A DATA SOURCE DEFINITION TOOL FOR REPORTING TOOLS

Final Report

for

Bachelor of Technology

(Information Technology)

Haoxiang Zhu

Department of Computer Science

University Of Auckland

New Zealand

October, 2007

Supervisors:

Gareth Cronin (Industrial)

Development Team Leader

Kiwiplan Ltd.

New Zealand

Dr. Xinfeng Ye (Academic)

Department of Computer Science

University Of Auckland

New Zealand

ABSTRACT

Kiwiplan GUI framework provides a crystal reports style report designer that uses Kiwiplan's flexible table system as its data source. Everyday users are able to create reports based on existing customizable tables within application user interfaces. However, power users in organizations often wish to report on custom data sources. Traditionally power user reporting tools allow the definition of an SQL query and then reports are built on the resultset from this query.

In this project, we are going to build a Data Source Definition Tool which allows the users to link together business objects in a SQL join style fashion to achieve custom data sources based on the business objects, rather than just a SQL query on the database backend. We are also going to enhance the original reporting facility by building in a more powerful sub report facility in to the system.

Contents

1. INTRODUCTION ..... - 3 -

1.1. THE COMPANY ..... - 3 -

1.2. MOTIVATION ..... - 3 -

1.3. PROJECT GOAL ..... - 4 -

2. HIGH LEVEL SYSTEM OVERVIEW ..... - 5 -

3. INVESTIGATING IREPORT AND JASPERREPORTS ..... - 7 -

4. SYSTEM REQUIREMENTS SPECIFICATION ..... - 10 -

5. INTEGRATE IREPORT INTO THE SYSTEM ..... - 13 -

5.1 DATA SOURCE SERVICE V.S. DATA SOURCE ..... - 13 -

5.2 IDATASOURCESERVICEPROVIDER ..... - 14 -

5.3 USING JRDATASOURCE ..... - 15 -

5.4 USING IREPORTCONNECTION ..... - 16 -

5.5 WORKING WITH FLEXIBLETABLE IN KIWIPLAN FRAMEWORK ..... - 17 -

6. SUPPORT FOR SUB-REPORTS ..... - 19 -

6.1 MASTER REPORT AND SUB REPORT DATA SOURCE ..... - 19 -

6.2 SETTING UP JRPARAMETERS ..... - 21 -

6.3 BUILDING A SUB REPORT MANUALLY IN IREPORT ..... - 22 -

6.4 DATA SOURCE EXPRESSION ..... - 24 -

6.5 SUB REPORT PARAMETERS ..... - 26 -

6.6 OBJECT LEVEL JOIN FOR MULTIPLE DATA SOURCES ..... - 26 -

7. AUTOMATING BUILDING SUB-REPORTS ..... - 30 -

7.1 THE WORKFLOW ..... - 30 -

7.2 IMPLEMENTATION OF AUTOMATION PROCESS ..... - 33 -

8. FUTURE WORKS ..... - 39 -

9. CONCLUSIONS ..... - 40 -

10. ACKNOWLEDGEMENTS ..... - 41 -

11. REFERENCES ..... - 42 -

1. Introduction

This report documents the outcomes from the final year ***Bachelor of Technology*** project. In this section, I am going to briefly introduce the company which offers this project, followed by the motivation and the final goal of this project.

1.1. The company

Kiwiplan is a software development company that services the corrugating and packaging industries. Typical customers include firms that produce corrugated cardboard products such as boxes, display stands, and other packaging products.

Their product range covers the entire business process for a packaging firm, from order entry to shipping. The core products relate to controlling the plant and scheduling the corrugating machinery. These products communicate with the equipment on the factory floor to control production and collect data.

In the 1970's, Kiwiplan was starting business as a small corrugating firm. As their throughput increased they developed computer systems to help them keep up with demand. There was considerable interest in these computerised systems from other packaging companies, and the IT department grew and eventually separated from the box plant division.

Kiwiplan is now one of the world's leading software suppliers to the packaging industry. They have customers in 28 countries and four international offices. All research and development work is done in New Zealand.

1.2. Motivation

- Currently, Kiwiplan uses *DataVision* as the reporting tool. This type of reporting tool only supports reports to be built from a single data source. There is no way of combining data that obtained from different data stores (e.g. different databases) together to produce the desired reports. However, power-users in organizations often wish to report across multiple data sources, this is typically the case, for big organization such as Kiwiplan, where a report may require data stored at different database servers located in different counties (e.g. US office and NZ office).

- Sub-reports are widely used in many organizations. A sub-report is an entire report that is placed in the detail area of another report. Its main purpose is to display data from data sources linked in a set using one-to-many links at the same level. The existing reporting facility in Kiwiplan does not have the build-in support for creating a sub-report. Having the ability of producing sub-reports not only make the resulting report more visually understandable, it also saves time

and effort for technical staffs to prepare lots of complicated reports.

1.3. Project Goal

The goal of this project is to enhance the existing reporting facility in Kiwiplan, so that:

- Report user can use multiple data sources to create a single report.
- Report user can design the desired report using an easy-to-use tool.
- Sub-report should be supported.

A key difference between this reporting facility and the traditional reporting facility is that the tool works on-object-basis.

Traditional reporting tool usually has a connection to a back-end database, and the report is generated from the resultset of executing some query languages, e.g. SQL. In this tool, we do not have a back-end database connection. All the data sources that the reporting tool works with are on object level. An example of such objects can be a collection of JavaBean objects.

In the next section, a high level view of the proposed system is to be shown to get us familiar with the proposed system.

2. High Level System Overview

In this section, we are going to have a general structure of the proposed system, and also have some ideas of how the system can achieve the goals that I specified in the project goal.

There are a few very important components in this system, let us now look at them one by one:

- Raw Data Source: The raw data source is the place where the data that we use in the report is originated. This can typically be some relational databases.
- Custom Data Source: As mentioned earlier, report users typically want some kind of custom data source(s) for the report, and therefore, the custom data source here, will be the data source(s) that the reporting tool makes use of. The custom data source in this case, can typically be a collection of java objects, since our proposed tool works on object level.
- Reporting Tool: A tool that allows the users to design the report using the supplied data source(s).
- Report: The result the system should produce. Note that the content of the report may come from different raw data source(s).

Having explained these important components, let us have a look at how our proposed system links them all together, and achieve our goal:

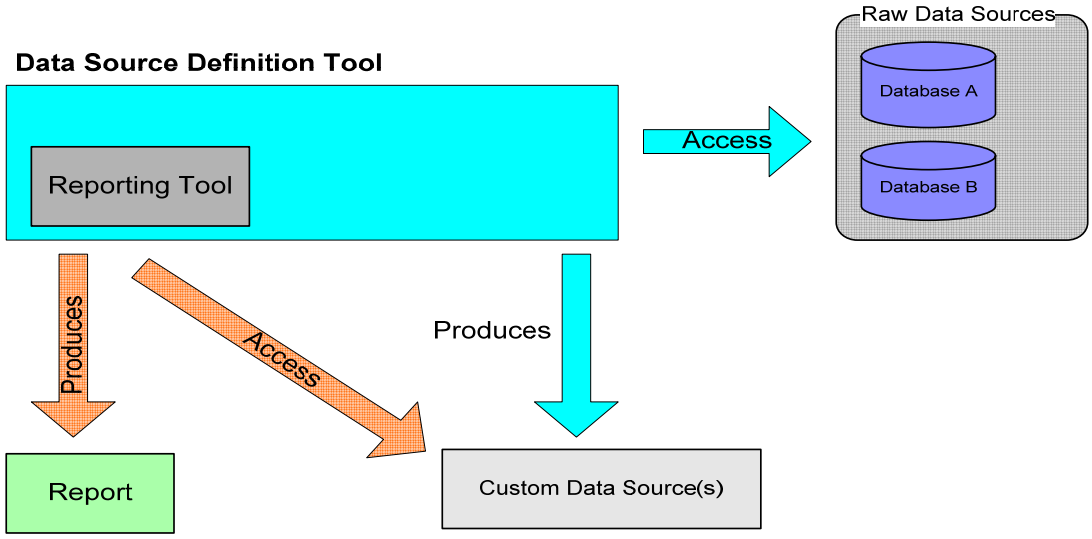


Figure 2.1 High level overview of data source definition tool

As we can see from Figure 2.1, the Data Source Definition Tool has linked all the important components together. This can be further explained by the following workflow:

1. The Data Source Definition Tool accesses some raw data sources.
2. The Data Source Definition Tool produces some custom data source(s). Note that there may be multiple custom data sources produced.

3. The reporting tool uses the custom data source(s) produced by the tool.
4. A report is generated based on the custom data source(s).

We can also notice that the reporting tool is actually part of the Data Source Definition Tool. This means that in order to develop such a reporting facility, we either have to develop a reporting tool of our own, or we have to use one of the existing reporting tools that are publicly available, which can be easily embedded into our system. There are many reporting tools that are publicly available, such as DataVision, iReport, and BIRT (as an Eclipse plug-in). They all have very similar functionalities, but in this particular project, iReport has been chosen as our underlying reporting tool, due to the fact that it is better suited to this project. The following table shows a comparison between iReport and DataVision on some selected features (Note that new version of DataVision may have some more features, this comparison is done based on the DataVision version that Kiwiplan uses.):

Tools / ***Features***	**iReport**	**DataVision**
Drag-n-drop report design	√	√
Language	Java	Java
Custom Data Source	Very well supported	Supported, but limited
Embeddability	√	√
Sub report	√	No
Report Parameters	√	√
Complexity	Heavy-weighted	Relatively light-weighted
Databases	Any with JDBC driver defined	Any with JDBC driver defined
Report Engine	Jasperreports	Build-in

Figure 2.2 Comparison between iReport and DataVision

As we can see from Figure 2.2, although the major functionality of these two reporting tools are quite similar, iReport is a better option due to its sub-report and custom data source support, hence, I have chosen iReport as the reporting tool for this project.

It is also shown in the table that iReport uses JasperReports as its report engine, in next section, we are going to look at the connection between iReport and JasperReports, and how they are combined together to produce an end-user report.

3. Investigating iReport and JasperReports

JasperReports and iReport are two widely used open source software developed by JasperSoft. They are purely written in Java, and their existences have made reporting in Java applications. In this section, we are going to look how JapserReports and iReport work.

JasperReports is an open-source Java class library designed to aid developers with the task of adding reporting capabilities to Java applications. Since it is not a standalone tool, it cannot be installed on its own. Instead, it is embedded into Java applications by including its library in the application's CLASSPATH. JasperReports is a Java class library, and is not meant for end users, but rather is targeted towards Java developers who need to add reporting capabilities to their applications.

JasperReports takes in a report design as an XML file (jrxml file), and compile into a jasper report file (jasper file). Through a JasperFillManager, a report print is produced for the end report users. The following diagram shows the work flow of how JasperReports work.

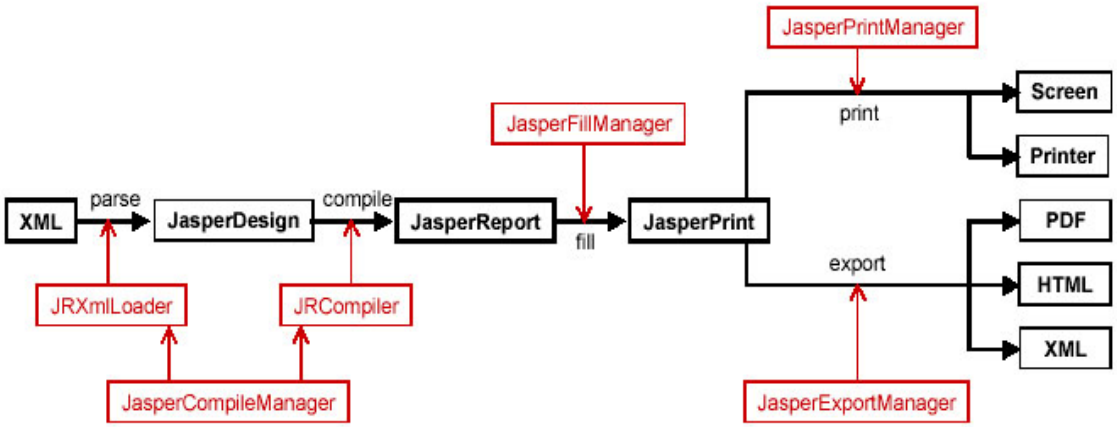


Figure 3.1 Work flow for JasperReports

(www.hisp.info/confluence/download/attachments/3330/seminar.ppt)

iReport, on the other hand, provides a front-end Graphical User Interface, for the end report users to define the design of a report, unlike JasperReports, iReport is targeted towards any report users, i.e. not necessary Java developers. The primary job of iReport is to produce the jrxml file for JasperReports to use. Hence, we can see iReport has JasperReports build-in as its report engine. The following diagram illustrates how iReport and JasperReports work together to produce a report for the end report users:

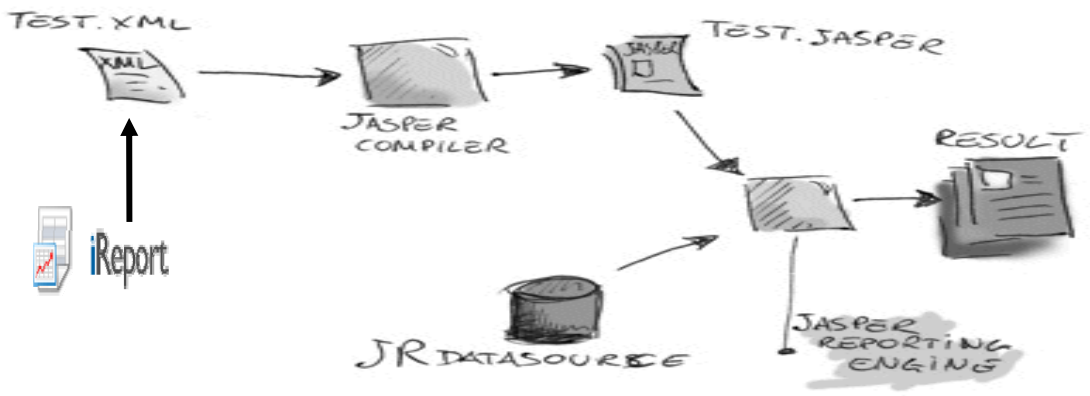
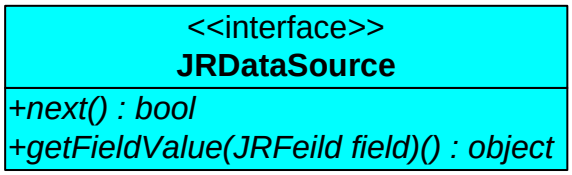


Figure 3.2 iReport and JasperReports
(http://ireport.sourceforge.net/cap3.html)

In Figure 3.2, report users define a report structure using iReport. The result that iReport provides is a jrxml file (in the diagram, it is shown as TEST.XML). Then this jrxml file is passed alone to JasperReports, it compiles the design of the report into a .jasper file (in the diagram above, it is shown as TEST.JASPER). This compiled report design is combined with a JRDataSource, to produce a final report (RESULT in the diagram), also known as a jasper-print. It is not hard to see that in JasperReports:

JasperPrint = Jasper file + JRDataSource



Now, let us have a look at another very important concept in JasperReports, JRDataSource. JRDataSource is an interface provided by JasperReports, it is the data source used for JasperReports to produce a print. Therefore, any possible data sources need to implement this interface, to provide the compatibility to JasperReports.

Some examples of implementations of JRDataSource can be:

- JRResultSetDataSource – wraps a JDBC ResultSet object as the data source.
- JRXMLDataSource – wraps an XML document as the data source.
- JRTableModelDataSource – wraps a TableModel as the data source.

Through this interface, JasperReports has provided the users with the ability to define custom data sources. Having this interface, we can therefore implement any kind of data source of our own, and those custom data sources that defined by ourselves can be used by JasperReport as the data source to produce reports. This is one of the most important reasons that I chose iReport/JasperReports as the reporting tool in this project.

Figure 3.3 shows the system structure when iReport is embedded:

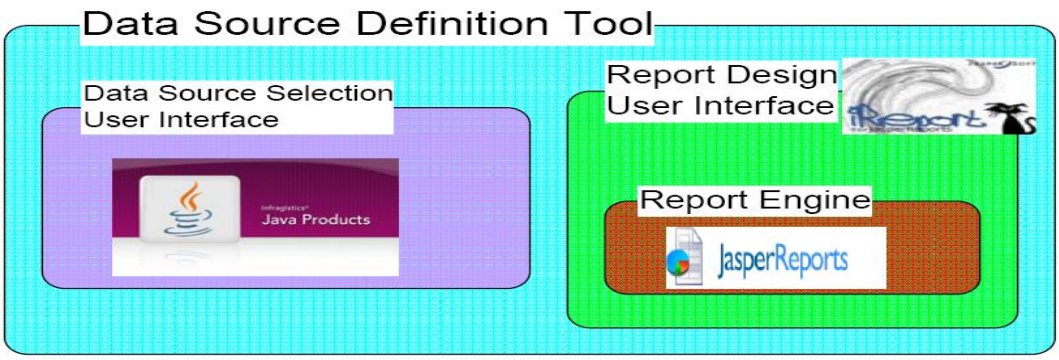


Figure 3.3 System embedded iReport

As shown in Figure 3.3, the system includes the following main components:

- **Data Source Selection UI**: This user interface allows the users to select the appropriate data sources from available data sources services. (Refer to Section 5.1 later in this report).

- **Report Design UI**: This user interface provides a front end environment for users to design the report graphically (e.g. by simply drag and drop fields), in our case, this feature is already provided by iReport.

- **Report Engine**: Every report is generated using a report engine, iReport has its build-in report engine as JasperReports.

4. System Requirements Specification

Requirement Engineering has been one of the most important components throughout Software Development Life Cycle (SDLC). In order to detail the requirements of the project, I have had quite a number of meetings with the development leader within Kiwiplan. The resulting software requirement of this project is summarized using the following use case diagram:

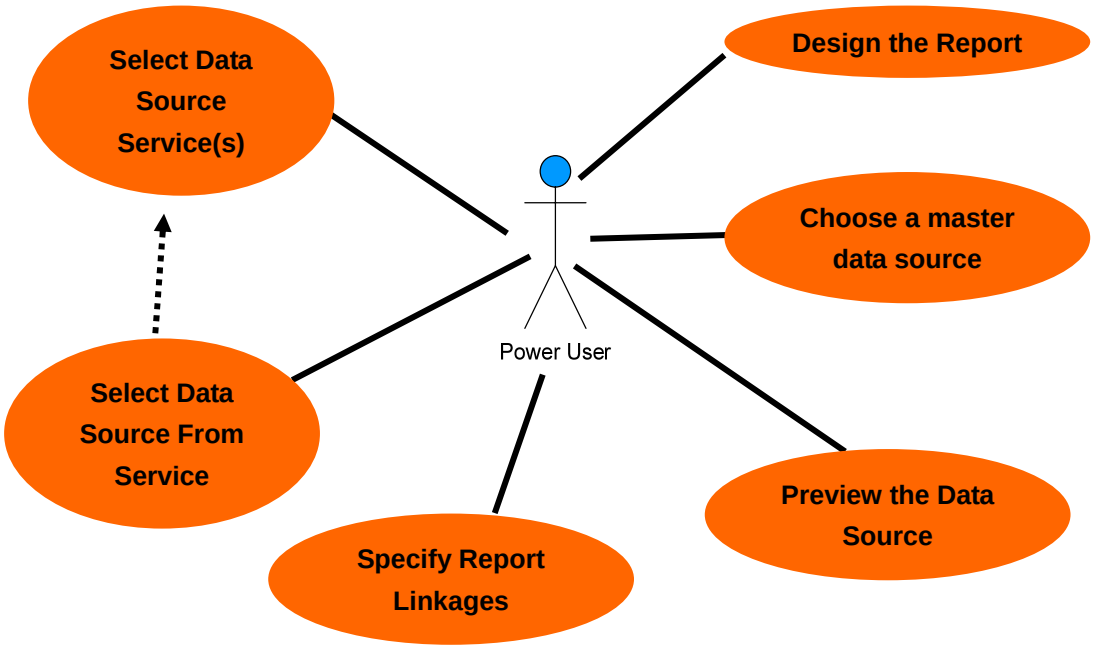


Figure 4.1 Use Case Diagram

Now, let us have a look at each of the use case in more detail:

- **Select Data Source Service(s):** As previously mentioned, the user is able to design the report across multiple data sources. Therefore, the user needs to be able to select different data source services which provide accesses to different data sources. Typically, a data source service provides access to more than one data sources.

- **Select Data Source From Service:** When the users have chosen the data source service, they should be able to select the corresponding data source from the particular service.

- **Preview Data Source:** After the user has chosen the data source, they should be able to preview the selected data source(s). The data source(s) are typically displayed in a JTable under the current implementation.

- **Choose Master Data Source:** The functionality is primarily used for the sub-report support of the tool. A master data source is the data source used for the

master report. When the user has chosen more than one data sources from the services, they should (they are required) choose a master data source for iReport to use, and any other data sources are treated as detail data sources used for sub-report. This feature will be discussed in more detail later this report.

- **Specify Report Linkage:** When users choose to design a sub report, they should specify how the sub report is linked to its master report, particularly, which fields from the master report is linked to the ones in the sub report. This feature will again be discussed in more detail later this report.

- **Design Report:** Of course, the users need to be able to design the report structure.

After the use cases have been decided. The User Interface has been developed according to those use cases. The following figure is a screen-shot of the current implementation of the user interface (Note that this user interface design might change later):

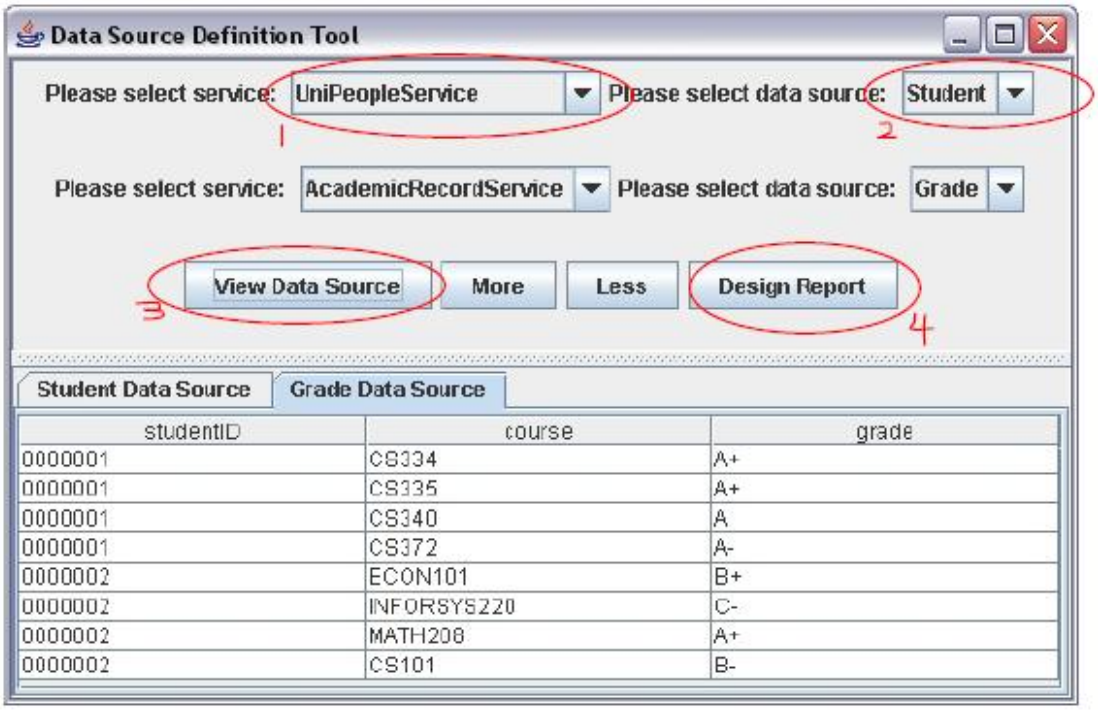


Figure 4.2 User Interface

In the above User Interface design:

- The component that is labeled "1" corresponds to the **Select Data Source Service** use case where user is able to choose among different data sources. For example, if there are four data source services available, each of the dropdown lists will have the four services available for users to choose from. By default, the system shows one pair of "select service" option and "select data source" option, the user can ask to choose from more/less data source services/sources by

clicking the “More”/ “Less” buttons. In this example shown in Figure 4.2, the user has clicked the “More” button once, and has chosen a service called “UniPeopleService” which provides the people service.

- The component that is labeled “2” corresponds to the “**Select Data Source From Service**” use case where the user has chosen the “Student” data source.

- The component that is labeled “3” corresponds to the “**Preview Data Source”**, where the user is able to preview the selected data source(s) in JTable, in the example above, as the user has chosen two data sources from two different services, there are two data sources table displayed in a tabbed pane.

- The component that is labeled “4” corresponds to the “**Design Report**” use case. This button will launch iReport report design panel, providing the users with a graphical user interface to design the desired report. However, if more than one data sources are chosen, a wizard will pop up asking the users to pre configure the report settings. This wizard will be discussed in more details in the next sections.

The following is a brief class diagram for the user interface design:

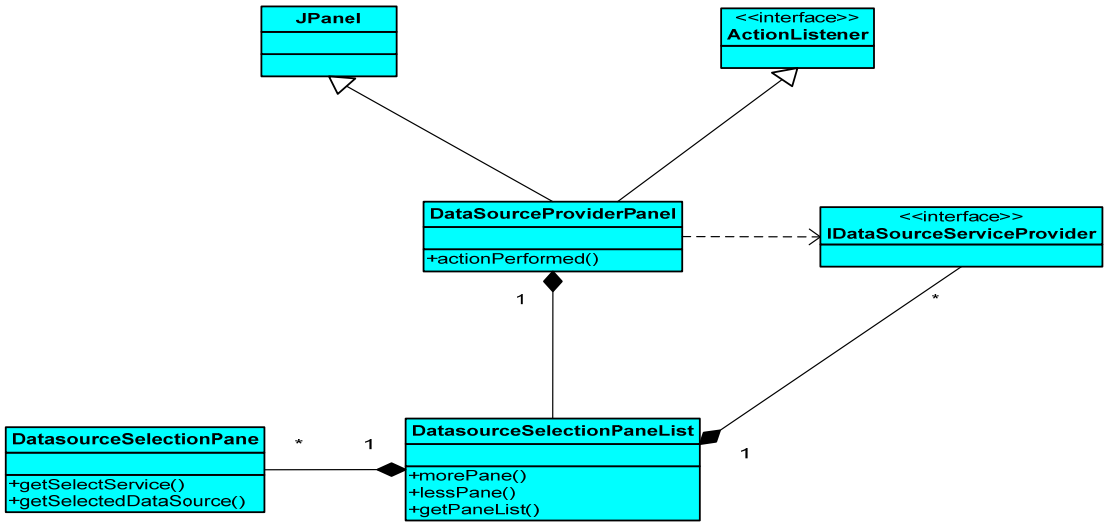


Figure 4.3 class diagram for user interface design

5. Integrate iReport into the System

Up until this point, we have had the data source(s) ready for the iReport to use. In this section, we are going to see how iReport uses our custom data source(s) produced by the Data Source Definition Tool, and how those data source(s) are passed along to iReport.

5.1 Data Source Service V.s. Data Source

Before I introduce how iReport is integrated into the system, we have to clarify two very important terms that we used: **Data Source Service** and **Data Source**, first, let us have a look at the diagram:

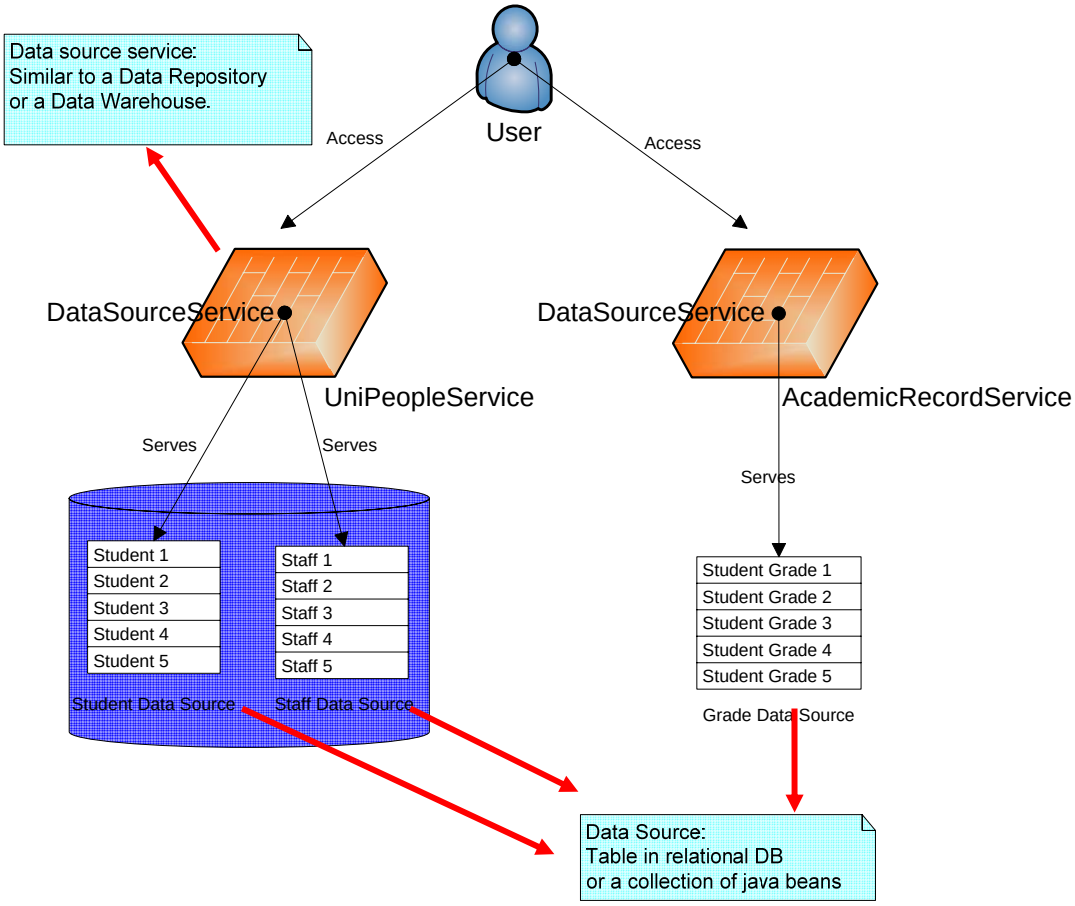


Figure 5.1 Data Source Service V.s. Data Source

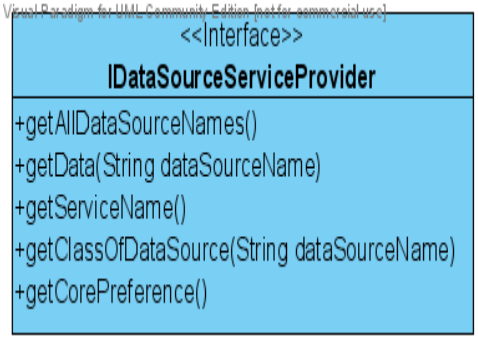
A **data source service** is a **service access layer** that provides accesses to the data sources it serves. In the above diagram, there are two data source services, UniPeopleService and AcademicRecordService, the UniPeopleService provides two data sources: Student and Staff data sources. And the AcademicRecordService provides only one data source StudentGrade data source. A data source service is similar to a data repository or a data warehouse, in the sense that it stores all the data

sources corresponding to that particular service.

A **data source** is the actual objects have the data information stored. Typical examples of data sources can be a table in a relational databases or a collection of java objects. In the above diagram, all the data sources are provided as a list of simple JavaBean objects.

5.2 IDataSourceServiceProvider

Now, let us have a look at how the data source service is implemented. As we noticed from Figure 4.3, there is an Interface called *IDataSourceServiceProvider*. In the use case diagram (shown in Figure 4.1), the user is able to select from different data source services. This interface provides such compatibility for any class that provides this kind of service. The class diagram of this interface is shown on the left:



For any classes that implements this interface, they should specify how each of the data sources that this service provide is obtained, for example, from a JDBC resultset, or from a collection JavaBean objects in the getData() method. A sample implementation of this interface is shown in Figure 5.2, which provides the data sources shown in Figure 5.1.

```
public List getData(String dataSourceName) {
  List list = new ArrayList();
  if (dataSourceName.toLowerCase().equals("student")) {
    Student s1 = new Student("0000001","James", "Bond");
    Student s2 = new Student("0000002","Bill", "Gates");
    list.add(s1);
    list.add(s2);
  }
  else if (dataSourceName.toLowerCase().equals("staff")) {
    Staff st1 = new Staff("4545674","Abc","Def","Computer Science");
    Staff st2 = new Staff("8745374","Kkk","Hhh","Computer Science");
    Staff st3 = new Staff("3524364","Loo","Ccc","Economics");
    list.add(st1);
    list.add(st2);
    list.add(st3);
  }
  return list;
}
```

Figure.5.2 Sample implementation of getData method for IDataSourceServiceProvider

In this implementation, the data sources are generated from a List of JavaBean objects.

5.3 Using JRDataSource

By having the data source services, we have the ability to enable users to choose data sources from different services. However, after these selected data sources are in place, we need to make use of these custom data sources in iReport. As I mention in Section.3 of this report, iReport make use of custom defined data sources through an interface called "*JRDataSource*".

In our case, when the user has selected the desired data sources, they are displayed in a JTable. Therefore, we can make use of a pre-defined data source called *JRTableModelDataSource* to warp the table model as the data source for iReport to use.

Typically, any kind of *JRDataSource* is provided through a corresponding JRDataSourceProvider, which is another interface provided by JasperReport. In my implementation, I have implemented a *TableModelDataSourceProvider* to provide the *JRTableModelDataSource***.**

The detail of the structure is shown in the following class diagram:

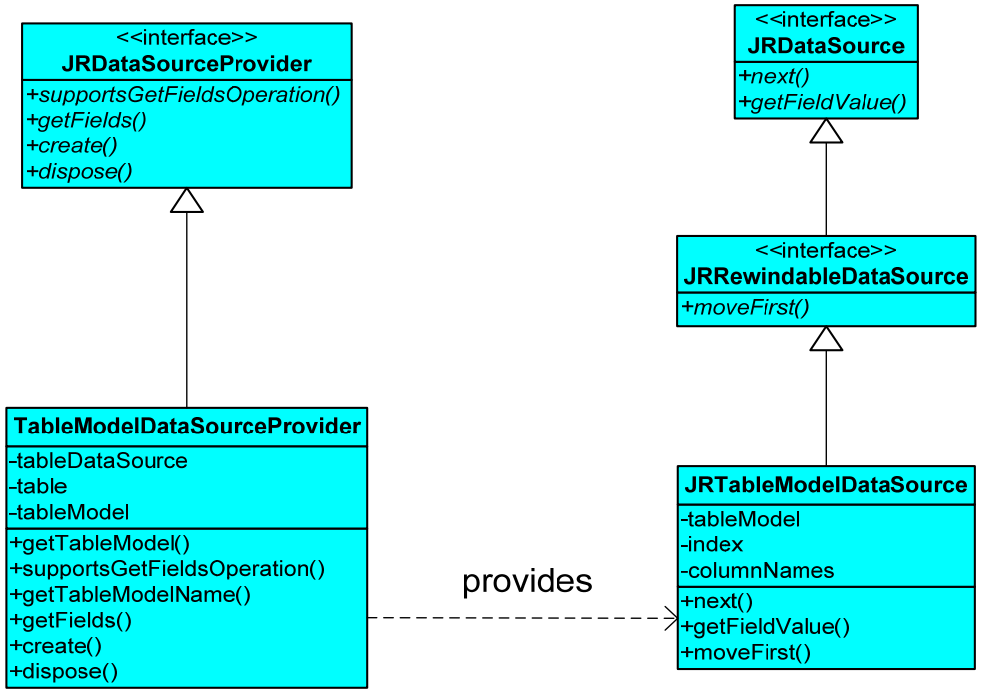


Figure 5.3 Class diagram for JRDataSource and its provider

- The *JRRewindable* interface is an extension to *JRDataSource*, any class that implements this interface is a data source that can move back to its very first element. *JRTableModelDataSource* is such a class.

- *TableModelDataSourceProvider* provides the *JRDataSource* through the *create*() method. The following code segment shows the constructor and the create

method of *TableModelDataSourceProvider*:

```
/* constructor */
 public TableModelDataSourceProvider(TableModel tableModel){
    this.tableModel = tableModel;
    this.tableDataSource = new
                          JRTableModelDataSource(this.tableModel);
  }
/* create a JRDataSource */
public JRDataSource create(JasperReport report) throws JRException{
    return new JRTableModelDataSource(tableModel);
  }
```

Figure 5.4 code segment showing constructor and create method from TableModelDataSourceProvider

5.4 Using IReportConnection

Using *JRDataSourceProvider* provides the *JRDataSource* for iReport to use, however, we still have to find out a way to pass the *JRDataSource* from our Data Source Definition Tool to iReport.

iReport has provided class called "*IReportConnection*" which enables the custom connections to iReport. Typically, each *IReportConnection* (and its sub classes) warps a *JRDataSourceProvider* instance in to, so that when this connection gets connected to iReport, the *JRDataSource* is provided through the provider that is wrapped inside this connection. In this project, I have implemented a *TableDataSourceConnection* which wraps a *TableModelDataSource* in it as the provider of *JRTableModelDataSource*. The following class diagram shows how iReport uses *IReportConnection* to produce data source through the provider:

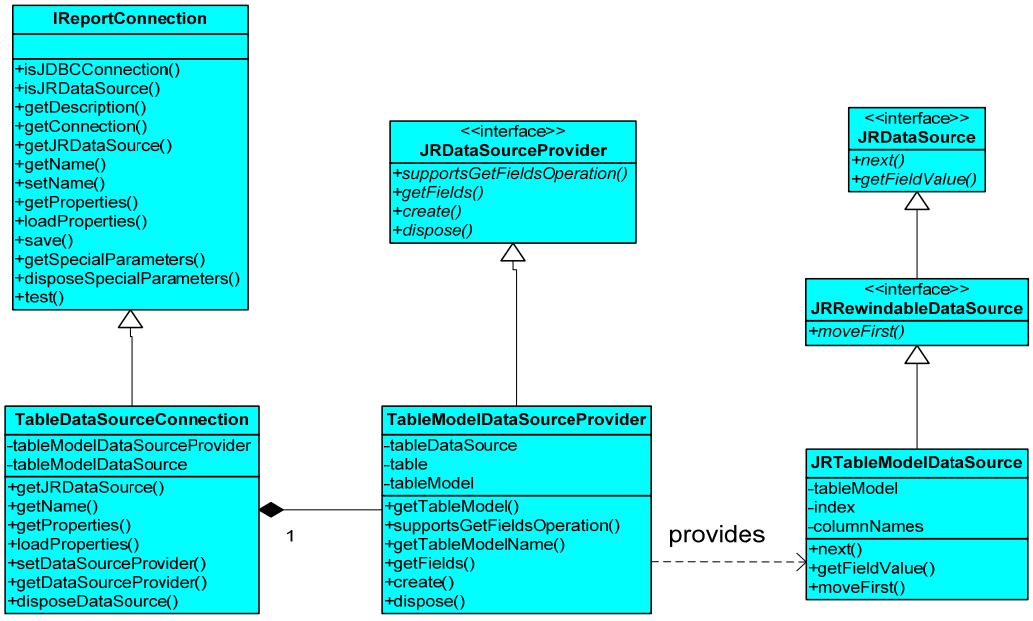


Figure 5.5 Class diagram: connects to iReport

When the users choose more than one data sources from the Data Source Definition Tool, each of those data sources will be created into a *JRDataSourceProvider* and wrapped into an *IReportConnection*, i.e. every data source is passed alone to iReport as a separate connection. The following code segment shows how iReport is launched with the pre-defined custom data source(s) that user has chosen using the Data Source Definition Tool:

```
public static void launchIReportWithDefaultConnection(…){
  … …
     Vector connList = new Vector();
     for (Iterator iterator = tableModels.iterator();
                                  iterator.hasNext();) {
     TableModel dataViewTableModel = (TableModel) iterator.next();
     TableModelDataSourceProvider provider = new
                 TableModelDataSourceProvider(dataViewTableModel);
     IReportConnection conn = new
                            TableDataSourceConnection(provider);
     connList.add(conn);
    … …
}
frame.setConnections(connList);
... …
```

Figure 5.6 Code segment: launch iReport using custom data sources

5.5 Working with FlexibleTable in Kiwiplan Framework

In Kiwiplan, instead of displaying data using a *JTable*, we use a *FlexibleTable* to display the data source(s). The *FlexibleTable* is similar to *JTable* but with more sophisticated functionalities, e.g. the table headers are defined from a preference bundle XML file. The following screen shot is an example of using the Data Source Definition Tool with *FlexibleTable*:

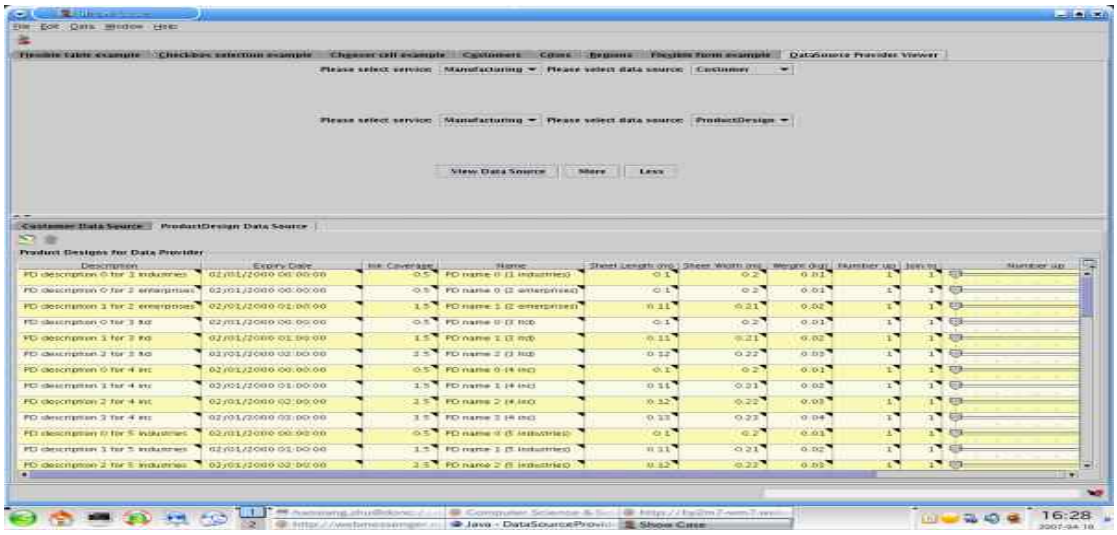


Figure 5.7 Data Source Definition Tool with *FlexibleTable*

The implementation using *FlexibleTable* is not fully completed yet. However, the concepts of using the *FlexibleTable* are exactly the same as using *JTable*. The following class diagram shows the design using *FlexibleTable*:

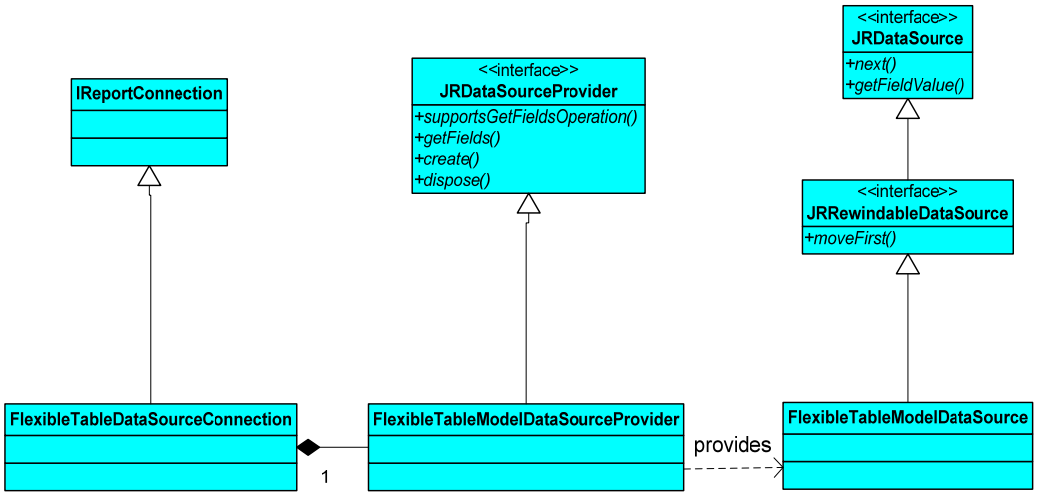


Figure 5.8 class diagram: connects to iReport using FlxibleTable

This feature is not yet included in the end due to time constraint, however, converting from JTable into FlexibleTable is not considered to be a complex task, and this feature is to be included in the future development.

6. Support for Sub-reports

Nowadays, sub reports have been widely used in many organizations, therefore, it is necessary to allow the Data Source Definition Tool to have the ability to build sub-reports. A sub report usually uses multiple data sources that have some kind of relationship. In this section, we are going to see how the Data Source Definition Tool supports the user in creating sub reports using the custom data sources that it provides.

6.1 Master Report and Sub Report Data Source

As described in Section 4, when the user chooses more than one data sources, a wizard will pop up. The first wizard step is to ask the user to choose one of the selected data sources as the master data source to iReport. The wizard step is shown below:

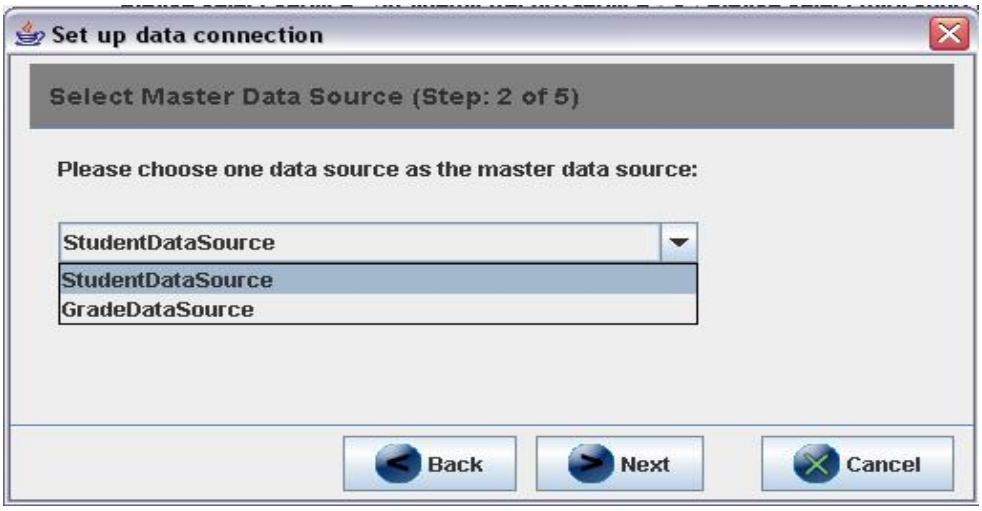


Figure 6.1 Wizard step for choosing master data source

As shown in Figure 6.1, the user has previously chosen two data sources from the data source selection UI, therefore, the wizard will prompts the user to choose one of them as the master data source.

Choosing more than one data sources will inform the Data Source Definition Tool that the current user tries to create a sub report using iReport. Typically, sub-report uses a different data source to the one master report uses when the report is being filled with data, however, in iReport, there can be only one active connection at a time, this means that any data sources that are used by the sub reports will therefore needs to be recreated at run time. Thus, when the user has selected the master data source, this data source will be passed alone to iReport as an active connection.

The following are the event flows after the user has chosen a master data source:

1. The user chooses one of the data sources as the master data source.
2. This data source is wrapped into an *IReportConnection,* and set to be the active connection, which is the connection that the master report uses.
3. The *TableModel* of any non-master data sources are stored into a parameter list of the master report, and this parameter list is being passed alone to iReport as the parameters of the master report.

(a) (b)

Figure 6.2 (a) active connection and (b) fields from active data source

Figure 6.2(a) shows that when user choose student as the master data source in the Data Source Definition Tool, the student data source has become the data source that the active connection uses for the master report. Figure 6.2(b) shows the available fields from the current active connection. In this example, there are studentID, firstName, and lastName from student data source.

Figure 6.3 parameters for master report

In iReport, each report has a list of parameters. Each of the parameters is a *JRParameter* object.

Figure 6.3 shows the parameter list for the master report. Notice at the bottom of the list, there is a parameter named: *SUBREPORT_GRADEDATASOURCE_ TABLEMODEL.* This means that the user has previously chosen two data sources from the Data Source Definition Tool and the grade data source is not chosen as the master data source (the user chose StudentDataSource in the wizard). Therefore, the **table model** of this data source is stored in the parameter list of the master report. This table model is used later to recreate the data source.

The entire process of creating report connection, passing table model into parameter can be better explained in Figure 6.4:

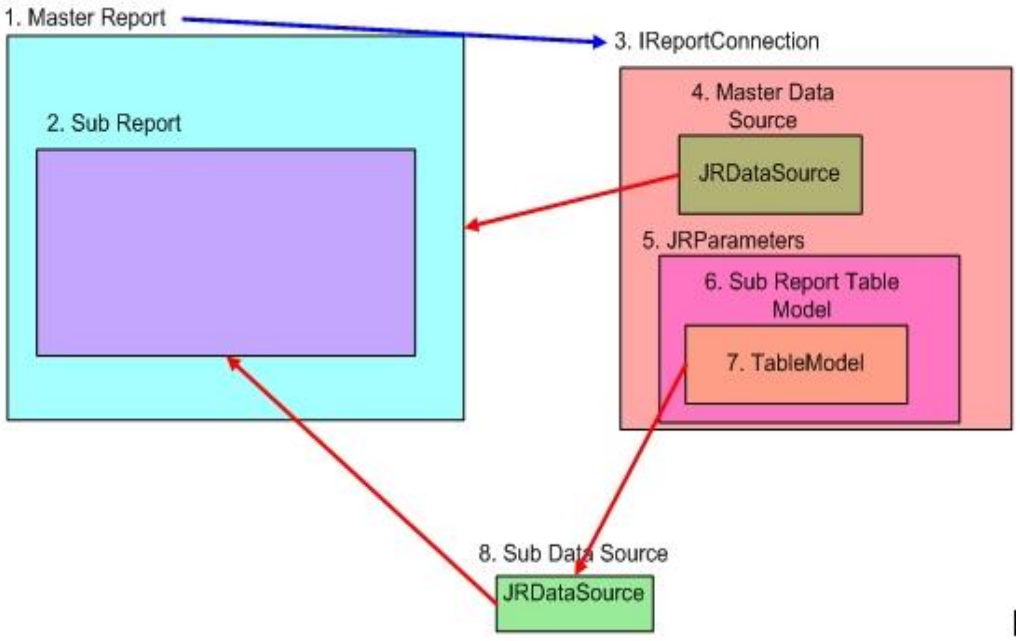


Figure 6.4 Setting up connection and data sources

- Our goal is to produce a master report (1) which has a sub report (2) embedded.
- Each master report has its active connection as a IReportConnection (3)
- Each IReportConnection (3) object wraps in a Master Data Source (4) which is of type JRDataSource (refer to Section 5.3). This JRDataSource therefore acts as the data source for the master report.
- Each IReportConnection object is associated with a list of JRParameters (6), this list records the information of the report attributes such as report page number, report date etc. One of the elements inside the list is a SubReportTableModel (6) which is of type TableModel (7). This table model is used every time to recreate the data source for the sub report (8) into a JRDataSource. This data source (8) is used as the data source for the sub report.

6.2 Setting up JRParameters

A *JRParameter* is constructed using the name of the parameter and the corresponding class type of that particular parameter. The actual value (or reference address) of each parameter in the list are stored in the first JRParameter: *REPORT_PARAMETERS_MAP* which is itself a *JRParameter*. This is a special parameter of type *HashMap*. Before the report is getting filled, this hash map is iterated through, and each value is assigned to the corresponding parameter if there is a match, otherwise, that parameter will have a null value. The follow diagram illustrates the structure of the parameter list:

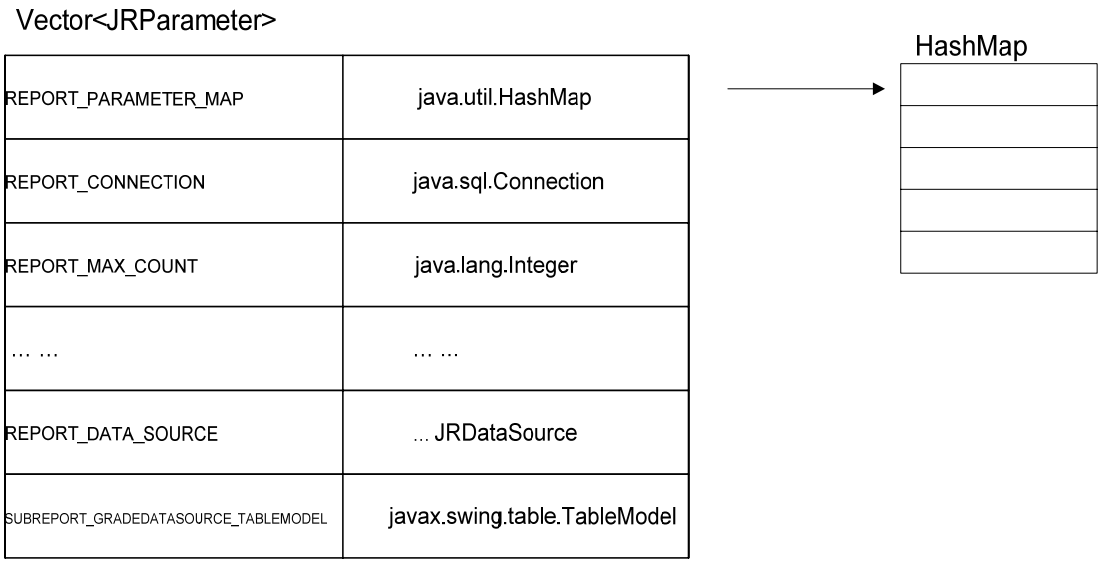


Figure 6.5 JRparameter list structure for master report

The sample of the above HashMap is shown in Figure 6.6

HashMap

REPORT_PARAMETER_MAP	java.util.HashMap@3f23a
REPORT_CONNECTION	null
REPORT_MAX_COUNT	15
… …	… …
REPORT_DATA_SOURCE	... JRDataSource@4a3d2
SUBREPORT_GRADEDATASOURCE_TABLEMODEL	javax.swing.table.TableModel@3ea2c

Figure 6.6 HashMap storing the actual value of *JRParameters*

6.3 Building a sub report manually in iReport

Before we explain how a sub report is built, we should first of all make sure how sub report is structured in iReport and clarify a few terminologies:

- **Master report**: A master report is a report that has its fields with a sub report element embedded, as shown in Figure 6.7 (a).
- **Sub report element**: A sub report element is a report element of a report. Whenever this element appears in a report, it means that this report is a master report and it has a sub report embedded.

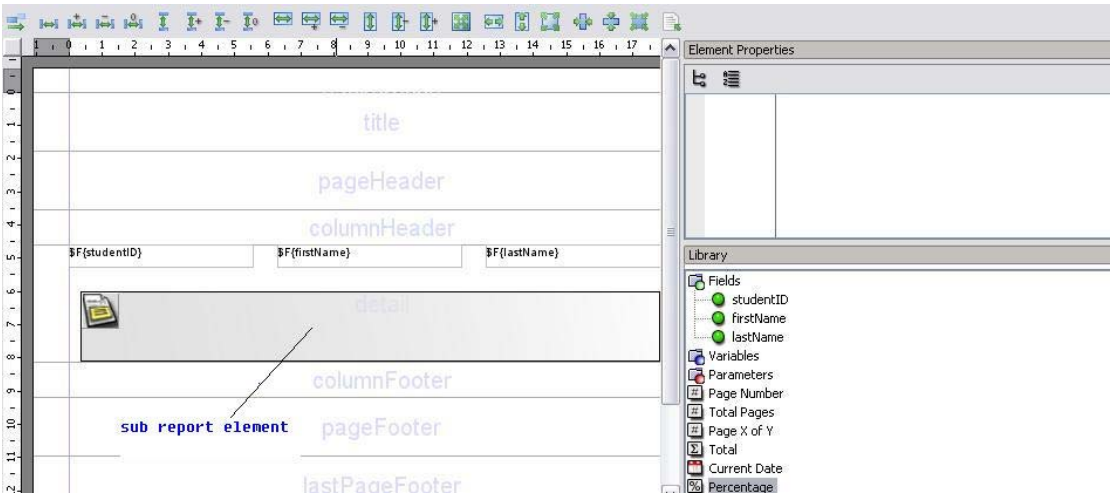


Figure 6.7 (a) master report

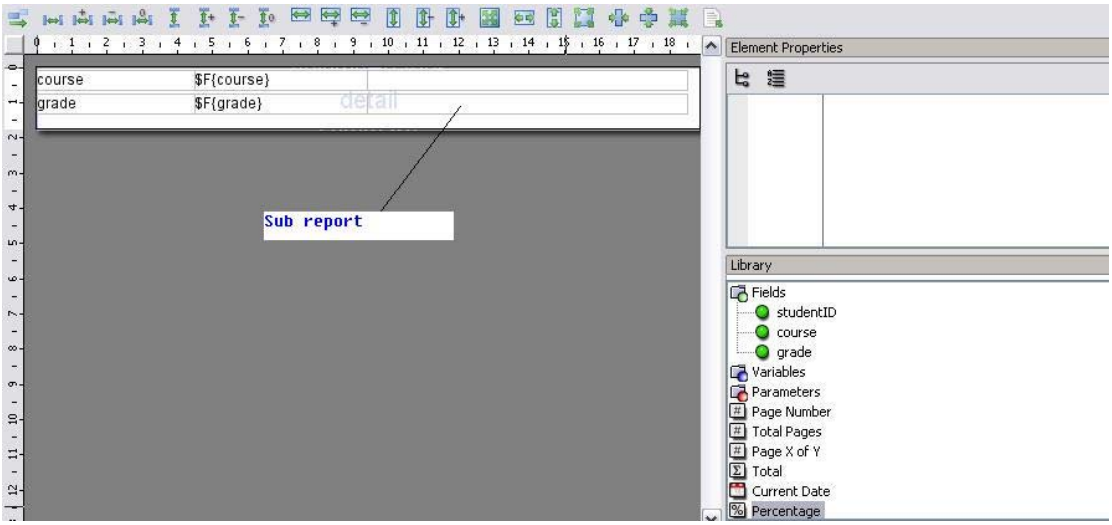


Figure 6.7 (b) sub report

- **Sub report**: Sub report is itself a report. It is always being referred to by a corresponding sub report element (Figure 6.7 (b)). In iReport, sub report only has detail band which has all the fields of the sub report. This is because a sub report is always being embedded inside a master report.

In order to build a sub report in iReport environment, a user needs to complete the workflow as described below (a live demo on how to create a sub report in iReport, refer to http://ireport.sourceforge.net/swf/Subreport_viewlet_swf.htm):

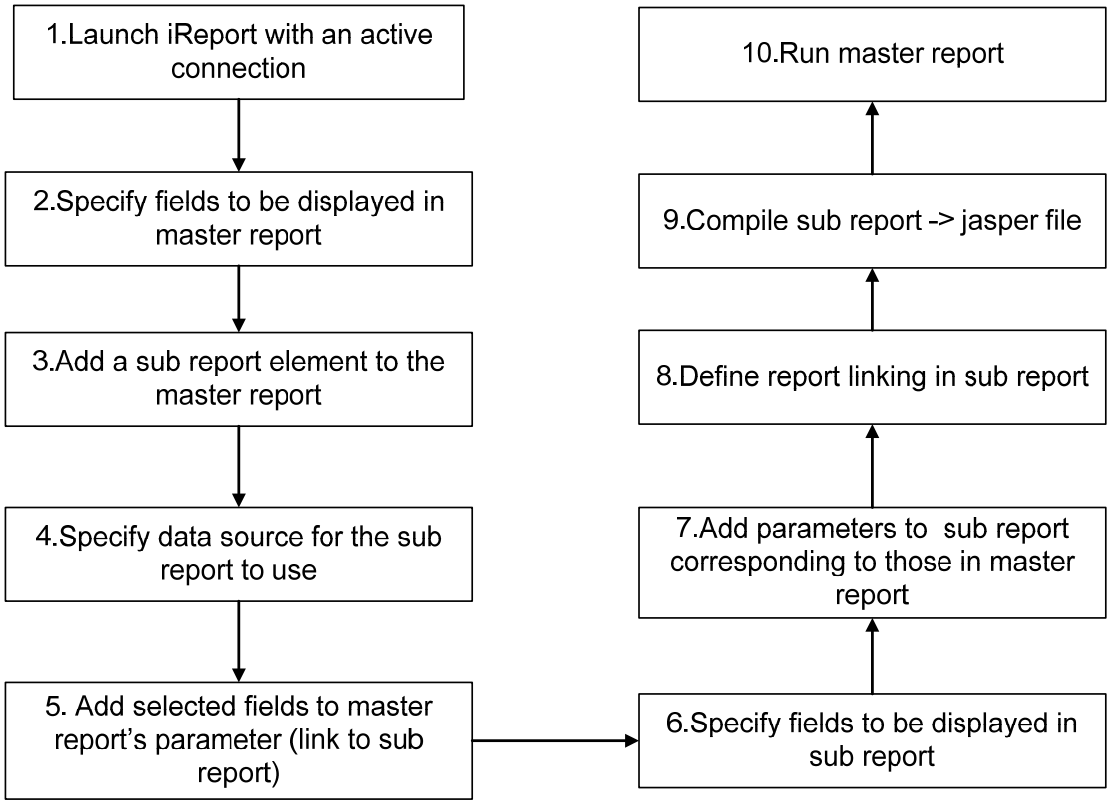


Figure 6.8 Workflow on building a sub report manually

There are a few important issues we have to emphasize here:

- Step 4 is done by defining a Data Source Expression by reusing the TableModel of the sub report which was wrapped in IReportConnection (refer to Section 6.1), the data source will be recreated every time a sub report is generated. Data Source Expression is discussed in more detail in Section 7.2.
- Step 5 is done by adding fields into the parameter list of the sub report element. This list is passed alone to its sub report. Another word, the sub report of the current master report will have access to every element in the list. This is discussed more in Section 7.3.
- Step 2 and 6 are done by dragging and dropping fields from the field lists to the report design panel. This feature is provided by iReport.
- Step 8 is done by defining a filter expression in the sub report. This is discussed in more details later in Section 7.4.

6.4 Data Source Expression

Since iReport can only have one active connection at a time, therefore, when we are filling the sub-report with data, we need to create the data source for sub report at run time. The recreation of sub report data source is done through constructing a **Data Source Expression** provided in iReport. The following diagram is a screen-shot from iReport when the user is setting up the connection for sub report manually.

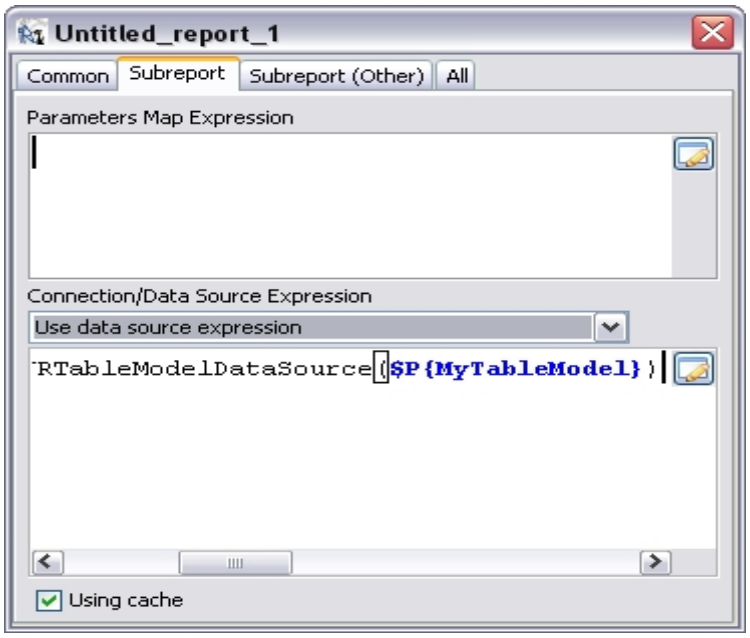


Figure 6.9 Set up connection for sub report

As we previously discussed, iReport uses *JRDataSource* when building the report, therefore, we need to create a *JRDataSource* object in order to fill the sub report. This *JRDataSource* is recreated every time the report is filled using the table model for that particular data source. The table model has been saved as a parameter of the master report that the current sub report belongs to. The following screen-shot is the data source expression editor, where a new *TableModelDataSource* is created using the table model:

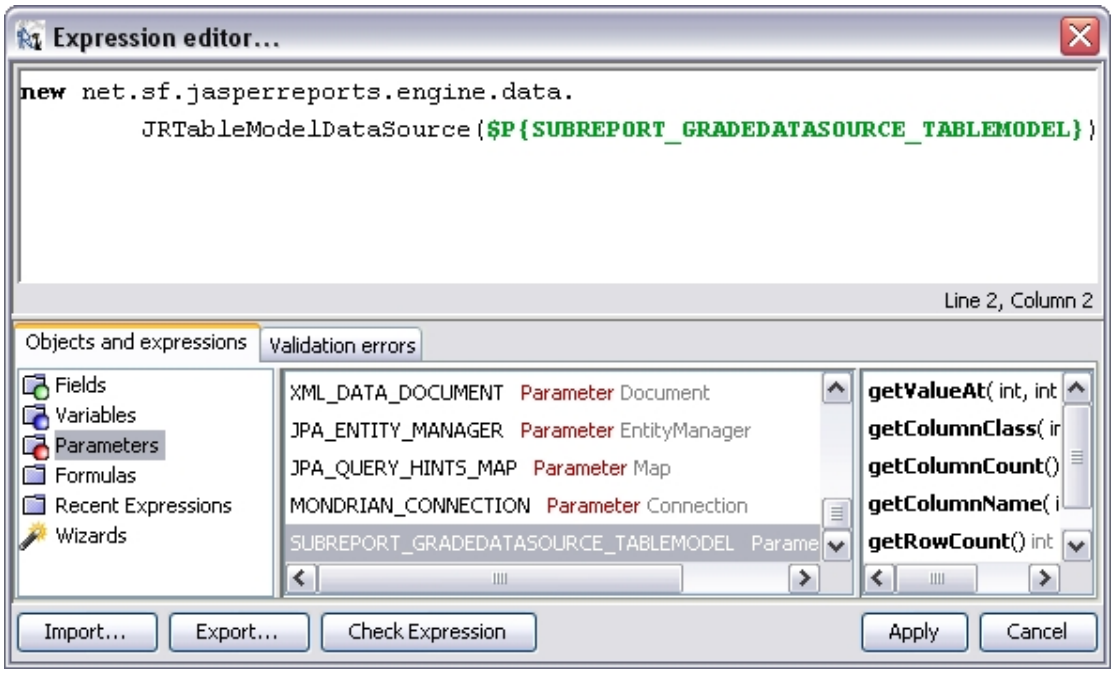


Figure 6.10 Data Source Expression Editor

In this example, we are building a *JRTableModelDataSource* using the table model

passed as the parameter from the master report parameter list (shown in Figure 6.10).

This **Data Source Expression** is effectively a java statement that creates a new JRTableModelDataSource object every time the sub report is built. Therefore the master report is filled with the JRDataSource embedded in the active connection, and the sub-report is filled with this newly created JRDataSource created using this data source expression.

6.5 Sub Report Parameters

In order to establish some linkages between a master and a sub report, we need to have some way to "communicate" between them. This is done through **Sub Report Parameters.** We can specify sub report parameters in the sub report element, to pass the field values from the master report to its sub report, as shown below:

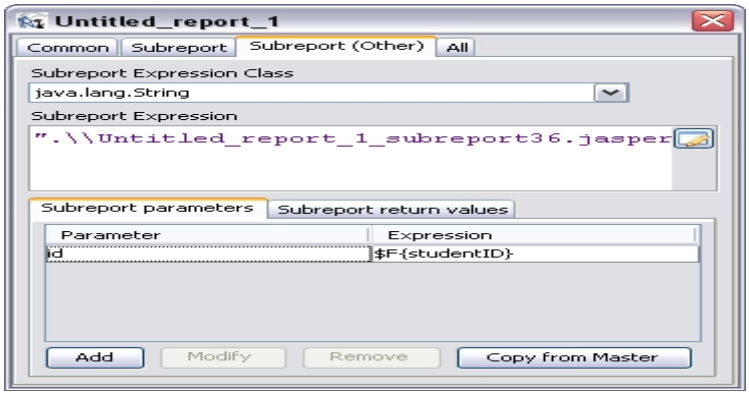


Figure 6.11 Sub report parameters

In the above diagram, we have assigned the "studentID" field from the master data source to a parameter called "id" to the sub report. The sub report can later access this field by referring to "id".

6.6 Object Level Join for Multiple Data Sources

When we build a sub-report, typically, there are some kind of relationship between the master report and the sub report. This is similar to a foreign constraint in relational databases. We cannot issue SQL commands to achieve joins among different data sources due to the fact that we use JRDataSource object as our custom data source in creating both master and sub reports; we therefore need to achieve the same type of join in object level.

Let us look at the following example, imagine we have two data sources displayed in *JTable* as follow, and keep in mind that these two data sources are both of type of JRDataSource in the context of iReport:

Student Data Source:

studentID	firstName	lastName
0000001	James	Bond
0000002	Bill	Gates

Grade Data Source:

studentID	course	grade
0000001	CS334	A+
0000001	CS335	A+
0000001	CS340	A
0000001	CS372	A-
0000002	ECON101	B+
0000002	INFORSYS220	C-
0000002	MATH208	A+
0000002	CS101	B-

And we want to build a report as shown in the diagram below:

0000001 James Bond

studentID	course	grade
0000001	CS334	A+
0000001	CS335	A+
0000001	CS340	A
0000001	CS372	A-

0000002 Bill Gates

studentID	course	grade
0000002	ECON101	B+
0000002	INFORSYS220	C-
0000002	MATH208	A+
0000002	CS101	B-

Figure 6.12 Sample sub report

As we can see in the above report, the student information is the master report and the corresponding student grade information for that particular student is displayed in the sub report. In order to achieve this, we have to do a join on the student ID between these two data sources when filling the sub report. If we were doing this in a relational database, we will effectively issue the following SQL commands in the sub report:

```
Select GradeTable.studentID, GradeTable.course, GradeTable.grade
From GradeTable, StudentTable
Where StudentTable.studentID = GradeTable.studentID
```

Figure 6.13 SQL for join between tables

However, we cannot execute the SQL commands against our data source simply because they are not tables in relational database. Therefore we need to find a similar way to achieve same goal as stated in the above SQL commands.

Remember here we are dealing with two JRDataSource objects, so if we translate the above SQL commands (executed to produce results for sub report) into plain English,

we mean:

Filter: the original GradeDataSource
Condition: every element in the resulting data source has the ID same as that specified in the master report.

Figure 6.14 Filter condition

Therefore, in order to achieve joins on data source object level, we reuse the **Filter Expression** feature provided by iReport. Effectively, a filter expression is a Java statement that specifies a condition to the sub report data source (JRDataSource), so that every time when data is displayed in the report, the data is filtered in advanced according to the condition specified in such an expression. We can view the idea graphically as follow:

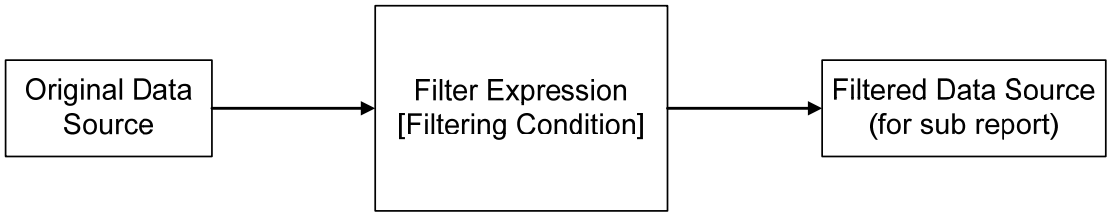


Figure 6.15 Applying Filter Expression

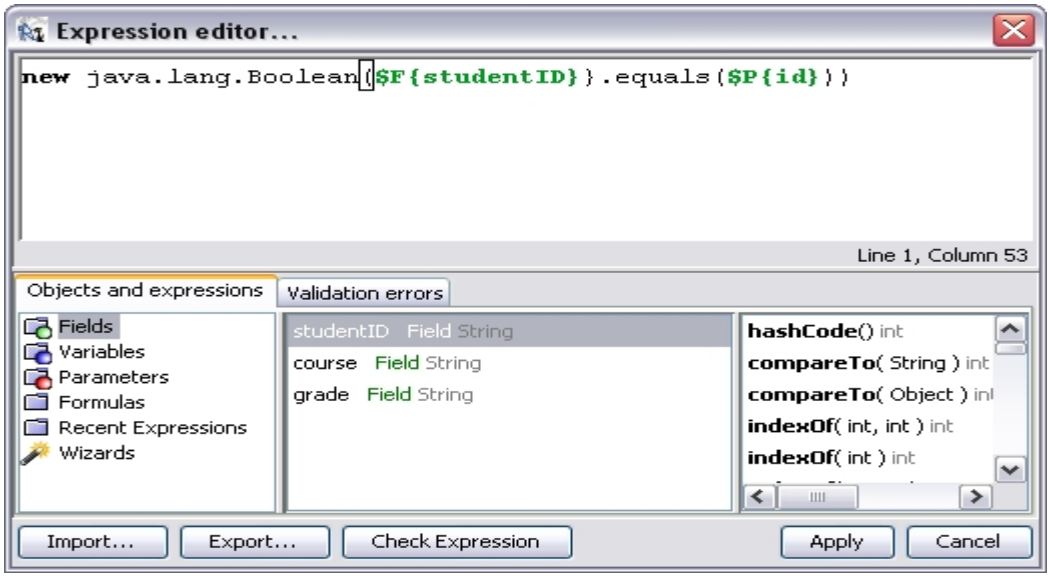


Figure 6.16 Filter Expression Editor

The filter expression shown in Figure 6.16 is defined for the Grade Data Source. $F{studentID} represents the studentID in the context of the data source for the sub report, namely Grade Data Source. $P{id} is the studentID from the Student Data Source (the master data source) that we have to previously passed from the master report as a sub report parameter (refer to Section 7.3).

When this expression is executed, it will filter out all rows from the grade data source wherever its StudentID is not the same as the current row displayed in the master report (refer to Figure 6.12).

7. Automating Building Sub-Reports

Building a sub report in iReport is quite a complex manual process. We are targeting at simple user experiences so that all the users need to do is simply specify data sources and then start designing the report, they are not required to have any prior knowledge on how iReport works. This has been identified as a major shortcoming after the first half of this project. Hence, simplifying user experience has been greatly addressed during the second half of the project. In this section, we are going to discuss how the process has been automated.

7.1 The Workflow

Our simplified user experience of building a sub reports can be summarized in the following workflow diagram:

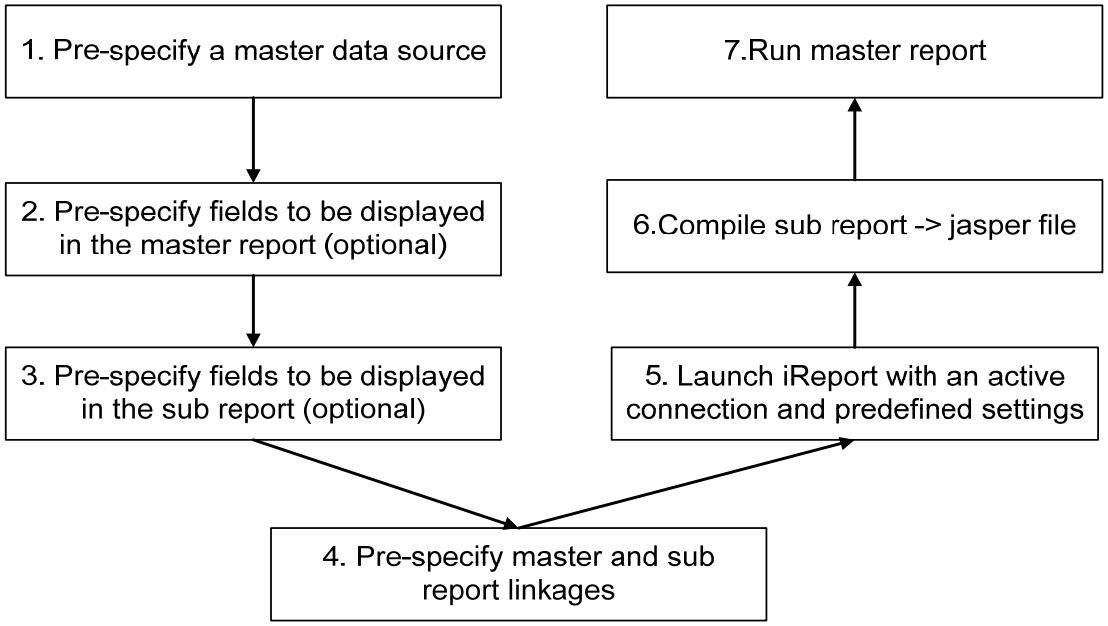


Figure 7.1 Workflow on building a sub report automatically

As we can see from the above workflow, our target here is to pre-configure the report settings, so that after iReport is launched, we can have the desired reports ready for the users to modify/design without them having to configure the sub report settings.

In another word, we only require the end users to perform the following:

1. Specify master data source in a simple user interface.
2. Specify fields to be displayed in the master and sub reports in a simple user interface.
3. Specify linkages between master and sub report in a simple user interface.
4. Design report by dragging and dropping fields in iReport environment.

As mentioned earlier, a wizard has been designed in our system to achieve simple user experiences. Now let us look at a few important aspects of those workflows in association with the wizard:

- When more than one data sources are chosen, user needs to specify a data source to be the master data source. This is already discussed previously in Section 6.1, selecting a master data source is the first wizard step.

- After selecting the master data source, the wizard will prompt the user to specify the fields to be displayed in the master report, as shown below:

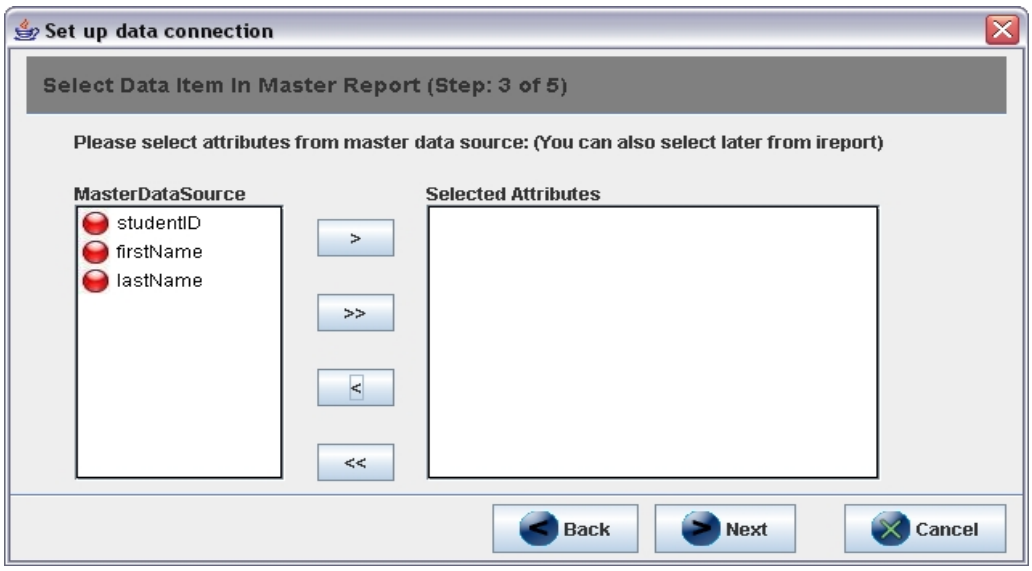


Figure 7.2 Pre-specify fields in master report

In the above diagram, the left list box has all the fields available in the master data source, the user can select the desired fields by clicking the ">" button (or ">>" to select all the fields) to add to the selected attributes. This will result those fields to be added automatically to the master report when iReport is launched. This step can however be skipped (select none), this is because the user can still add those fields into the report design panel (in iReport) by dragging them into the design panel.

- The next wizard step is to select the fields to be displayed in the sub report, as shown in Figure 7.3. In this case, the list box on the left lists all the available fields in the sub report data source, the user can choose desired fields to be displayed by adding them to the selected attributes list, which locates on the right on side of the wizard panel. This step can also be skipped due to the same reason described earlier.

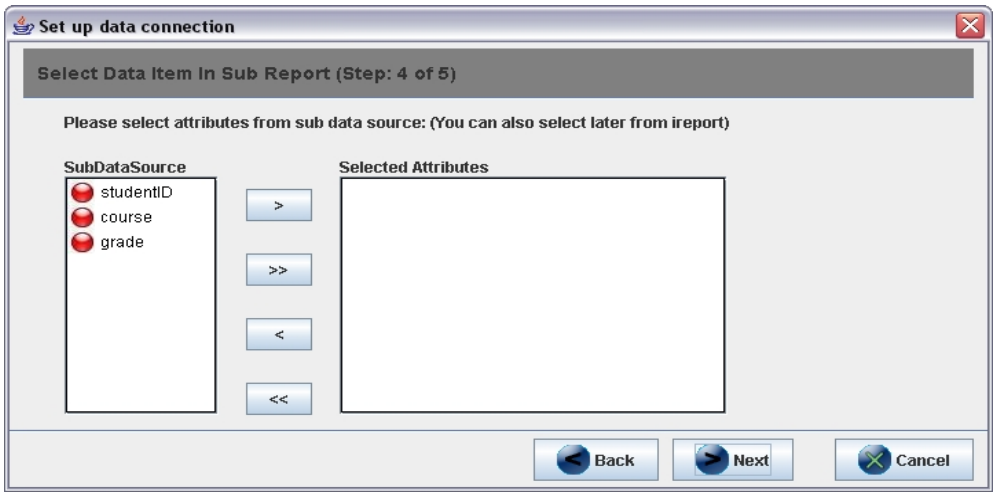


Figure 7.3 Pre-specify fields in sub report

- We also need the user to specify the linkages between the master and the sub report by selecting the connecting fields. The next wizard step provides the user a graphical view of doing this:

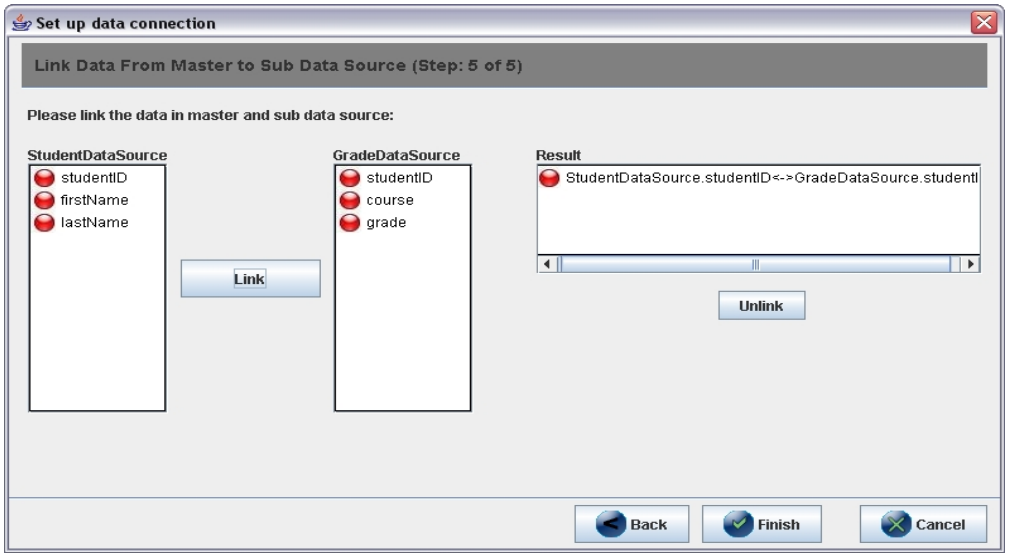


Figure 7.4 Wizard step for linking fields in master and sub report

This wizard step has three list boxes. The very left one lists all the available fields from the master data source; the middle one lists all the available fields from the sub data source. The user can select the fields to be linked together by choosing one field from the master data source and one from the sub data source followed by clicking the link button, this link is later converted into a filter expression described earlier to achieve object level join between different data sources. The list box on the right hand side shows all resulting links chosen by the user previously. It is displayed using a format similar to: *MasterDataSourceName.fieldName* <-> *SubDataSourceName.fieldName*

- After the user clicks the "Finish" button from the wizard panel, iReport will launched with those settings specified in the wizard loaded, including a pre-configured sub report. The user can now start designing the report by simply dragging and dropping fields into the report design panel, without needing to manually set up the sub report settings (if they are satisfied with the report layout and positioning, they can even compile and run the report). Figure 7.5 shows a pre-configured report.

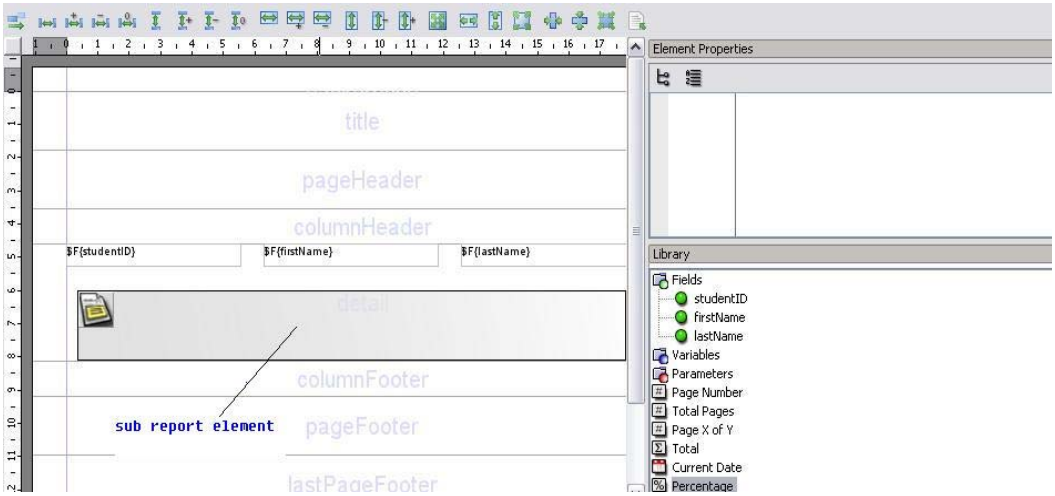


Figure 7.5 Pre-configured report

7.2 Implementation of Automation Process

In this section, we are going to look at how the automation process is implemented in detail. In order to make the process of building the sub report automatic, we effectively need to programmatically achieve the following:

- Add displaying fields into master and sub report programmatically by reading wizard results.
- Add sub report element into master report programmatically.
- Initialize data source expression programmatically.
- Initialize sub report parameters in its master report programmatically.
- Add the corresponding parameters into the parameter list of the actual sub report.
- Initialize filter expression programmatically according to the linkage specifications from the wizard results.

In order to achieve the above goals, we have come up with the class design shown in Figure 7.6. **Report**, **ReportElement** and **SubReportElement** are original classes from iReport framework. **ConfiguredReport**, **ConfiguredSubreport** and **ConfiguredSubReportElement** are the three new classes created to achieve our specific goal. Each of the three classes is configured in the following ways:

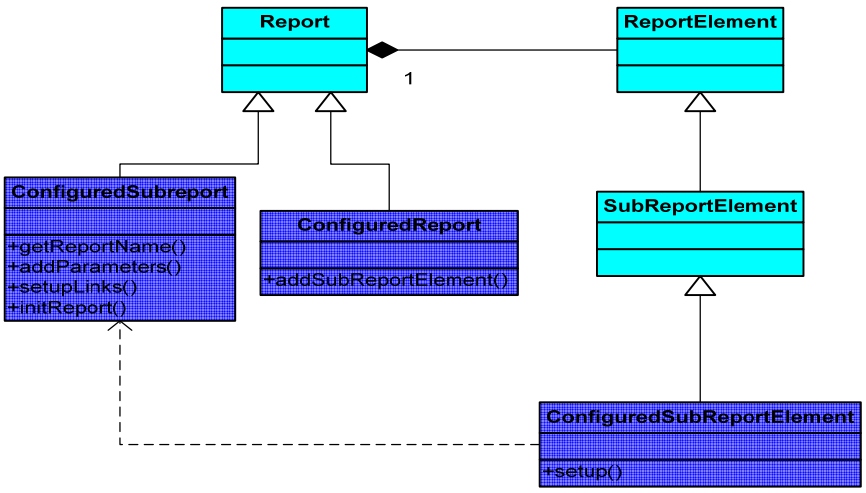


Figure 7.6 class diagram for automating sub report

- **ConfiguredReport**
  - ■ Displaying Fields

Fields are added in advanced into the report according to the user's previous selection from the wizard. The following code segment illustrates this:

```
for (int i = 0; i < selectedMasterAttributes.size(); i++) {
        String field = (String) selectedMasterAttributes.get(i);
        TextFieldReportElement re =
                new TextFieldReportElement( marginX + i*fwidth + 10,
                                            yPos + 10,
                                            fwidth - 20,
                                            20
                                           );
        re.updateBounds();
        re.setText("$F{"+field +"}");
        re.setBand(detailedBand);
        re.setMatchingClassExpression( "java.lang.String", true );
        elements.add(re);
        offset += 10;
    }
```

Figure 7.7 Code segment illustrating how to add predefined fields

In the above code segment, *selectedMasterAttributes* is an object of type Vector<Object>. It is one of the results from the wizard. This vector contains the names of fields selected previously by the user in the wizard which are to be displayed in the master report (refer to Figure 7.2).

Each field is constructed into a TextFieldReportElement and positioned accordingly in the report frame.

- **ConfiguredSubreportElement**
  - Initialize Data Source Expression

    We knew earlier that each sub report will use a JRDataSource as its underlying data source, and this data source is constructed every time when the sub report is produced using a data source expression, we therefore need to construct this data source expression programmatically, we do this by examining every parameter in the parent report until we find out a parameter is of type TableModel, we can then construct the data source expression and embed it into the expression editor as if we were doing this manually (refer to Figure 6.10). The following code segment shows how this is achieved:

```
        Vector reportParams = parentReport.getParameters();
        for (Iterator iterator = reportParams.iterator(); iterator.hasNext();) {
           JRParameter para = (JRParameter) iterator.next();
           String paraName = para.getName();
           if(paraName.startsWith("SUBREPORT_")
                                    &&
paraName.endsWith("_TABLEMODEL")){
           String connExpression =
     "new net.sf.jasperreports.engine.data.JRTableModelDataSource($P{"+
                                                              paraName+"}) ";
           MainFrame.getMainInstance().getElementPropertiesDialog().
                                              configureSetup(connExpression,
                                                          dataLinksInMaster);
           }

        }
```

Figure 7.8 Initialize data source expression

  - Add Sub Report Parameters

    We also notice from previous sections that we need to pass fields from the master report to the sub report for them to be linked together through sub report parameters. The following code segment shows how sub report parameter is added into the element.

```
for (int i = 0; i < dataLinks.size(); i++) {
   String linkAttribute = dataLinks.get(i);
   JRSubreportParameter srp = new
              JRSubreportParameter("param"+i, "$F{"+linkAttribute+"}");
   element.getSubreportParameters().addElement( srp );
   dtm.addRow(new Object[]{srp, srp.getExpression()});
}
```

Figure 7.9 Adding sub report parameters

In the above code fragment, we iterate through the dataLinks, which are the links specified by the user in the wizard dialog and refer to the link fields in the master data source. We construct a new JRSubreportParameter object for each of the link fields with the naming convention "paramN" where N is the number index of the field. The value of each of the parameter is the actual field name that is to be linked. Note that this naming convention needs to be strictly followed when constructing **ConfiguredSubReport** (next point to be discussed).

- **ConfiguredSubReport**
  - Displaying Fields

    Similar to that in the master report, the sub report also adds the fields into the report frame according to user selection from the wizard. The following code segment illustrates how this is done:

```
for (Iterator iterator = allAttributes.iterator(); iterator.hasNext();) {
        Object attribute = (Object) iterator.next();
        JRField field = new JRField((String)attribute, "java.lang.String");
        addField(field);
}
…
for (int i=0; i<fieldList.getModel().getSize(); ++i) {
    // FIELD
    JRField f = (JRField)fieldList.getModel().getElementAt(i);
    TextFieldReportElement
                        re = (TextFieldReportElement)detailField.cloneMe();
                        re.setPosition(…);
                        re.setWidth(fwidth);
                        re.updateBounds();
                        re.setText("$F{"+ f.getName() +"}");
                        re.setBand(detail);
                        re.setMatchingClassExpression(      f.getClassType(),
true );
                        getElements().addElement(re);
…
}
```

Figure 7.10 Code segment illustrating how to add predefined fields in sub report (some code omitted)

In the above code segment, *allAttributes* is again an object of type Vector<Object>. It is another result from the wizard. This vector contains the names of fields selected previously by the user in the wizard which are to be displayed in the sub report (refer to Figure 7.3). Each of the fields is constructed into a JRField, and added into the field list of the sub report.

Each of the field is then constructed into a TextFieldReportElement and positioned accordingly into the report frame.

- Adding Parameters

  As we mentioned earlier, sub report needs have access to the fields in the master report which are used as linking fields between master and sub report. All we need to do here is to assign that many parameters in the parameter list of the sub report and give them an appropriate name, the actual assignment is done in the process of constructing a **ConfiguredSubreportElement** (see previous point). The following code segment shows how the parameters are added into the parameter list of the sub report:

```
/**
 * Add the linking parameters for the sub report to link with master report
 */
private void addParameters(){
   for(int i = 0; i < dataLinks.size(); i++){
      JRParameter param =
                  new JRParameter("param"+i, "java.lang.String", false);
      addParameter(param);
   }
}
```

Figure 7.11 Code illustrating how to add parameters into sub report's parameter list

In the above code fragment, dataLinks are the result from the wizard (refer to Figure 7.4) where user specified the linking fields between the master and sub report. Here, we only count the name of the linking fields, and construct the same number of parameters in the sub report's parameter list. The naming convention of the parameter is "paramN" where 'N' represents the index number of the parameter, starting from 0. Note that this naming conversion has to be exactly the same when assigning the sub report parameters in the **ConfiguredSubreportElement** (described earlier)**.**

- Setting Up Report Links (Filter Expression)

  We also need to set up the links between master and sub report. We achieve this by programmatically construct a filter expression, the following code segment illustrates this:

```
private void setupLinks(){
   String expressionStr = "new java.lang.Boolean(";
   for (int i = 0; i < dataLinks.size(); i++) {
      DataItemLinkage link = (DataItemLinkage) dataLinks.get(i);
      String param = link.getSubData();
      String linkStr =
            String.format("$P{%s}.equals($F{%s})",
                                          "param"+i, param);
```

```
        if(i > 0) expressionStr += " && ";
        expressionStr += linkStr;
    }
    expressionStr += ")";
    setFilterExpression(expressionStr);
}
```

Figure 7.12 Programmatically construct Filter Expression

In the above code segment, DataItemLinkage is simply a wrapper class that wraps around each linkage specified by the user in the wizard, the getSubData() method return the name of the fields to be linked in the sub report data source. For example, if getSubData() is called on a link represented as: *DataSourceA.fieldA* <-> *DataSourceA.fieldB*, it will return fieldB, because the linking field name in the sub report data source is fieldB. We therefore iterate through the linking list and construct a similar filter expression shown in Figure 6.16 programmatically. Note that here we use the parameter in the sub report parameter list which was previously assigned, following the naming convention of "paramN".

The following sequence diagram shown in Figure 7.13 summarizes what we discussed in the above section.

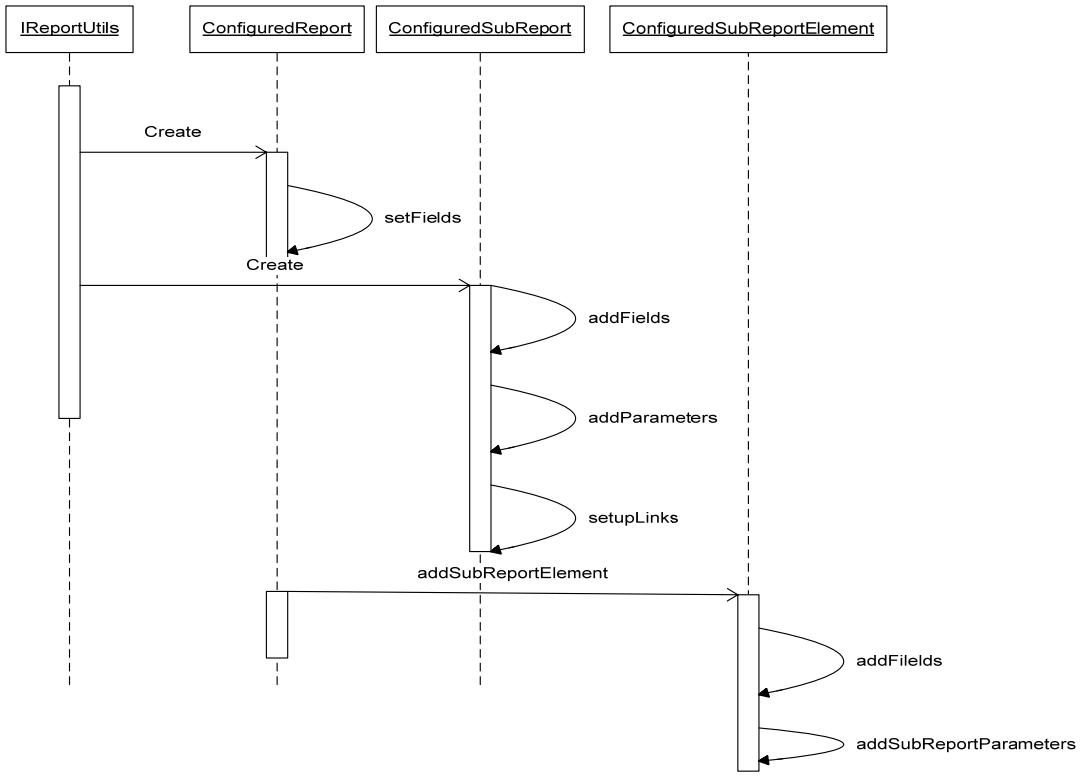


Figure 7.12 Interaction among three classes

8. Future Works

- **Using FlexibleTable For Displaying Data Sources**

In the initial plan of the project was to embed reporting facility into the existing framework. However due to the time constraint, this has not been completed yet. The major tasks that to be undertaken are to display the data source in FlexibleTable under Kiwiplan GUI framework instead of the current JTable implementation. A few concerns might arise from the conversion process:

- Define table attributes in core preference bundle (an XML file specifying table structure) rather than using TableModel straightaway.
- Using module class inside Kiwiplan framework to display table.
- A few more classes to enable FlexibleTable support for displaying data in FlexibleTable.

These tasks described above are not considered to be particularly difficult, and therefore require only minor effort either by me or by other developers from Kiwiplan to complete.

- **Saving The Report Definition**

After reports have been generated, we can only save the report structure into jasper file. We did not have any mechanism on saving the embedded data sources behind the report. This would not be desired if we want to reproduce a same report and viewing the same data in the report. Therefore, in the future, we should have some mechanism on saving the report definition effectively so that data sources can be retrieved back easily when the report is rerun.

- **Modified iReport Code**

While every effort has been made throughout the project to not modify the original code of iReport, I did come across a few places that I have not found out any way to solve the problem while not modifying the original code. This can be very inflexible due to the fact that our code will not work with a newer version of iReport if our modification has not been included in the new release.

Therefore, in the future, we need to identify those modifications and migrate them to our own code base. This can typically be done by subclass the original iReport classes.

9. Conclusions

I am actually quite excited when I am writing under this title, because it gives me a sign that this is the end of my final year Bachelor of Technology project.

The most valuable thing that I gained from the project is possibly not that I understand how to use iReport or JasperReports, but more importantly, the skills that I gained to develop new ideas, ways to solve problems, working with leading edge technologies. I guess this is the most important spirit for anyone in IT industry to survive.

Before this BTech project, I have completed an undergraduate project within the university as well as an industrial project with Kiwiplan already, but I have to say this whole year project really gave something I have never tasted. I am actually involved in the entire software development life cycle! From the first day I was sitting in Gareth's office trying to figure out what does the data source definition tool do to today I am writing the conclusion of this project, I feel I have learnt so much. I had gone to a completely wrong way on solving this problem, had experienced hard time when a problem cannot be solved for days, even months. Those are the unique and valuable experience I gained through this project.

I also realized that planning, designing are very important aspects throughout the development process. At the very beginning, I was struggling on how on earth iReport works, and how on earth I can make custom data sources to make them work within the context of iReport. I visited iReport forum, honestly, every single post about custom data source, I even tried to contact iReport authors when I really don't get what they are doing by looking at the code. Finally, I started to understand the problem, and had some basic ideas on how to solve them. Then I started to build some classes, gradually, more problems raised, I then tried to build more classes based on the original design, without scarifying what I already had. This kind of gradual approach is important, and it is also important to make flexible designs in every single system so that they are scalable, flexible, easy to maintain, easy to understand and easy to debug.

This project has really prepared me well to work in the real industry. I have had much more new thoughts on how to solve problems, in particular, some more design and analysis skills. I have enjoyed this project a lot and I also recommend everyone who is doing IT project either next year or a few years later to take your project as a valuable opportunity, because it might really give your something in return.

10. Acknowledgements

I wish to extend my great thanks to my supervisors: Gareth Cronin from Kiwiplan and Dr. Xinfeng Ye, for their continuous support throughout the entire project.

I would also like to thank Dr. S. Manoharan, the BTech coordinator, for providing lots of useful information on the BTech programme.

I would also like to thank everyone who helped me throughout the year.

11. References

1. JasperReport Homepage, "***JasperReports Documentation"***, Available from: http://jasperforge.org/sf/wiki/do/viewPage/projects.jasperreports/wiki/HomePage. Accessed Apr 2007

2. Elizabeth Montalbano, "***Eclipse Developers To Get Open-Source Reporting Tool. (Business Intelligence and Reporting Tool)***", Computer Reseller News Sept 13, 2004 p37.

3. iReport Home, ***"iReport Tutorial"***, Available from: http://ireport.sourceforge.net/tutorial1.html. Accessed Apr 2007