returns a result set that can be saved into result set object (JDBC). The retrieved data needs to be saved into some data structure; simply, it can be saved in array of objects, where every object maps to a single row.

DRAFT

OperationInfo object is our data structure which was built over result set object. It has two attributes;

a. isValid
b. result

isValid attribute is a Boolean file that indicates operation success; it will be "True" if the database operation is executed successfully, otherwise it will be "False".

Result attribute is responsible of data saving; it accepts any object structure. DAFv2 passes DataTable structure to the result object. DataTable is a Vector of Objects structure; it reads values from the result set rows, forms new objects rows, and inserts the formed objects into Vector, which builds eventually the DataTable itself. Accessing elements in DataTable object can be done using two ways:

a. Determine Row and Column index
b. Determine Row index and Column name

Check the following example;

```
1.    OperationInfo oprInf = new OperationInfo();
2.    JobType jobType = new JobType();
3.    DataTable dt = new DataTable();
4.    oprInf = jobType.loadAll();
5.    if (oprInf.isValid()) {
6.          dt= (DataTable) oprInf.result();
7.          int JTID = Integer.parseInt(dt.getValueAt(0, 0).toString());
8.    }
```

Fig. 4.4.1 OperationInfo Example

Lines 1, 2 and 3 initialize OpertionInfo, JobType entity and DataTable objects.
Line4 executes loadAll method in JobType entity, which gets all Job Types in DAFv2. This method returns OperationInfo object.
Line5 checks if the returned object is valid or not.
Line6 casts OperationInfo result object into DataTable object.
Line7 reads value from the DataTable (location 0,0) and parses it into integer. This line can be written as
int JTID = Integer.*parseInt*(dt.getValueAt(0, "JT_ID").toString());
Where "JT_ID" is the column name maps to column 0.

All methods interacting with database have to prepare argument list and choose stored procedure then forward them all to executeProcedure method (DBConnection class), which returns always an OperationInfo object (isValid value + DataTable value).

The User who is calling a method interacting with database and receiving the OperationInfo has to check isValid value at first to make sure operation was executed successfully (Fig. 4.4.1).

DRAFT