# UWS Library v3.0

## Documentation

8th March 2011

**Web-site:** http://saada.u-strasbg.fr/uwstuto



**Author:** Grégory Mantelet
**email:** gregory.mantelet@astro.unistra.fr
**Institute:** CDS (Strasbourg – France)

# CONTENTS

# A. Brief reminder of UWS

## 1. An asynchronous service

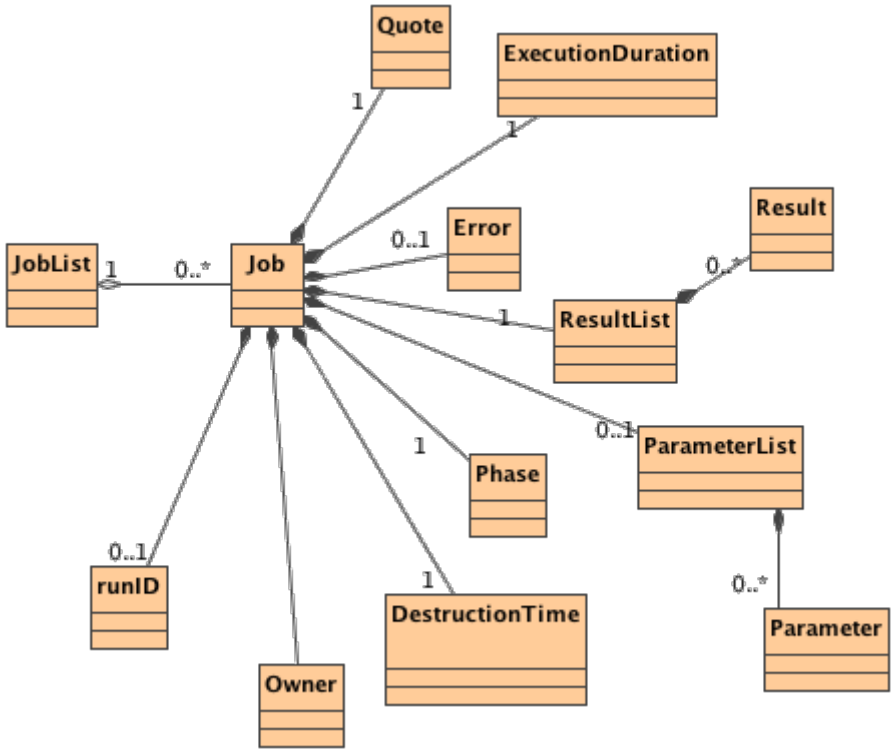
> *The Universal Worker Service (UWS) pattern defines how to manage asynchronous execution of jobs on a service.*

In a synchronous service the client must wait a response for each sent request. Thus he can make only one request at a time and if he disconnects from the service the request is abandoned.

UWS is an asynchronous service. That means the client has not to wait the response of one request before making another one. So it is possible that a request is executing for hours or days even if the client disconnects from the service. Indeed UWS is also (as most of asynchronous services) a state-full service that is to say it stores the state of each received request and so for the client addresses.

## 2. A job-oriented service

> *UWS consists logically of a set of objects that may be read and written to in order to control jobs. The objects are represented by elements within XML schema [...].*




A UWS is organized in one or more JobList objects. A JobList is a collection of Job. A job is described by:

- **an ID:** a unique identifier of the job in its JobList
- **a RunID:** a kind of label or name of the job given by its owner (it is not used to identify the job)
- **an Execution Phase:** the state of the job
- **an Execution Duration:** the maximum authorized duration of the job
- **a Deletion Time:** the time at which the job must be aborted and removed from its JobList

- **a Quote:** the estimated duration of the job
- **one or more results:** all the results of the job execution
- **one or zero error:** the error which has occurred during the job execution
- **zero or more additional parameters:** parameters useful for the job execution

## 3. Resources and URIs

> *In a REST binding of UWS, each of the objects defined above is available as a web resource with its own URI. These URIs must form a hierarchy [...]*

Considering *{jobs}* as a name or an identifier of a JobList and *(job-id)* a JobID:

| URI | Description | Value | Writable |
|---|---|---|---|
| /{jobs} | List of all jobs contained in the JobList *{jobs}* | | NO |
| /{jobs}/(job-id) | Description/Summary of the Job *(job-id)* | | NO |
| /{jobs}/(job-id)/runid | RunID of the Job *(job-id)* | A String value | YES |
| /{jobs}/(job-id)/phase | Execution Phase of the Job *(job-id)* | PENDING, QUEUED, EXECUTING, COMPLETED, ERROR, ABORTED, UNKNOWN, HELD or SUSPENDED | YES |
| /{jobs}/(job-id)/owner | Owner of the Job *(job-id)* | ? *(by default a String value)* | NO |
| /{jobs}/(job-id)/quote | Quote of the Job *(job-id)* | Integer number of seconds ≥ 0 ; a negative value means there is no quote information | NO |
| /{jobs}/(job-id)/executionduration | Execution Duration (in seconds) of the Job *(job-id)* | Integer number of seconds ≥ 0 ; 0 means unlimited execution duration | YES |
| /{jobs}/(job-id)/destruction | Destruction Time of the Job *(job-id)* | A date with the format "yyyy-MM-dd'T'HH:mm:ss.SSSZ" (ISO:8601) | YES |
| /{jobs}/(job-id)/error | Description/Summary of the error of the Job *(job-id)* | | NO |
| /{jobs}/(job-id)/results | Results list of the Job *(job-id)* | | NO |
| /{jobs}/(job-id)/parameters | List of the additional parameters of the Job *(job-id)* | | YES |

All these web resources are formatted in XML according to the UWS Schema available in the Appendix B of the IVOA Recommendation. Now almost all browsers are able to transform on the client side a XML document into a HTML document by using a XSLT resource *(Click here for a XSLT tutorial).*

# B. Using a UWS

In this part the usability of UWS will be described through an example: the UWS Timers. Its URL is http://saada.u-strasbg.fr/uwstuto/basic. It manages only one JobList named *timers*. A HTML form to interact with this UWS is available at http://saada.u-strasbg.fr/uwstuto/basic.html. With this form you can easily manipulate the URL, the HTTP method and the parameters, so that you can experiment yourself the different types of request.

**So, now, let's see the basic operations of a UWS !**

## 1. List jobs

To list all the jobs of a JobList you only have to select the web resource associated with the JobList. Here you just have to type in your browser the URL:

*http://saada.u-strasbg.fr/uwstuto/basic/timers*

## 2. Create a job

To create a job you just need to send a **HTTP-POST** request to the JobList. No parameter is required, but if you want you can give one of the following:

- *RUNID*
- *EXECUTIONDURATION*
- *DESTRUCTION*
- one or several additional parameters
- *PHASE=RUN* (to start the job just after its creation)

The JobID must be generated automatically so that it can identify only one job in the whole jobs list.

Once created the response to the request must be a redirection to the summary of the Job, that is to say: an HTTP code 303 ("See other") and the URL *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345* (supposing the created job has the ID 12345). If the parameter *PHASE* has been given with the value *RUN*, the Job must start directly.

## 3. Get a job summary

As said previously UWS is a service based on REST. That means that any resource is available thanks to a URI build in a hierarchical manner. To get a job summary, you need to know the URL of the UWS, the name of the JobList and the ID of the job:

*UWS_URL+"/"+JobList_name+"/"+Job_ID.*

So you get the following URL: *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345*.

***Note 1: The other resources***

*All other objects which compose a Job object are available in the same way:*

- *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345/runid*
- *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345/phase*
- *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345/executionduration*
- *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345/destruction*
- ...

***Note 2: The error summary case***

*An ErrorSummary object is represented with 3 main pieces of information:*

- *an error type (fatal or transient),*
- *a message,*
- *and the attribute hasDetail - a boolean value which indicates if there are more details about the error (for instance: a stack trace or an execution log).*

The detailed message must be available at the URI */{jobs}/(job-id)/error* (in our example the corresponding URL is *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345/error*).

## 4. Change some job parameters

Some job parameters are writable. That means the client can change their value by sending a **HTTP-POST** request to the URL of the job parameter to change. The request must have exactly one parameter whose the key must be the name of the parameter to change. The response will always be a redirection to the job summary.

For example: To change the parameter Execution Duration you must send a request to *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345/executionduration* with the parameter *EXECUTIONDURATION=120* (for 120 seconds). The response will be a redirection to *http://saada.u-strasbg.fr/uwstuto/basic/timers/12345*.

Additional parameters must be updated in a different way. A **HTTP-POST** request must be sent to the URI */{jobs}/(job-id)/parameters* with one key-value pair which associates the name of the parameter and its value. An alternative is to send a **HTTP-PUT** request with the same parameter but at the following URI */{jobs}/(job-id)/parameters/{paramName}*.

**Warning:**

The value of a writable parameter can be changed ONLY IF the job has not been yet executed !

## 5. Start a job

Starting a job implies sending a **HTTP-POST** request to the job phase with the parameter *PHASE=RUN*. In this case the execution phase of the Job goes from PENDING to QUEUED when the server starts to execute the Job, and finally to EXECUTING when the execution finishes successfully.

## 6. Abort a job

Aborting a job implies sending a **HTTP-POST** request to the job phase with the parameter *PHASE=ABORT*. In this case the execution must finished and the execution phase must be set to *ABORT*.

## 7. Destroy a job

There are two possible ways to remove a job from a jobs list:

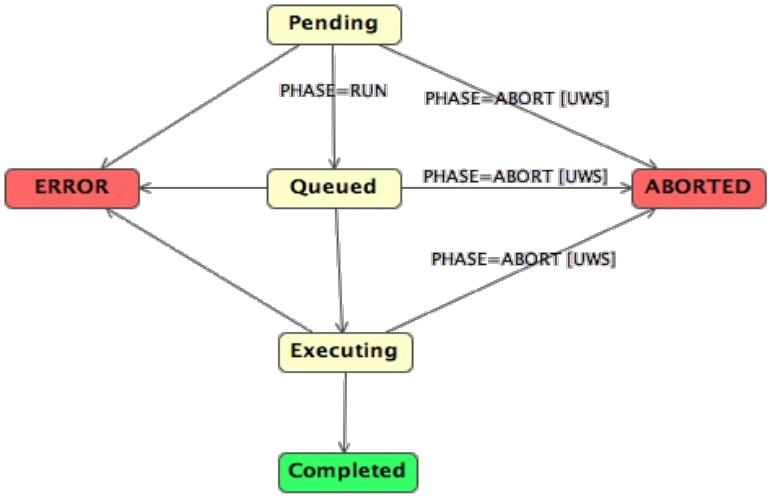
- by sending a **HTTP-DELETE** request to job resource
- by sending a **HTTP-POST** request with the parameter *ACTION=DELETE* to the job resource

When receiving one of these two request the Job execution must abort, all resources associated with the Job (result file, error file, ...) must be deleted and finally the Job must be removed from its JobList. The response to these requests is a redirection to the JobList: *http://saada.u-strasbg.fr/uwstuto/basic/timers*.

## 8. Important notes

- When the execution is longer than the given Execution Duration, the job must be aborted. But all previously generated results are retained.
- When the Destruction Time of a Job is reached, the job must be aborted - if running - and destroyed (included all generated results and errors).
- The Execution Phase cannot be changed freely !

> *The Job is treated as a state machine with the Execution Phase naming the state.*
> *[...] A successful Job will normally progress through the PENDING, QUEUED, EXECUTING, COMPLETED phases in that order. At any time before the COMPLETED phase a job may either be ABORTED or may suffer an ERROR.*

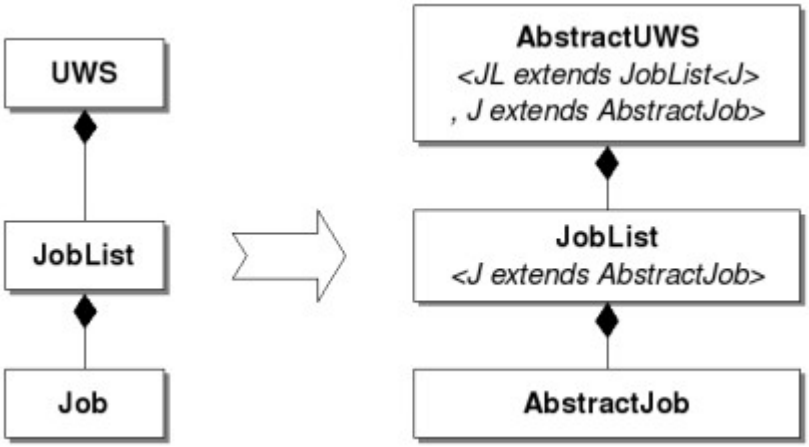


# C. Getting started

## Introduction

This library is a convenient implementation of a UWS which already implements all behaviors and functionalities described by the IVOA Recommendation. It is designed to be as quick and easy to use as possible so that a developer of a UWS has not to worry with the UWS management.

Furthermore the representation of a UWS by the IVOA Recommendation remains the same in this library (even if the name of the objects are slightly different): a UWS is a set of jobs lists which are sets of jobs.




In practice all that change between two UWServices is the execution task of their Jobs. The service management - *included all possible requests it may receive* - never changes. That's why this library already manages all the behaviors of the IVOA Recommendation and why a job is represented by an abstract class (AbstractJob).

So basically to make your own UWS with this library you will have to define the task of a job by extending the class AbstractJob. Once done, you will write a Java servlet in which you will create a UWS *(instance of an AbstractUWS sub-class: one of the default or your own)* and forward all HTTP requests to the built UWS. And that's all !

**Let's take an example !** In this part of the tutorial, we will build, step by step, a very simple UWS which has to manage only one jobs list of timers. A timer is an object which executes an action when the given time is elapsed. In our example, a timer will write a file with a basic message: *"X seconds elapsed"* (where X is the given time).

More concretely, to make the described UWS we will follow the next three steps:

1. Choosing the "type of UWS"
2. Writing the servlet
3. Defining the job (its parameters, its results and its task)

# 1. Choosing the "type of UWS"

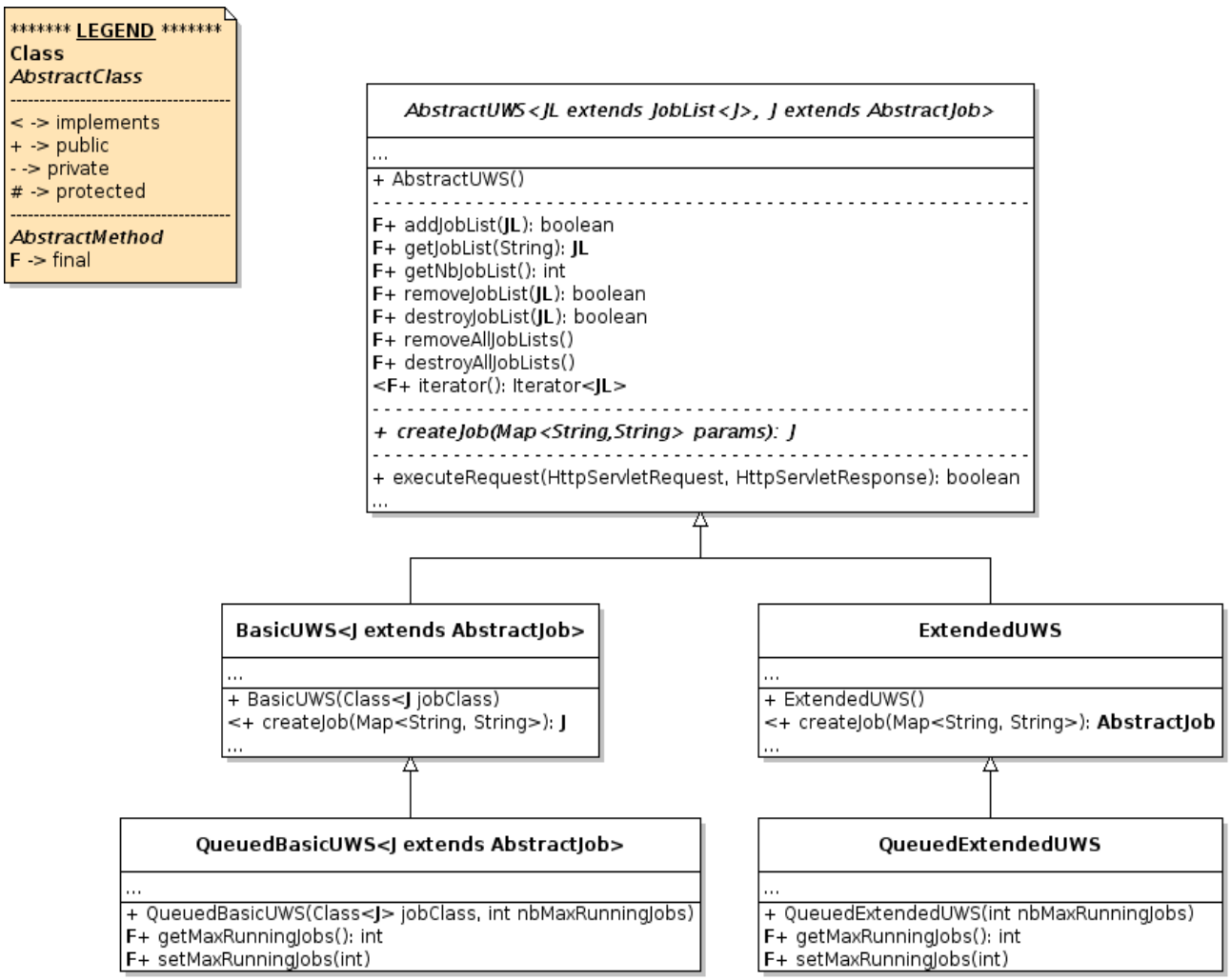
This UWS library provides five classes which are able to represent a UWS: AbstractUWS, BasicUWS, ExtendedUWS, QueuedBasicUWS and QueuedExtendedUWS. The first one is an abstract class whereas the four others are its sub-classes. Depending of your needs you are going to use one class rather than another one, hence this first step: *Choosing the "type of UWS"*. So that helping you to make your choice, lets introduce these classes...

## AbstractUWS

The main goal of a UWS is to manage a set of jobs lists in addition to all jobs resources. AbstractUWS already defines all methods required to easily manage its jobs lists. Particularly you can add new jobs list thanks to the method: addJobList(JL). Besides, it may also manage an execution queue. That implies that some jobs may be put in a QUEUED state until enough resources has been freed. Obviously all default behaviors of a UWS described by the IVOA Recommendation are already fully implemented.

The only abstract method of AbstractUWS is createJob(Map<String, String> params). It lets creating a new job with the given parameters each time a such request is sent to the UWS. This function is abstract because the type of job to create is not predictable in a generic way. For that, a concrete service is needed ! That means you would have to make an extension of AbstractUWS for each service.

However in spite of extending AbstractUWS at each time, you can use one of its default sub-classes: BasicUWS, ExtendedUWS, QueuedBasicUWS or QueuedExtendedUWS. Below a simplified class diagram of all theses classes:



## BasicUWS and ExtendedUWS

BasicUWS and ExtendedUWS propose a simplified definition of a UWS. So you will never need to extend AbstractUWS ! Indeed both have been designed so that each jobs list can manage ONLY ONE GIVEN kind of job. Their only difference lies in the fact that in BasicUWS ALL managed jobs lists MUST be of ONLY one given type of job, whereas ExtendedUWS allows that each jobs list may be of a different type of job.

Let's take some examples:

- To make a UWS which manage ONE jobs lists of JobA, you may use BasicUWS:

```
BasicUWS<JobA> uws = new BasicUWS<JobA>(baseUWSUrl, JobA.class);
uws.addJobList(new JobList(jlUrl_A));
```

- But to make a UWS which manages TWO (at least) jobs lists, the first of JobA and the second of JobB, you may use ExtendedUWS:

```
ExtendedUWS uws = new ExtendedUWS(baseUWSUrl);
uws.addJobList(new JobList(jlUrl_A), JobA.class);
uws.addJobList(new JobList(jlUrl_B), JobB.class);
```

**IMPORTANT**

**These both classes use the Java Reflection in their method createJob to create the jobs of the good type. Hence: JobA.class and JobB.class. Thus they expect to find the constructor of AbstractJob with only one parameter (of type Map): see AbstractJob(Map<String, String>). So the used job types must have at least this constructor to be correctly managed with BasicUWS, ExtendedUWS and their sub-classes !**

## QueuedBasicUWS and QueuedExtendedUWS

QueuedBasicUWS and QueuedExtendedUWS extend respectively BasicUWS and ExtendedUWS. They just have one additional functionnality: they can manage an execution queue. A job is going to run only if there are less running jobs than a given number. Otherwise the job is put in a queue until a running job ends. The maximum number of running jobs may be initialized at the creation of the UWS and may be changed afterwards with setMaxRunningJobs(int).

The execution queue management can be customized quite easily for any sub-class of AbstractUWS. For more information see C.3.b. Execution WITH queue.

**IMPORTANT**

**As for BasicUWS and ExtendedUWS, the used job types must have at least the constructor of AbstractJob with only parameter of type Map<String,String> !**

## Which one choosing ?

As said previously to make a UWS you always need an instance of an AbstractUWS sub-class. You have two solutions: either you extend AbstractUWS or you use directly one of its sub-classes. To help you determining which solution is the best you can ask yourself the two following questions:

1. **«May my UWS have to manage different kinds of job ?»**
   - YES: ExtendedUWS or QueuedExtendedUWS
   - NO: BasicUWS or QueuedBasicUWS
2. **«Need I an execution queue ?»**
   - YES: QueuedBasicUWS or QueuedExtendedUWS
   - NO: BasicUWS or ExtendedUWS

In our example, we will use only one kind of job and we do not require an execution queue *(because a timer does not take much resources)*. So we will use BasicUWS !

# 2. Writing the servlet

A UWService is a Web-Service. Since this library is developed in Java, a UWService will be implemented by a HttpServlet. This part of the tutorial explains how to write a Servlet using this library to create and to manage a UWService.

To execute a servlet, Tomcat must be installed on your server. This UWS library has been tested with the versions 6 and 7 of Apache/Tomcat.

## The HttpServlet class

It lets defining a dynamic web-resource for the protocol HTTP. Since this class is abstract, you must override it. There is already one constructor with no parameter. In the most cases it is not needed to override it or even to add more constructor.

It has several methods but only three of them will interest us:

- *init()* or *init(ServletConfig)* : called only at the first use of the servlet
- *destroy()* : called at the end of a Tomcat session, particularly when Tomcat stops or restarts
- *service(HttpServletRequest, HttpServletResponse)* : called at each request on the servlet. It is used as a hub towards the following methods, in function of the used HTTP method: doGet (HTTP-GET) , doPost (HTTP-POST) , doPut (HTTP-PUT) , doDelete (HTTP-DELETE), ...

**Warning**

The same HttpServlet is used for all clients. It means that all class attributes will have the same value for all clients. So beware of the way you use them !

## UWSTimers

UWSTimers is the name of the servlet we want to create in this tutorial. It will have only one attribute whose the type has been choosen in the previous part: *BasicUWS<JobChrono> uws*. Now all we have to do is to initialize this variable and to forward it all requests the servlet receives.

*JobChrono* is the type of the job our UWS must manage. It will be defined in the next part of this tutorial.

### Initializing the UWS

The initialization of a UWS must be done at the first request sent to the servlet. So our UWS will be initialized in the init(ServletConfig) method of the servlet UWSTimers:

```
@Override
public void init(ServletConfig config) throws ServletException {
    super.init(config);
    try{
        // Create the UWS [required]:
        uws = new BasicUWS<JobChrono>(JobChrono.class);

        // Set a description [optional]:
        uws.setDescription("This UWS aims to manage one (or more) JobList(s) of
JobChrono." +
            "JobChrono is a kind of Job whose the execution task consists to wait a
given time" +
            "before executing an action.");
```

```
        // Create the job list "timers" [required]:
        uws.addJobList(new JobList<JobChrono>("timers"));
    }catch(UWSException ex){
        throw new ServletException(ex);
    }
}
```

### Forwarding requests to the UWS

**Forwarding request to the UWS is actually easier than meets the eye !** AbstractUWS has been designed to be able to interpret HTTP servlet requests and to answer them in consequence. So to do that, you have only to call the method AbstractUWS.executeRequest(HttpServletRequest, HttpServletResponse). This method can interpret any kind of request, whatever is the HTTP method (GET, POST, PUT or DELETE). If a request is invalid - according to the IVOA Recommendation - this method throws an exception which specifies the HTTP error code in addition to a message.

Here is the service(HttpServletRequest, HttpServletResponse) method of UWSTimers in which requests are forwarding to our UWS:

```
@Override
protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
    try{
        // Forward the request to the uws [required]:
        uws.executeRequest(req, resp);
    }catch(UWSException uwsEx){
        // Display properly the caught UWSException:
        resp.sendError(uwsEx.getHttpErrorCode(), uwsEx.getMessage());
    }
}
```

## Customize your UWS

By using this UWS library, the minimal code of a servlet implementing a UWService is only limited to the above code *(except for the uws description which can be omitted)*. But so that making more controls while using the UWS, the full source code of UWSTimers is a little longer than the explained one. For instance the execution duration and the destruction time have a maximum and a default value. Besides each user of this UWS is identified so that they can manage only their own jobs.

All customizations of your UWS should be done at its initialization, in the init(ServletConfig) method. Besides the part E. How to customize a UWS ? lists and explains the most usefull UWS customizations. However to help you writing quickly your servlet you can download the following template. It contains the above minimal code and some UWS customizations.

# 3. Defining the job

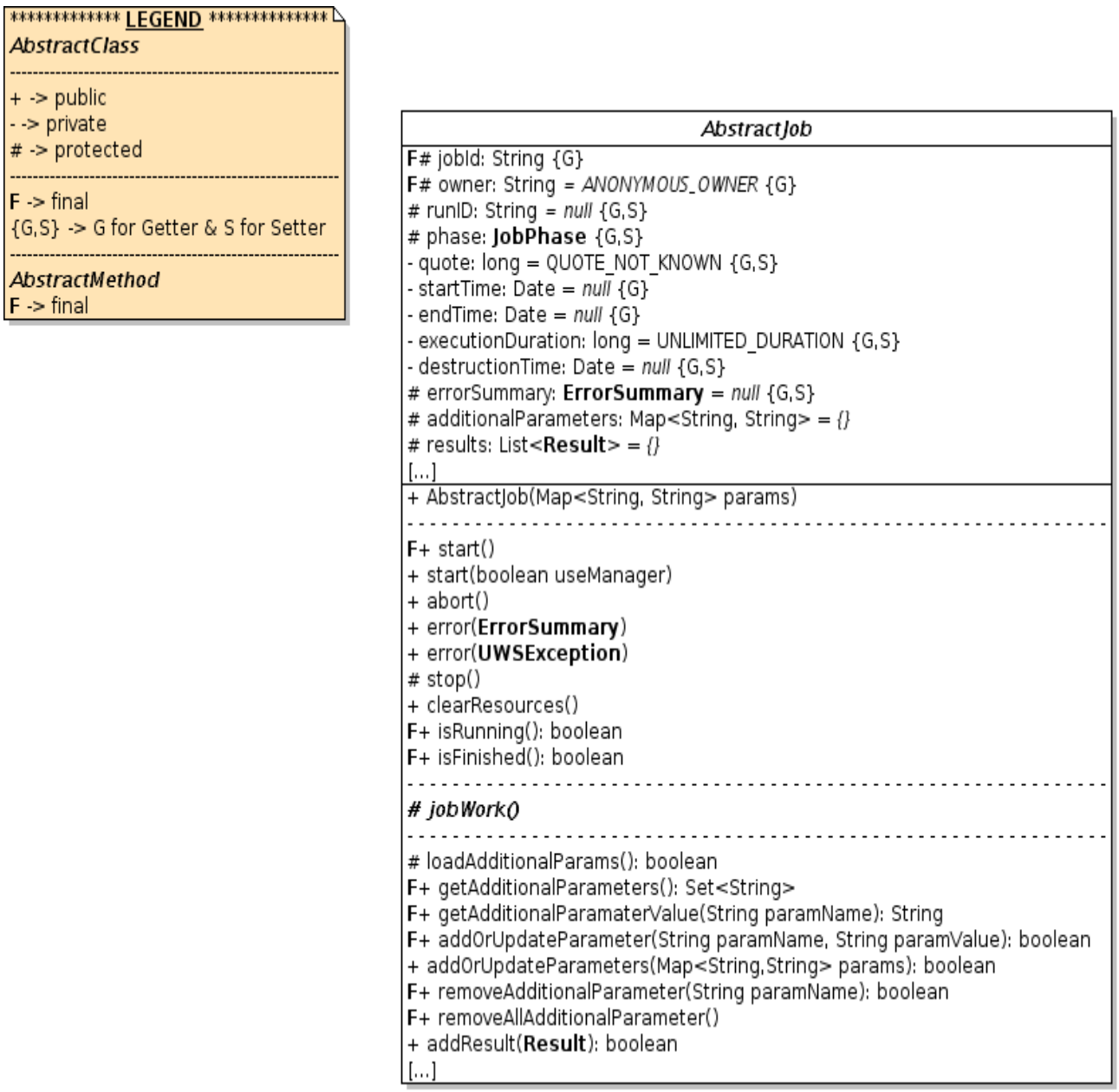
Now we have our servlet and our UWS all that's missing is the job ! With this library a job is an instance of an AbstractJob sub-class. So the job our UWS must manage - named previously "JobChrono" - must be an extension of AbstractJob. In this last part of the tutorial you will see how to define a UWS job through the example of JobChrono.

## AbstractJob

The IVOA Recommendation describes the main attributes of a job as well as its behaviors. All this description has been preserved in AbstractJob.

As said in the introduction, the task of a job is the only thing that really change between two UWS. It is for this reason that this library defines a job in an abstract class. The function jobWork(), which lets defining the task of a job, is the only abstract method.

Here is a simplified class diagram of AbstractJob:



**Now let's see how to extend** AbstractJob **!**

## The constructor

AbstractJob has four constructors. Three of them require a map of parameters (default and/or additional). If the job has to be used by BasicUWS, ExtendedUWS or one of their sub-class you must ensure that the constructor with only the map of parameters exists. Indeed these classes use the Java Reflection to create jobs. They expect to use the constructor with only one parameter of type Map: AbstractJob(Map<String,String>).

The last constructor is discouraged because it lets initializing manually all the fields of the new job. No processing or check is done on the attributes. Consequently it can produce errors or "paradoxes" like a job with a phase COMPLETED but with an error summary, or worse a job whose the job ID is already used by another job *(in this case the job can not be added in its job list)*.

**In the most cases, the constructor with only one parameter of type Map (AbstractJob(Map)) is clearly enough !**

```
public JobChrono(Map<String, String> lstParam) throws UWSException {
    super(lstParam);
}
```

## Managing job parameters

Any timer has to wait a given time. This time must be set at least at the job initialization and may be updated before the job execution. As it is not a default attribute of a UWS job, it is considered as an additional parameter and so it is stored in the attribute additionalParameters. With AbstractJob you can check or process these parameters by overriding loadAdditionalParams(). This function does nothing by default and is called by addOrUpdateParameters(Map<String,String>) when one or more job parameters (additionnal or not) has to be added or updated.

addOrUpdateParameters(Map<String,String>) always does the following actions:

1. Call loadDefaultParams(Map<String,String>) which loads all job attributes described by the IVOA Recommendation. All the corresponding items are removed from the given map.
2. Add/Update all remaining items to the job attribute additionalParameters
3. Call loadAdditionalParams()

This function is also called to initialize the job attributes in the main constructor of AbstractJob: AbstractJob(Map<String,String>).

So if you want to manage yourself some additional job attributes, you must extend loadAdditionalParams(), extract the items that interest you and do what you want with them. That is exactly what we are going to do to initialize (or update) the "time to wait" of JobChrono:

```
protected int time = 0;

protected boolean loadAdditionalParams() throws UWSException {
    // JobChrono needs only one parameter for its execution: the time:
    if (additionalParameters.containsKey("time")){
        try{
            time = Integer.parseInt(additionalParameters.get("time"));
            if (time < 0)
                time = 0;
            // If you want you can remove this parameter from the map
additionalParameters:
```

```
            // additionalParameters.remove("time");
        }catch(NumberFormatException nfe){
            throw new UWSException(UWSException.BAD_REQUEST, "The given TIME value
(\""+additionalParameters.get("time")+"\") is incorrect: it must be a positive integer
value !");
        }
    }

    return true;
}
```

***Notes:***

- *Rather than creating a class attribute for the "time to wait" and extracting its value from the map additionalParameters, you can merely leave the corresponding item in the map and get its value when needed.*
- *In loadAdditionalParams() you have a full access to the map additionalParameters, and so you are able to remove an item if it is not correct or if you have already used its value. Besides you can also add or update some items if needed.*

## Writing the task

Writing the task means overriding the abstract method jobWork(). This method will be then called in a separated thread during the job execution. The whole thread execution and the phase transitions are already managed by this library. However you should beware to the following points when defining a job task:

- You should not have to change the job phase !
- You should check as often as possible whether the thread has been interrupted *(which would mean the user has aborted the job)*. In this case you must throw an InterruptedException so that the job execution can be stopped with the status ABORTED:

```
if (thread.isInterrupted())
    throw new InterruptedException();
```

- Any error (exception or not) must be thrown in an UWSException, so that the job execution can be stopped with the status ERROR. The error message is fetched directly from the exception.
- Writing the results is your responsibility ! It MUST be done in this method.

Now let's see what looks like the implementation of this method for JobChrono ! I remind you that the task of JobChrono is to wait a given time before writing a file with the following content: *X seconds are elapsed*.

```
protected synchronized void jobWork() throws UWSException, InterruptedException {
    int count = 0;

        // 1. EXECUTION TASK = to wait {time} seconds:
        while(!thread.isInterrupted() && count < time){
            Thread.sleep(1000);
            count++;
        }

    // If the task has been canceled/interrupted, throw the corresponding exception:
    if (thread.isInterrupted())
        throw new InterruptedException();

    // 2. WRITE THE RESULT FILE:
    String fileName = "JobChrono_n"+getJobId()+"_result.txt";
    File f = new File(resultsDir, fileName);

    try {
        // Build the directory if not existing:
```

```
        if (!f.getParentFile().exists())
            f.getParentFile().mkdirs();

        // Write the result:
        BufferedWriter writer = new BufferedWriter(new FileWriter(f));
        writer.write(time+" seconds elapsed");
        writer.close();

        // Add it to the results list of this job:
        addResult(new Result("Report", "Info", "/uwstuto/jobResults/"+fileName));

    } catch (IOException e) {
        // If there is an error, encapsulate it in an UWSException so that an error
summary can be published:
        throw new UWSException(UWSException.INTERNAL_SERVER_ERROR, e, "Impossible to
write the result file at \""+f.getAbsolutePath()+"\" !", ErrorType.TRANSIENT);
    }
}
```

As you can see this method is divided in two parts. In the first part we wait as many seconds as needed until the given time is elapsed. In the second part the result file is written. Once done it is set to the job *(line 29)* thanks to the method addResult.

Furthermore you can notice at lines 5 and 11 that the interrupted flag is checked. If it is *true* the job is aborted at line 12 by throwing an InterruptedException. Besides if an error occurs while writing the result file a UWSException is thrown at line 33. By doing that the job is immediately stopped with the execution phase ERROR.

As you have surely noticed the result file is written in *resultsDir* while the address given at the creation of the Result object is *"/uwstuto/jobResults"*. Actually the Result address (absolute or relative) is the one to use to access the result. It must be a public address (i.e. http://...). In the other hand the address used to write the file is a file path (file://...), whose the direct access is forbidden out of the server.

However both addresses point exactly on the same file ! Like here, in the most cases, you will have to make a such difference between the address used to write the file and the one used to read it. What you must remember is that the Result address must always be used at least to read the result file !

*resultsDir* is the path of the directory which has to contain all job results files. It is a static attribute of JobChrono which is set at the initialization of the servlet UWSTimers.

## Clearing resources

When the job is removed from its jobs list, the UWS stops it and frees all its resources by calling its method clearResources(). In our example we create a file at the end of the execution. This file must be deleted when the job is destroyed otherwise we risk to have some space problems...

```
protected void clearResources() {
    // 1. STOP THE JOB (if running):
    super.clearResources();

    // 2. DELETE THE RESULT FILE (if any):
    try {
        File f = new File(resultsDir+"JobChrono_n"+getJobId()+"_result.txt");
        if (f.exists() && f.canWrite())
            f.delete();
    } catch (Exception e) {
        System.err.println("### UWS ERROR: "+e.getMessage()+" ###");
        e.printStackTrace();
    }
}
```

# Conclusion

This tutorial has been done on a very simple example of a UWService. Obviously this UWS library offers more functionalities *(like adding/changing/removing actions to your UWS)*. To know more about them see the parts D. How to customize a Job ? and E. How to customize a UWS ?. **However among this example you have learnt to use basically this UWS library, and...**

...here are the most important things you should now know:

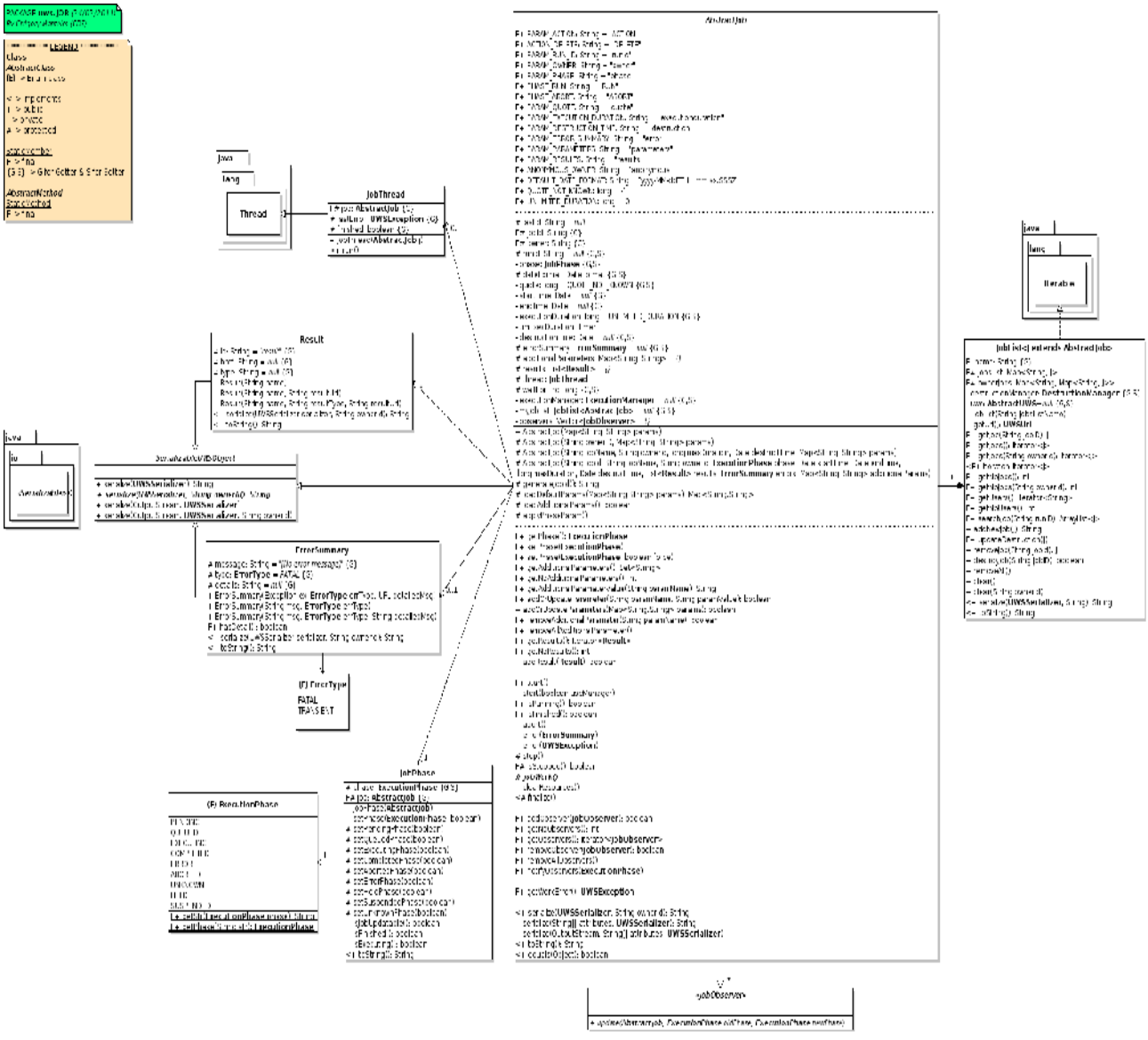
# D. How to customize a Job ?

In the part C.3. Defining the job you have learnt that by extending AbstractJob and then overriding the abstract method jobWork(), you can describe what a job must do during its execution. In this part you will see that the extension of AbstractJob lets you also customizing your job. For instance: changing the job ID generation, the date format or how keeping an eye on some jobs with a JobObserver.

Below are the possible job customizations you will see:

1. Job ID
2. Date format
3. Execution WITHOUT queue
4. Execution WITH queue
5. Error summary
6. Destruction
7. Keeping an eye on jobs

Obviously this list is not exhaustive ! Indeed, this library has been designed to leave the most freedom as possible to the developers. Only some sensitive behaviors or attributes are definitely unmodifiable. Thus if you have to modify some of the objects defined in this library for your UWS, you may do it by extending the good classes. For that you will surely need the following class diagram:



All examples shown in this part of the tutorial are extracted from another UWS example: Algorithms. This UWS has three jobs lists which each manages a different kind of algorithm (Chvatal, Syracuse and Ackerman).

# 1. Job ID

The ID of a job is generated at its creation by the function generateJobId(). By default it is the creation time in milliseconds and a upper-case letter *(starting with 'A')* which is incremented if the resulting ID is already used.

For instance: if the current date is *2011-01-18T16:50:12.163+0100* you will get: *1295365812163**A*** ; if it is already used by another job, *1295365812163**B***, and so on.

However you can change the ID generation by overriding the function generateJobId(). It is what is done for the jobs of the UWS Algorithms:

```
public class JobChvatal extends AbstractJob {
...
    @Override
    protected synchronized String generateJobId() {
        return "Chvatal_"+super.generateJobId();
    }
...
}
```

The resulting IDs will be the same than the formers, but prefixed with the name of the type of job *(here: Chvatal)*.

**IMPORTANT**

**You must ensure that all generated IDs are UNIQUE ! A Job whose the ID is already used is NOT added to the jobs list.**

# 2. Date format

According to the IVOA Recommendation, all Job dates must have a format compatible with the standard **ISO-8601**.

By default AbstractJob follows this standard by using the format **yyyy-MM-dd'T'HH:mm:ss.SSSZ**. It is stored in the attribute DEFAULT_DATE_FORMAT, and is used to parse it when the user sets/updates an attribute of type Date, and to format it when a date has to be displayed (*particularly during the serialization of a Job*).

This date format can be changed thanks to the method setDateFormat(DateFormat). Here is an example of how to use it:

```
public class JobChvatal extends AbstractJob {
...
    public JobChvatal(Map<String, String> lstParam) throws UWSException {
        super(lstParam);
        setDateFormat(new SimpleDateFormat("dd/MM/yyyy HH:mm:ss.SSS"));
    }
...
}
```

Thus any date of Chvatal jobs must follow the format: **dd/MM/yyyy HH:mm:ss.SSS** *(ex: instead of*

*"2011-01-18T16:50:12.163+0100" , we must have "18/01/2011 16:50:12.163").*

**Warning !**

All job attributes given in a HTTP request **must have the same format** than the one defined in the implied type of Job !

# 3.a. Execution WITHOUT queue

## How does it work ?

As seen previously the task of a job is described by the method jobWork(). During its life the status of a job may change, but always starts with PENDING. This phase is stored in the attribute phase. In addition to the getter of this attribute, isRunning() and isFinished() lets indicating whether the job is currently executing or finished.

The execution of a job can be managed thanks to the following methods:

- start() *(*=start(false)*)*: Starts the execution of the job.
    1. *if* isRunning()=*true then nothing is done !*
    2. *Change the execution phase to EXECUTING (*setPhase(ExecutionPhase)*)*
    3. *Create the thread and start it (its execution will call* jobWork()*)*
    4. *Set the start time (*setStartTime(Date)*)*
    5. *Start the timer for the maximum execution duration if positive and different from 0*
- abort(): Aborts/Interrupts the job.
    1. *Stop the thread (*stop()*)*
    2. *if* isStopped()=*false then return here*
    3. *Change the execution phase to ABORTED (*setPhase(ExecutionPhase)*)*
    4. *Set the end time (*setEndTime(Date)*)*
- error(ErrorSummary)**:** Stops immediately the job with the given error.
    1. *Stop t*he thread (stop())
    2. if isStopped()=false then return here
    3. Set the error summary (setErrorSummary(ErrorSummary))
    4. Change the execution phase to ERROR (setPhase(ExecutionPhase))
    5. Set the end time (setEndTime(Date))
- error(UWSException)**:** Stops immediately the job with the given UWSException.
  *Call* UWSToolBox.publishErrorSummary(AbstractJob, String, ErrorType).

As you can notice there is no method for the case the job ends successfully. Actually it is the role of the thread ! When the call to jobWork() has just finished, it sets the phase to COMPLETED (setPhase(ExecutionPhase)) and sets the end time (setEndTime(Date)).

In a UWS to start or to abort a job, a POST request with the parameter PHASE=RUN or PHASE=ABORT must be sent to the job. As said in the part C.3. Defining the job, this parameter is stored in the additionalParameters attribute. To apply this parameter, the method applyPhaseParam() must be called, which is done by default after each update of any job parameter. Thus, we get:

- PHASE=RUN => start()
- PHASE=ABORT => abort()

All these methods can also be used if the job is executed alone (that's to say: independently from a UWS) !

## Stopping job

The method stop() is used to stop the thread when the user asks to abort the job or when the maximum

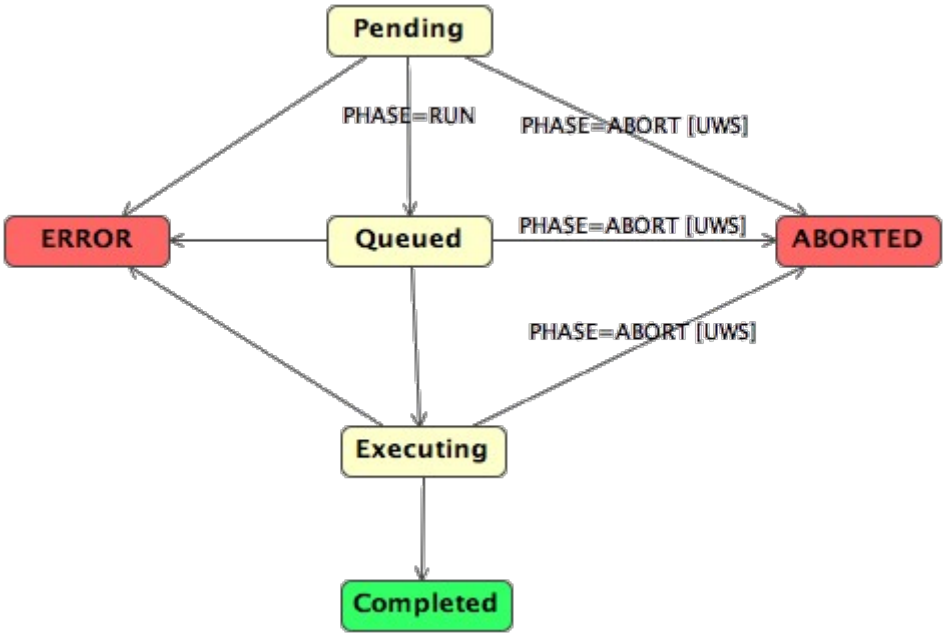
execution duration is elapsed or when an error occurs. However stopping a thread is not so trivial. That's why you should know some things about the way it is done in this library through the method stop().

Firstly the thread is stopped by calling the function Thread.interrupt(). If the thread is waiting (Thread.wait()) or sleeping (Thread.sleep(long) then, it will receive an InterruptedException. Otherwise its interrupted flag is set. That's why you should check as often as possible this flag in jobWork(). If it is true, it is your responsibility to throw either an InterruptedException or a UWSException. The other solution is to call yourself the abort() or error(ErrorSummary) function.

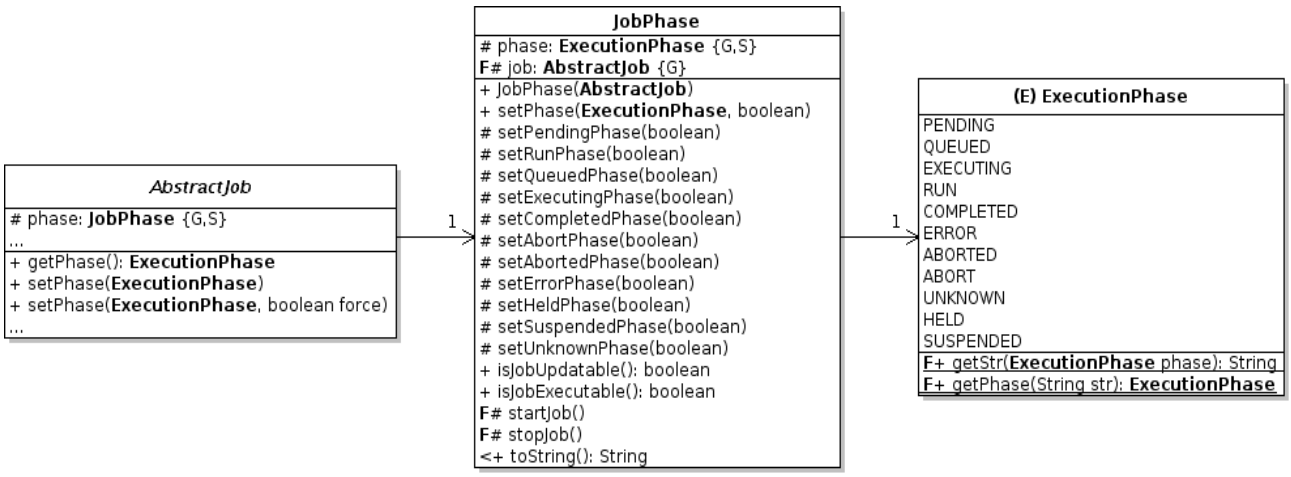
Secondly after the interruption of the thread, stop() waits until the thread is really stopped by using Thread.join(long). The time to wait is by default 1000ms (1 second) and can be changed thanks to setTimeToWaitForEnd(long).

## Changing the execution phase

A job has an attribute which indicates its current status *(pending, queued, executing, completed, error, ...)*, also called *execution phase*. All the default possible execution phases are listed in the enumeration class: ExecutionPhase. Transitions between phases are imposed by the IVOA Recommendation:




In order to manage more easily these transitions and particularly to allow their modifications, they are not managed by AbstractJob but by the class JobPhase. Consequently the current execution phase of a job is got and set through an instance of JobPhase. So we get the following simplified class diagram:




## How to customize ?

All the above functions can be overrided. Thus with start(boolean), abort(), error(ErrorSummary) and error(UWSException) you can customize what to do when starting or aborting (with or without error) the job. Besides the interruption of the thread can also be modified by overriding stop(). You should also remember that with setTimeToWaitForEnd(long) you can change the maximum time the stop() method must wait until the end of the thread.

About the execution phases you should know that all the default transitions are already implemented, except for HELD, SUSPENDED and UNKNOWN which are not used in this library. As you have seen, to change the current phase you must use the setPhase(ExecutionPhase, boolean) function. Actually it is a kind of hub between the *set...Phase(boolean)* functions *(i.e. setExecutingPhase(boolean))*. There is one function *set...Phase(boolean)* per execution phase. Thus if you want to change a phase transition action:

1. Override the corresponding function. *For instance to change what is done when going to the* EXECUTING *phase you have to override* setExecutingPhase(boolean).
2. Once done, set the extension of JobPhase to your uws object thanks to the function setPhaseManager(JobPhase).

You must take care to the current execution phase before changing it ! For instance: it would be a non-sense to allow a job going from ERROR to COMPLETED !

Finally if some actions must trigger an action *(i.e. RUN* => start()*)*:

1. Override applyPhaseParam(),
2. Call the super method,
3. Fetch the PHASE parameter from the additionalParameters attribute,
4. And in function of its value execute the appropriate action.

# 3.b. Execution WITH queue

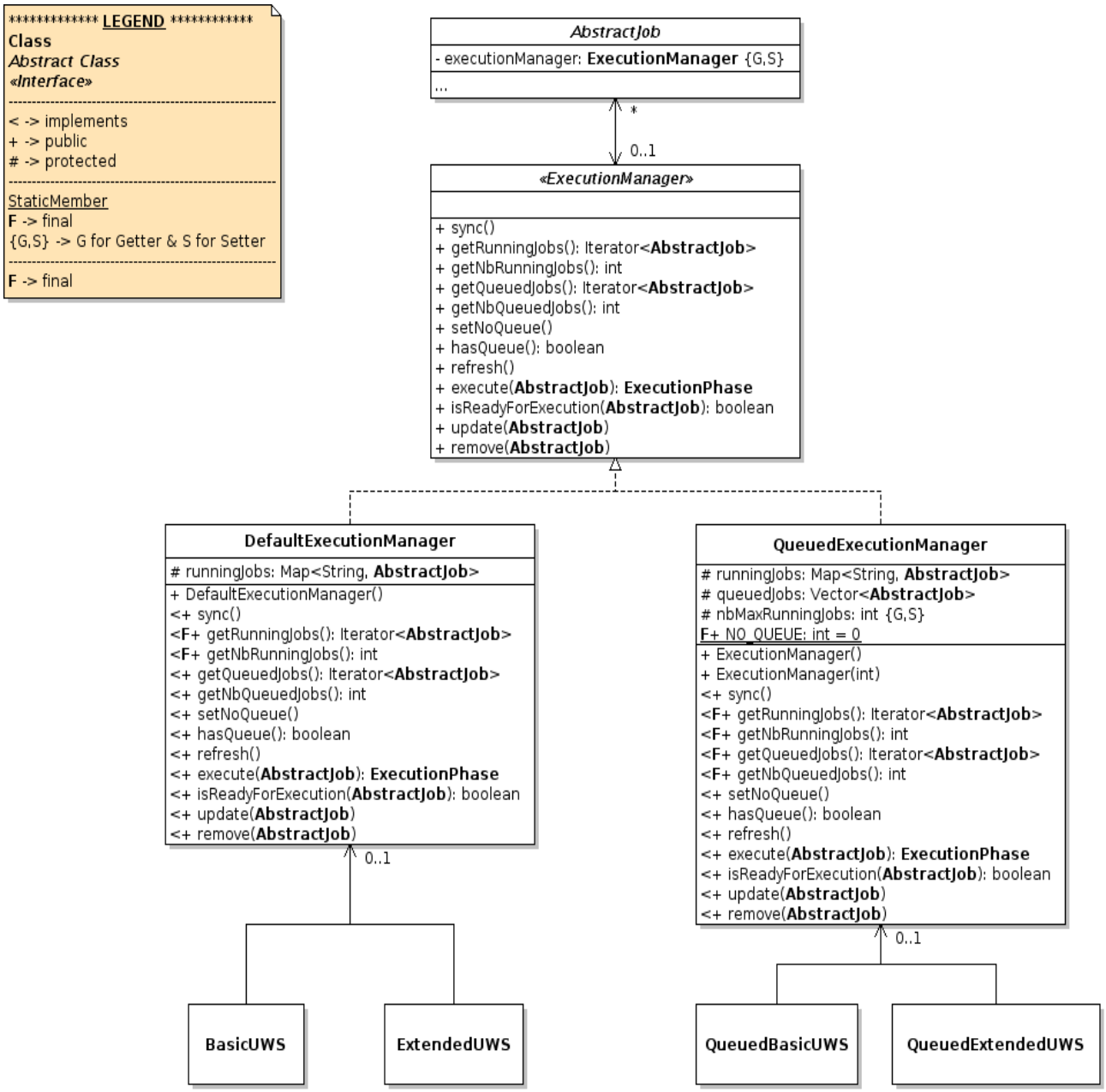
## How does it work ?

You have surely noticed that there are two functions to start a job: one with no parameter and another with a boolean parameter. start() always calls the second start method with *true* if there is an execution manager, *false* otherwise.

An execution manager is an object which implements the ExecutionManager interface. Its goal is to gather all executing jobs. There is one execution manager per UWS and it is propagated to the managed jobs.

There are already two default implementations of this interface: DefaultExecutionManager and QueuedExecutionManager. The first one is the most simple implementation. It has no queue and so it manages only running jobs. On the contrary QueuedExecutionManager is able to queue any given job if the number of running jobs exceeds a given number. When a running job ends, the first queued job is removed from the queue and then executed.

Here is a simplified class diagram for these classes:



Now, lets supposing your UWS has an execution manager which can manage a queue *(*ExecutionManager.hasQueue() *returns true)*. When you ask to start the job thanks to start(), start(boolean) is actually called with *true* as parameter. Then execute(AbstractJob) of the execution manager is called. This function has to execute or to queue the job according to the value returned by its function isReadyForExecution(AbstractJob): if *true*, the job is executed, otherwise the job is queued. Finally to start a job, the execution manager calls start(boolean) with *false*. In this case the starting is exactly the same than with no queue *(see D.3.a. Execution WITHOUT queue)*.

***Notes:***

- *Even if the set execution manager has no queue: start() = start(true). In that way it is possible to have a list of all running jobs through the used execution manager. It may be very usefull for a UWS administrator !*
- *Only if there is really no execution manager: start() = start(false) !*
- *The methods update(AbstractJob) and remove(AbstractJob) are used only when a job is added or removed to/from a UWS or when a job ends. Both update the list of running jobs and the list of queued jobs (if any). See their javadoc for more details.*

### How to customize ?

QueuedExecutionManager proposes one possible management of an execution queue. If your needs are different you can either extend QueuedExecutionManager or implement the interface ExecutionManager. Once done you just have to use the function setExecutionManager(ExecutionManager) on your UWS or on one or several given jobs *(if you want to manage them independently from a UWS)*.

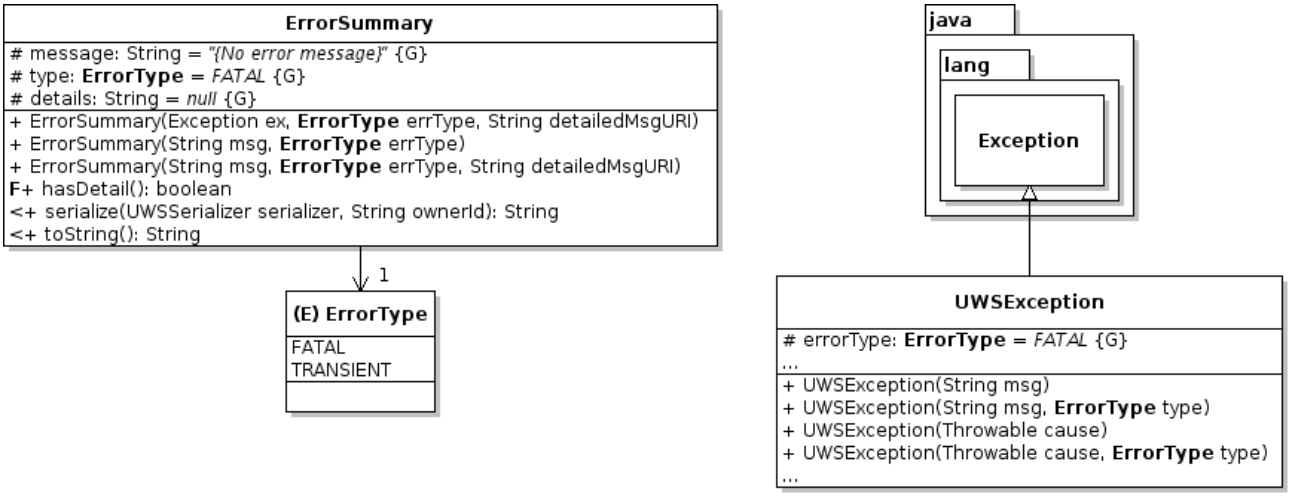
When changing the execution manager of a UWS, the manager is automatically set to all managed jobs !

# 4. Error summary

### How does it work ?

An error summary is published when any error occurs during the job execution. A full error summary is composed of three pieces of information: a brief message, an error type *(transient or fatal)* and a URI/URL of a detailed message. In all cases if a job has ended with an error, the type of the error and a boolean which indicates whether there is a detailed message must always be given. Although an error is intended to always have a detailed message, the "brief message" is optional: it is only a sum up of the full message (or its title).

Now, in this library an error summary is published when any exception - except InterruptedException which stops the job with the phase ABORTED - occurs during the execution of a job, that is to say during the call of jobWork(). The publication is done by the function error(UWSException). When an exception is thrown in jobWork(), it is caught and encapsulated in a UWSException (if needed), which is then given as parameter of the function error(UWSException).




By default the function error(UWSException) only calls the function UWSToolBox.publishErrorSummary(AbstractJob, String msg, ErrorType):

```
public abstract class AbstractJob extends SerializableUWSObject {
...
    protected synchronized boolean publishExecutionError(UWSException ue) throws
UWSException {
        boolean published = UWSToolBox.publishErrorSummary(this, (ue.getCause() !=
null)?ue.getCause().getMessage():ue.getMessage(), ue.getUWSErrorType());
        if (!published)
            throw new UWSException("[Set an error] Impossible to set the given UWS
exception to the job "+jobId+" !");
    }
...
}
```

UWSToolBox.publishErrorSummary(AbstractJob, String msg, ErrorType) builds an error summary and sets the job phase to ERROR. Because it is impossible, in a generic manner, to know where to write a file with the detailed message of an error, by default only the "brief message" and the error type are set: there is no detailed message.

## How to customize ?

To change the default error publication, you must override the function error(UWSException).

For instance to write a detailed message you can call the other function of UWSToolBox: publishErrorSummary(AbstractJob, Exception, ErrorType, URL errorFileUrl, String errorsDirectory, String errorFileName) In this case, the "brief message" and the error type will be filled in the same way than with the other function, but in addition the stack trace of the given exception will be written in the specified file *(in errorsDirectory/errorFileName)*. The URL *errorFileUrl* is used to indicate a public access to read the detailed message. It will be used to make a redirection when the user types the URI */uws/jobList/job/error*.

**IMPORTANT**

**You must take care to check the current execution phase of the job before doing anything ! Indeed the error summary can be set only if the job has not been already stopped before: the publication of an error summary MUST set its phase to ERROR.**

# 5. Destruction

## How does it work ?

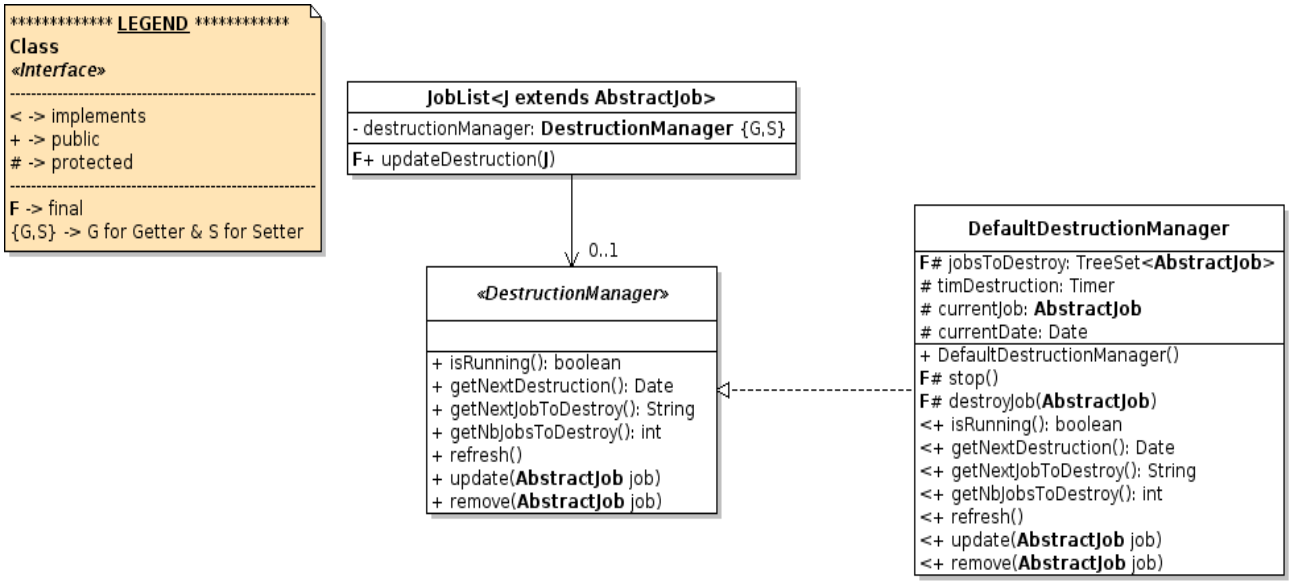
Destroying a job means that nothing in relation with the job must live or stay any more. In other words the job is aborted if running, all results files are destroyed and the job is removed from its jobs list. All these actions are done by the method clearResources(). By default, it is called by the jobs list when JobList.destroyJob(String) is called or when the whole jobs list is destroyed.

## Automatic destruction

A job has by default an attribute *destruction* which corresponds to the date/time at which the job must be destroyed. That means that when the date/time is reached the job must be destroyed by its jobs lists. In this library the responsibility to destroy automatically jobs is given to an object which implements the interface DestructionManager. By default, a jobs list has one attribute of this type.

When a job list is added to a UWS its destruction manager is replaced by the one of the UWS. Consequently all jobs list of a UWS have the same destruction manager.

Below is a simplified class diagram of this interface:




DefaultDestructionManager is a default implementation of the interface DestructionManager. It contains a list of all known jobs whose the destruction date is set. This list is sorted by ascending destruction time. This implementation has also a timer which has to destroy the first job of the list *(so the job with the earliest destruction date/time)*.

When a job is added into a jobs list, the destruction manager is updated by calling DestructionManager.update(AbstractJob). Then, since the job knows its jobs list, it will notify it at each modification of its destruction field, by calling JobList.updateDestruction(AbstractJob).

## How to customize ?

- To add the destruction of results files or to do any other action which must executed while destroying the job, you must override the clearResources() method. This has been done for the first UWS example Timers whose the source code is displayed below:

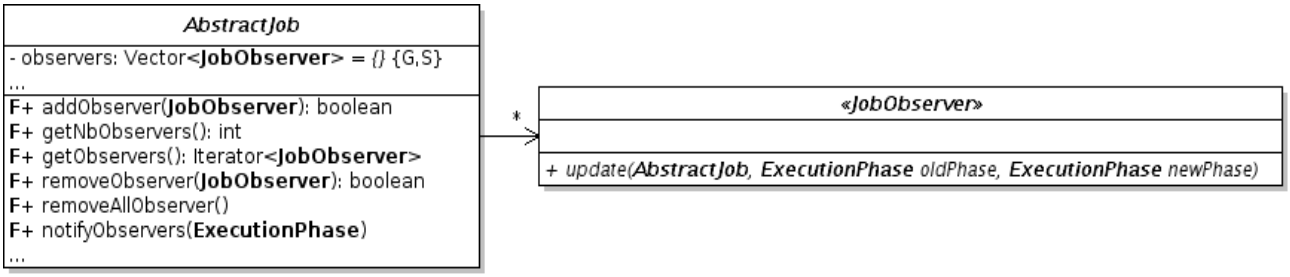
```
public class JobChrono extends AbstractJob {
    ...
    @Override
    public void clearResources() {
        // 1. STOP THE JOB (if running):
        super.clearResources();

        // 2. DELETE THE RESULT FILE (if any):
        try {
            File f = new File(resultsDir+"JobChrono_n"+getJobId()+"_result.txt");
            if (f.exists() && f.canWrite())
                f.delete();
        } catch (Exception e) {
            System.err.println("### UWS ERROR: "+e.getMessage()+" ###");
            e.printStackTrace();
        }
    }
    ...
}
```

- As for the job execution, the job automatic destruction can be modified by implementing the DestructionManager or by extending its default implementation (DefaultDestructionManager). Then you just have to set it to your UWS (setDestructionManager(DestructionManager)) or to a jobs list (setDestructionManager(DestructionManager).
  When changing the destruction manager on a UWS, it is also set to all its managed jobs lists !

# 6. Keeping an eye on jobs

Any object is able to keep an eye on jobs, but more precisely on its execution phase ! Indeed each time the execution phase of a job changes, this job notifies all the interested objects. These objects must:

1. implement the interface JobObserver
2. and subscribe as observer of the jobs to observe, thanks to the method addObserver(JobObserver).



# E. How to customize a UWS ?

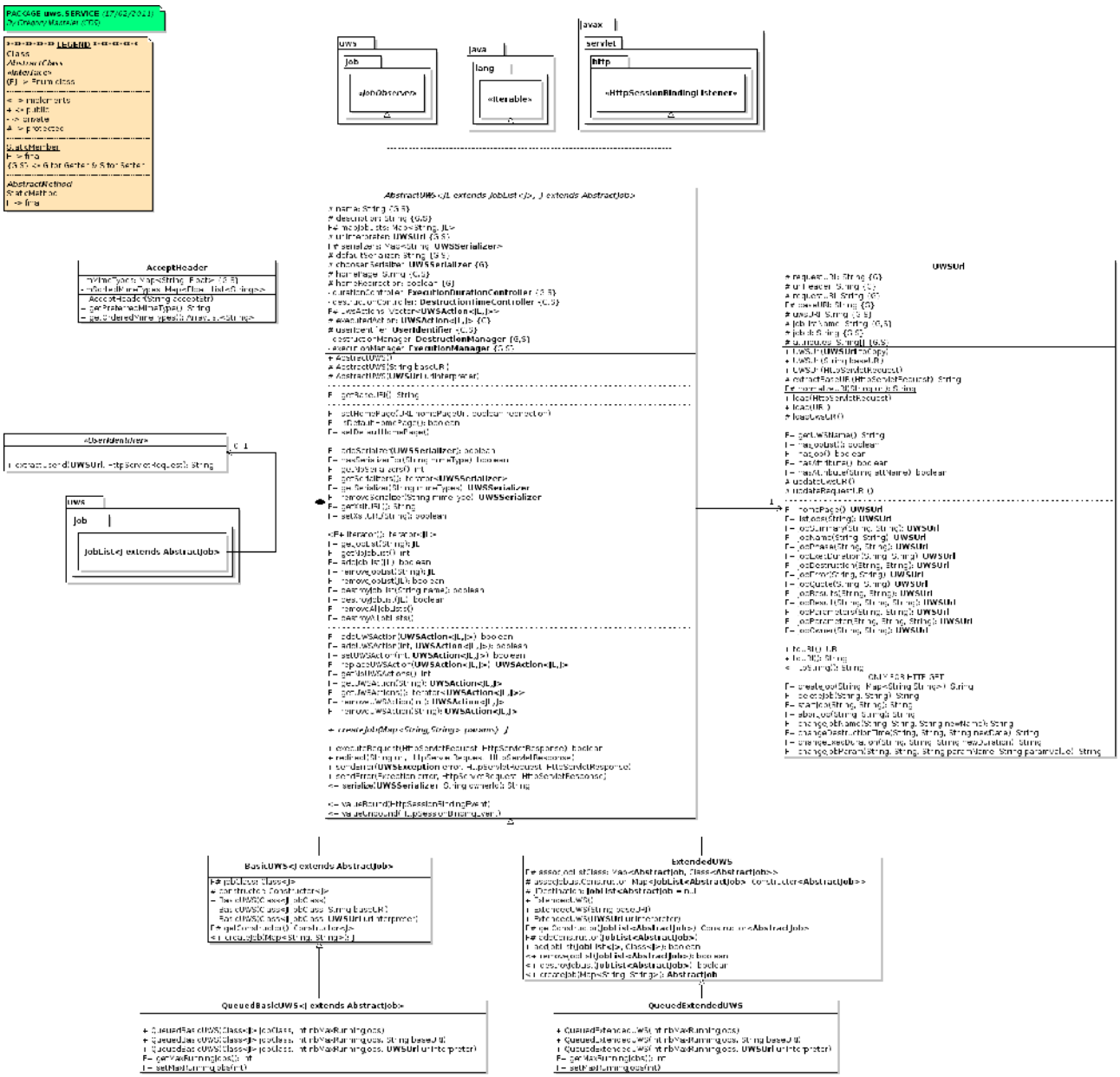
In the part C.2. Writing the servlet you have learnt to create a UWS. Then in the section D. How to customize a job ?, you have seen that you can change the global destruction manager and the global execution manager.

However by extending AbstractUWS (or one of its subclasses) more points of a UWS can be customized. For instance you can put more controls on the fields *destruction* and *executionDuration* by setting a default and a maximum value. Besides you will see that a UWS can be viewed as a set of actions *(listing jobs lists, creating jobs, starting jobs, ...)*. It is also to its responsibility to catch and to manage request errors and to return UWS resources in the asked format *(i.e. XML, JSON, ...)*.

In this part of the tutorial these and other features of a UWS will be explained so that you can customize them as you want:

1. Name, description and home page
2. UWS administration
3. User identification
4. Request interpretation
5. UWS URL interpretation
6. Actions
7. Serialization
8. Redirection and errors
9. The interface HttpSessionBindingEvent

Since this list is not exhaustive at all, you will need the following class diagram to customize other points of your UWS:



# 1. Name, description and home page

## UWS resource

In the IVOA Recommendation, jobs list and jobs are described as web resources. Thus there is an XML schema for both of them. About the UWS itself nothing exist: no URI, no XML schema and no actions.

However in this library the UWS can be viewed also as a web-resource. This resource corresponds to the URI */{uws}* which is the base of all the other URIs: */{uws}/{job-list1}, /{uws}/{job-list2}, /{uws}/{job-list1}/{job1}*, .... Like a jobs list and a job, this resource returns by default a XML document whose the format is:

## UWS Name & Description

The UWS name is by default the corresponding part of the URI. For instance: the URI of the job list

*timers* of the first UWS example is */basic/timers*, so the default UWS name is *basic*. This name can be changed thanks to the method setName(String).

Changing the UWS name does not change the URI ! If the URI of a UWS is /basic and if you have changed its name into *My first UWS*, the URI will still be /basic !

By default a UWS has no description. But if you want, you can set one by using the method setDescription(String).

Both methods have been used for the second UWS example of this tutorial named Algorithms. Here is the corresponding source code:

```
public class MyExtendedUWS extends QueuedExtendedUWS {
    ...
    public MyExtendedUWS(int nbMaxRunningJobs) throws UWSException {
        super(nbMaxRunningJobs);

        // Set name:
        setName("Algorithms");

        // Set description:
        setDescription("This UWS aims to manage several JobLists. Each one manages one
different type of Job. Jobs are \"famous\" algorithms which may take a long time in
function of their parameters.");
        ...
    }
    ...
}
```

## Home Page

As said previously the home page of a UWS is its corresponding resource *(by default a XML document)*. However, you can replace this resource by another one with the method setHomePage(String) or setHomePage(URL, boolean). With the second method you can specify the URL of the replacement resource and whether a redirection to this resource must be done. If *false*, the full content of the specified resource will be copied when the home page of the UWS will be asked. In another hand, the first method always does a redirection to the specified resource *(which can be either a URL or a URI)*.

In the second UWS example of this tutorial, the home page of the UWS has been changed in the following way:

```
public class UWSAlgorithms extends HttpServlet {
    ...
    @Override
    public void init(ServletConfig config) throws ServletException {
        ...
        // Create our UWS (with a user identification):
        uws = new MyExtendedUWS(3);
        ...
        // Set the UWS home page:
        uws.setHomePage(contextPath+"/extended.html");
        ...
    }
    ...
}
```

To go back to the default resource, use the method setDefaultHomePage().

## Add actions

Provided this resource has not been described in the IVOA Recommendation, there is no actions for this

resource contrary to the jobs list and job resources. Nevertheless with this library it is possible: for more details see the part E.6. Actions.

# 2. UWS administration

## Jobs lists management

As a jobs list is a container of jobs, a UWS is mainly a container of jobs lists. Thus to add a jobs list you must use the method addJobList(JobList).

**Warning !**

A job list can be added in a UWS only if there is not already another job list with the same name. Indeed the name of a job list is used to build its URI. If two jobs list have the same name, there would be no way to determine the corresponding jobs list from the same URI.

In AbstractUWS two kinds of method can be used to remove jobs lists: removeJobList(String), removeJobList(JobList) and removeAllJobLists() ONLY remove the specified jobs list(s) from the UWS whereas destroyJobList(String), destroyJobList(JL) and destroyAllJobLists() also destroy its (their) job(s).

Remember that destroying a job means stopping the job if running and destroying all its resources (thread, files, ...) *(see D.5. Destruction for more details)*.

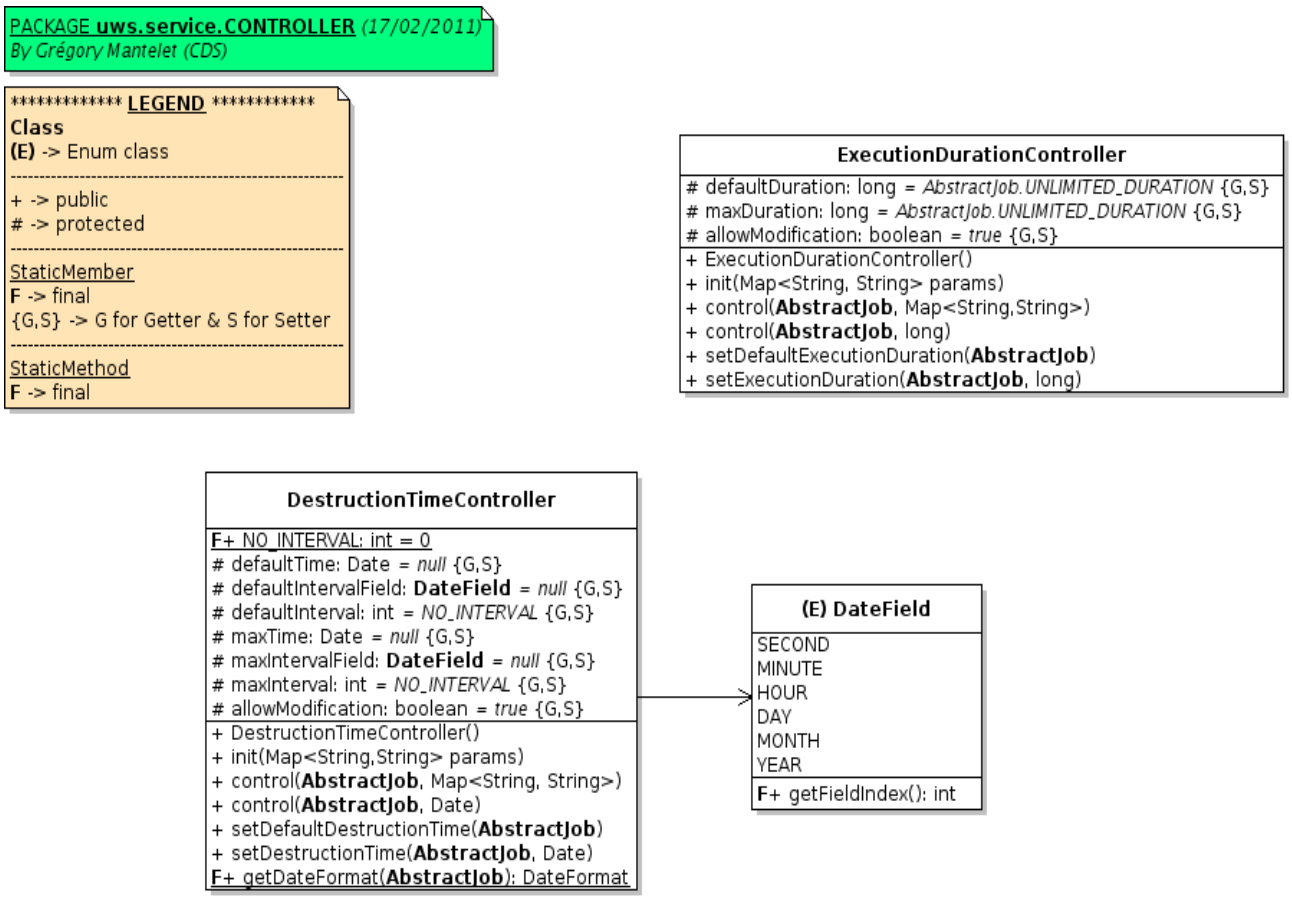
And finally, jobs lists can also be retrieved either by their name *(hence the name unicity)*, with the function getJobList(String), or thanks to an iterator with the function iterator().

## Controllers

Since a web-service may be used by many clients *(human or software)*, all the occupied resources may increase in an unpredictable manner. This, is particularly true for a UWS whose the clients may create more than one job. Indeed, each job is associated to one thread for its execution and to two managers - one for its execution and the other for its destruction - which must be notified of any modification of all their managed jobs. Besides the execution itself may need to generate some additional threads. And obviously one job may write one or several results files whose the size is likelly unpredictable *(it depends of the type of job)*.

However the computing and the memory resources can be more controlled thanks to the attributes *executionDuration* and *destruction* of all managed jobs. Indeed, provided that all managed jobs are created, updated and destroyed by requests interpreted in first by the UWS, it is possible for a UWS to allow or to forbid some ranges of value for these job attributes.

For that a UWS is associated to two objects which let it setting a default and a maximum value for these attributes. The first one is an instance of ExecutionDurationController and the second one of DestructionTimeController. Here is a UML class diagram of these classes:




Both classes have two common methods: *init(Map<String, String>)* and *control(AbstractJob, Map<String,String>)*. The first method is used to initialize the destruction or the execution duration attribute at the creation of a new job. The map parameter contains all the parameters with which the new job must be initialized. And the second method checks the destruction or the execution duration attribute given in the map parameter when a request has for goal to set the specified attribute to the given job.

***Note:***

*Even if these controllers have common methods, there is no common interface because there are only two job attributes for which there is a real reason to be controlled by a UWS (additional parameters are not considered here because they are specific to a given type of job, and so there is no reason that they can be controlled globaly by a UWS). However if it becomes needed to have a common interface and an extendable list of controllers in a UWS, please send me a mail so that changing the library in this way.*

For both of these attributes there is a default value *(used only at the job creation)* and a maximum value. They have been both set in the second UWS example - Algorithms:

```
public class UWSAlgorithms extends HttpServlet {
    ...
    @Override
    public void init(ServletConfig config) throws ServletException {
        ...
    // Set the destruction time for all jobs:
        DestructionTimeController destController = uws.getDestructionTimeController();
        // = job destroyed 1 month after its creation
        destController.setDefaultDestructionInterval(1, DateField.MONTH);
        // = no modification
        destController.allowModification(false);

    // Set the execution time for all jobs:
        ExecutionDurationController execController =
```

```
uws.getExecutionDurationController();
        // = job execution limited to 3600s (1 hour)
        execController.setDefaultExecutionDuration(3600);
        // = no modification
        execController.allowModification(false);
        ...
    }
    ...
}
```

### Information about a UWS

The following functions give several information about the current status of the UWS which can be usefull for the administrator:

- **AbstractUWS**:
  - getName()
  - getDescription()
  - getBaseURI(): gets the base URI (*that's to say the URI of the UWS*)
  - getExecutedAction()**:** gets the last executed action *(see E.6. Actions)*
  - getChoosenSerializer()**:** gets the serialize used by the last executed action *(see E.7 Serialization)*
- **ExecutionManager**:
  - getNbRunningJobs()
  - getRunningJobs()
  - getNbQueuedJobs()
  - getQueuedJobs()
- **QueuedExecutionManager**:
  - getNbMaxRunningJobs(): gets the maximum number that can run in the same time. The other jobs are put in the queue.
- **DestructionManager**:
  - getNextDestruction(): gets the date of the next automatic job destruction
  - getNextJobToDestroy()
  - getNbJobsToDestroy()
- **JobList**:
  - getNbJobs(): gets the number of all the jobs
  - getNbJobs(String): gets the total number that the specified owner has in this jobs list
  - getNbUsers(): gets the number of users that have at least one job in this jobs list
  - getUsers(): gets the list of users that have at least one job in this jobs list
- **AbstractJob**:
  - getPhase(): gets the current phase of the job

All these information can be used to display an information page about the UWS, as it has been done in the second UWS example Algorithms *(see E.6. Actions for more details)*. We could also imagine an administrator page where some points of the UWS can be dynamically set *(for instance: set the number of maximum running jobs, set the default value for the execution duration or for the destruction time, ....)*.

# 3. User identification

Without a user identification all jobs are visible by anyone. So everybody is able to modify a job *(start/stop/delete it, change its parameters, ...)* as he wants. However with this library you can specify a way

to identify a user in function of the received request.

### How does it work ?

Before interpreting a request the UWS tries to identify the user thanks to an object which implements the interface UserIdentifier. In this interface there is only one function: extractUserId(UWSUrl, HttpServletRequest). This function must return the ID of the user identified from the given UWS URL and request.

***Notes:***

- *By default, a UWS does not identify the user (there is no default identifier). Consequently all the jobs are visible and modifiable by everybody.*
- *With the function JobList.getJobs(String) you can retrieve only the jobs of the specified user. By default, it is done by all requests which have to manage jobs lists.*

### How to customize ?

You have to implement the interface UserIdentifier and to set the resulting class to your UWS thanks to the method setUserIdentifier(UserIdentifier). Here is the way a user is identified in the second UWS example - Algor ithms:

```
public class MyExtendedUWS extends QueuedExtendedUWS {
    ...
    public MyExtendedUWS(int nbMaxRunningJobs) throws UWSException {
        ...
        // Add a user identification (by IP address):
        setUserIdentifier(new UserIdentifier() {
            private static final long serialVersionUID = 1L;

            @Override
            public String extractUserId(UWSUrl urlInterpreter, HttpServletRequest
request) throws UWSException {
                return request.getRemoteAddr();
            }
        });
    }
    ...
}
```

***Note:***

*In this example, the user is actually a machine because the user identification is based on an IP address. Obviously it is not the best way to identify a user, because he can not access its jobs from another machine. Besides if there are several users on the same machine any of them can access to the same jobs. However this type of user identification is widely enough for this tutorial !*

# 4. Request interpretation

In C.2. Writing the servlet you have seen that at each request, you have only one thing to do: just call the method executeRequest(HttpServletRequest, HttpServletResponse). This method is able to interpret any HTTP request and to execute the corresponding UWS action.

### How does it work ?

When a request is sent to the servlet, it is forwarded to the UWS thanks to the method executeRequest(HttpServletRequest, HttpServletResponse). To interpret the received request, the UWS does

three things:

1. Interpret the URL,
2. Identify the user,
3. Look for the corresponding UWS action.

To interpret the URL the method load(HttpServletRequest) of the object UWSUrl is called. This class is explained in more details in the next page *(E.5. The UWS URL interpretation)*, but what you have to known is it lets simplifying the given request so that extracting the UWS URI (*/{jobListName}/{jobName}/{jobAttributes}...*). This simplification will help to determine which UWS action must be executed.

The identification of the user may be important for some UWS actions like displaying a jobs list. It is done by the function extractUserId(UWSUrl, HttpServletRequest) of UserIdentifier, explained previously in E.3. User Identification.

Finally to determine the corresponding UWS action, we loop on the list of all the available actions. Indeed a UWS can also be viewed as a set of actions. Each action is represented by an extension of the abstract class UWSAction. Only two functions of this class interest us here (but if you want more information, see E.6. Actions):

- match(UWSUrl urlInterpreter, String userId, HttpServletRequest request)
- and apply(UWSUrl urlInterpreter, String userId, HttpServletRequest request, HttpServletResponse response).

The first function is used in the loop to choose the UWS action corresponding to the received request, whereas the second one is called on the found action. If no matching action can be found, a UWSException is thrown.

executeRequest(HttpServletRequest, HttpServletResponse) returns a boolean: *true* if the found action has been successfully executed, *false* otherwise.

### How to customize ?

All steps of the request interpretation have been designed to be customizable independently. So theoretically you have to override the method or to extend the class which correspond to what you want to customize. For that you need to look at the corresponding part of this tutorial:

- E.5. UWS URL interpretation
- E.3. User identification
- E.6. Actions

Only if you have to add some operations to the current request interpretation or if you want completely replace it, you would have to override executeRequest(HttpServletRequest, HttpServletResponse).

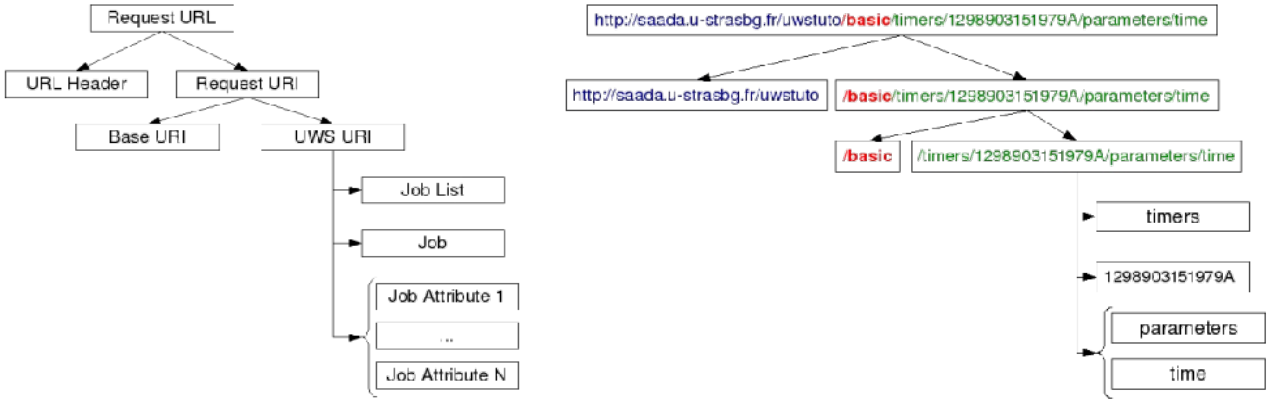
# 5. UWS URL Interpretation

In this library there is one very usefull and important class: UWSUrl. Indeed it is used each time a URL of a UWS must be interpreted and/or generated. So you should be very carefull if you want to customize it ! So that helping you in the customization, you will find below some explanations about the way a UWS URL is represented by this kind of object.

### URL splitting

As said in the IVOA Recommendation all UWS URIs must be built in a hierarchical manner, according to REST. UWSUrl splits and generates URLs in the same way. Below are two schemas: the **left schema** describes the way UWS URLs are split by UWSUrl ; the **right schema** shows the splitting of a UWS URL

example:




The base URI *(in bold and red in the example)* is used as separator between the URL header and the UWS URI. It is the key of the URL splitting in this class. It is required for any URL interpretation or generation !

Thus you can create a UWSUrl directly with the base URI *(or a URL)* or by copying another UWSUrl. Besides it can also be extracted automatically from a HttpServletRequest object thanks to the function getServletPath(). This path is set in the *web.xml* file of your servlet:

By default, the UWS URL interpreter of an AbstractUWS is initialized with the first received request. No interpreter exists before that ! However you can set one at any moment with the method setUrlInterpreter(UWSUrl).

## URL interpretation

Once the base URI is known, you can use load(URL) or load(HttpServletRequest) to load and to interpret respectively a URL or a request. The two functions return always the same result except for the URL header which may be more complete with a request.

***Note:***

*In load(HttpServletRequest), the base URI is always extracted from the given request and is then compared to the one stored in this class. If they are different nothing is done except calling load(URL).*

Each part of the URL can be retrieved individually thanks to its corresponding getter function (*i.e. getUwsURI(), getJobListName(), getJobId(), ...*). Besides you can also know if some parts are valued or not: hasJobList() tells whether the UWS URL indicates at least a job list name, hasJob() tells the same thing but about a job ID, ...

getUWSName() returns the presumed name of the UWS. This "name" is the last item of the base URI. However it may not be the real name of the UWS which can be set at any moment with AbstractUWS.setName(String). Actually getUWSName() is only used by AbstractUWS.getName() to returns a default value.

## URL generation

UWS URLs can be generated in two ways: by modifying the current instance of a UWSUrl or by using it as a base for a new UWSUrl.

### a. With modification

A UWSUrl object can be updated thanks to its setter methods. But contrary to the getters, setters exist only for the UWS URI part *(part of the request URL which starts just after the base URI)*.

**Lets take an example !** Supposing we have created a UWSUrl object named *uwsUrl* with */basic* as base URI. It has been initialized with a UWS URL which lets getting the additional parameter *time* of the job 1298904240779A. Then only the job ID of the UWS URL is modified thanks to setJobId(String). Here is the corresponding code:

```
UWSUrl uwsUrl = new UWSUrl("/basic");
uwsUrl.load(new URL("http://saada.u-
strasbg.fr/uwstuto/basic/timers/1298904240779A/parameters/time"));

System.out.println("BEFORE MODIFICATION:");
UWSToolBox.printURL(uwsUrl);

uwsUrl.setJobId("1298971587132A");

System.out.println("AFTER MODIFICATION:");
UWSToolBox.printURL(uwsUrl);
```

And the result is:

```
BEFORE MODIFICATION:
***** UWS_URL (/basic) *****
Request URL: http://saada.u-
strasbg.fr/uwstuto/basic/timers/1298904240779A/parameters/time
Request URI: /basic/timers/1298904240779A/parameters/time
UWS URI: /timers/1298904240779A/parameters/time
Job List: timers
Job ID: 1298904240779A
Attributes (2): parameters time

AFTER MODIFICATION:
***** UWS_URL (/basic) *****
Request URL: http://saada.u-
strasbg.fr/uwstuto/basic/timers/1298971587132A/parameters/time
Request URI: /basic/timers/1298971587132A/parameters/time
UWS URI: /timers/1298971587132A/parameters/time
Job List: timers
Job ID: 1298971587132A
Attributes (2): parameters time
```

As you can notice only the job ID has been changed. Indeed when using a such function all the other parts of the main request URL are automatically updated.

***Notes:***

- *As shown in the above example, you can use the function UWSToolBox.printURL(UWSUrl) to display the content of a UWSUrl.*
- *Rather than setting individually each part of a UWS URI you can set the whole URI thanks to the method setUwsURI(String) !*

### b. Without modification

To avoid modifying a UWSUrl object you have three solutions:

1. **To create a new UWSUrl()** and to initialize it with a base URI and any request URL you want. *(**warning:** the base URI must be contained in the URL !)*
2. **To make a copy of an existing UWSUrl() object** *(thanks to the constructor UWSUrl(UWSUrl))* and to use the methods described above, directly on the copy.
3. **To use one of the following methods** which return a modified copy of the current UWSUrl() instance:
    - homePage()

- listJobs(String jobListName)
- jobSummary(String jobListName, String jobId)
- jobName(String jobListName, String jobId)
- jobPhase(String jobListName, String jobId)
- jobExecDuration(String jobListName, String jobId)
- jobDestruction(String jobListName, String jobId)
- jobError(String jobListName, String jobId)
- jobQuote(String jobListName, String jobId)
- jobResults(String jobListName, String jobId)
- jobResult(String jobListName, String jobId, String resultId)
- jobParameters(String jobListName, String jobId)
- jobParameters(String jobListName, String jobId, String paramName)
- jobOwner(String jobListName, String jobId)

**Warning:**

The following methods lets generating a URL corresponding to an action on a JobList or a Job:

- createJob(String jobList, Map params)
- deleteJob(String jobList, String jobId)
- startJob(String jobList, String jobId)
- abortJob(String jobList, String jobId)
- changeJobName(String jobList, String jobId, String newName)
- changeDestructionTime(String jobList, String jobId, String newDate)
- changeExecDuration(String jobList, String jobId, String newDuration)
- changeJobParam(String jobList, String jobId, String paramName, String paramValue)

Since these actions needs to add some HTTP parameters, all these methods return a String which corresponds to the final request URL.

Besides they are designed for a **HTTP-GET** request rather than HTTP-POST. But you can use these functions and then call UWSToolBox.getParameters(String queryPart) to extract the HTTP-GET parameters into a Map. Then you can use the URL and the parameters map to send a HTTP-POST request through a servlet.

# 6. Actions

At each sent request, the UWS identifies the corresponding action (i.e. listing jobs, starting, creating or displaying a job, ...) and executes it. This library allows you to add and/or remove actions to/from your UWS. So in this part of the tutorial you will learn how to define your own action(s) and how to add it (them) to your UWS.

## The class UWSAction

So that offering more flexibility, a UWS contains an extendable set of actions. Each one is identified by a unique name and is able to indicate whether it can be applied considering a given HTTP request. So when receiving a request, a UWS loops on all its available actions and chooses the one which matches. The match condition and the action execution are both defined in one class which must extend UWSAction.

To sum up, an extension of UWSAction must define two functions:

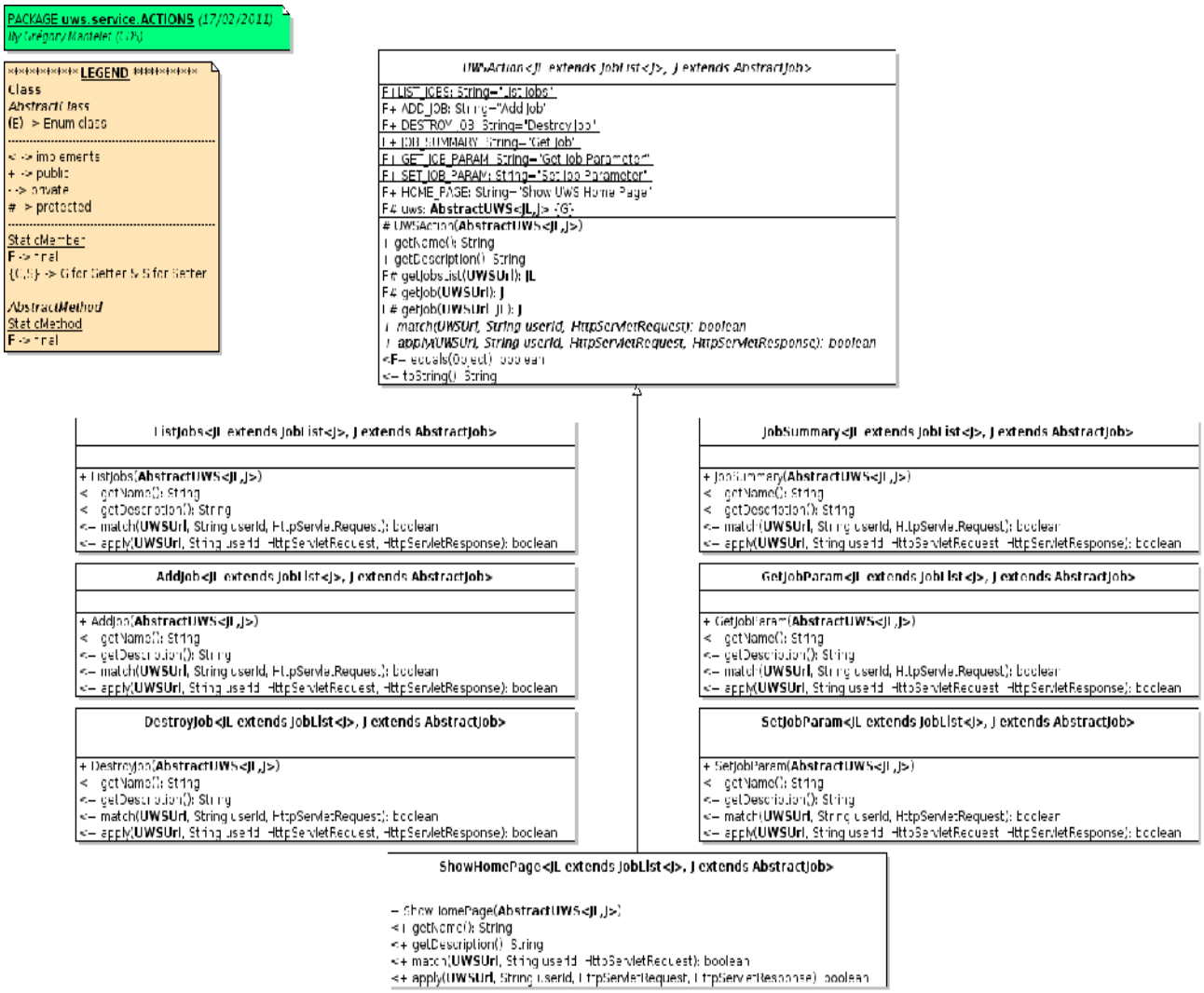
- **match(UWSUrl, HttpServletResquest):** *Indicates whether the given request (with the given UWS URL) corresponds to this action.*
- **apply(UWSUrl, HttpServletRequest, HttpServletResponse):** *Does what the action is supposed to do.*

Obviously all UWS actions described by the IVOA Recommendation are already implemented. Here is their corresponding class:

- Listing jobs: ListJobs
- Adding a job: AddJob
- Destroying a job: DestroyJob
- Getting a job summary: JobSummary
- Getting the value of a job parameter: GetJobParam
- Adding/Updating the value of a job parameter: SetJobParam

ShowHomePage is also a sub-class of UWSAction, but is not an action described by the IVOA Recommendation. This action corresponds to the URI: */{uws}* (the base UWS URL actually).

Below is the class diagram of all existing actions:




## How does it work ?

AbstractUWS is an ordered set of UWSActions. Its attribute uwsActions groups together all its possible actions. When a user sends a request to the servlet, the function AbstractUWS.executeRequest(...) is called. This one will choose the action corresponding to the received HTTP request by iterating in uwsActions:

```
public abstract class AbstractUWS<JL extends JobList<J>, J extends AbstractJob> {
...
    public boolean executeRequest(HttpServletRequest request, HttpServletResponse
```

```
response) throws UWSException, IOException {
    ...
        boolean actionFound = false;
        for(int i=0; !actionFound && i<uwsActions.size(); i++){
            if (uwsActions.get(i).match(urlInterpreter, userId, request)){
                actionFound = true;
                actionApplied = uwsActions.get(i).apply(urlInterpreter, userId, request,
response);
            }
        }

        if (!actionFound)
            throw new UWSException(UWSException.NOT_IMPLEMENTED, "[Execute UWS request]
This UWS action is not supported by this UWS service !");
    ...
    }
...
}
```

As you can notice, the actions of a UWS are evalutated in the order with their function UWSAction.match(UWSUrl, String, HttpServletRequest). If it returns *true* the method apply(UWSUrl, String, HttpServletRequest, HttpServletResponse) is called to execute the action. If *false* we try with the next action, and so on.

## How to customize ?

UWSAction is an abstract class: the functions match(UWSUrl, String, HttpServletRequest) and apply(UWSUrl, String, HttpServletRequest, HttpServletResponse) must be overrided. Besides the attribute uwsActions of AbstractUWS is a vector and several getters and setters let managing easily the list of actions available for a given UWS. Thus to customize this part of your UWS you have to:

1. Extend UWSAction (or one of its sub-classes)
2. Update your UWS

To illustrate a such customization lets see how the action *AboutAction* has been added to UWSAlgorithms. This additional action has to display some information about UWSAlgorithms *(name, description, number of jobs lists, ...)* but also some statistics about its jobs.

### 1. Extend UWSAction

Any type of UWS action must have a name, a kind of action ID. So **this name must be unique for each type of action**, that is to say for each extension of UWSAction. The name does not have to change at each instantiation, it is used to distinguish the different type of action in a given UWS ! It is returned by the function getName(), which is not abstract, but returns by default the absolute name of the java class.

The name of the default UWS actions are stored as final class variable in UWSAction:

- LIST_JOBS="List Jobs" for the class ListJobs
- ADD_JOB="Add Job" for the class AddJob
- DESTROY_JOB="Destroy Job" for the class DestroyJob
- JOB_SUMMARY="Get Job" for the class JobSummary
- GET_JOB_PARAM="Get Job Param" for the class GetJobParam
- SET_JOB_PARAM="Set Job Param" for the class SetJobParam
- HOME_PAGE="Show UWS Home Page" for the class ShowHomePage

The name of our additional action - *AboutAction* - will be:

```
public class AboutAction<JL extends JobList<J>, J extends AbstractJob> extends
UWSAction<JL, J> {
```

```
    public AboutAction(AbstractUWS<JL, J> u) {
        super(u);
    }

    @Override
    public String getName() {
        return "About UWS";
    }
...
}
```

After the name of the action you have to define at which condition it must be applied: the function UWSAction.match(UWSUrl, String, HttpServletRequest). You must be very careful when overriding this function ! Indeed **the tested condition has to be as precise as possible** so that avoiding to forget this action or to trigger this action rather than another one. For instance, the difference between ListJobs and AddJob is thin: the HTTP method *(GET for ListJobs and POST for AddJob)*. To avoid the confusion the HTTP method must absolutely be tested.

Here is their match(...) function:

```
public class AddJob<JL extends JobList<J>, J extends AbstractJob> extends UWSAction<JL,
J> {
    public boolean match(UWSUrl urlInterpreter, String userId, HttpServletRequest
request) throws UWSException {
        return (urlInterpreter.getJobListName() != null    // jobs list specified
        && urlInterpreter.getJobId() == null                // no job specified
        && request.getMethod().equalsIgnoreCase("post"));    // HTTP-POST
    }
...
}

public class ListJobs<JL extends JobList<J>, J extends AbstractJob> extends
UWSAction<JL, J> {
    public boolean match(UWSUrl urlInterpreter, String userId, HttpServletRequest
request) throws UWSException {
        return (urlInterpreter.getJobListName() != null    // jobs list specified
        && urlInterpreter.getJobId() == null                // no job specified
        && request.getMethod().equalsIgnoreCase("get"));    // HTTP-GET
    }
...
}
```

In our example, *AboutAction* will be applied when the URL is the base UWS URL and only when the parameter *ACTION* has the value *ABOUT*. The HTTP method does not matter here, because the parameter is enough to distinguish this action from ShowHomePage (which has no parameter).

```
public class AboutAction<JL extends JobList<J>, J extends AbstractJob> extends
UWSAction<JL, J> {
...
    @Override
    public boolean match(UWSUrl urlInterpreter, String userId, HttpServletRequest
request) throws UWSException {
        return urlInterpreter.getJobListName() == null    // no jobs list specified
        && request.getParameter("ACTION") != null    // at least, 1 parameter = ACTION
        && request.getParameter("ACTION").equalsIgnoreCase("about");    // its value
must be ABOUT (not case sensitive)
    }
...
}
```

And finally: the function apply(UWSUrl, String, HttpServletRequest, HttpServletResponse). *AboutAction* must write a HTML page with some information about the UWS Algorithms:

- name, description and base UWS URL
- status of the execution queue and of the executing jobs list
- available UWS serializers
- available UWS actions (name + description)
- name and content of all the jobs lists

Here is the full source code:

```
public class AboutAction<JL extends JobList<J>, J extends AbstractJob> extends
UWSAction<JL, J> {
...
    @Override
    public boolean apply(UWSUrl urlInterpreter, String userId, HttpServletRequest
request, HttpServletResponse response) throws UWSException, IOException {
        response.setContentType("text/html");

        PrintWriter out = response.getWriter();

        BufferedReader reader = new BufferedReader(new
FileReader(request.getSession().getServletContext().getRealPath("/about_header.txt")));
        try{
            String line = null;
            while((line = reader.readLine()) != null)
                out.println(line);
        }finally { reader.close(); }

        out.println("<h1>About the UWS Algorithms</h1>");

        out.println("<h2>UWS Algorithms</h2>");
        out.println("<ul>");
        out.println("<li><b>Name:</b> "+uws.getName()+"</li>");
        out.println("<li><b>Description:</b><p style=\"margin-bottom: 0; padding-bottom:
0\">"+uws.getDescription()+"</p></li>");
        out.println("<li><b>Base UWS URL:</b> "+uws.getBaseURL()+"</li><br />");
        out.println("<li><b>Nb Queued Jobs:</b> "+uws.getNbQueuedJobs()+"</li>");

        out.print("<li><b>Nb Running Jobs:</b> "+uws.getNbRunningJobs());
        if (uws.getNbRunningJobs() > 0){
            Iterator<J> it = uws.getRunningJobs();
            String runningJobs = null;
            while(it.hasNext())
                runningJobs = ((runningJobs==null)?"":(runningJobs+", "))
+it.next().getJobId();
                out.print(" ("+runningJobs+")");
        }
        out.println("</li>");

        out.println("<li><b>Nb Max Running Jobs:</b> 3</li>");
        out.println("<li><b>Nb Jobs Lists:</b> "+uws.getNbJobList()+"</li>");
        out.println("</ul>");

        out.println("<h2>"+uws.getNbSerializers()+" Available Serializers</h2>");
        out.println("<ul>");
        Iterator<UWSSerializer> itSerializers = uws.getSerializers();
        while(itSerializers.hasNext()){
            UWSSerializer serializer = itSerializers.next();
            out.println("<li><b>"+serializer.getMimeType()+"</b></li>");
        }
        out.println("</ul>");

        out.println("<h2>"+uws.getNbUWSActions()+" Available Actions</h2>");
        out.println("<ul>");
        Iterator<UWSAction<JL,J>> itActions = uws.getUWSActions();
        while(itActions.hasNext()){
            UWSAction<JL,J> action = itActions.next();
```

```
            out.println("<li><b>"+action.getName()+"</b><p style=\"margin-bottom: 0;
padding-bottom: 0\"><i>"+action.getDescription()+"</i></p></li>");
        }
        out.println("</ul>");

        out.println("<h2>Jobs Lists</h2>");
        out.println("<table style=\"text-align: center; width: 100%;\">");
        out.println("<tr><th></th><th>Users</th><th>Jobs</th><th>Pending</th><th>Queued<
/th><th>Running</th><th>Complete</th><th>Error</th><th>Aborted</th><th>Others</th></tr>"
);
        for(JL jl : uws){
            int nbPending=0, nbQueued=0, nbRunning=0, nbComplete=0, nbError=0,
nbAborted=0, nbOthers=0;
            for(J job : jl){
                switch(job.getPhase()){
                    case PENDING: nbPending++; break;
                    case QUEUED: nbQueued++; break;
                    case EXECUTING: nbRunning++; break;
                    case COMPLETED: nbComplete++; break;
                    case ERROR: nbError++; break;
                    case ABORTED: nbAborted++; break;
                    default: nbOthers++; break;
                }
            }
            out.println("<tr><td><b><a href=\"extended/"+jl.getName()+"\">"+jl.getName()
+"</a></b></td><td>"+(jl.getNbUsers()<=0?"-":jl.getNbUsers())+"</td><td>"+
(jl.getNbJobs()<=0?"-":jl.getNbJobs())+"</td><td>"+(nbPending<=0?"-":nbPending)
+"</td><td>"+(nbQueued<=0?"-":nbQueued)+"</td><td>"+(nbRunning<=0?"-":nbRunning)
+"</td><td>"+(nbComplete<=0?"-":nbComplete)+"</td><td>"+(nbError<=0?"-":nbError)
+"</td><td>"+(nbAborted<=0?"-":nbAborted)+"</td><td>"+(nbOthers<=0?"-":nbOthers)
+"</td></tr>");
        }
        out.println("</table>");

        reader = new BufferedReader(new
FileReader(request.getSession().getServletContext().getRealPath("about_footer.txt")));
        try{
            String line = null;
            while((line = reader.readLine()) != null)
                out.println(line);
        }finally { reader.close(); }

        out.close();

        return true;
    }
...
}
```

## 2. Update your UWS

Once finished, your extension of UWSAction can be added/setted in your UWS. For that you have three ways:

- **An addition:** addUWSAction(UWSAction)
- **An insertion:** addUWSAction(int, UWSAction)
- **A replacement:** setUWSAction(int, UWSAction) or replaceUWSAction(UWSAction)

All these methods work ONLY IF no action with the same name already exists in the UWS ! The only exception is replaceUWSAction(UWSAction) whose the goal is to replace a UWSAction by another one with the same name.

In UWSAlgorithms *AboutAction* is inserted:

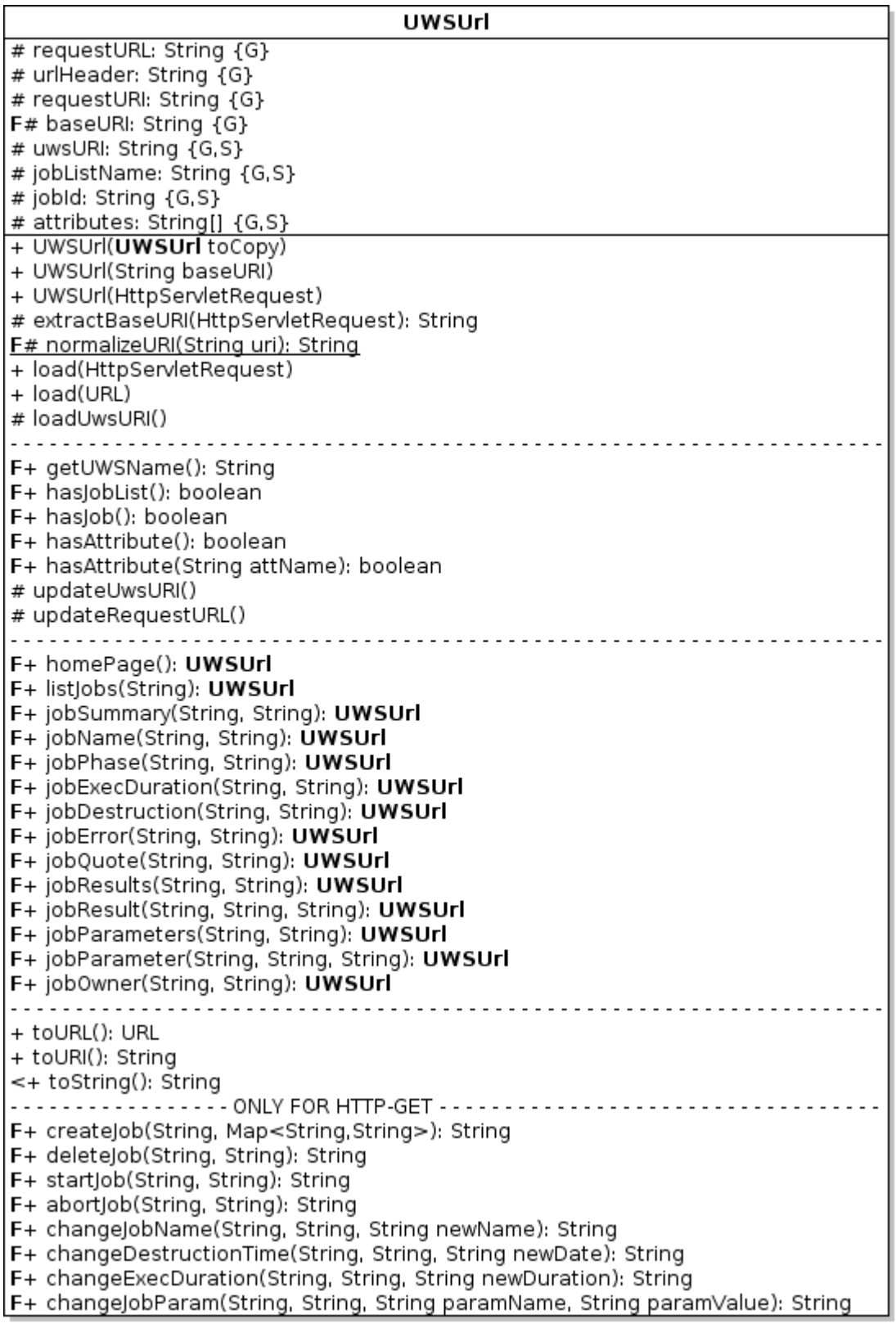
```
public class MyExtendedUWS extends ExtendedUWS {

        public MyExtendedUWS(URL baseURL) throws UWSException {
                super(baseURL);
                addUWSAction(0, new AboutAction<JobList<AbstractJob>,
AbstractJob>(this));
        }
...
}
```

**Note: Be very careful with the action position !**

The action has been inserted at the first position, to be sure it will be always evaluated ! Here it is necessary that this action is tested, at least, before ShowHomePage, because ShowHomePage matches even if there are parameters.

Obviously in AbstractUWS you can also:

- iterate on all actions: getUWSActions()
- search the action which has a given name: getUWSAction(String)
- and remove actions: removeUWSAction(int) and removeUWSAction(String).




## How to customize ?

By default the base URI is extracted from a HttpServletRequest by extractBaseURI(HttpServletRequest). This function only calls HttpServletRequest.getServletPath(). Consequently if you want to change the extraction of this URI, you just have to override extractBaseURI(HttpServletRequest).

However if you intend to change the URL splitting or the whole UWS URL structure, you will have to override all the other functions:

- **load(HttpServletRequest) and load(URL):**
  *They both fetch the request URI and the UWS URI. Then they call* loadUwsURI().
- **loadUwsURI():**
  *extracts all parts (jobListName, jobId and attributes) of the stored UWS URI.*
- **updateUwsURI():**
  *updates the UWS URI with the stored job list name, job ID and attributes. At the end, updateRequestURL() is called.*
- **updateRequestURL():**
  *updates the request URL and URI with the stored UWS URI, base URI and URL header.*

One more time, this class is very important and you should take care of any of your modifications ! By the way I hope the UML class diagram opposite will help you to better understand this class and to extend it if needed.

# 7. Serializations

As said previously all UWS resources are formatted by default in XML according to the IVOA Recommendation. With this library it is possible to manage more formats. Besides the format to use is choosen in function of the HTTP header *Accept*, and so, of the specified MIME types.

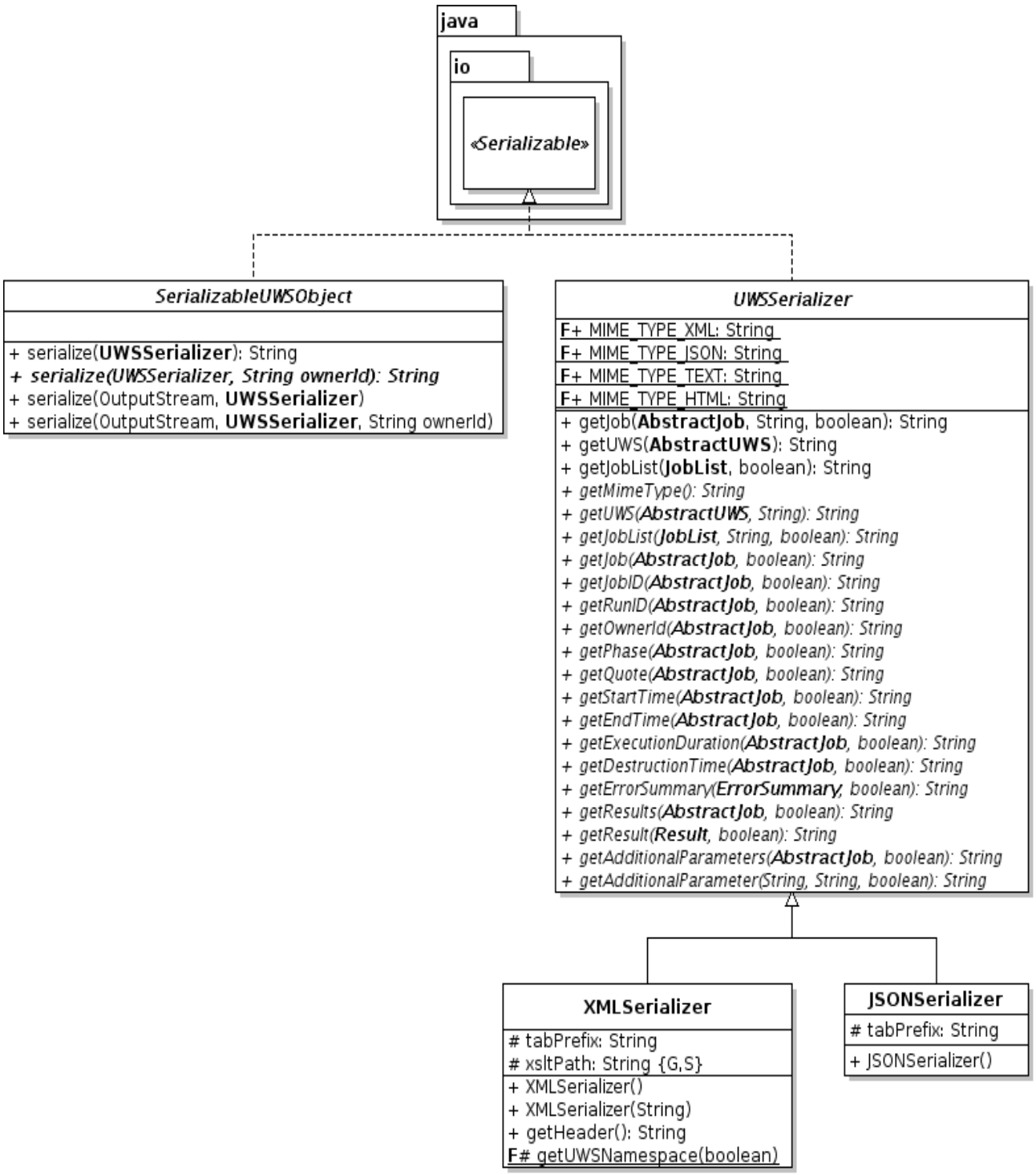
## The classes UWSSerializer and SerializableUWSObject

The class UWSSerializer is the abstract class used to define the serialization of any UWS resource in a given format. XMLSerializer and JSONSerializer correspond respectively to the XML and the JSON formats.

***Notes:***

- *The XML format is defined by the IVOA Recommendation thanks to this XML schema.*
- *The JSON format is already managed in this library. It respects the following format: JSON Model*
- *The most common MIME types are already stored as constant class attributes in UWSSerializer:*
  - *MIME_TYPE_XML: "application/xml"*
  - *MIME_TYPE_JSON: "application/json"*
  - *MIME_TYPE_TEXT: "text/plain"*
  - *MIME_TYPE_HTML: "text/html"*

All UWS objects extend the abstract class SerializableUWSObject. Only one function is abstract: serialize(UWSSerializer, String ownerId). It lets calling the good method of the given UWSSerializer *(ex: in* JobList *the function* UWSSerializer.getJobList(...) *is called)* and may consider the given owner ID to adapt the returned content to the current user *(ex: in* JobList *a user can not get the job of another user, he gets only its own jobs)*. Thus you just have to call any method of SerializableUWSObject on the UWS objects to "serialize" them.




## How does it work ?

Any HTTP request sent to a UWS ends with the serialization of a UWS resource. So the corresponding UWS action has to make a serialization of the asked objects in the specified format. As said previously this format is given by the HTTP header *Accept*. It gives a list of allowed formats which is actually an ordered list of MIME types. The choice of the format to apply is done by the method AbstractUWS.getSerializer(String) which takes the full MIME types list.

Below is the way that the action ListJobs returns the serialization of the specified job list:

```
public class ListJobs<JL extends JobList<J>, J extends AbstractJob> extends
UWSAction<JL, J> {
...
    @Override
    public boolean apply(UWSUrl urlInterpreter, String userId, HttpServletRequest
request, HttpServletResponse response) throws UWSException, IOException {
        // Get the jobs list:
        JL jobsList = getJobsList(urlInterpreter);

        // Write the jobs list:
        UWSSerializer serializer = uws.getSerializer(request.getHeader("Accept"));
        response.setContentType(serializer.getMimeType());
        jobsList.serialize(response.getOutputStream(), serializer, userId);

        return true;
    }
...
}
```

AbstractUWS has a list of UWSSerializer instances: the attribute serializers. AbstractUWS.getSerializer(String) chooses the serializer corresponding to the prefered MIME types among the given list. If there is no match the default serializer *(specified by the attribute* defaultSerializer*)* is returned.

## How to customize ?

You can add or remove easily some serializers to you UWS thanks to the functions: addSerializer(UWSSerializer) and removeSerializer(String mimeType).

It is also possible to iterate on all existing serializers thanks to getSerializers() as it is done in the additional action About of the UWS Algorithms.

Besides you can change the default serializer with setDefaultSerializer(String).

As in the version 2 of this library, it is also possible to associate a XML document to a XSLT style-sheet. It must be done in the instance of XMLSerializer used by your UWS, with the function XMLSerializer.setXSLTPath(String). As in the previous version, the function AbstractUWS.setXsltURL(String) still works. But now it calls the function XMLSerializer.setXSLTPath(String) on the used instance of XMLSerializer.

The UWS Algorithms uses this function to set the XSLT style sheet to any XML output:

```
public class UWSAlgorithms extends HttpServlet {
...
    @Override
    protected void service(HttpServletRequest req, HttpServletResponse resp) throws
ServletException, IOException {
        ...
        // Sets the XSLT URL:
        uws.setXsltURL(req.getContextPath()+"/styles/uws.xsl");
        ...
    }
...
}
```

**Warnings:**

- All serializers of a UWS must have a different MIME type. Otherwise addSerializer will not add the given serializer.
- The value of the attribute defaultSerializer is a MIME type and it MUST correspond to an existing serializer.

- Take care that there is always one serializer (the default one, of course) !

***Trick !***

*Some formats are often associated with several MIME types, like XML (application/xml and text/xml). If you want to associate several MIME types to one serializer, you can add manually some entries in the map attribute serializers. For instance:*

```
public class MyExtendedUWS extends ExtendedUWS {

    public MyExtendedUWS(URL baseURL) throws UWSException {
        super(baseURL);
        addUWSAction(0, new AboutAction<JobList<AbstractJob>, AbstractJob>(this));
        if (hasSerializerFor(UWSSerializer.MIME_TYPE_XML))
            serializers.put("text/xml", getSerializer(UWSSerializer.MIME_TYPE_XML));
    }
...
}
```

(MIME_TYPE_XML="application/xml")

# 8. Redirection and errors

In your servlet, after the initialization of your UWS, the only interaction with it is done at each HTTP request when calling executeRequest(HttpServletRequest, HttpServletResponse). Many occasions to throw an exception may occur during this call and particularly while executing an action. That's why this method is able to catch these errors. Its behavior in function of the occurred errors can be customized. Hence this part of the tutorial which will explain you in first how errors are caught and what is done with them.

**Warning**

The errors explained in this part are only about the UWS management. They must not be confused with the errors which occur during the execution of a job (those they are thrown within AbstractJob.jobWork()). These last one are already managed: they are stored as ErrorSummaries in the corresponding job. See C.3. Defining the job - Writing the task and D.4. Error summary for more details.

## How does it work ?

The whole content of executeRequest(HttpServletRequest, HttpServletResponse) is put into a *try* block, so that any instance of UWSException can be caught. This kind of exception is thrown only by this library, and particularly when a UWS action is executed. So when an exception occurs the method sendError(UWSException, HttpServletRequest, HttpServletResponse) is called. By default, it displays an Apache error with the given HTTP error code and the given message, except if the HTTP error code is 303 *(= See Other)*. In this last case a redirection is made thanks to the method redirect(String, HttpServletRequest, HttpServletResponse):

```
public abstract class AbstractUWS<JL extends JobList<J>, J extends AbstractJob> {
...
    public void sendError(UWSException error, HttpServletRequest request,
HttpServletResponse response) throws IOException, UWSException {
        if (error.getHttpErrorCode() == UWSException.SEE_OTHER)
            redirect(error.getMessage(), request, response);
        else
            response.sendError(error.getHttpErrorCode(), error.getMessage());
    }
...
}
```

Nevertheless any other kinds of exception may also occur. For that reason, a second *catch* block has been added to catch Exception. In this block the function sendError(Exception, HttpServletRequest, HttpServletResponse) is called, which prints the stack trace of the exception in the standard output and then displays an Apache error with the given error message:

```
public abstract class AbstractUWS<JL extends JobList<J>, J extends AbstractJob> {
...
    public void sendError(Exception error, HttpServletRequest request,
HttpServletResponse response) throws IOException, UWSException {
        error.printStackTrace();
        response.sendError(UWSException.INTERNAL_SERVER_ERROR, error.getMessage());
    }
...
}
```

Obviously the second function can also be used for UWSException, but the result will be different, especially about the management of the HTTP error code: no more redirection will be done anymore.

***Note:***

*Because of redirect(...), sendError(Exception, ...) and sendError(UWSException, ...), the method executeRequest(HttpServletRequest, HttpServletResponse) may also throw exceptions (UWSException and IOException). That's why you should put the call to this method in try...catch block !In that way you can still manage errors as you wish.*

## How to customize ?

The three methods sendError(Exception, HttpServletRequest, HttpServletResponse), sendError(UWSException, HttpServletRequest, HttpServletResponse) and redirect(String, HttpServletRequest, HttpServletResponse) can be overrided. However you should still beware of the specified status code with the UWSException so that errors with the 303 status code always make a redirection to the URL given in the exception message.

For instance, the UWS Algorithms has been modified so that errors are displayed in a different way:

```
public class MyExtendedUWS extends ExtendedUWS {
...
    @Override
    public void sendError(UWSException error, HttpServletRequest request,
HttpServletResponse response) throws IOException, UWSException {
        // Reset the whole response to ensure the output stream is free:
        if (response.isCommitted())
            return;
        response.reset();

        // If HTTP status code = 303 (see other), make a redirection:
        if (error.getHttpErrorCode() == UWSException.SEE_OTHER)
            redirect(error.getMessage(), request, response);

        // Else, display properly the exception:
        else{
            // Set the HTTP status code and the content type of the response:
            response.setStatus(error.getHttpErrorCode());
            response.setContentType(UWSSerializer.MIME_TYPE_HTML);

            PrintWriter out = response.getWriter();

            // Header:
            out.println("<html>\n\t<head>");
            out.println("\t\t<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=UTF-8\" />");
            out.println("\t\t<link
```

```
href=\""+UWSToolBox.getServerResource("styles/uwstuto.css", request)+"\"
rel=\"stylesheet\" type=\"text/css\" />");
            out.println("\t\t<title>UWS ERROR</title>");
            out.println("\t</head>\n\t<body>");

            // Title:
            String errorColor = (error.getUWSErrorType() ==
ErrorType.FATAL)?"red":"orange";
            out.println("\t\t<h1 style=\"text-align: center; background-
color:"+errorColor+"; color: white; font-weight: bold;\">UWS ERROR -
"+error.getHttpErrorCode()+"</h1>");

            // Description part:
            out.println("\t\t<h2>Description</h2>");
            out.println("\t\t<ul>");
            out.println("\t\t\t<li><b>Type: </b>"+error.getUWSErrorType()+"</li>");
            String msg = error.getMessage();
            int start=msg.indexOf("["), end=msg.indexOf("]");
            String context=null;
            if (start >= 0 && start < end){
                context = msg.substring(start+1, end);
                msg = msg.substring(end+1);
            }
            if (context != null)
                out.println("\t\t\t<li><b>Context: </b>"+context+"</li>");
                out.println("\t\t\t<li><b>Exception: </b>"+error.getClass().getName()
+"</li>");
                out.println("\t\t\t<li><b>Message:</b><p>"+msg+"</p></li>");
                out.println("\t\t</ul>");

                // Stack trace part:
                out.println("\t\t<h2>Stack trace</h2>");
                out.println("\t\t<table style=\"width: ihnerit;\">");
                out.println("\t\t\t<tr><th>Class</th><th>Method</th><th>Line</th></tr>")
;
                StackTraceElement[] trace = error.getStackTrace();
                for(int i=0; i<trace.length; i++){
                String className = trace[i].getClassName();
                if (className.startsWith("uws."))
                    className = "<a
href=\"/uwstuto/javadoc/"+className.replaceAll("\\.", "/")+".html\"
class=\"javadoc\">"+className+"</a>";
                out.println("\t\t\t<tr"+((i%2 != 0)?" class=\"alt\"":"")
+"><td>"+className+"</td><td>"+trace[i].getMethodName()
+"</td><td>"+trace[i].getLineNumber()+"</td></tr>");
            }
            out.println("\t\t</table>");
            out.println("\t</body>\n</html>");

            out.close();
        }
    }
...
}
```

***Trick !***

*You can test this customization by typing a bad jobs list name (for a fatal error) or a bad attribute name (for a transient error ; ex: params rather than parameters) in the URL.*

# 9. The interface HttpSessionBindingEvent

- This part of the documentation is useful only if you put any instance of AbstractUWS subclass in a session *(so it is stored in the navigator of the client)*.
- A better way to keep your UWS in state is to use saveUWs and restoreUWS of UWSToolBox. See the part F.1. Save & Restore a UWS.

When a session is finishing the contained UWS is not accessible anymore through a servlet. It can cause some problems. Indeed if all jobs of the previous session are still running, they continue to hold resources and so the performances of your computer are decreasing.

In order to clear all UWS resources *(stop threads and timers, delete files, ...)* when a session is finishing, the class AbstractUWS implements the interface HttpSessionBindingListener. This interface contains two methods:

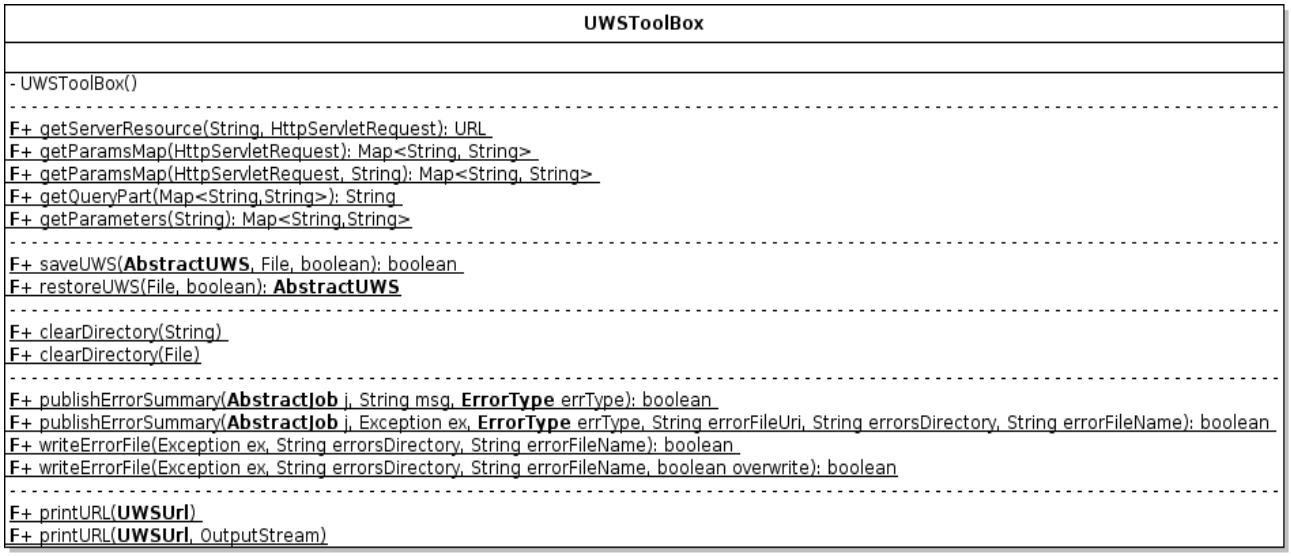
- **valueBound(HttpSessionBindingEvent)**: *called when the UWS is added as attribute of a session.*
- **valueUnbound(HttpSessionBindingEvent)**: *called when the UWS is removed from a session.*

These two methods are already implemented but can be overrided. By default valueBound(HttpSessionBindingEvent) does nothing except printing a message. valueUnbound(HttpSessionBindingEvent) calls the method removeAllJobLists() which calls the method clearResources() for all Jobs of all managed JobLists.

# F. UWS Tools-Box

The class UWSToolBox gathers several usefull functions. All of them are static, so you don't need to create an instance of this class. Some of them have already been used in the previous sections of this tutorial:

- **printURL(UWSUrl) and printURL(UWSUrl, OutputStream):**
  *They display the content of the given UWSUrl in the given output stream.*
- **publishErrorSummary(AbstractJob, String, ErrorType) and publishErrorSummary(AbstractJob, Exception, ErrorType, String, String, String)**
  *They set the phase to ERROR and set an error summary of the given job in function of the given parameters.*




## Error tools

The both error publication functions use actually writeErrorFile(Exception error, String dirPath, String fileName, boolean overwrite) which only writes the stack trace of the given exception in the specified file. The second writeErrorFile function calls this one but with *true* as last parameter which means that if the specified file exists, it will be overwritten.

## URL tools

In addition to the printURL methods, this class provides the following functions to manipulate more easily URLs:

- **getServerResource(String, HttpServletRequest):**
  *It returns the full URL to access to the server resource (a directory, a file, an image, a html page, ...) which corresponds to the given URI.*
- **getParamsMap(HttpServletRequest):**
  *It extracts the parameters map from the given HTTP request. The keys (parameter name) and values (parameter value) of the returned map are of type String. (note: HttpServletRequest.getParameterMap() returns a map whose the values are an array of String. Hence this tool function !)*
- **getParamsMap(HttpServletRequest, String):**

*It does the same thing than getParamsMap(HttpServletRequest) and add the given owner ID, if it is not already into the parameters map.*

- **getQueryPart(Map):**
  *It generates the query part of an URL and returns it. In a URL, this part starts with **?**. The parameters follows the syntax name=value and are separated by a **&**.*

- **getParameters(String):**
  *It does the contrary of getQueryPart(Map). The given query part is parsed so that parameters can be returned in a map (keys are parameter names whereas values are parameter values).*

## Saving & Restoring a UWS

At each stop *(or even during a simple restart)* of Tomcat, the value of all class attributes are lost. Consequently all pending/running/stopped jobs are destroyed. To avoid this effect you can save the UWS in the destroy() function, by using saveUWS(AbstractUWs, File, boolean). Below is an example:

```
public void destroy() {
    // Save the current state of this UWS:
    UWSToolBox.saveUWS(uws, restoreFile, true);
    super.destroy();
}
```

To restore a UWS you just have to call restoreUWS(File, boolean) in the init(ServletConfig) function:

```
public void init(ServletConfig config) throws ServletException {
    // Restore the last saved UWS:
    restoreFile = new File(config.getServletContext().getRealPath("/"), "uwsRestore");
    uws = (BasicUWS<JobChrono>) UWSToolBox.restoreUWS(restoreFile, true);
    super.init(config);
}
```

***Notes:***

- *All the UWS objects of the library are serializable (they implement the interface java.io.Serializable). The functions UWSToolBox.saveUWS and UWSToolBox.restoreUWS use this particularity to save the UWS in the specified file and then to restore it.*
- *Another solution is to use the Tomcat Persistent Manager which uses also the java Serialization mechanism. However according to the Tomcat website this functionnality "should be considered experimental" !*