Per G Jönsson, AS-IDS
Derek Mathieson, AS-IDS

ART

AIS Reporting Toolkit

or

Administrative Reporting Toolkit

or

A Reporting Toolkit

Design Proposal

Overall Architecture

Goals

The CERN Reporting Toolkit aims at fulfilling the ART User Requirements Document as well as:

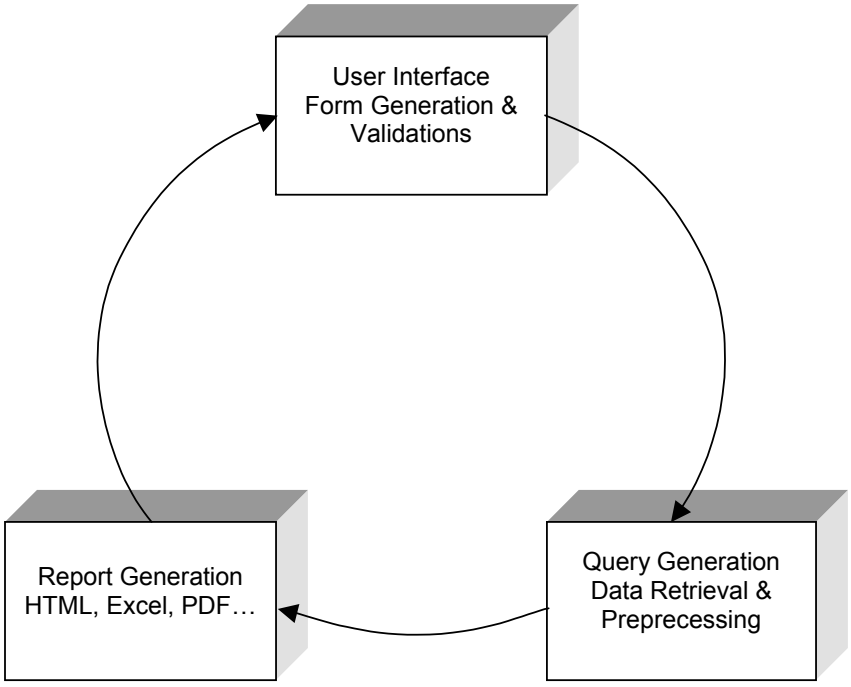
- Capitalise on existing development in EDH, MRT etc.
- Provide an easy to use reporting framework for AIS developers.
- Increase maintainability, by applying existing design/code and documentation standards and ensure in-house knowledge of the product.
- Work towards one development environment and technology. Any AS-IDS developer should easily be able to rotate between projects.
- Use of the framework should be possible with knowledge of standard languages only such as Java, SQL, XML.
- Ensure usability with commercial products such as Oracle Reports, NetCharts etc.
- Ensure that the use of commercial publishing tools can be used to provide HTML templates and definitions files.

Three Main Components

The core of the reporting toolkit will consist of three main blocks as shown on the next page. Each component will in turn have several sub-components. They will be discussed in more detail later in this document.

These three components will in effect reuse much code and ideas from existing work although in a repackaged form.

Also in addition to these main components, utility components such as log managers, query monitoring, object life cycle management etc. will exist. These components already also exists in the current EDH architecture and/or in the current MRT framework in a form that should be useable by the new framework with relatively minor modifications.



The user interface component

This component will draw on the proven EDH servlet architecture for form generation and validations. It will use the existing CBO's and CIO's meaning that the core of the architecture is already in place.

A modified version of the EDH generic servlet superclass will be created which will encapsulate the default services provided by the reporting toolkit.

This servlet will then likely be extended to create project specific servlets such as for instance a HRTGenericServlet adding on project specific features.

The query generation, data retrieval and pre-processing

This component will have the responsibility for producing a database query based on the information in the report and table definition files supplied, combined with the actual user data supplied by the user interface component.

The query component will either retrieve the data or pass to the report generation component or it will supply the constructed query to some other tool such as Oracle Reports for processing.

It is possible that the query component might do some prepossessing of the data in order to, for instance interpret Oracle 8 cross-tab queries etc.

The report generation component.

This component is responsible for producing the final output. It can do this by handing the data set or produced query to another application (NetCharts, Oracle Report etc.) or by using the data to produce the report which will be the case for, for instance, HTML and Excel report generation.

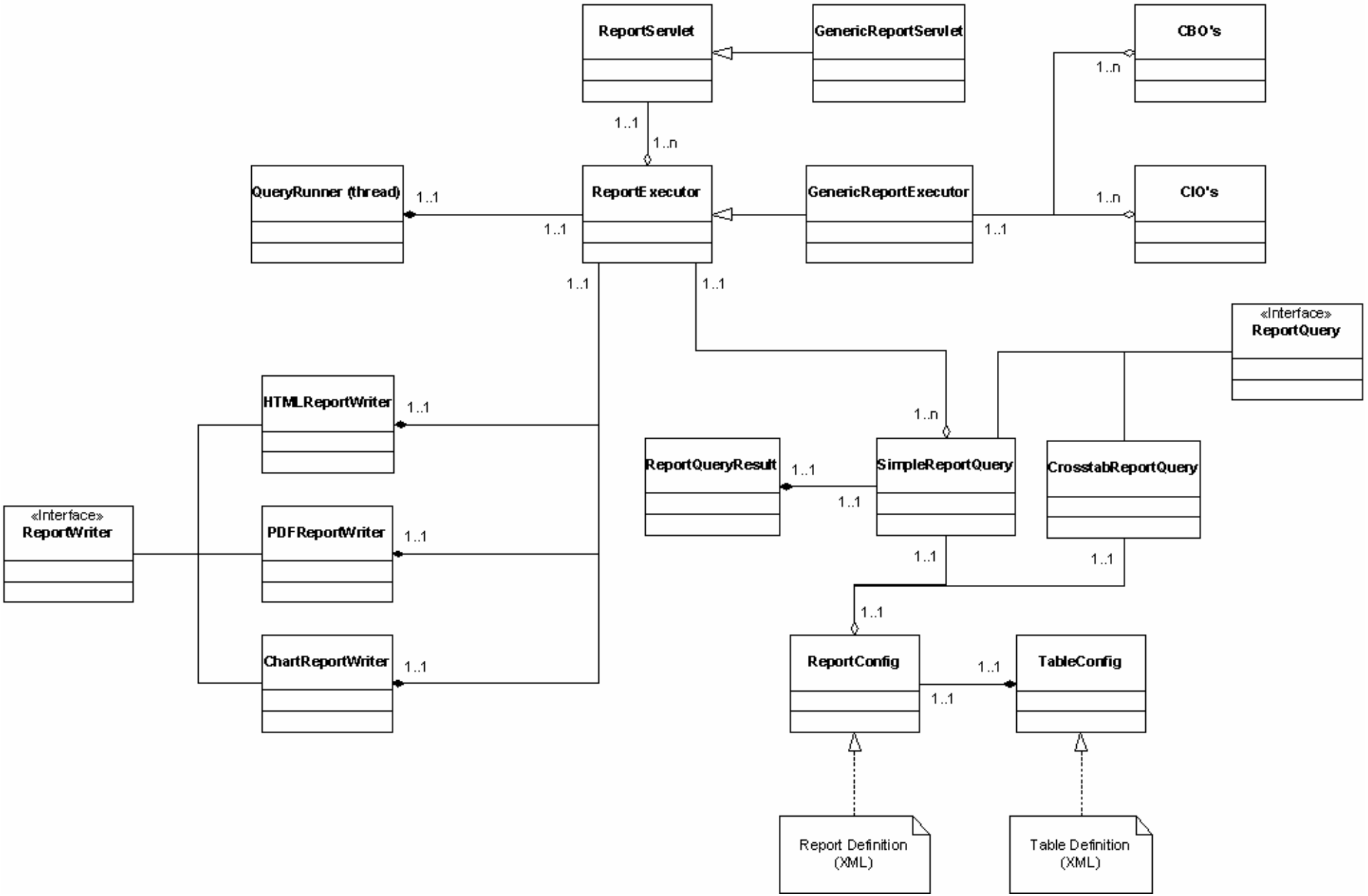
A More Detailed Overview.

On the following pages is a more detailed overview of the design.

More Detailed Class Diagram.

On the next page is a more detailed class diagram. Although this is more detailed it do not lay out all classes involved in the design, nor all references between them.

This diagram should however be sufficient for the normal user of the reporting toolkit to understand the general structure.


ReportServlet
GenericReportServlet
CBO's
1..n
1..1
1..n
QueryRunner (thread)
1..1
ReportExecutor
1..1
GenericReportExecutor
1..1
1..n
CIO's
1..1
1..1
«Interface»
ReportQuery
HTMLReportWriter
1..1
1..n
ReportQueryResult
1..1
SimpleReportQuery
CrosstabReportQuery
«Interface»
ReportWriter
PDFReportWriter
1..1
1..1
1..1
1..1
ChartReportWriter
1..1
ReportConfig
1..1
TableConfig
1..1
Report Definition
(XML)
Table Definition
(XML)

Class Responsibility Charts

Below are class responsibility charts which briefly outlines the responsibilities of each class.

User Interface Components

ARTGenericServlet
Generic handling of post and get. Obtains user identity. Instantiation and registration/deregistration of executor. Interpretation and feeding of default commands to executor.

ARTGenericExecutor
Generic parsing of HTML form and feeding of form data to ReportQuery. Generic generation of HTML output, common tag replacement etc. Handling of common tasks such as user polling/query interruption, loading of report configurations, container for ReportQuery and ReportWriter instances etc.

ReportServlet (implemented by the application)
Supplies the ARTGenericServlet with information about which executor to instantiate. Customizes the ARTGenericServlet if application specific processing is needed.

ReportExecutor (implemented by the application)
Supplies the ARTGenericExecutor with information about resources, templates and reports to use. Customizes the ARTGenericExecutor if application specific processing is needed.

Query Generation Components

TableConfiguration
Container for table definitions (meta data) describing a table. Loads, parses and interprets table definition files expressed in XML as defined by the TableConfiguration.dtd.

ReportConfiguration
Container for report definitions describing a report. Loads, parses and interprets report definition files expressed in XML as defined by the ReportConfiguration.dtd. Factory for ReportQuery and ReportWriter objects.

ReportQuery
Interface defining the api a query object in the reporting toolkit must implement.

SimpleReportQuery
A simple implementation of the ReportQuery interface handling “standard” queries. Produces a SQL query string from the data in a ReportQuery object and input data given to it by the caller, usually a ReportExecutor. Executes the query on demand and returns the resulting data in the form of a ReportQueryResult object.

CrosstabReportQuery
A crosstab implementation of the ReportQuery interface handling “crosstab” queries. Performs the same functions as the SimpleReportQuery but handles the extra tasks needed to execute and interpret a crosstab query (probably using Oracle 8 crosstab functionality).

ReportQueryResult
Container for a result row returned by the ReportQuery object. Hides physical database names and allows the user to refer to data by the names given in the report definition. Provides additional utility methods such as for example simple tests if a data value has changed since the last result etc.

Report Generation Components

ReportWriter
Interface defining the api a writer object in the reporting toolkit must implement.

HTMLReportWriter
Uses the data in a ReportConfig object together with a prepared and executed ReportQuery object in order to produce the report in HTML form onto a given output stream. Handles the necessary template management, tag replacement, template splitting and reconstruction etc.

PDFReportWriter
Uses the data in a ReportConfig object together with a prepared ReportQuery to produce a report in PDF form. Will interface with some external tool (Oracle Reports?) to perform its task. Will probably never execute the query but pass the query string to Oracle Reports.

ChartReportWriter
Uses the data in a ReportConfig object together with a prepared ReportQuery to produce a graph. Will interface with some external tool (NetCharts, Oracle Reports?) to perform its task.

QueryRunner
Thread class that executes the query and generates the result. Used by the executor to asynchronously execute the query. Can be polled to know if data is available and not. Capable of aborting the query on demand.

XML definition files and their use

The prototype DTD's for the definition files can be found in the appendixes at the end of this document.

In the design two different XML files are currently used.

1. The Table Definition files.

Below is an example extract of a table definition file.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE tableconfig SYSTEM "../Dtds/TableConfig.dtd">

<tableconfig name="Persons" dbname="EDHPER">
  <column name="Idn" dbexpr="IDN" type="NUMBER">
    <label eng="CERN id"/>
  </column>

  <column name="Nam" dbexpr="NAM">
    <label eng="Family name"/>
    <case type="upper"/>
  </column>
```

```
  <column name="Age" dbexpr="SYSDATE - Bth" type="NUMBER">
    <label eng="Age"/>
    <selectonly/>
  </column>
</tableconfig>
```

These files describe the tables (or views) and their content. It's used by the ReportQuery objects to construct a reasonable query as well as information that can be used by the user interface such as default labels in English and, optionally, in French.

For instance the <case type="upper"> tag on the Nam field can indicates to the ReportQuery object that any comparison for equality must be done with a to_upper construct to succeed.

The <selectonly/> tag on the Age indicates that it is an error if someone has written a report definition that uses that field in the select part of the query.

Note that the table definition can contain "virtual columns" such as the Age column, which does not map directly to a column but to an expression SYSDATE minus birth date to get the age.

Naturally the exact DTD for the table definitions are not yet complete.

2. The Report Definition files.

This file defines the actual report; its output options (HTML, Excel…), the possible in parameters, the fixed and variable query criteria (where clause), etc.

Below is an example of a simple report definition file.

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE reportconfig SYSTEM "../Dtds/ReportConfig.dtd">

<reportconfig name="TestReport">

  <queryclass name="cern.crt.SimpleQuery"/>

  <datasrc>
    <table ref="cern/crt/Tests/TestTable"/>
  </datasrc>

  <input>
    <indata name="Name" type="STRING"/>
    <indata name="FirstName" type="STRING"/>
    <indata name="OrgUnit" type="STRING"/>
  </input>

  <output>
    <option name="StandardReport" type="HTML"
            writerclass="cern.crt.HTMLWriter">
      <label eng="Standard HTML" fre="Standard HTML"/>
      <layout groupby="Persons.Org">
        <template ref="cern/crt/Main.html"/>
        <header>
          <templatesection name="mainreport_header"/>
          <report ref="cern/crt/Tests/TestSubReport"
                  outputoption="StandardReport">
            <link src="Persons.Org" dst="Name"/>
          </report>
        </header>
        <line>
          <templatesection name="mainreport_line"/>
        </line>
        <footer>
          <templatesection name="mainreport_footer"/>
        </footer>
      </layout>
    </option>
  </output>
```

```
  <select>
    <selectcol name="Persons.Nam">
    </selectcol>
    <selectcol name="Persons.Frn">
    </selectcol>
    <selectcol name="Persons.Org">
    </selectcol>
    <selectcol name="Persons.Age">
    </selectcol>
    <selectcol name="Persons.Email">
    </selectcol>
  </select>

  <where>
    <wherecol name="Persons.Nam" inputname="Name"
              conditiontype="LIKE">
    </wherecol>
    <wherecol name="Persons.Frn" inputname="FirstName"
              conditiontype="LIKE">
    </wherecol>
    <wherecol name="Persons.Org" inputname="OrgUnit"
              conditiontype="LIKE">
    </wherecol>
    <custom txt="Persons.Atc = 'Y'"/>
  </where>

  <order>
    <ordercol name="Persons.Org">
    </ordercol>
    <ordercol name="Persons.Nam">
    </ordercol>
    <ordercol name="Persons.Frn">
    </ordercol>
  </order>
</reportconfig>
```

This report is a simple report, which retrieves a number of persons, based on any combination of name, first name and organic unit. It orders them by organic unit, name and first name.

It also breaks the report into groups for each individual organic unit. The header of each group contains a sub-report, which retrieves the full description of the organic unit for the current group.

Some notes on some of the sections currently defined.

<datasrc>
Contains one or more tables to select from. If more than one table is used then the join condition must be included in the definition.

<input>
Defines the input parameters

<output>
Defines the various output options available. Several option can be available. For instance when interfacing with other products such as Oracle Reports, the flexibility for the user to control the report can be lesser than what we can provide with our HTML writers. Thus it could be necessary to provide more than one output option for various users.

<select>
Enumerates the columns to select. Can also contain hints to the writer such as “display in red when negative”, or decoration commands such as “decorate this data with a hyperlink (anchor-tag) using the data found in column XYZ” as the URL.

<where>
Describes the possible options when constructing the where clause. Can contain both fixed directives such as the <custom txt="Persons.Atc = 'Y'"/> tag in the example above, and <wherecol> tags that causes certain columns to be included only if the corresponding input data is available.

It also contains information to the ReportQuery objects on how to apply certain conditions. The “conditiontype=”LIKE”” part in the example will for example force a LIKE clause and percentage onto the condition. The argument to the conditiontype can be linked to an indata value, thus enabling the user interface to select the condition type.

The use of XML as basis for the definition language has several distinct advantages.

- It is a standard. Anyone familiar with XML will from the DTD be able to understand the syntax of the definition files. Considering the fast growth
- Editors are already starting to be available. You do not need to hand-edit XML-files no more than you need HTML-files.
- It is extensible. If one for some reason needs to implement a specialised ReportQuery or ReportWriter the XML can be extended so that the definition can contain any extra information needed. Existing code will ignore the parts of the DOM-tree that they do not know about and it will be the specialised objects responsibilities to interpret its private data.
- Supplied with a proper XSL, that you write once, XML files can, to a certain extent, be self-documenting. Via an XSL browsers such as IE 5 can display a definition file in a nice human-readable fashion.

Mapping of user interface to query

The actual mapping of the user interface data to the query is the responsibility of the Executor (unless a completely different type of user interface is used, which is possible) and is normally done by code in the ARTGenericExecutor unless custom behaviour is needed.

The default behaviour is for the named <indata> parameters in the <input> section of the query definition to be mapped to the names of the Common Input Objects (CIO’s) of the user interface.

Any mapping’s not supplied, i.e. the user did not type any data or data was not available for any other reason, will cause the corresponding parts of the <where> section definitions to be discarded from the query.

Mapping of indata parameters is not only done to columns used in the where clause but also to order by clauses, group clauses by and conditions types.

For instance the below example maps an indata parameter to a order clause. It is expected that the parameter should contain null (not used) or the name of a selected column as defined by the ReportDefinition.

Just for the sake of the example the order caused by this definition will be first by the column “Persons.Org" and then by the column given by the Name indata parameter if one is supplied.

```
<input>
    <indata name="Name" type="STRING"/>
    <indata name="FirstName" type="STRING"/>
    <indata name="OrgUnit" type="STRING"/>
</input>
.
```

```
.
.
<order>
    <ordercol name="Persons.Org">
    </ordercol>
    <ordercol name="Indata.Name">
    </ordercol>
</order>
```

Template management

As could be seen from the XML example above, specifically the HTML output option and it's layout definition, we talk about templates and templatesections.

The goal is that you should be able to use any publishing tool to produce you HTML files with all the advantages of WYSIWG.

In order to do this all of the HTML for a report should be in a single file that can be separated apart by the writer object and then recomposed with certain sections repeated the number of times necessary or even certain section left out.

On the next page is an example of a prototype HTML file. Please note that the <templatesection> tags are NOT in its final form and that the file is not complete but it illustrates the principle.

```
<templatesection name="report_header">
  <tr bgcolor=lightgrey>
    <td colspan=4>
      <h4>#SUBREPORT#</h4>
    </td>
  </tr>

  <tr bgcolor=white>
    <th>#Persons.Nam.LABEL#</th>
    <th>#Persons.Frn.LABEL#</th>
    <th>#Persons.Org.LABEL#</th>
    <th>#Persons.Email.LABEL#</th>
  </tr>
</templatesection name="report_header>

<templatesection name="report_line_1">
  <tr bgcolor=white>
    <td>#Persons.Nam#</td>
    <td>#Persons.Frn#</td>
    <td>#Persons.Org#</td>
    <td><a href='mailto:#Persons.Email#'>#Persons.Email#</a> </td>
  </tr>
</templatesection name="report_line_1">
<templatesection name="report_line_2">
  <tr bgcolor=white>
    <td>#Persons.Nam#</td>
    <td>#Persons.Frn#</td>
    <td>#Persons.Org#</td>
    <td><a href='mailto:#Persons.Email#'>#Persons.Email#</a> </td>
  </tr>
</templatesection name="report_line_2">
<templatesection name="report_line_3">
  <tr bgcolor=cornsilk>
    <td>#Persons.Nam#</td>
    <td>#Persons.Frn#</td>
    <td>#Persons.Org#</td>
    <td><a href='mailto:#Persons.Email#'>#Persons.Email#</a> </td>
  </tr>
</templatesection name="report_line_3">
<templatesection name="report_line_4">
  <tr bgcolor=cornsilk>
    <td>#Persons.Nam#</td>
    <td>#Persons.Frn#</td>
    <td>#Persons.Org#</td>
    <td><a href='mailto:#Persons.Email#'>#Persons.Email#</a> </td>
  </tr>
</templatesection name="report_line_4">

<templatesection name="report_footer">
</templatesection name="report_footer">
```

The template contains several parts, the report_header and report_footer, which are repeated once for each group, several report_line sections, which are repeated for each line.

The use of multiple line section is to create a banding effect, the first two are white, the next two are cornsilk. These are then iterated over by the writer.

Note that the sub-report is a single tag in this example. In the final product, the entire sub-report as such will be included in the same HTML file.

Customising

Customising queries

It is the intention that small set of different ReportQuery objects should be supplied which can handle the majority of cases.

In the event that something very special should be needed it might be necessary to create specialised ReportQuery objects.

The ReportQuery objects will be designed to minimise the effort in doing this. It should not be necessary to start from scratch with the ReportQuery interface and begin coding.

A set of methods will be supplied in the ReportQuery object implementation which can then be overridden by a extending object in order to hook into any part or any level of the query preparation process such as when

- preparing the entire query
- preparing the select clause
- preparing the where clause
- processing a single item in the where clause
- preparing the order by clause
- binding the input data to the query

etc….

Customising writers

What was said for the customising of queries also applies to customisation of writers.

An extending object will be able to hook in when

- producing the header
- producing a line
- producing a single column in the line
- executing a sub report

etc….

Miscellaneous

Query lifecycle management (query abort).

The lifecycle of a query must be monitored. Especially must a query where the user have gotten tired of waiting or extremely long running queries be able to be aborted.

The EDH infrastructure already have an executor lifecycle management tool where executor are registered and deregistered (killed) if the are idle for more than a certain amount of time which is given during registration.

MRT is deploying a technique where a polling interface is defined which is den continuously polled by a daemon. The implementer of this interface then has the responsibility of determining if the thread should be terminated or not.

Determining whether a query should abort or not is done by sending a space character to the client. If successful, the client is still assumed to be listening. This does not work on many platforms, i.e. Mac, and cannot be used if the return result is not HTML such as when sending Excel data etc.

The solution proposed in ART is to use client driven polling as opposed to servlet driven.

The core of the mechanism is the EDH lifecycle manager. When executing a query the executor of a report will register by the EDH lifecycle manager and asked to be removed after a certain maximum time (such as one minute). This mechanism is well proven and understood.

It will be augmented such that if the executor to be destroyed implements an interface (Abortable), the lifecycle manager will invoke a method (abort) defined by this interface so that the executor can take actions to properly abort the query.

The executor will then spawn a thread (QueryRunner) that performs the actual query execution.

It will subsequently wait a number of seconds, such as five seconds, if the thread executing the query has not finished, or at least started to produce results, after this time, a page will be sent back to the client with an appropriate wait message. If the query finishes within the initial few seconds the result will be sent back immediately.

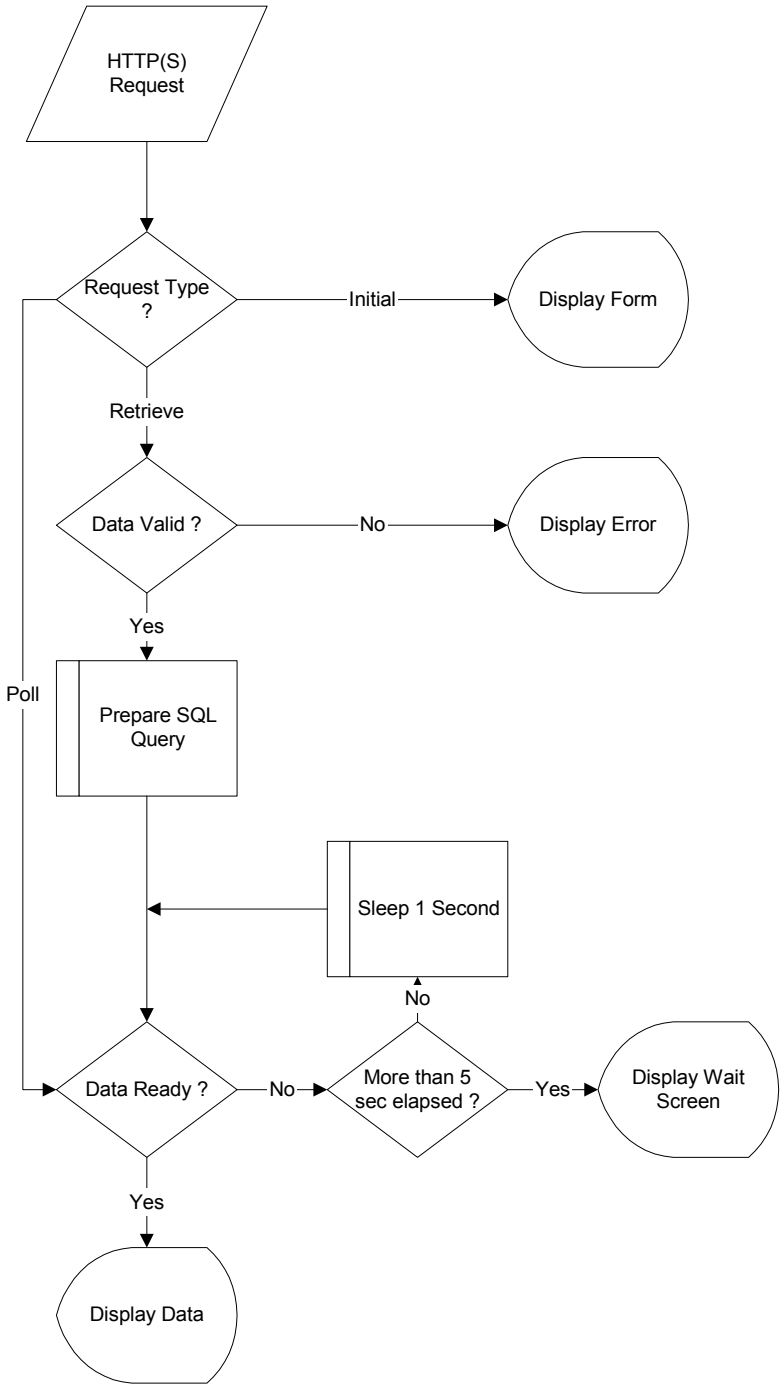
This page will contain a meta-tag telling the browser to resubmit the page after a number of seconds. When this ocurres, the executor will check the state of the query. if ready to deliver data, data will be sent back to the client, otherwise the same wait page will be sent back.

During each of these wait invokations, the executor will ask the life cycle manager to reset that executors “death-time”.

Should the client during this process abort, then no more submits from the wait page will be done and within one minute the life cycle manager will call upon the executor to abort the query and finally terminate the executor itself.

The entire process will be managed by the shared code in the ARTGenericServlet, ARTGenericExecutor and QueryRunner.

The flowchart below illustrates the process.



Logging

The current framework supports logging of certain events and keeps statistics of the usage of each report.

The majority of "internal" logging, which is vital to track down unexpected problems, is however done via "add hoc" calls to System.err.println.

The EDH infrastructure already has a central log manager, which classifies the log messages into different types, decorates the logs with the user identity, aids in including stack traces into the log.

It is expected that this tool, with some possible modification will be used as a central log manager in the new reporting toolkit.

Access, especially HRT style access.

Due to the access mechanism in HRT, a system with different user accounts having views of the same data but with varying degrees of visibility is deployed.

This system creates a maintenance burden in that a number of accounts and views must be maintained. It also inflates the number of connections that will be opened when connection pools are used since one pool must exist per access level.

In the new design the ReportQuery implementation could easily implement a simple access mechanism where the table definition contained an access level for each column and where the construction of the query excluded, or rather replaced with null, the columns that were not accessible.

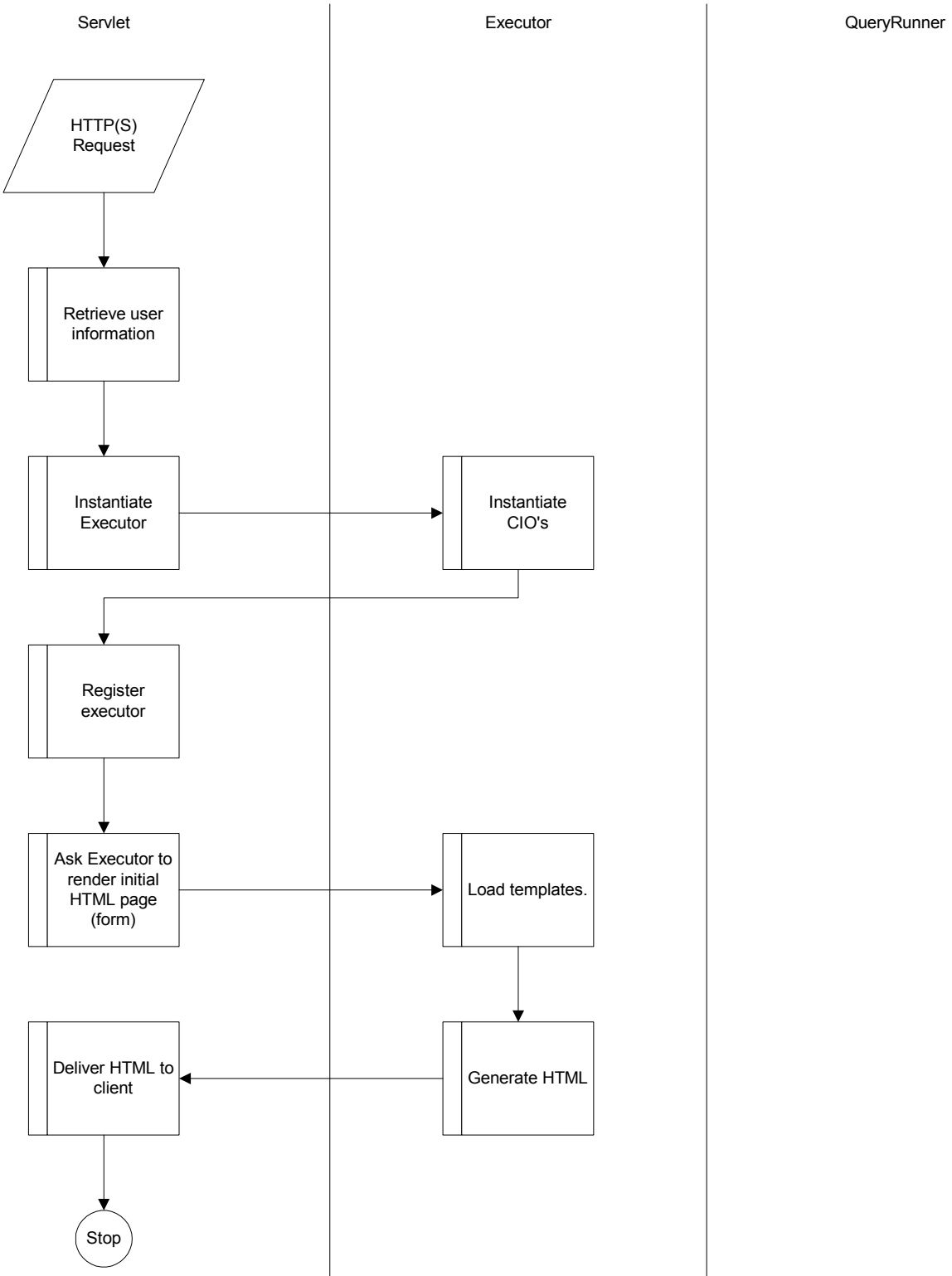
Having this data in the table definition, i.e. the table meta-data also makes it accessible to the user interface, i.e. greying out inaccessible columns in the option screens.

Appendix

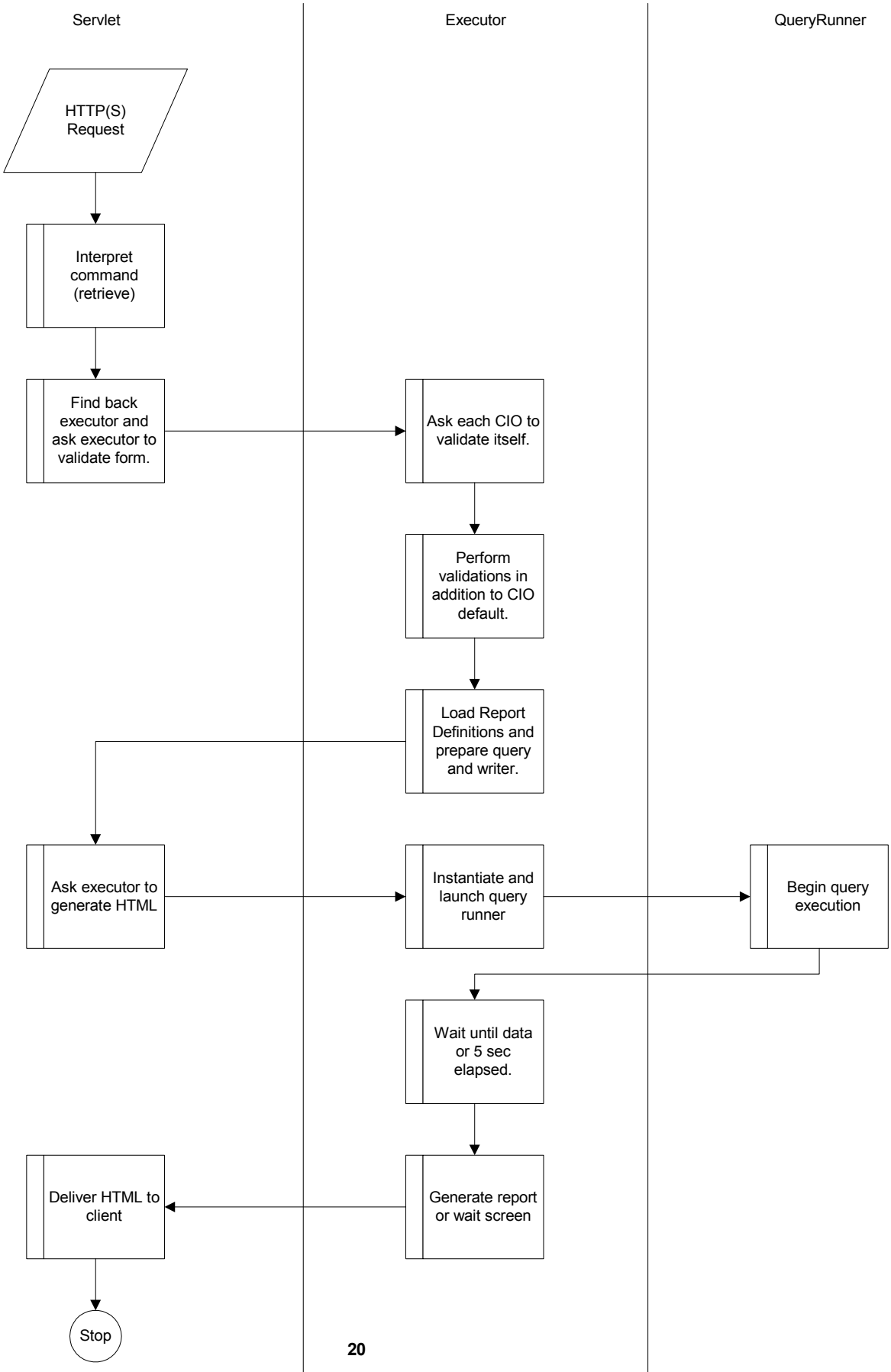
On the following pages flowchart pictures of the three normal servlet invocations (initial, retrieve and pool) and how the servlet, executor and query runner interacts.

The pictures do not outline all of the steps taken in detail. They are only meant to give a overview-type understanding of the mechanisms.

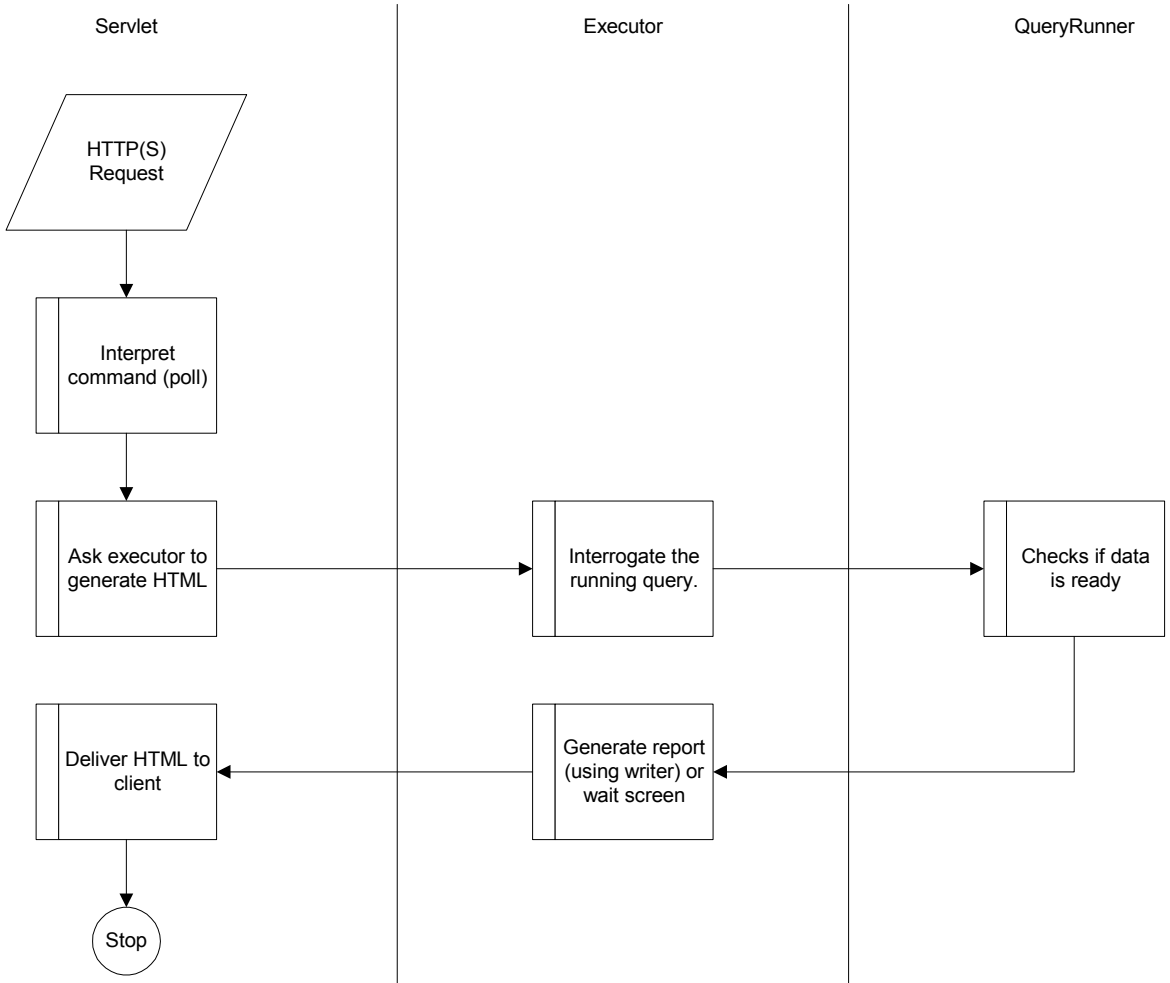
Initial Invokation of Servlet



Retrieve Invokation of Servlet (shows a situation without validation errors)



Poll Invokation of Servlet



Below follows the current state of the DTD's for table and query definitions.

Note that these definitions are NOT complete and changes not yet reflected in the blow examples have already been discussed.

```
<!-- DTD for XML's defining CRT Table Configurations -->
<!--                                                 -->
<!ELEMENT tableconfig (column+)>
<!ATTLIST tableconfig
  name    CDATA #REQUIRED
  dbname  CDATA #REQUIRED
>

<!-- Column source section                                  -->
<!-- This section defined a single column within the table -->
<!--                                                        -->
<!ELEMENT column (label, selectonly?, case?)>
<!ATTLIST column
  name    CDATA #REQUIRED
  dbexpr  CDATA #REQUIRED
  type    CDATA "STRING"
>

<!ELEMENT label EMPTY>
<!ATTLIST label
  eng CDATA #REQUIRED
  fre CDATA ""
>

<!ELEMENT selectonly EMPTY>
<!ELEMENT case EMPTY>
<!ATTLIST case
  type ( upper | lower | initcap ) #REQUIRED
>



<!-- DTD for XML's defining CRT Report Configurations -->
<!--                                                  -->
<!ELEMENT reportconfig (queryclass, datasrc, input?, output, select,
where, order?, group?, sql?, crosstab?)>
<!ATTLIST reportconfig
  name CDATA #REQUIRED
>

<!-- Definition of the query class to use for this report -->
<!--                                                      -->
<!ELEMENT queryclass EMPTY>
<!ATTLIST queryclass
  name CDATA #REQUIRED
>

<!-- Data source section                                          -->
<!-- This section defined which tables are included in the report -->
<!-- By enumerating a number of Table Configuration files         -->
<!-- See TableConfig.dtd for more details                         -->
<!--                                                              -->
<!ELEMENT datasrc (table+)>

<!ELEMENT table EMPTY>
<!ATTLIST table
```

```
  ref CDATA #REQUIRED
>

<!-- Input section                                        -->
<!-- This section describes the input to the report  -->
<!--                                                      -->
<!ELEMENT input (indata+)>

<!ELEMENT indata EMPTY>
<!ATTLIST indata
  name CDATA #REQUIRED
  type (STRING | DATE | NUMBER) #REQUIRED
>

<!-- Output section
-->
<!-- This section defines the possible output options for this report
-->
<!--
-->
<!ELEMENT output (option+)>

<!ELEMENT option (label, layout)>
<!ATTLIST option
  name        CDATA #REQUIRED
  type        ( HTML | EXCEL | PDF ) #REQUIRED
  writerclass CDATA #REQUIRED
>

<!ELEMENT label EMPTY>
<!ATTLIST label
  eng CDATA #REQUIRED
  fre CDATA ""
>

<!ELEMENT layout (template, header?, line, footer?)>
<!ATTLIST layout
  paginateby CDATA #IMPLIED
>
<!ELEMENT header (templatesection, report?)>
<!ELEMENT line (templatesection, report?)>
<!ELEMENT footer (templatesection, report?)>

<!ELEMENT template EMPTY>
<!ATTLIST template
  ref   CDATA #REQUIRED
>

<!ELEMENT templatesection EMPTY>
<!ATTLIST templatesection
  name   CDATA #REQUIRED
>

<!ELEMENT report (templatesection, link*)>
<!ATTLIST report
  ref           CDATA #REQUIRED
  outputoption  CDATA #REQUIRED
>
<!ELEMENT link EMPTY>
<!ATTLIST link
  src CDATA #REQUIRED
```

```
  dst CDATA #REQUIRED
>

<!-- Select section
-->
<!-- This section defines what is selected and any formatting options
-->
<!--
-->
<!ELEMENT select (selectcol+)>
<!ELEMENT selectcol ANY>
<!ATTLIST selectcol
  name CDATA #REQUIRED
>

<!-- Where section.
-->
<!-- This section defines which columns are involved in the where
clause -->
<!-- Plus any fixed custom where clauses
-->
<!--
-->
<!ELEMENT where (wherecol*, custom*)>
<!ELEMENT wherecol ANY>
<!ATTLIST wherecol
  name          CDATA #REQUIRED
  inputname     CDATA #REQUIRED
  conditiontype ( EQ | NEQ | GE | LE | GEQ | LEQ | LIKE | IN | ANY |
STD ) "STD"
>

<!-- Order and group section                                  -->
<!-- This section defined the order and group information -->
<!--                                                          -->
<!ELEMENT order (ordercol+)>
<!ELEMENT ordercol ANY>
<!ATTLIST ordercol
  name CDATA #REQUIRED
>

<!ELEMENT group (groupcol+)>
<!ATTLIST group
  type ( STD | CUBE | ROLLUP ) "STD"
>
<!ELEMENT groupcol ANY>
<!ATTLIST groupcol
  name CDATA #REQUIRED
>

<!-- Custom SQL section                    -->
<!-- Used to define custom SQL statements -->
<!--                                       -->
<!ELEMENT sql (custom+)>

<!-- Crosstab information -->
<!--                      -->

<!-- Some general elements used in varios sections of this DTD -->
<!--                                                            -->
<!ELEMENT custom EMPTY>
```

```
<!ATTLIST custom
  txt CDATA #REQUIRED
>
```