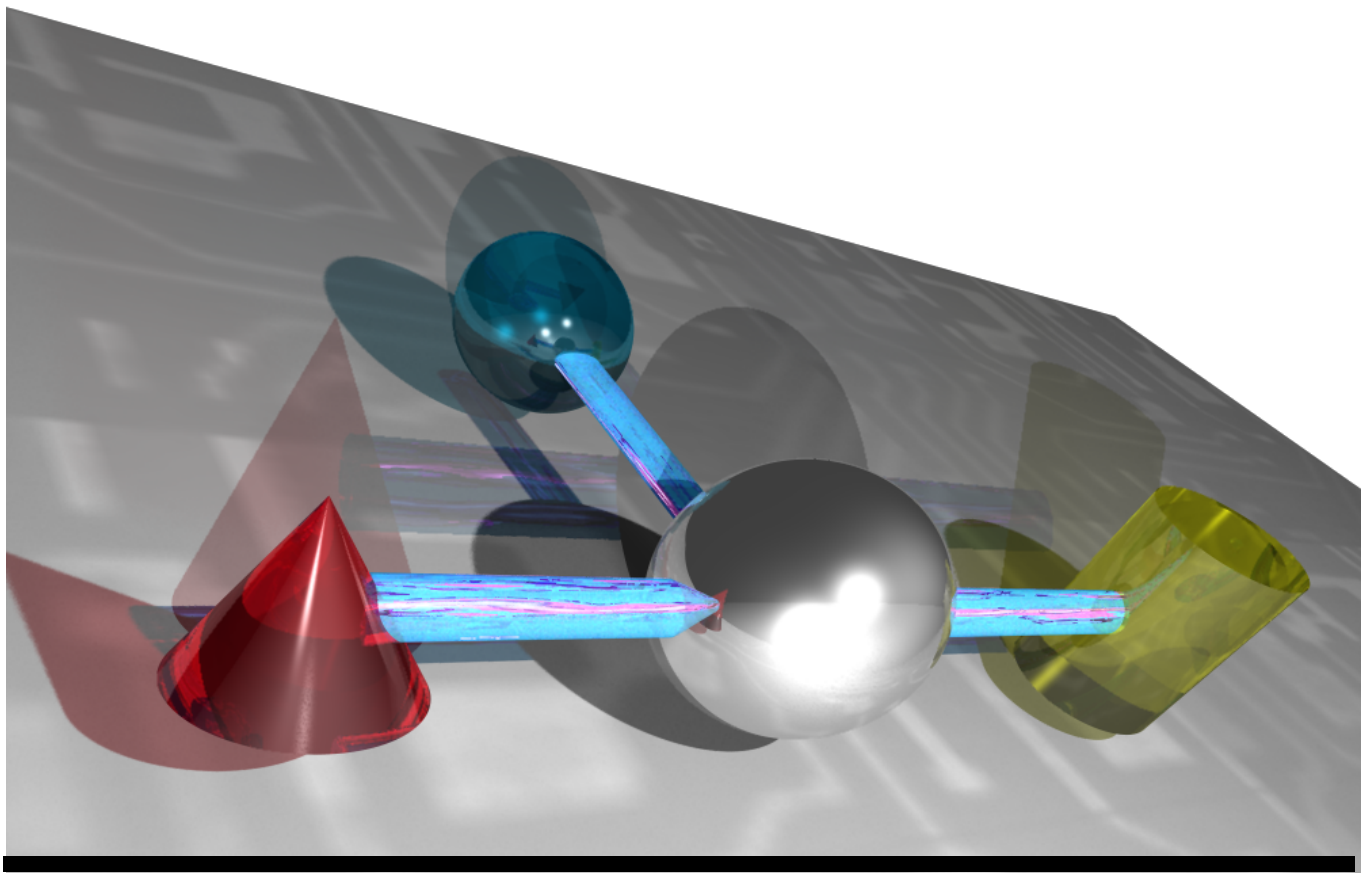


KFramework3

KF2tutorial3.0.doc

LGPL Free Software

Sofea conforming

sourceforge

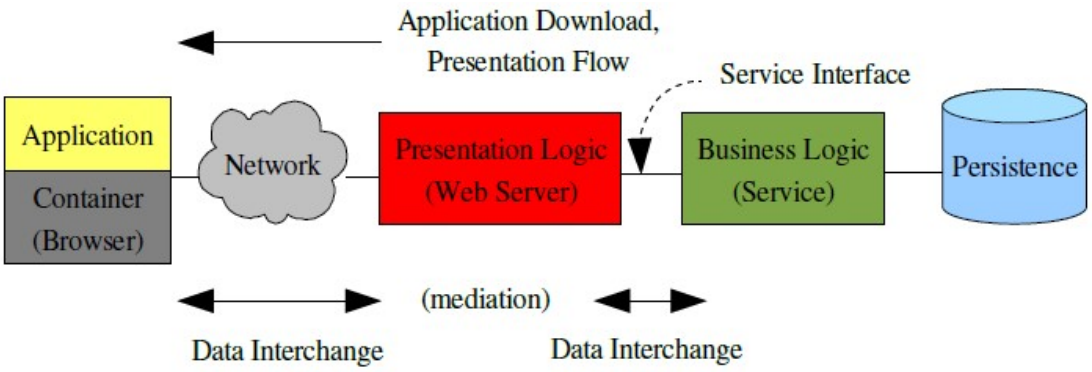
Contents

1 Introduction

The KFramework is an integral java/swing framework for complex business applications using the SOFEA/SOUI aechitectrure and Domain Driven Design.

1.1 SOFEA/SOUI Model

Today Web servers usually drive Presentation Flow in thin client applications, and a number of web frameworks have sprung up to augment basic web servers with such capability. Web servers also act as intermediaries between the Client and Business Logic during Data Interchange operations, often caching session data and performing other optimizations.



We will argue that this multiplicity of web server roles is a serious problem affecting end to end application architecture and impacting our ability to realize the full benefits of SOA.

1.1.1 Flaws of today´s thin based clients:

1.1.1.1 Flaw 1: No Mechanism to Ensure Data Integrity

Using traditional HTTP based interfaces the business data is transported embedded into a GET or POST request along with presentation markup code. This lack of respect for data as data is a fundamental characteristic of HTML over HTTP.

It has been with us since the inception of the web, but no one seems to mind. In a service oriented world, this is a serious shortcoming because strong data definitions form a very big part of the process of service enablement. If the Presentation Tier does not adequately enforce data integrity a lot of needless processing needs to occur near the Service Interface, where the two tiers meet. Wouldn't it be better if the Presentation Tier could adopt the data related mechanisms, if not the actual data definitions that the Service Tier uses?

1.1.1.2 Flaw 2: Coupling of Presentation Flow and Data Interchange

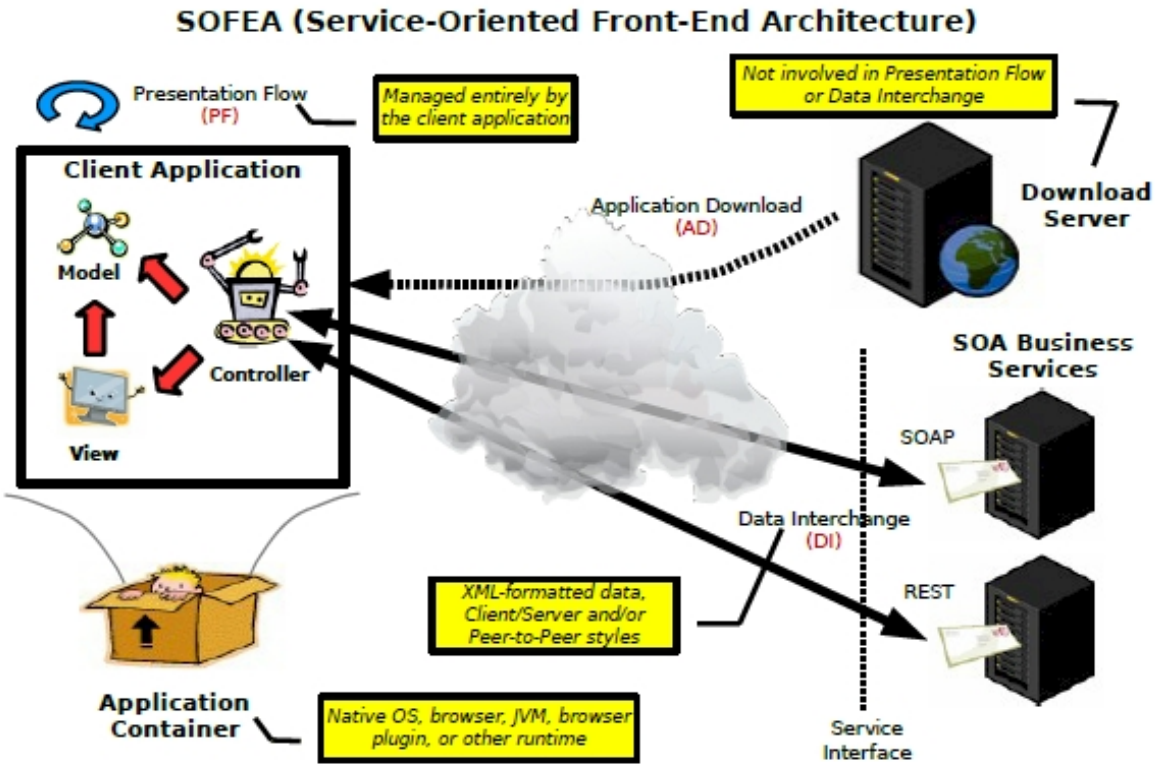
It is not possible to trigger a Presentation Flow in a web application without initiating a Data Interchange operation (e.g., a GET or a POST). More vexingly, every Data Interchange operation results in a Presentation Flow. It's a classic case of tight coupling between two orthogonal concerns.
It's only in very recent times, and thanks to AJAX, that our collective eyes have been opened to the possibility that Presentation Flow and Data Interchange can be decoupled. AJAX seems like magic!
Screens can display fresh data without blinking! We call this “greater interactivity” and praise it with high praise, whereas all that AJAX has really done is break the lockstep coupling between Presentation Flow and Data Interchange that should never have existed in the first place.

1.1.1.3 Flaw 3: Data Interchange Restricted to Request/Response Semantics

HTTP is a request/response protocol, which means that the Business Logic tier can only respond to requests from the Presentation tier and cannot initiate any unsolicited Data Interchange operation. However, there are any number of real world application use cases where this kind of behavior may be required, and these could be generically categorized as “event notification”.

One could view the entire client application as a “view” into a “model” that is held behind the service tier. When the model changes, the view must be notified. However, current thin client technology does not support server push. In most such cases, the workaround is to have the client poll the server periodically, converting “push” to “pull”. This is another example of a system that “sort of” works and thereby weakens the impetus for an overhaul. Having to implement server initiated notification

We propose a model that addresses what we believe to be the three flaws in current client technology. When these are fixed, there automatically arises a unified model for clients (the labels of “thin” and “rich” become moot) and integration with the Service Tier becomes seamless.



References:

Ganesh Prasad, Rajat Taneja, Vikrant Todankar (2007) *Life above the Service Tier*

1.2 Domain Driven Design DDD

Domain-driven design (DDD) is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts. The premise of domain-driven design is the following:

- Placing the project's primary focus on the core domain and domain logic
- Basing complex designs on a model
- Initiating a creative collaboration between technical and domain experts to iteratively cut ever closer to the conceptual heart of the problem.

Domain-driven design is not a technology or a methodology. DDD provides a structure of practices and terminology for making design decisions that focus and accelerate software projects dealing with complicated domains.

An important advantage of a domain model is that it describes and constrains the scope of the problem domain. The domain model can be effectively used to verify and validate the understanding of the problem domain among various stakeholders. It is especially helpful as a communication tool and a focusing point both amongst the different members of the business team as well as between the technical and business teams.

Tightly relating the code to an underlying model gives the code meaning and makes the model relevant.

If the design, or some central part of it, does not map to the domain model, that model is of little value, and the correctness of the software is suspect. At the same time, complex mappings between models and design functions are difficult to understand and, in practice, impossible to maintain as the design changes. A deadly divide opens between analysis and design so that insight gained in each of those activities does not feed into the other.

Therefore,

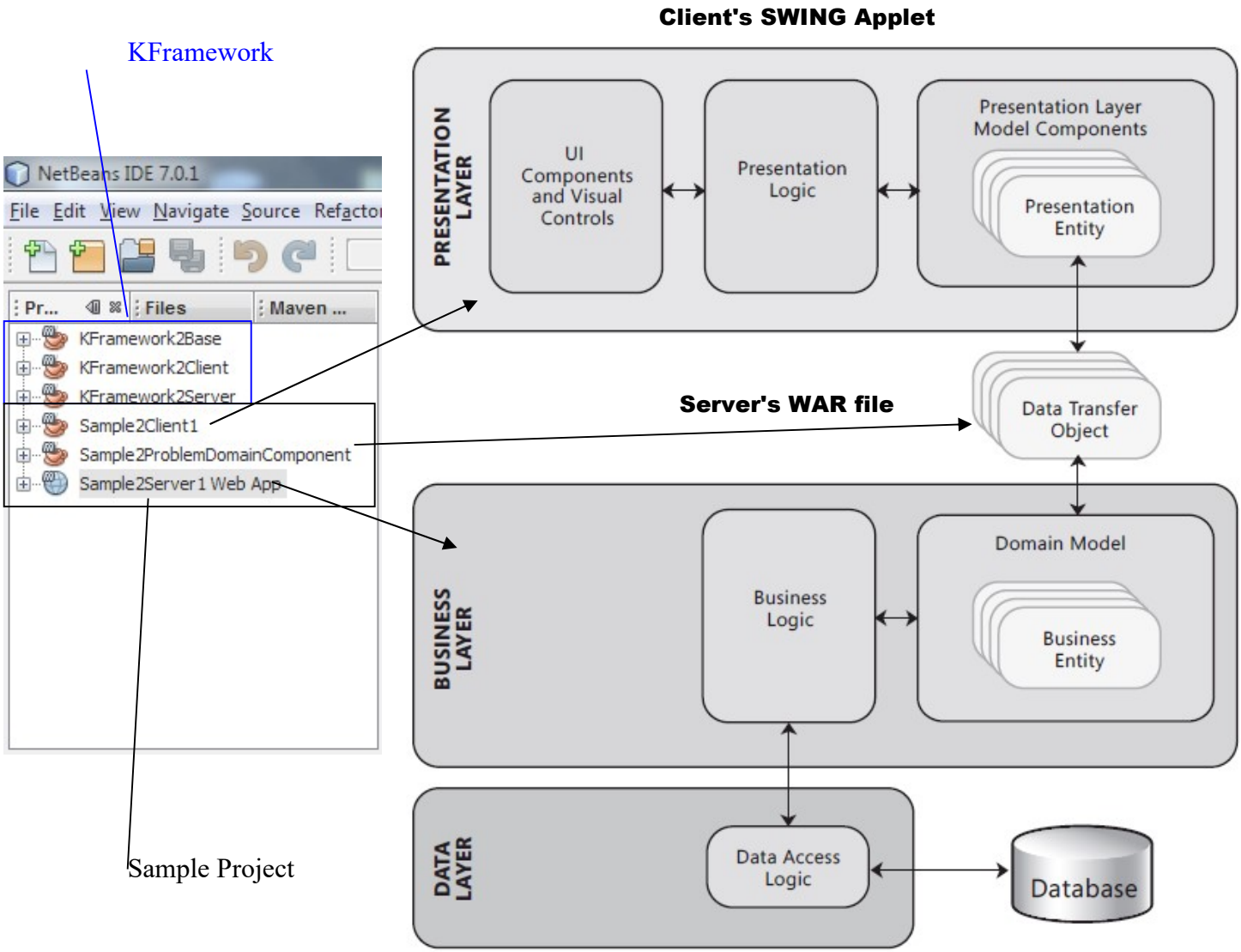
Design a portion of the software system to reflect the domain model in a very literal way, so that mapping is obvious. Revisit the model and modify it to be implemented more naturally in software, even as you seek to make it reflect deeper insight into the domain. Demand a single model that serves both purposes well, in addition to supporting a fluent UBIQUITOUS LANGUAGE.

Draw from the model the terminology used in the design and the basic assignment of responsibilities. The code becomes an expression of the model, so a change to the code may be a change to the model. Its effect must ripple through the rest of the project’s activities accordingly.

To tie the implementation slavishly to a model usually requires software development tools and languages that support a modeling paradigm, such as object-oriented programming.

Excerpted from Domain-Driven Design: Tackling Complexity in the Heart of Software, by Eric Evans, Addison-Wesley 2004.

1.3 Overview of Delivered Package



2 Supported Infrastructure and Configurations

2.1 Front End

The front end requires a JAVA enabled browser. The Framework has been tested with Firefox (UNIX/WINDOWS), Google Chrome and IE 7/8/9/10.

Operating systems tested include Solaris, Linux, OSX and Windows XP / Vista / 7.

2.2 Middle Tier

The middle tier requires a JEE5 complaint webservice / servlet container. Tested servers include:

1) Glassfish v2
2) Glassfish v3

Help needed to test more serves, and port to non J2ee servers.

2.3 DataBase Back-End

The KFramework does not require any specific DB server as long as it supports standard SQL, tested databases include:

1) ORACLE 7,8,9,10,
2) MS SQL Server 2005 and 2008.

Including the Express Versions.

No PL-SQL or Transact SQL code is required or used.

2.4 IDE tool

For best results we recommend **Netbeans 7**. It has been compiled with Eclipse but more testing is required.

For quickest setup, download the preconfigured Netbeans 7 & Glassfish 3.x bundle at: http://netbeans.org/downloads/

Help needed to build eclipse release and adapting to SWT and JAVA FX.

3 Getting Started

3.1 Setting up and trying the sample project

3.1.1 Get all the necessary components

1. Download and setup the Netbeans 7.x / Glassfish server bundle:

➲ http://netbeans.org/downloads/

2. If you don't already have a database we recommend the Oracle Express which is production quality and royalty free, even for production:

➲ http://www.oracle.com/technetwork/database/express-edition/downloads/102xewinsoft-090667.html

3. Download the latest KFramework3.X delivery .ZIP file from source forge:

➲ https://sourceforge.net/projects/k-framework/files/kframework3.0/

4. Find the database import or SQL script in the root of the delivered package. Import the database files to user "sample". If you want to try another database, then use the SQL scripts also included.

The delivery decompresses to 6 folders:

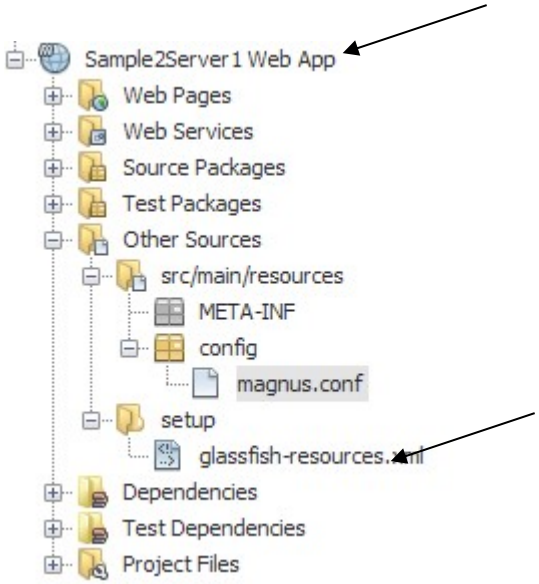
1.	KFramework3Base	*KF3 base library*
2.	KFramework3Client	*KF3 Client Library*
3.	KFramework3Server	*KF3 Server Library*
4.	Sample2Client1	*Demo Client*
5.	Sample2ProblemDomainComponent	*Demo Business Entities*
6.	Sample2Server1	*Demo Server*

Each folder is a netbeans maven project that provides an architectural component as depicted before in section *Overview of Delivered Package.*

1. Open all the projects in Netbeans:

3.1.2 Configure the sample project server

1. Locate the database configuration files. Open the sample project's server project, and locate the datasources file:

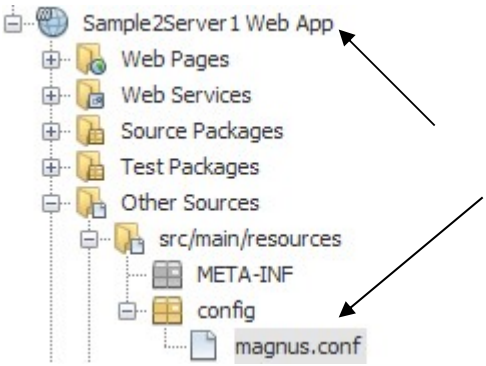


2. Configure the *"KFRAMEWORK_DATASOURCE3"* datasource. Examples are provided for Oracle and SQLServer. If you imported the database dump file for oracle in the schema sample with sample password you might not need to change anything, otherwise verify the datasource settings.

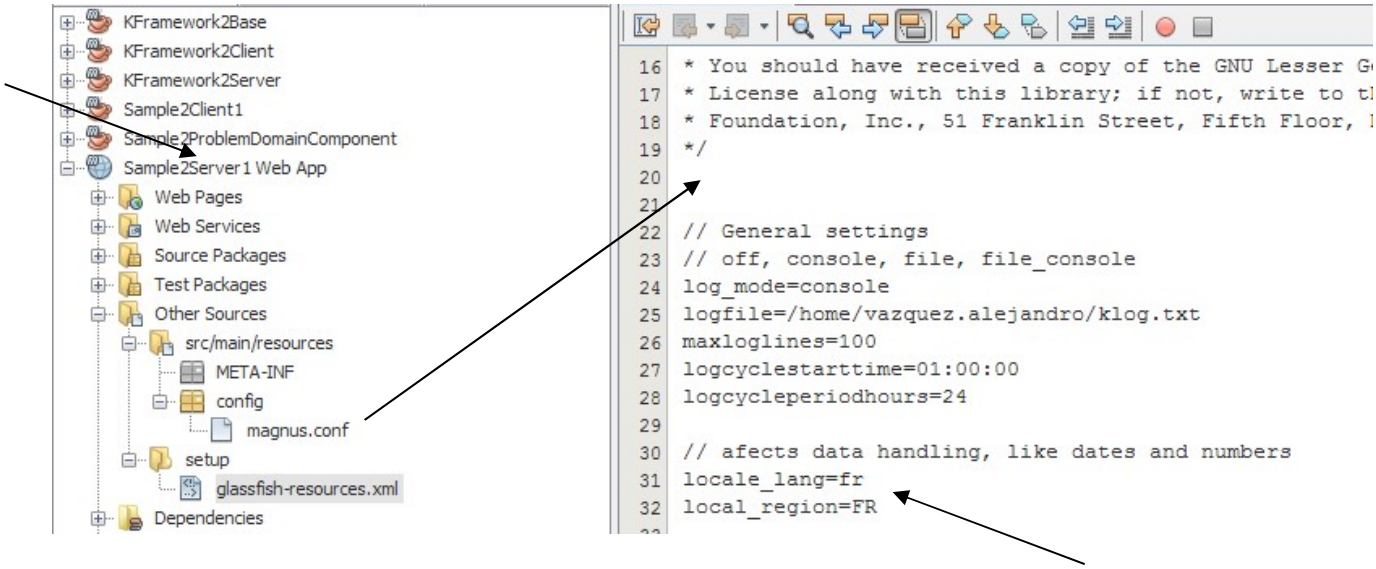
Make also sure that you installed the JDBC driver of your selected database in the .ext folder of your application server. For glassfish:

```
..\glassfish-X.X.X\glassfish\domains\{DomainName}\lib\ext
```

3. Locate the server's magnus.conf file:



4. Open and configure the magnus.conf. There are several things that you can configure in this file, but for the purposes of this demo just make sure the locale is setup properly.

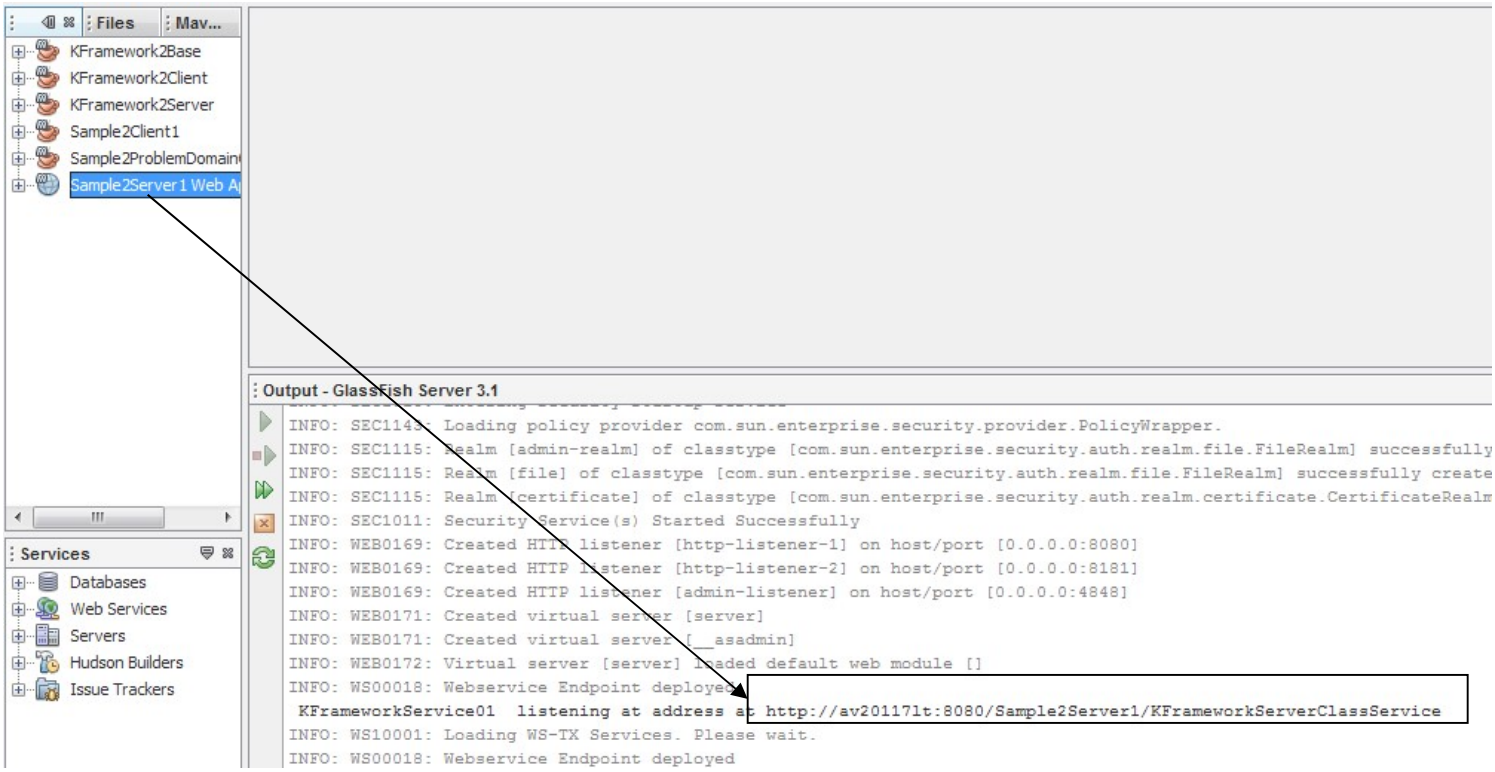


3.1.3 Compile all the modules

1. Compile projects in this order:

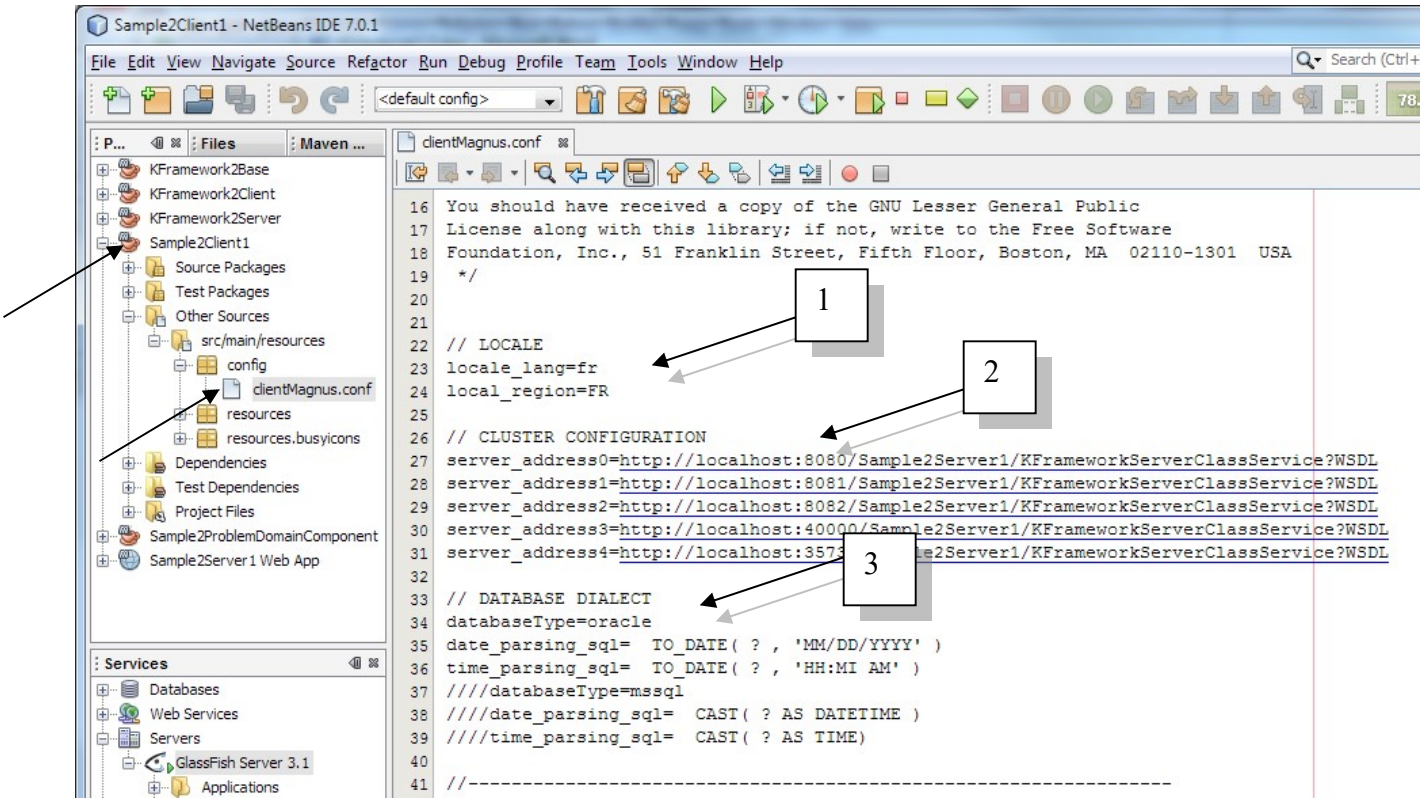
1.	KFramework3Base	*KF3 base library*
2.	KFramework3Client	*KF3 Client Library*
3.	Sample2ProblemDomainComponent	*Demo Business Entities*
4.	KFramework3Server	*KF3 Server Library*
5.	Sample2Client1	D*emo Client*
6.	Sample2Server1	D*emo Server*

2. Run the Sample Server and note the deployed end point, which might vary depending on your setup. This will now be used to configure the client.



3.1.4 Configure the sample project client

1. Open de clients configuration file, and configure user options:



1 Set the correct locale, and make sure it's the same as the server's

2 Set the servers addresses. For this test adjust only server_address0 with the endpoint published when testing the server, as explained above.
The KFramework supports server clustering with out needing the server's native clustering support. You can have as many servers as you want, even in different geographic locations or in different platforms. Only requirement is that all nodes "see" the same database.
You can delete the extra server entries or leave them; the framework will treat them as unavailable or standby servers with no impact to the user.

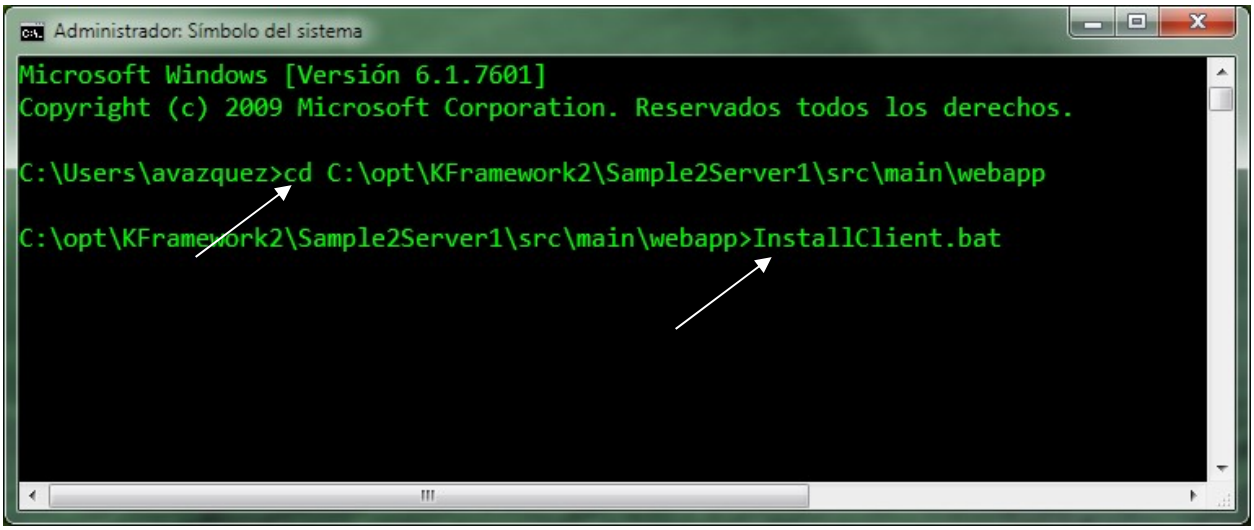
3 Set the database dialect and date handling functions to match your database

2. Recompile the client project

3. Install the client applet in the server project for web deployment.

Installing the client applet means signing all libraries and deploying to the server's website.

1. Locate the server web folder
2. Execute the client applet installation script for Windows or Unix




Once an applet is signed, using the above procedure, the applet will have access to almost all java functionality with out security pop-ups and the like. Only the first time that an applet is run in a client will java pop a dialog asking if you trust the certificate. It will then install the certificate in the computer and never ask for it again.

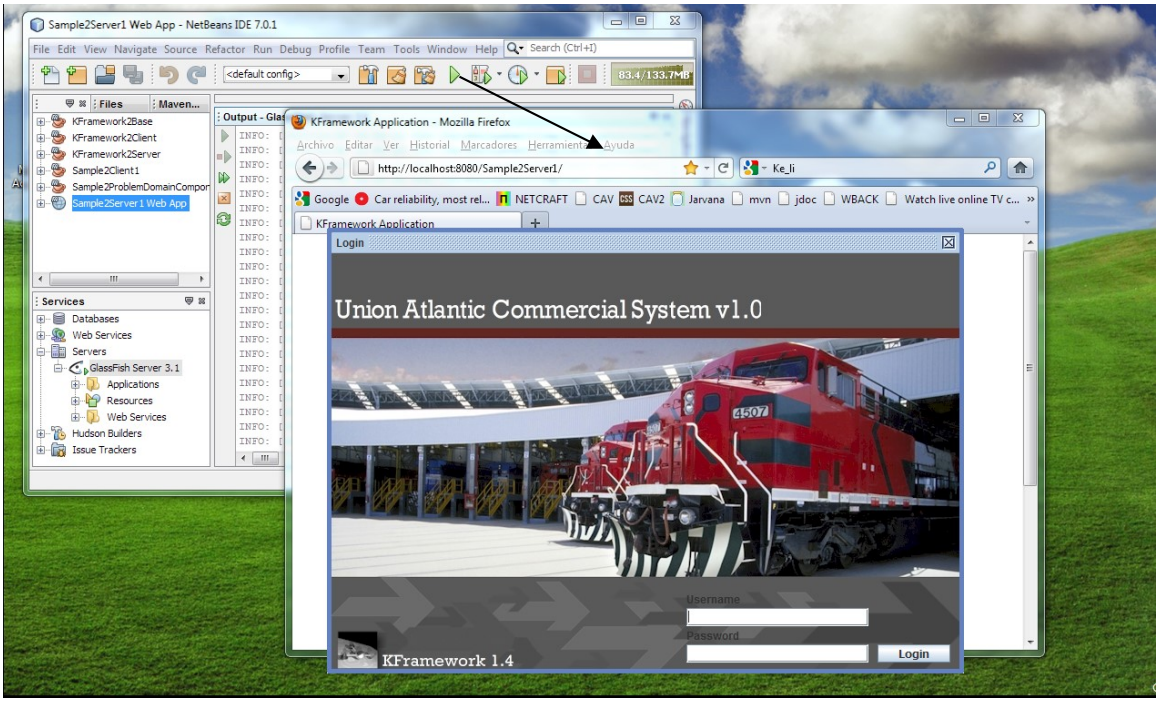
Applets are very restricted, ONLY if they are not signed. A signing certificate is provided. If you have your own, edit the script. If you have a certificate recognized by a security authority you will not even have the initial dialog.

4. Recompile the server to package the client for web deployment.

3.1.5 Give it a try !

Make sure:

1. You configured the server as explained above
2. You configured the client as explained above
3. Ran the clients install script as explained above
4. Recompiled the server again to replace the provided applet with the newly configured and signed one
5. Re ran the server, the application shall start and bring up a browser with the client application automatically:

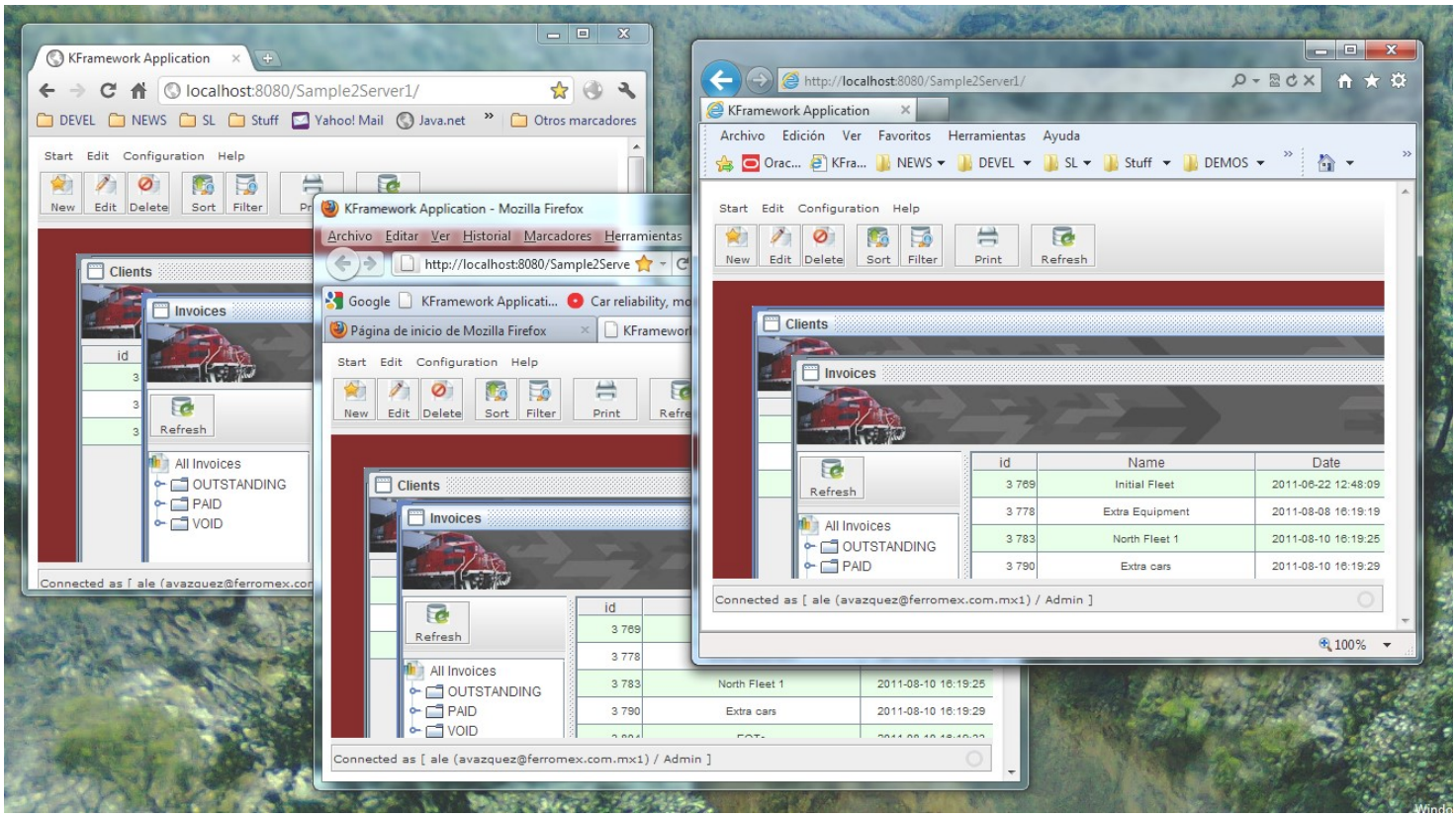


If you are running from the provided database dumps, log in with user/password: ale/ale. Otherwise, add a user to the **SYSTEMUSERCLASS** table.

Make also sure that you installed the JDBC driver of your selected database in the .ext folder of your application server. For glassfish:

```
..\glassfish-X.X.X\glassfish\domains\{DomainName}\lib\ext
```

Forget about compatibility issues with HTML / JAVASCRIPT…. Code with confidence _ONCE_ for Chrome, FIREFOX, IEXPLORER, SAFARI…



4 Tutorial: Building a minimum project from scratch

In this section we will walk you through making a project from scratch. The project will be as close as to the real thing as possible.

4.1 First, make a domain driven design

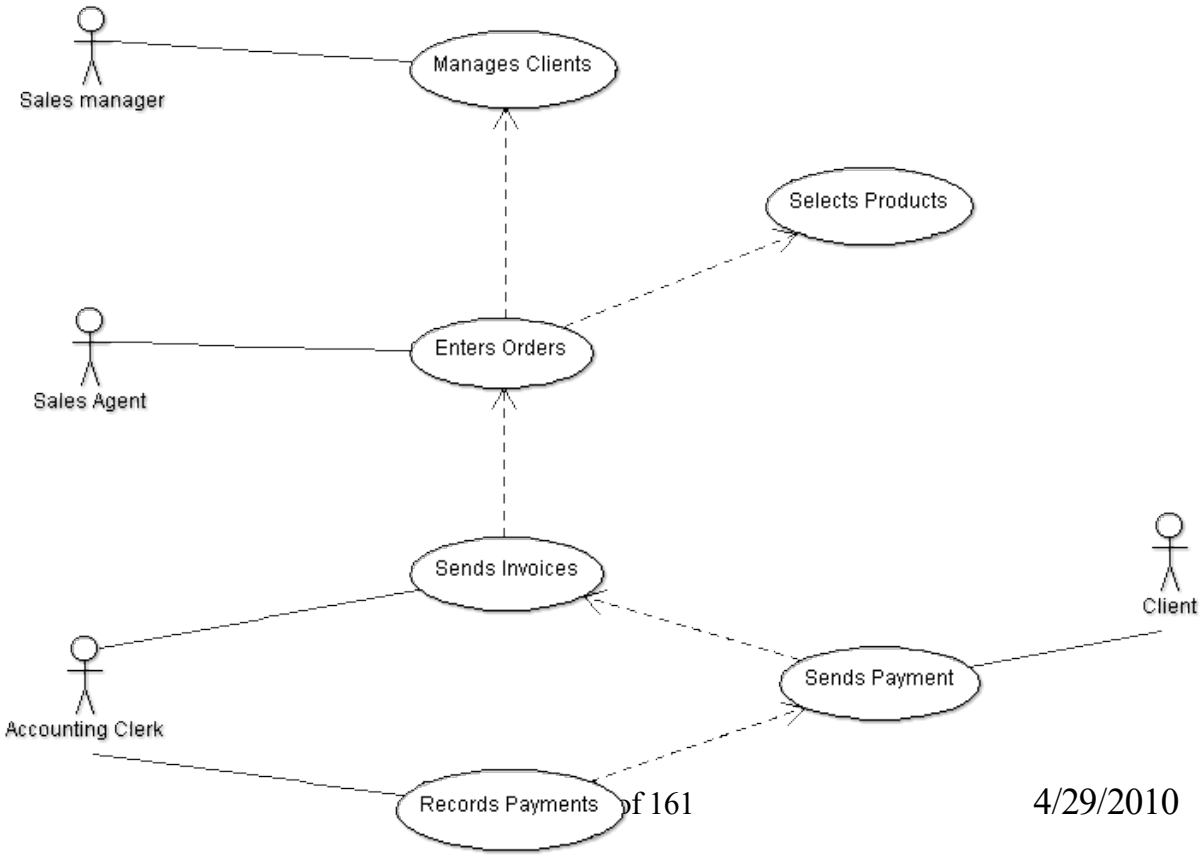
We will now present a demo design. We will quickly and in simplified form go through all the minimum analysis and documentation steps you normally follow in OO projects. This is important so that you see how Domain Driven Design will help you define and sequence all subsequent steps of construction.

4.1.1 Gather Requirements using UML use case diagrams

First let's come up with a business problem. All projects are justified because of a sensed performance gap in a business process. The most important thing is to have the scope of this process very well understood, and to make sure all efforts are only directed towards solving this problem and no other.

Let's assume that, after several interviews, we agree the following business cases with our stakeholders:

Demo Project's Use Case Diagram

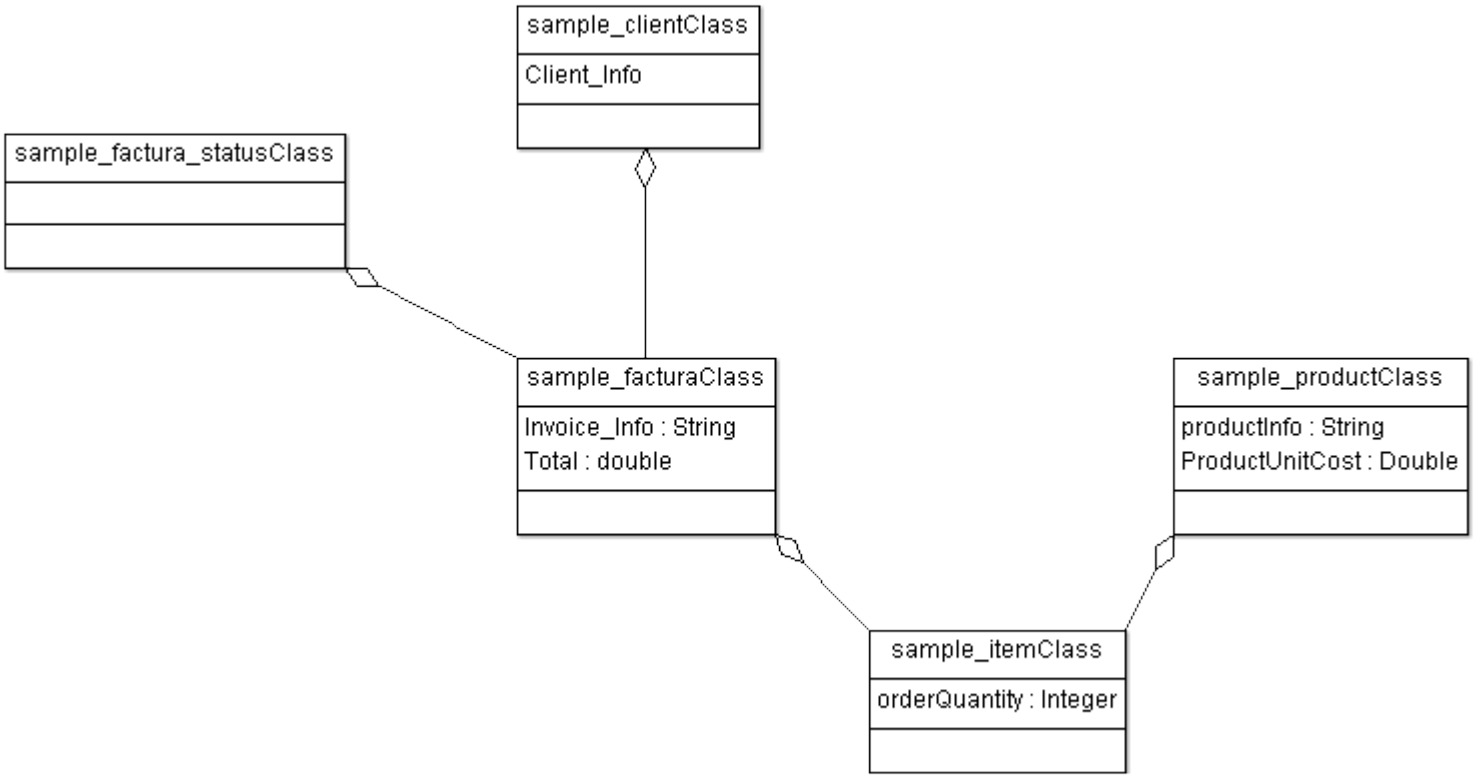


1. **Sales Manager:** Adds and Removes Clients form the Authorized Clients List
2. **Sales Agent:** Takes purchasing order from clients, which shall be found in the authorized list, and enters orders based on the product catalogue.
3. **Accounting Clerk:** Monitors Status of Orders for Invoicing and Payment Follow-up and record keeping.
4. **Client:** Puts orders and sends payments based on entered orders

4.1.2 Model the Domain using UML class diagrams

Now we take the business case model and find the domain objects, which will clearly delimit the scope of our project. All domain objects / entities live inside the circles of the use case diagram, we need to identify them, and then, specify their relationships between each other. This will be our domain model. How successful this project will be is directly related to how accurate this model is. Are we mixing two entities in one? Are we missing a relationship? Are there more domain objects that we missed? These are questions we need to keep doing all the way until the end of the project, refining the model step by step. Never bet a whole project on the initial definition of the domain model, use an iterative approach.

Demo Project's Problem Domain




Note that statuses have invoices (facturas). Statuses are required first and then invoices are attached to them, not the other way around.

Also note that link objects, like the invoice items, are treated as full objects with their own primary key, never using a complex key combining the foreign keys.

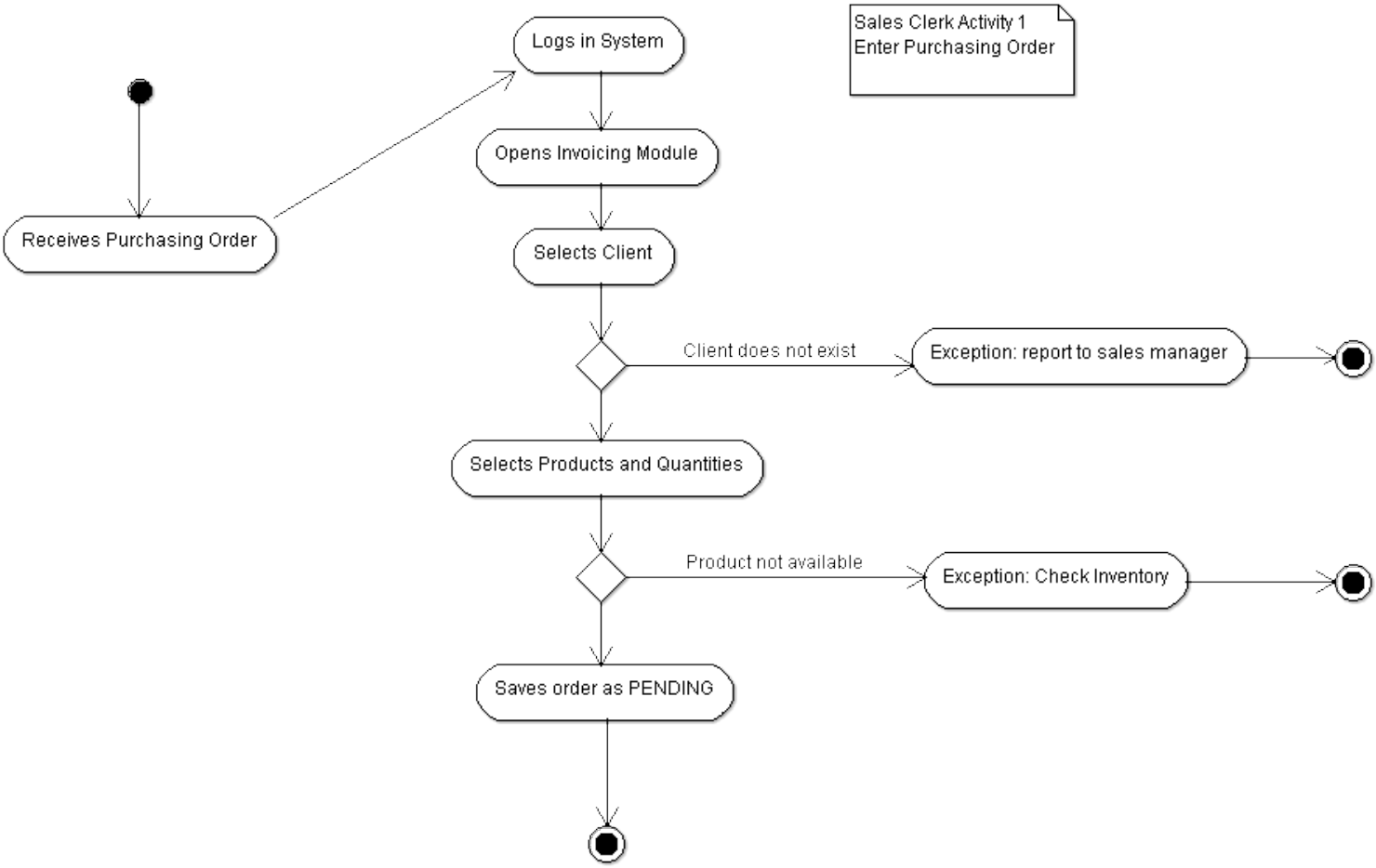

Note that only simple integer keys are supported as PK-FK, and there is no plan to support complex keys or non numeric keys, since they are not deemed a recommended approach


PK values of 0 or less than 0, -1, -2 etc. are ok, but they are used for special cases and become read only. When ever you need a record protected like a basic default value for a combo or something like that, set the PK to 0, the framework will prevent it from being deleted or edited.

4.1.3 Model activities -and validate the model using UML activity / sequence diagrams

Once we have a preliminary domain model, we model the business flows, always validating that we didn't miss and entity and that no flow violates a relationship.

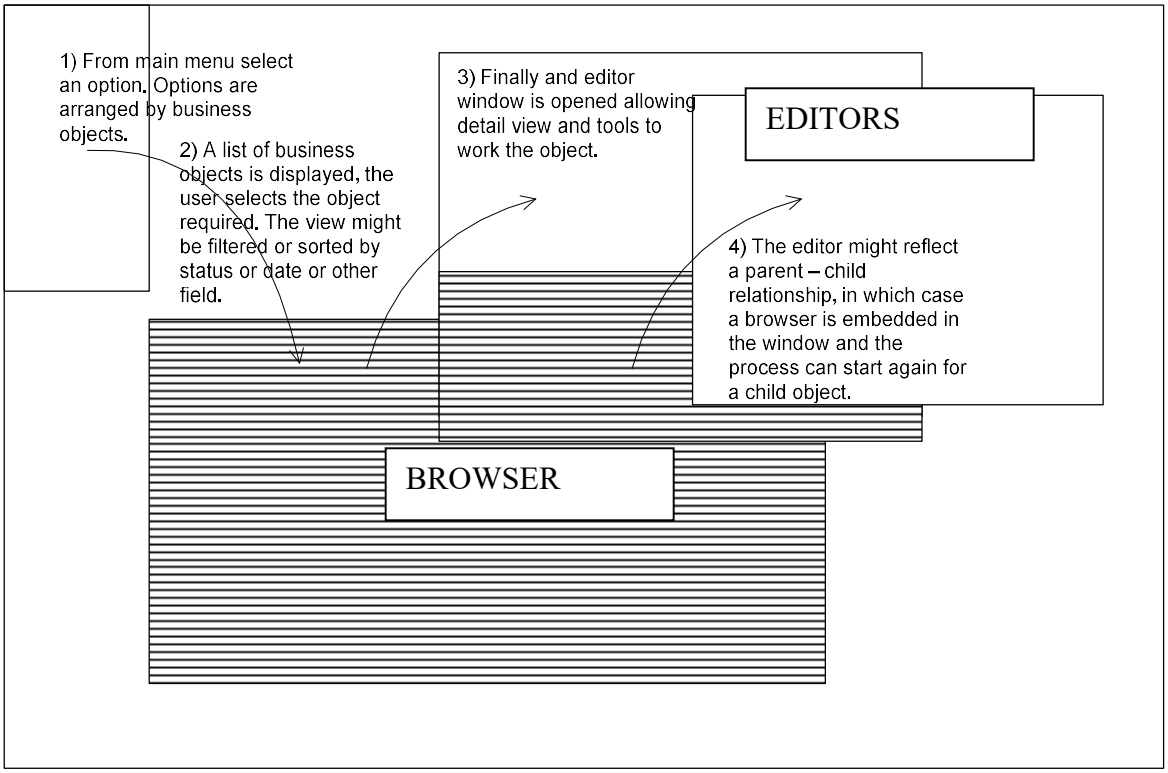
Sample Purchasing Flow



4.1.4 Model the application interface using UMLs activity diagrams

As explained before, we recommend doing a Problem Domain design first. The Human interface and DB design should be done later, when you already have a preliminary PDC design. A domain less GUI design is purely functional, not OO.

The framework's GUI support is built around a multiple document interface; the general guideline is as follows:



The options in the menu should represent root objects of the problem domain model.

The general idea is: Start from a problem domain object, say invoices, and open a table to select an invoice and then open the selected invoice editor.

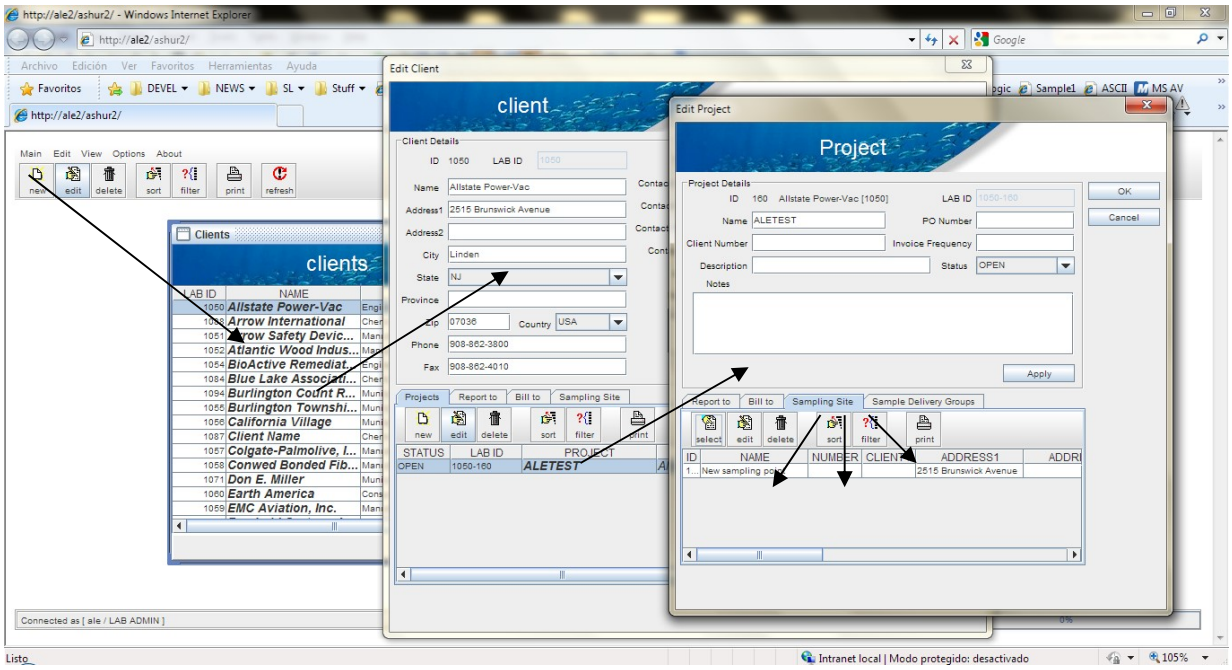
From the invoice editor you can have another table or tables, for the invoice items say, and start the flow again.

A good design will reflect the problem domain model diagram always. Any deviation can indicate an illogical view or a poorly designed problem domain model. The problem domain model is the map to guide all designs for all the developers.

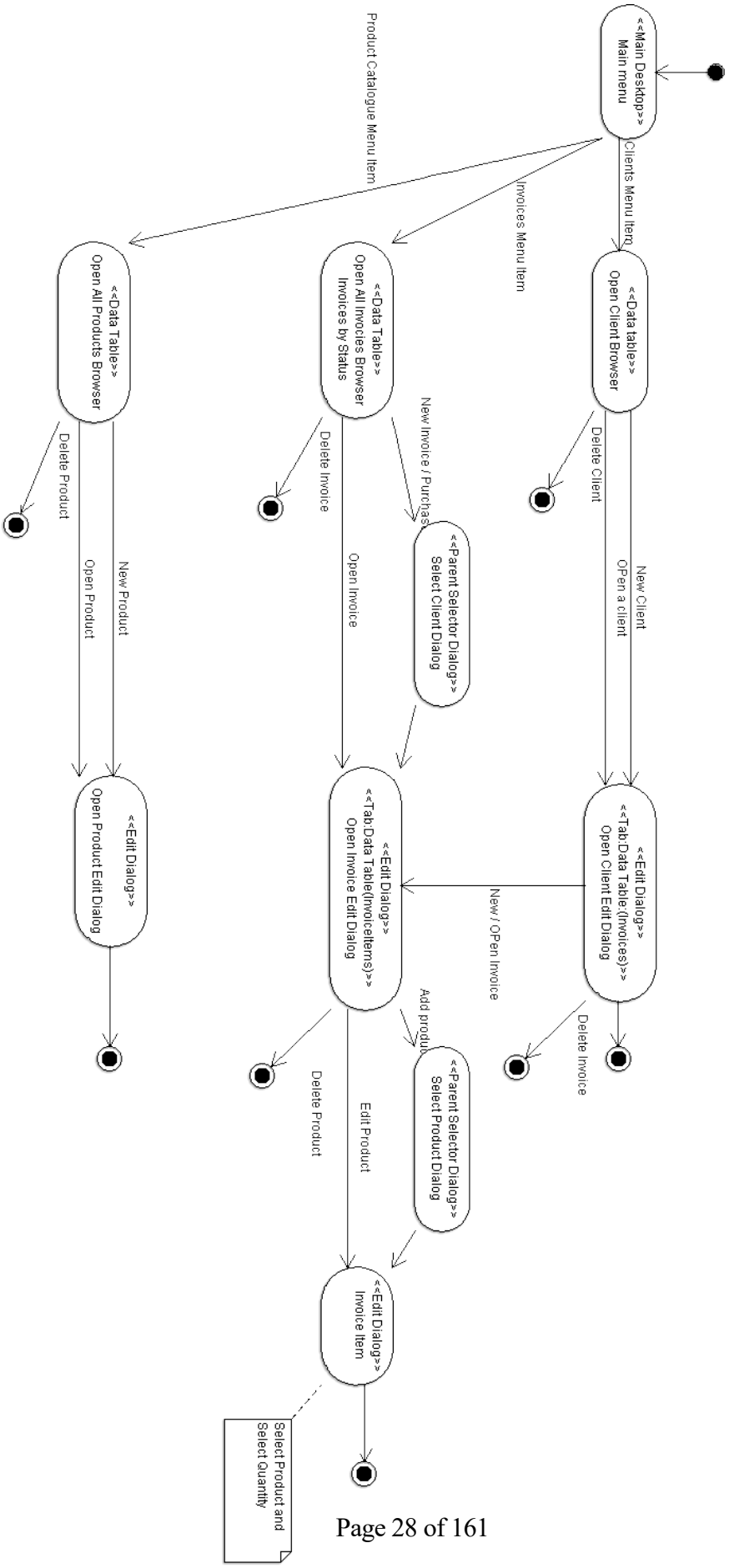
Remember that you can chain as many objects as needed in the domain model and GUIs.

There no limit to the depth of the problem domain, though a well define domain normally is not to deep but a little wide. If you have two unrelated trees of problem domain objects be careful: you are implementing two different systems at the same time.

Sample Complex Interface: One business object per editor, as many tabs as needed to browse child objects



UML activity diagrams are useful to diagram the navigation flow. For this example the GUI flow will be as follows:


<<Main Desktop>>
Main menu
Product Catalogue Menu Item
Clients Menu Item
Invoices Menu Item
<<Data Table>>
Open All Products Browser
<<Data Table>>
Open All Invoices Browser
Invoices by Status
<<Data table>>
Open Client Browser
Delete Product
New Product
Open Product
Delete Invoice
New Invoice / Purchase
Open Invoice
Delete Client
New Client
OPen a client
<<Edit Dialog>>
Open Product Edit Dialog
<<Parent Selector Dialog>>
Select Client Dialog
<<Edit Dialog>>
<<Tab:Data Table(InvoiceItems)>>
Open Invoice Edit Dialog
<<Edit Dialog>>
<<Tab:Data Table(Invoices)>>
Open Client Edit Dialog
New / OPen Invoice
Delete Invoice
Add produ
Delete Product
Edit Product
<<Parent Selector Dialog>>
Select Product Dialog
<<Edit Dialog>>
Invoice Item
Select Product and
Select Quantity

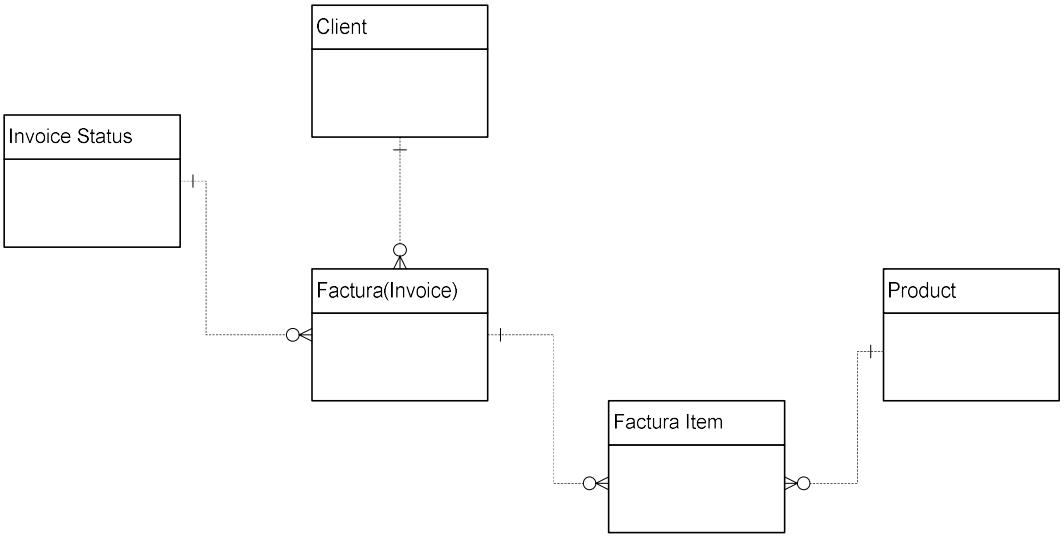
4.1.5 Model the database after the Problem Domain Model

This will be our database diagram. Sounds familiar? Sure it should match the problem domain diagram exactly. Some times we obviate it. All objects will have a simple integer ID and references will be implemented using traditional foreign keys.

More over, once you code the JAVA for the domain classes JPA will automatically generate the tables for you.

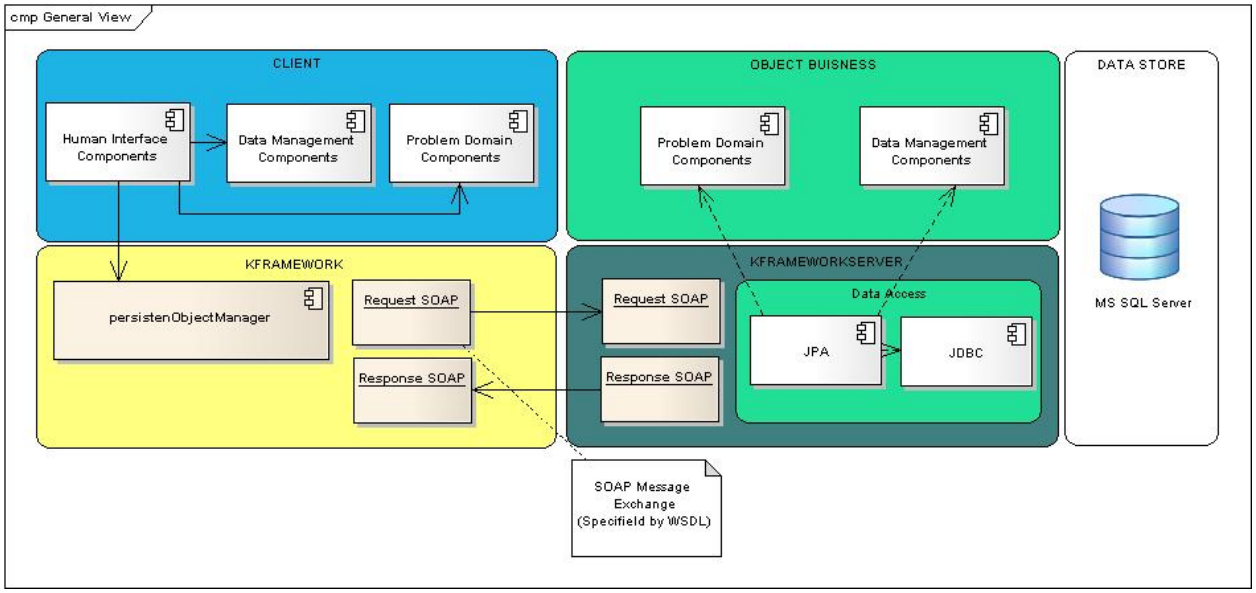
Note the rule where the crow's foot is the end of a has-uses relationship and it always points TO the bearer of the Foreign Key field.

Sample Project ERD		Edit Date: 10/12/2011 10:16:20 p.m.
Kframework's Sample Project database diagram		
DB ORACLE	Rev: 0.0	Creator: alejandro
Filename: VisioDocument		SARGON



4.1.6 Model the structural components

You don't have to worry about designing the structural part. The framework already provides a defined based on best practices. You just need to worry about your specific business design.



4.1.7 Model the IT systems requirements / Cross cutting concerns

1. Logging Scheme
2. Audit Trial
3. User and Role Management
4. Record Locking Strategy
5. Transaction Scope Management
6. Availability and Clustering of Servers
7. Reports framework

You need not to worry about any of these, the framework provides it all. There are very few frameworks that provide such integral framework for rapid application development. The KFramework is a real turn-key solution.

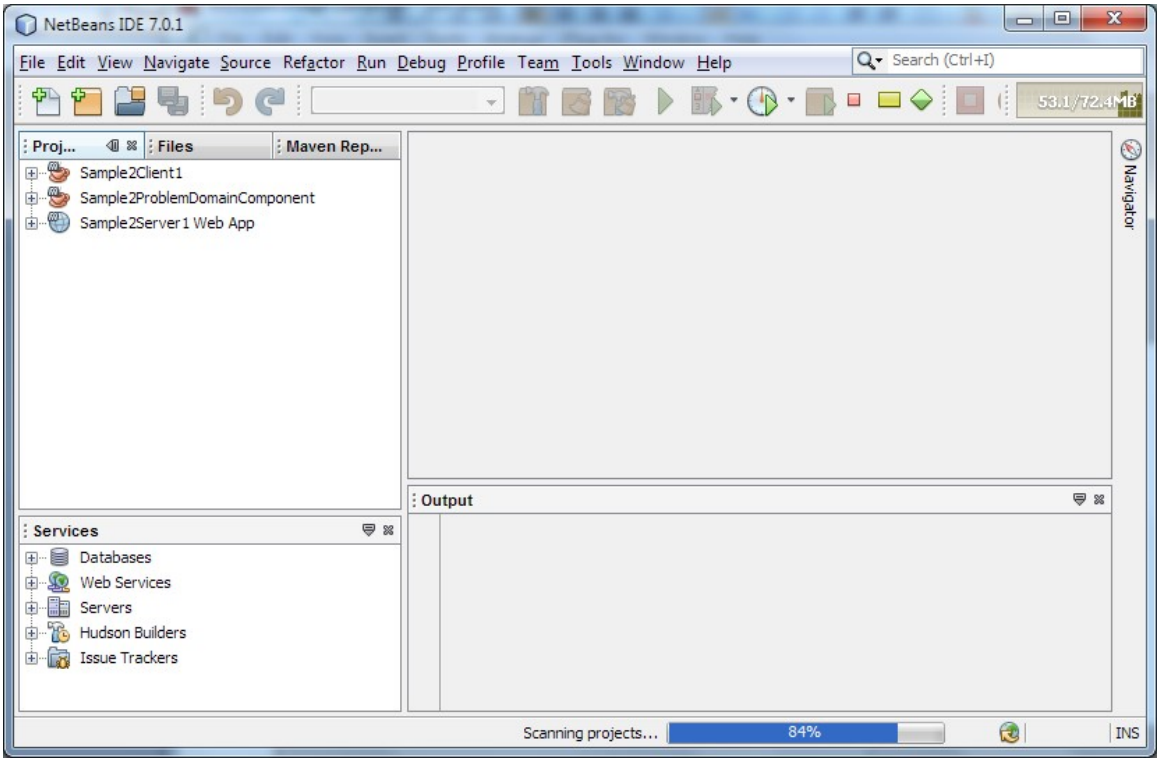
4.2 Get coding started!

Now we take the domain model, the GUI design and build the tool. With the KFramework you can have an application like this up and running in just a few hours, along with all IT systems, security and availability requirements.

This tutorial assumes you followed the previous section configuring the sample project, and that you have a functional sample project.

The running sample project will be the start point of all your projects, until you make your own customized templates.

Sample Projects open in Netbeans

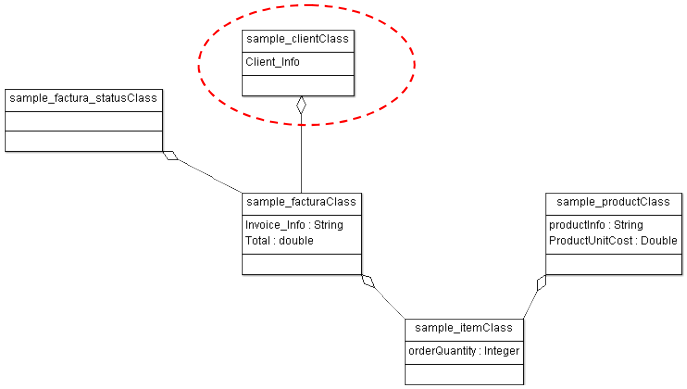


4.2.1 Build the domain model objects and database tables

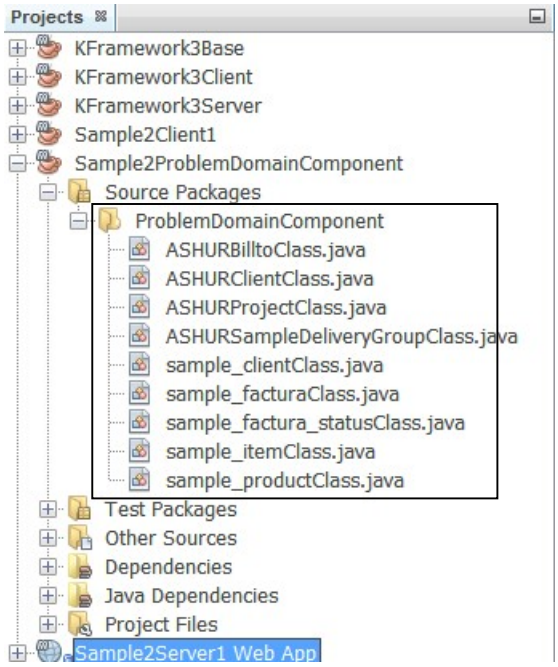
First we build the domain model. The domain model is all the business object classes defined in section *"Model the Domain using UML class diagrams".* Under DDD this model will guide all the development and ensure we are always under scope.

The database diagram is the same as the Domain Classes, and JPA assumes this, so it will create all the tables automatically after we finish.

Let's begin with the client entity:



1. ►Open the project named "Sample2ProblemDomainComponent" and enter the business classes as follows:



This module is compiled as JAR and shared between server and client, thus resolving the issue of propagating the entities to the server that other frameworks like FLEX have.

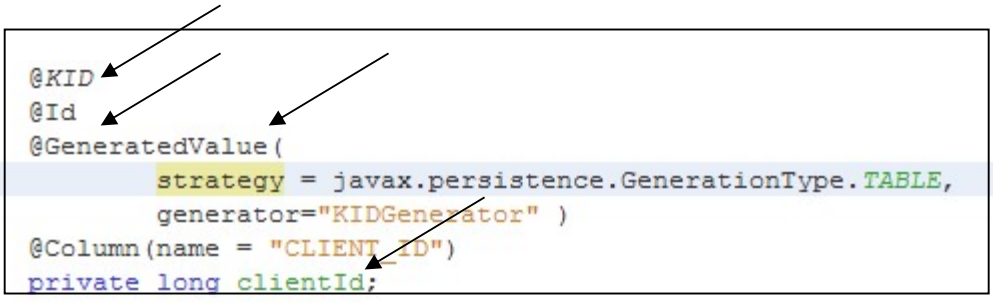
2. ►Now lets write de class declaration

```
@Entity
@TableGenerator( // SCOPE is Global to PU
    name="KIDGenerator",
    table="SEQUENCE",
    pkColumnName="SEQ_NAME",
    valueColumnName="SEQ_COUNT",
    pkColumnValue="SEQ_GEN_TABLE",
    allocationSize=1)
@Table(name = "SAMPLE_CLIENT")
public class sample_clientClass
extends KBusinessObjectClass
{
```

Note that the entity has to extend the ***KBusinessObjectClass***. This class provides the binding to widgets, serialization to server, etc.

Also note the **@TableGenerator** annotation. This tells JPA how to generate the primary key value. If you don't care, and you should not because it should not be intelligent, leave it, and a table value will be used. This is not very efficient but will work on all databases. See the JPAs documentation if you wan to use SQL's auto number or Oracle's sequences.

3. ►Then we add the client's attributes, starting wit the primary key:



```
@KID
@Id
@GeneratedValue(
        strategy = javax.persistence.GenerationType.TABLE,
        generator="KIDGenerator" )
@Column(name = "CLIENT_ID")
private long clientId;
```

```
@Column(name = "CLIENT_NAME")
private String clientName;
```

```
@Column(name = "CLIENT_ADDRESS")
private String clientAddress;
```

```
@Column(name = "CLIENT_EXPRESS_DELIVERY")
private String clientExpressDelivery;
```

```
@Column(name = "CLIENT_DISCOUNT")
private String clientDiscount;
```

```
@KObjectVersion
@Column(name = "version")
private long version;
```

```
@Override
public void validateInput(String currentField, Component currentComponent) throws KExcept
}
```

Note the primary key declaration. It has to be marked as **@KID** for the framework to know this is the PK and also **@id** for JPA for the same reason. Also, put the **@GeneratedValue** to take the number generator previously declared, or what ever other strategy you want to create the PKs.

Note that only simple integer keys are supported as PK-FK, and there is no plan to support complex keys or non numeric keys, since they are not deemed a recommended approach

For the framework to automatically tie the flow in and to perform automatic referential integrity all foreign keys MUST be named as the primary key they point to.

Note the final element marked **@KObjectVersion.** This optimal element indicates that we want optimistic locking, and defines the field to be used to keep object versions. The only thing you need to use optimistic locking is to mark a long attribute as **@KObjectVersion.**


Note the method validate input. Every time the UI tries to assign a value from a UI it will call this with the field to write and the widget from where the value will be taken. You can then centrally validate input to this domain object.

4. ►Finally we make a no args constructor and the getter and setters. The getters and setters need to be named exactly as the field they refer to plus the get and set prefix. Save your self some time and just right click on netbeans and select insertCode->Getter and Setters, and let netbeans write all of them for you.

```
public sample_clientClass() throws KExceptionClass {
}

public Long getClientId() {
    return clientId;
}

public void setClientId(Integer clientId) {
    this.clientId = clientId;
}
```

ETC…


Note that DATE fields need to be annotated as **@Temporal(TemporalType.DATE).** If you will also be using the hours and minutes part of the date change it to **@Temporal(TemporalType.TIMESTAMP).**

The framework transports the date with time always, but JPAs will only write and read the date if not set to timestamp. Note that the renderers and calendar component are not affected if you change this, only what is read and written to the database by JPA.

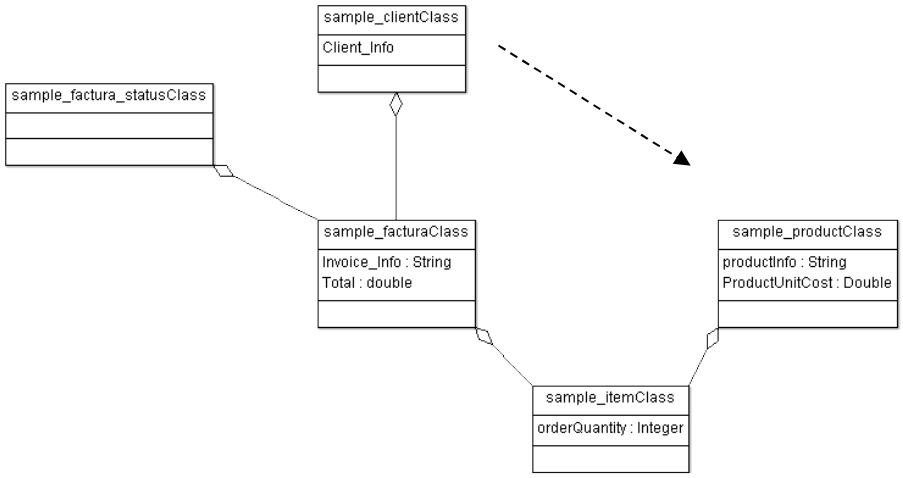
You can add validation that is used by the client and server to apply some rules to the fields. For example, you can set field types to automatically format and validate fields in the edit dialogs, as well as disabling text or combo boxes for read only fields. The advantage of doing it at the PDC object, and not at the editor, is that these options will work automatically on any window the object is displayed to.

For example:

Example:

```
        //set max size
        fieldTypes.put( "fac_total" ,CURRENCY_TYPE  );
        fieldMaxSize.put( "fac_name" ,30  );
        readOnlyFields.add( "fac_status" );

        editable = true;


}
```

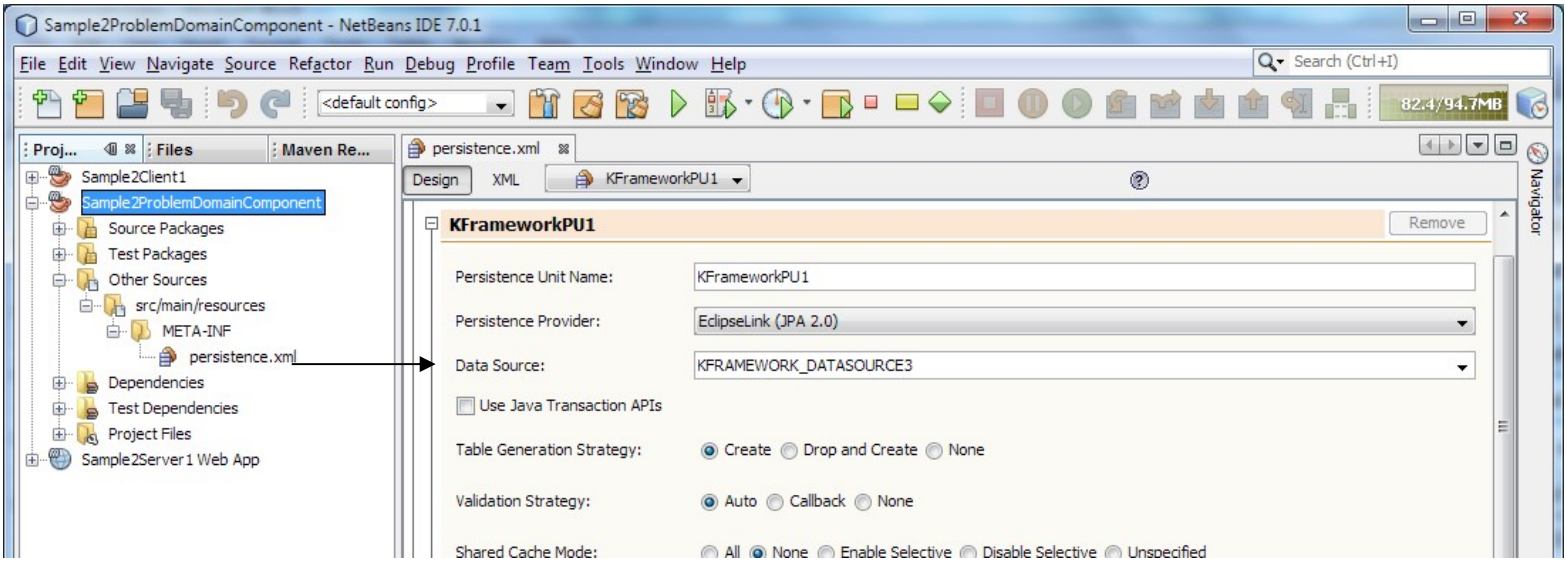
5. ►Proceed the same for all the entities



6. ►When finish compile the *Sample2ProblemDomainComponent* project and then recompile the *Sample2Server1*project to make sure it takes the changes.

7. ►Redeploy the *Sample2Server1,* JPA will then create the tables and PKs in the database. For changes in the entity classes, just drop the corresponding table and redeploy.

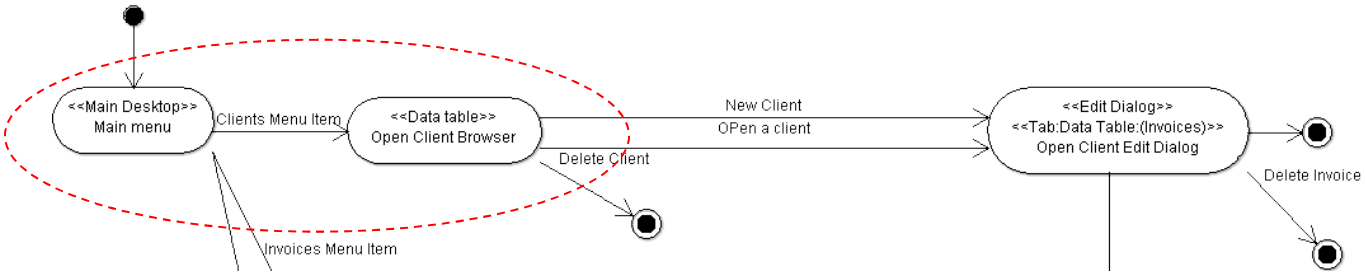
If you run into problems with JPA or just want to customize it, the persistent.xml is in the *Sample2ProblemDomainComponent* module as follows:



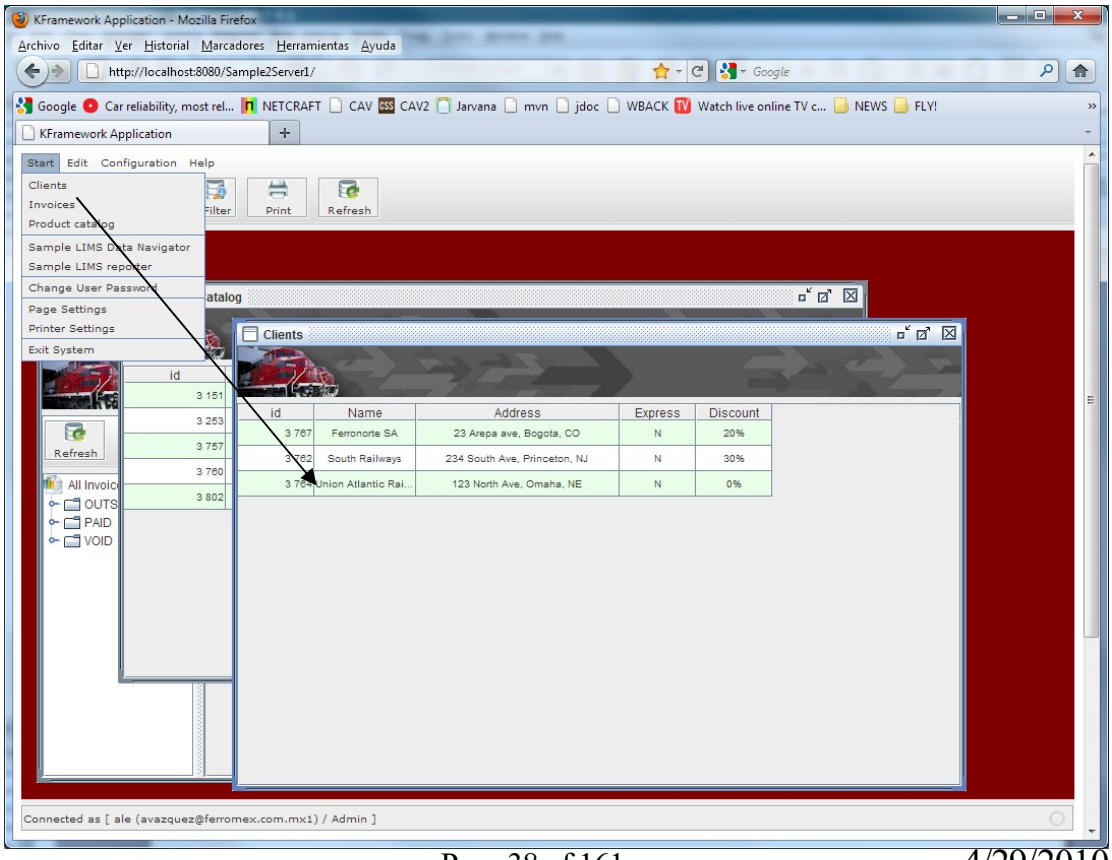
4.3 Build the GUI elements

Now let's make our first GUI element. We start from the top of the GUI hierarchy, and make our way down until we finish.

We will begin with the main menu and the initial client data table according to the GUI design presented before:



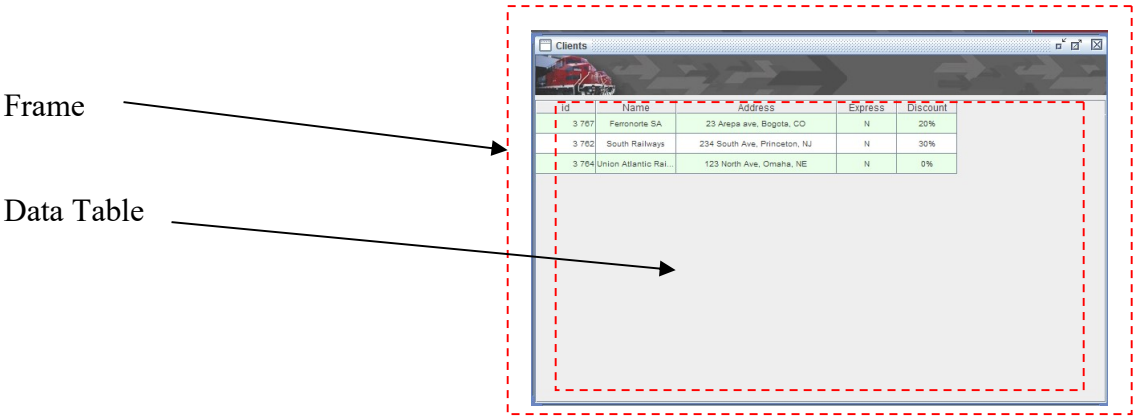
Our initial data table will look as follows: Note that the framework provides a MDI or multiple document interface, like a regular windowed app. It handles the state of all the documents for you and the user can navigate to any document with out you having to code weird HTML handling, multiple page sessions, and needing situational awareness tools like bread crumbs and the like.



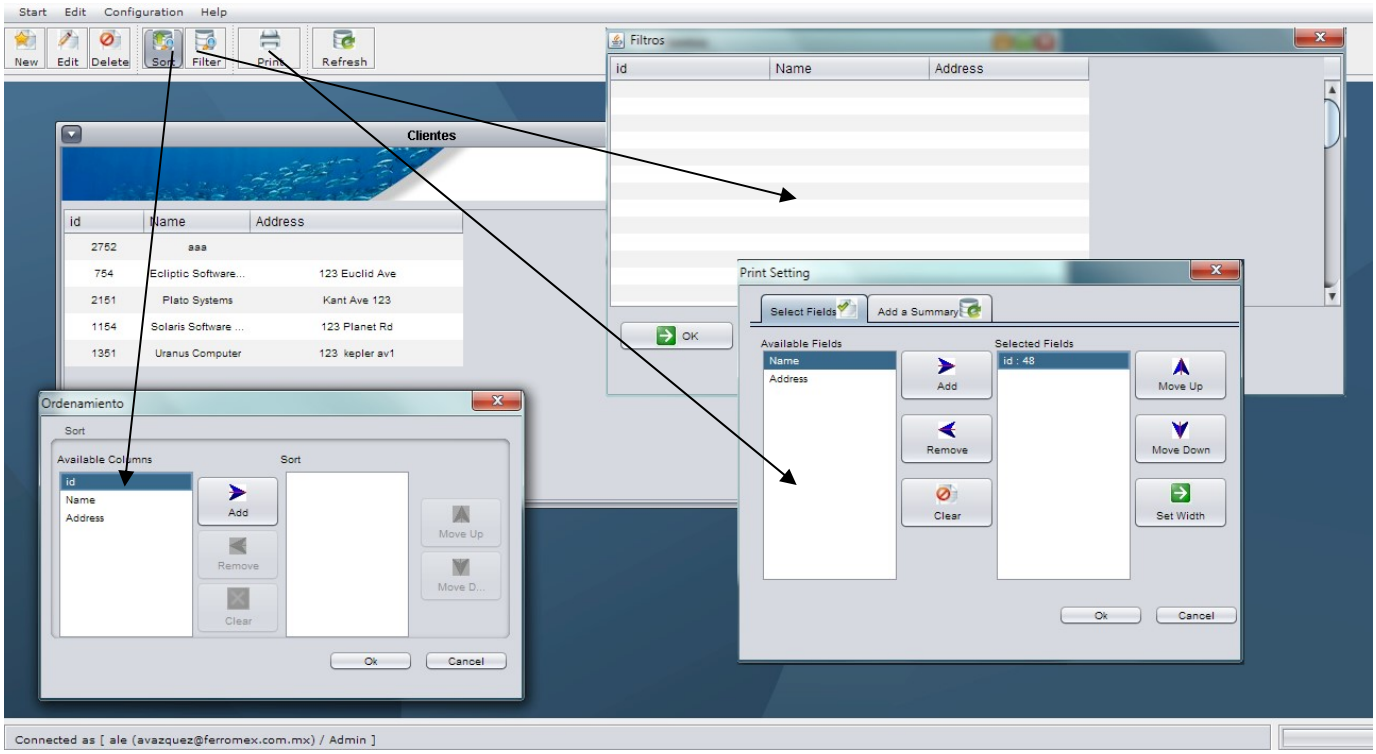
4.3.1 Building a data table

Initial data tables for the desktop are built with two classes, a generic data table and a desktop frame. The generic data table, for clients in this case, will be reused every where we need a client list. You only code one data table for a domain object through all the system, no matter where you need it and the specific filtering required. Maximize reuse!

Then we need a desktop frame to hold the data table. This frame provides the frame, GUI tools and state, and the nice logo at the top.

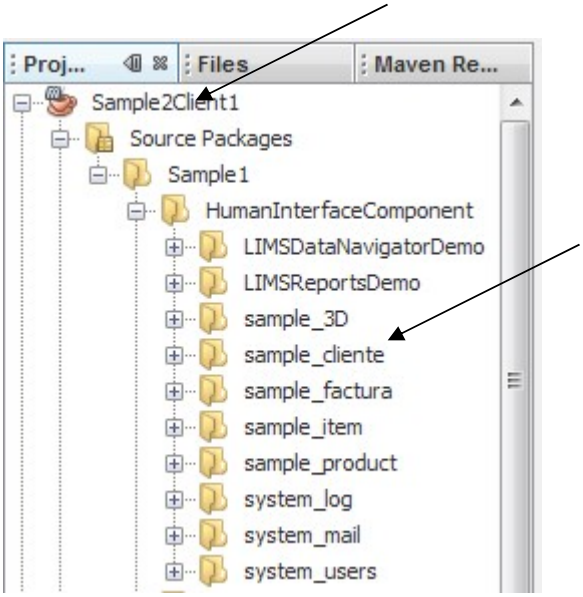


Once we have a data table the framework will automatically provide full CRUD, filtering tools, sorting and reporting, with 0 code needed from you:



Let's begin with the client's visual data table:

1. ►Open the *Sample2Client1* and make a package called "client" in the *HumanInterfaceComponent* package:



2. ►Make a class that inherits from KDataBrowserBaseClass like this:

```
public class clientBrowserClass
extends KDataBrowserBaseClass {


    public clientBrowserClass(
            KConfigurationClass configurationParam,
            KLogClass logParam,
            JTable tableParam,
            java.awt.Window parentWindow ) throws KExceptionClass
    {

            // inherits
            super(
                    configurationParam, logParam,
                    true, tableParam, parentWindow,
                    null, null
            );


    }
```

This class will hold our clients table. The super class will provide pagination, filtering, sorting multiple sessions customizable displays, display hooks etc.

You will need to pass configuration object, log object, the target JTable and a parent window. Don't worry; the framework will give you all that later.

Also note in the super constructor that you need to pass the corresponding entity's class and the entity's edit window, pass null for now.

3. ►Finally, you need to override the *initializeTable* method to define the tables data:

```
@Override
public void initializeTable()
throws KExceptionClass
{

        // set the SQL
        super.initializeSQLQuery(

            // 1 campos
            " client_id , client_name, client_address, CLIENT_EXPRESS_DELIVERY, CLIENT_DISCOUNT ",

            // 2 tablas and joins
            " sample_client cli ",

            // 3 llave principal (mayusculas)
            "CLIENT_ID"

                );

        // define column settings
        setColumnNames( "cli", "CLIENT_ID", "id" );
        setColumnNames( "cli", "CLIENT_NAME", "Name" );
        setColumnNames( "cli", "CLIENT_ADDRESS", "Address" );
        setColumnNames( "cli", "CLIENT_EXPRESS_DELIVERY", "Express" );
        setColumnNames( "cli", "CLIENT_DISCOUNT", "Discount" );

        setDefaultOrder( "  client_name " );


        // load data
        super.initializeTable();

        // some customization
        adjustColumnWidth( "Name", 100 );
        adjustColumnWidth( "Address", 200 );

}
```

1. First call the *initializeSQLQuery* method. This method requires the field list of the SQL, then the tables and joins if any and finally the name of the PK field, which should appear in the field list. This method makes the component aware of the data.

Add relationships as joins like this …

```
super.initializeSQLQuery(

    // 1 fields
    " fac.fac_id , fac.fac_name, status.facstatus_status, fac.fac_total ",

    // 2 tables and joins
    " sample_factura fac " +
    " left join sample_factura_status status on status.facstatus_id = fac.facstatus_id  "
```

Then we customize the columnar display:

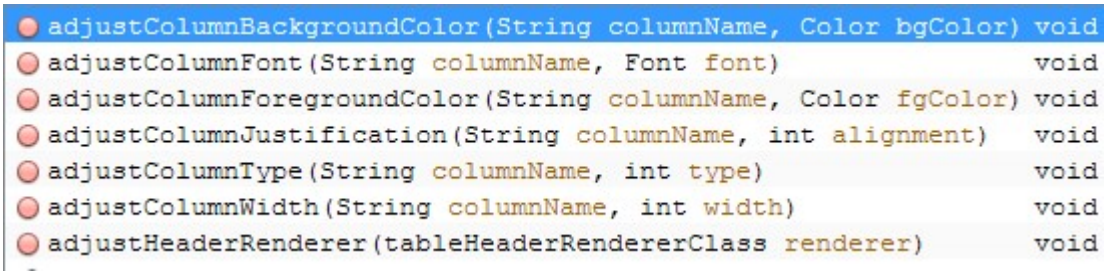
```
// define column settings
setColumnNames( "cli", "CLIENT_ID", "id" );
setColumnNames( "cli", "CLIENT_NAME", "Name" );
setColumnNames( "cli", "CLIENT_ADDRESS", "Address" );
setColumnNames( "cli", "CLIENT_EXPRESS_DELIVERY", "Express" );
setColumnNames( "cli", "CLIENT_DISCOUNT", "Discount" );

setDefaultOrder( "  client_name " );
```

2. The *setColumnNames* method takes the table alias and column name of a column from the previous SQL and allows you to set a display name for the column. This will be the name by which the column will be referred to and displayed to the user's in the GUI.

3. With the *setDefaultOrder*, set the initial default sort, this is a regular SQL order string. Just make sure you follow your database's rules on order bys.

```
        // load data
        super.initializeTable();

        // some customization
        adjustColumnWidth( "Name", 100 );
        adjustColumnWidth( "Address", 200 );

    }
```

4. At this point we call *super.initializeTable();* to load our table. At runtime the client will call the server and pull the initial cache of data.
5. Finally, by referring with the display name we can call the different "*adjust*" methods to customize the display.



Note that a browser will not load all records from the server. It grabs the necessary records depending on the view range, plus a small cache for smooth forward and backward scrolling. The user will "feel" he is traversing the whole set, even for millions of records, but in reality the browser is fetching only the required records in the background. You can jump to the end, middle and the browser will show the corresponding records, but it needs not load the whole set in the client to go to the end of the table, for example.

That's it for the table view. Now we need a frame to display it inside the desktop.

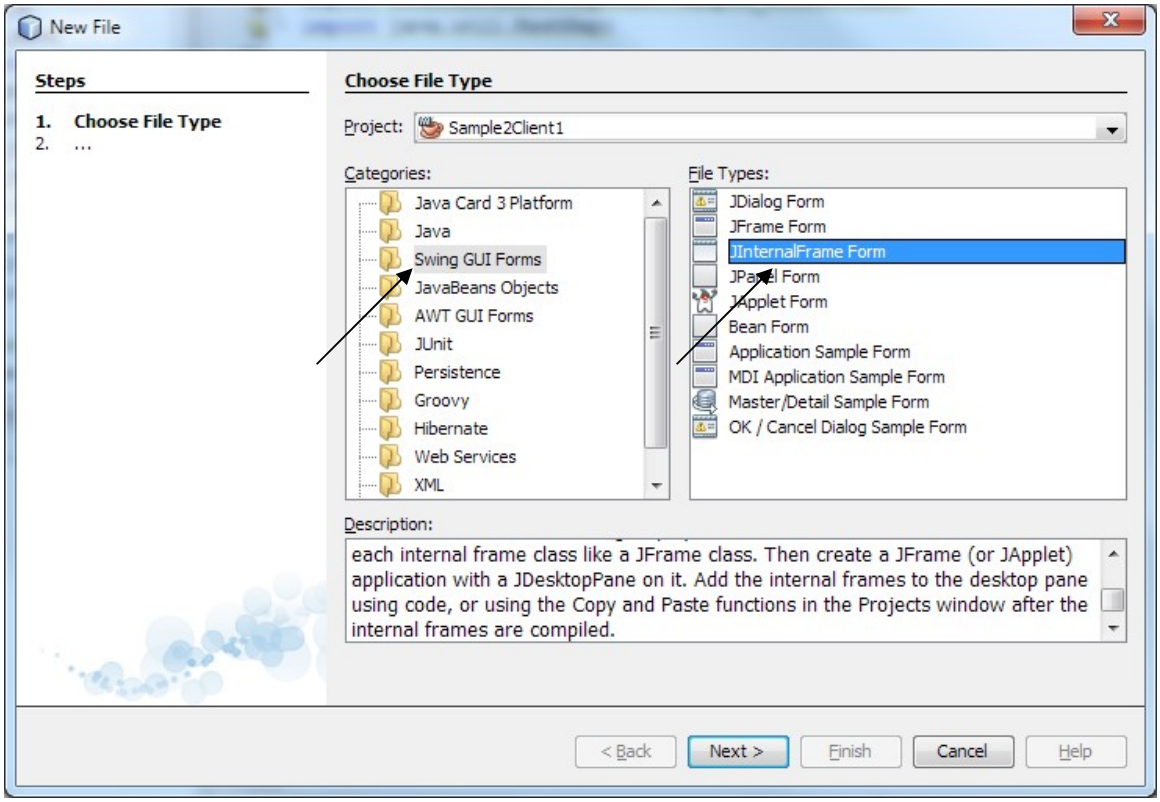
4.3.2 Building an internal MDI frame

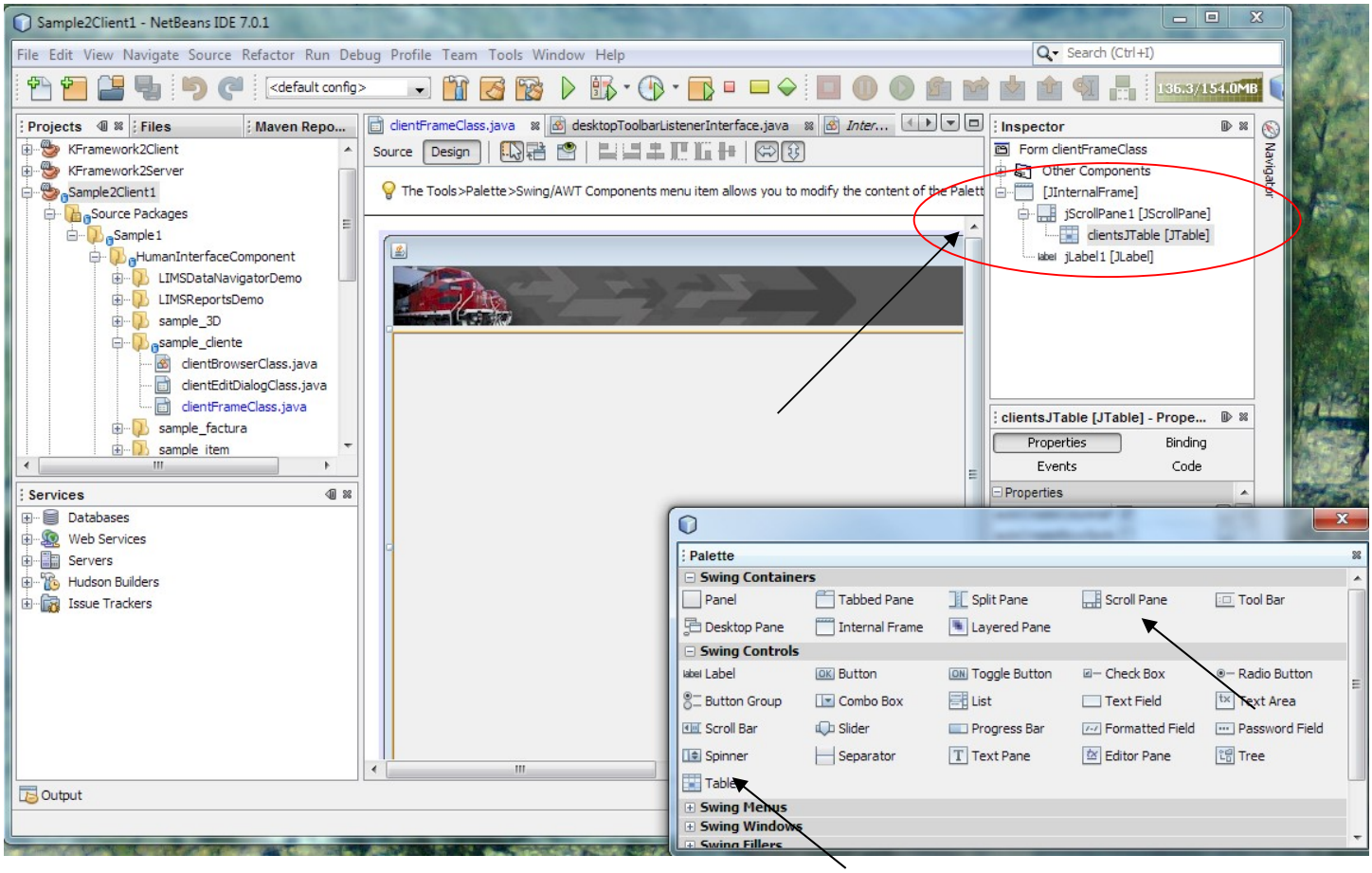
Now we need to make an internal frame to present the table inside the GUI. The table is separate of the frame because we will reuse the table in other places.

The framework provides a generic desktop with a generic CRUD buttons with filter, sort and print tools. The object you just built in the section before implements all this for you, when you extended the *KDataBrowserBaseClass*. We now make a frame to display or hold the table inside the desktop and interface it with the provided toolbar.

1. ► Open again the *client* package in the *HumanInterfaceComponent* that we used previously, and where we will put all the client GUI artifacts.

2. ►Right click in the package and create a new Swing internal Frame: In Netbeans, on the clients package, click new->Other, and select Swing GUI forms ->JInternalFrame Form as follows:





3. ►Visually design your frame. Make a nice top banner or you might need to add a special tool bar. Place all images and resources in the resources folder. The only required components for this example are a *JTable* inside a *JScrollPane*.

4. ►Now we need to hook this frame to the previously built data table and the framework's desktop. Switch your frame from Design View to Source View.

```
public class clientFrameClass extends javax.swing.JInternalFrame
implements desktopToolbarListenerInterface, InternalFrameListener
{

        // uses
        private KConfigurationClass             configuration;
        private KLogClass                       log;
        private desktopToolbarAccessInterface   mainToolbar;

        // has - defaulted
        private clientBrowserClass               browser;
```

5. ►Add and extends clause to the class for *desktopToolbarListenerInterface* and *InternalFrameListener*, you will then need to implement several methods, leave them empty for now. The *desktopToolbarListenerInterface* is used by the desktop to access your data table, and the *InternalFrameListener* is used by swing to notify you of internal frame actions.

6. ►Add the required attributes for configuration, log, toolbar and browser. Note that ALL objects in the framework "use" configuration and log. Specifically for this object will also "use" the desktop toolbar which will be passed in the constructor, and, of course, the table we built in the previous section. So our constructor will look as follows:

```
/** Creates new form justificacionFrameClass */
public clientFrameClass(
        KConfigurationClass configurationParam,KLogClass logParam,
        desktopToolbarAccessInterface systemDesktopParam )
        throws KExceptionClass
{
    initComponents ();

    // uses
    configuration = configurationParam;
    log = logParam;
    mainToolbar = systemDesktopParam;

    // has defaulted
    browser =  new clientBrowserClass( configuration, log,
            clientsJTable, mainToolbar.getDesktopsWindow()  );

    browser.initializeTable();

    addInternalFrameListener( this );

    log.log( this, "Clients frame constructed successfully." );
}
```

1. First we setup the "use" attributes, which are passed by the desktop as arguments to the constructor.

2. Then we initialize the data table, or browser, which we created in the previous section. Note how we pass the JTable and call the toolbar's *getDesktopWindow*() to set the parent window.

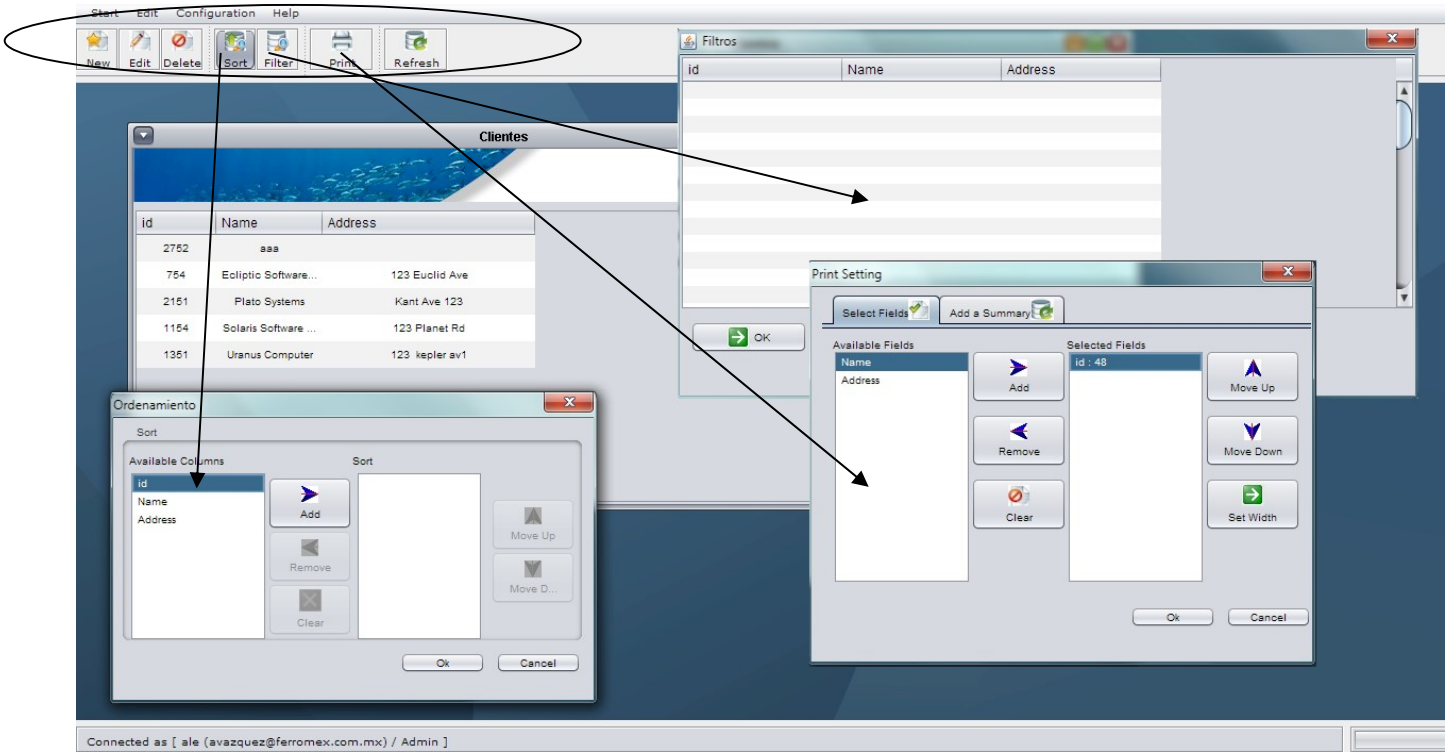
3. Then we set the frame as listener for frame events:

```
        addInternalFrameListener( this );

        log.log( this, "Clients frame constructed successfully." );
    }
```

4. Finally we need to implement the methods that are required by the *InternalFrameListener* and the *desktopToolbarListenerInterface* interface. Here is an example:

```
    public void internalFrameOpened(InternalFrameEvent e) {}
    public void internalFrameClosing(InternalFrameEvent e) {}
    public void internalFrameClosed(InternalFrameEvent e) {}
    public void internalFrameIconified(InternalFrameEvent e) {}
    public void internalFrameDeiconified(InternalFrameEvent e) {}
    public void internalFrameActivated(InternalFrameEvent e) {
            mainToolbar.enableDelete( true );
            mainToolbar.enableNew( true );
            mainToolbar.enableEdit( true );
            mainToolbar.enableSort( true );
            mainToolbar.enableFilter( true );
            mainToolbar.enableRefresh( true );
            mainToolbar.enablePrint( true );
    }
    public void internalFrameDeactivated(InternalFrameEvent e) {
            mainToolbar.enableDelete( false );
            mainToolbar.enableNew( false );
            mainToolbar.enableEdit( false );
            mainToolbar.enableSort( false );
            mainToolbar.enableFilter( false );
            mainToolbar.enableRefresh( false );
            mainToolbar.enablePrint( false );
    }


    public ActionListener getDesktopToolbarActionListener() {
        return( browser );
    }
```

Basically we are just enabling and disabling the desktop's toolbar buttons depending on the data displayed.



Specifically you might need to prevent edits on some data, deletes on others etc. For now just enable all on focus and disable on lost focus as shown previously.

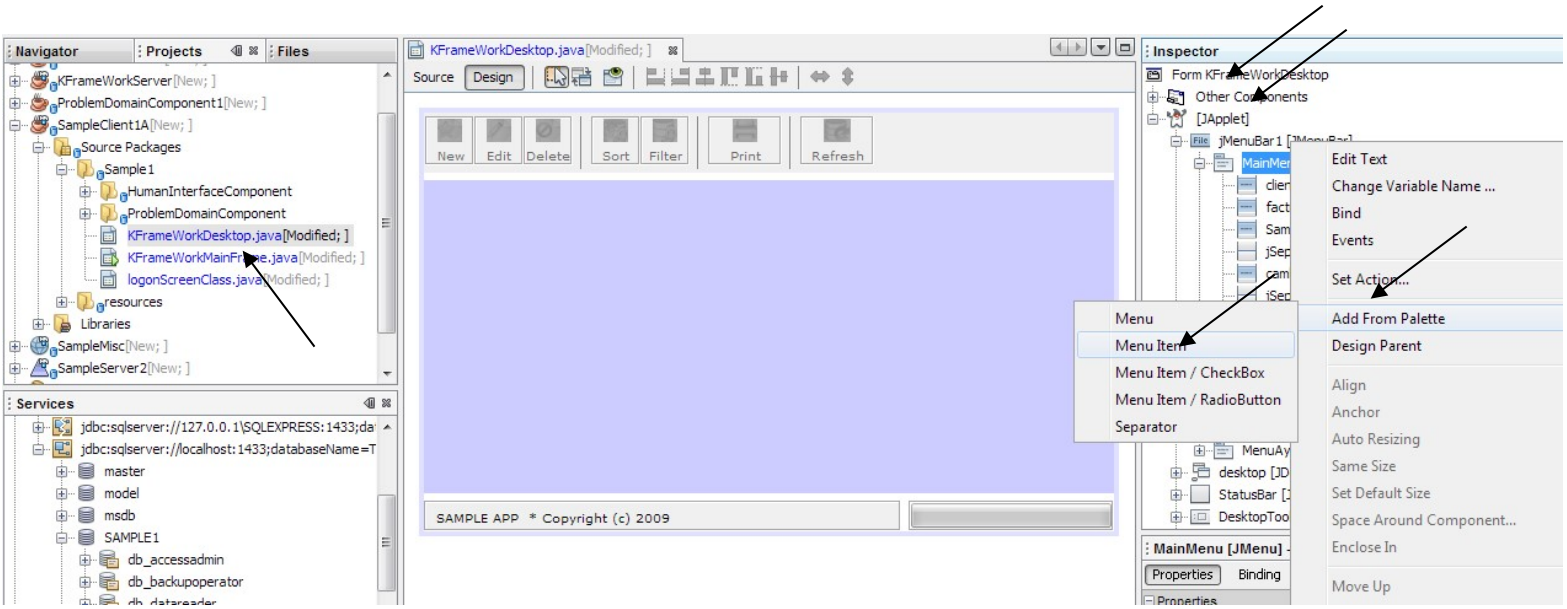
In the other hand *getDesktopToolbarActionListener* is used by the framework to forward the toolbar events to our data table. Don't worry, the data table we created already implements all the button actions, just return it in the method:

```
public ActionListener getDesktopToolbarActionListener() {
    return( browser );
}
```

7. ►Finally, Wire the Browser to the Main Menu

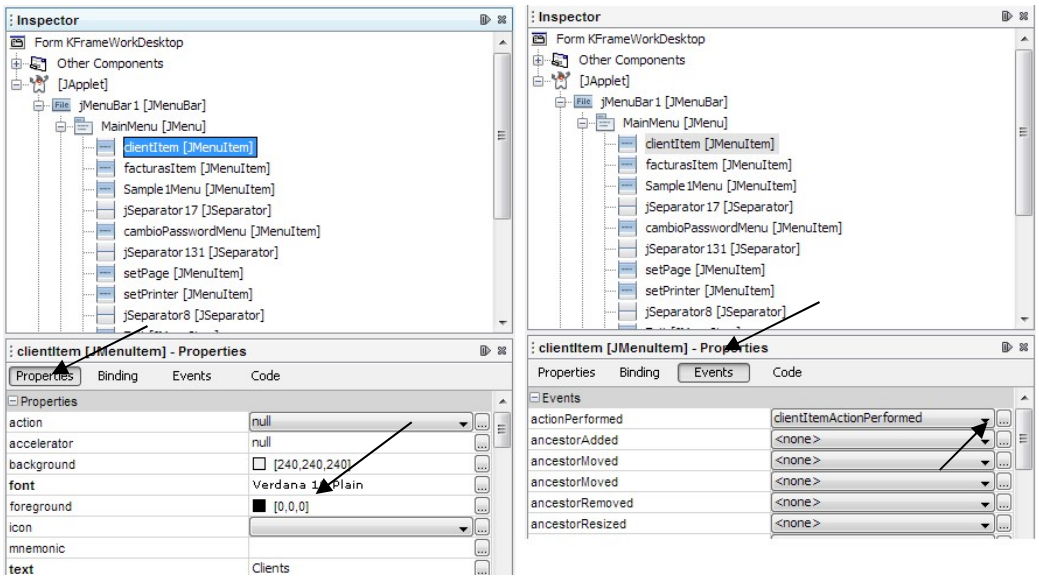
The desktop is provided. Aside from adding buttons, you don't need to change it.

1. Open the KFrameWorkDesktop.java from the sample client:
2. Open the menu tool and add a MenuItem



3. Fix the font type of the item to Verdana, 10pts, plain

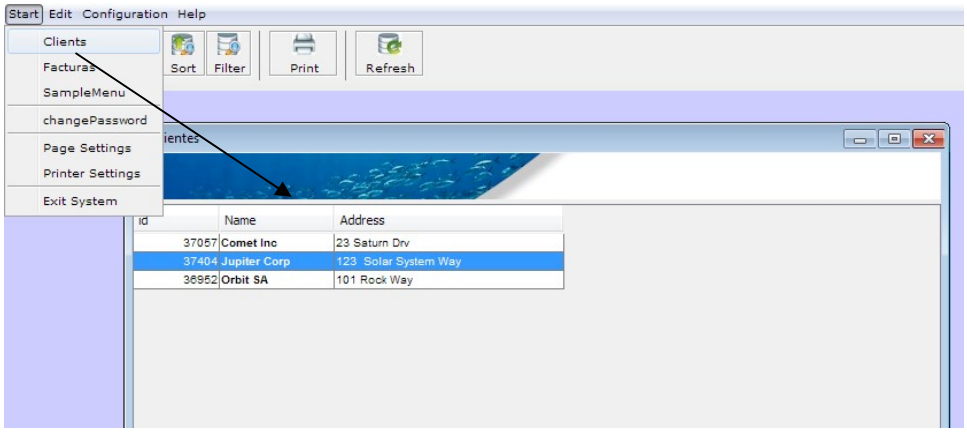
4. On the events tab, click the down arrow in action performed, it will open a function and the java editor.



5. Code the event to open the frame we just built. Call the *openInternalFrame* method and pass the class of the frame you just created, and give a title to the window.

```
private void clientItemActionPerformed(java.awt.event.ActionEvent evt) {

    openInternalFrame( clientFrameClass.class, "Clients" );
}
```

6. That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…

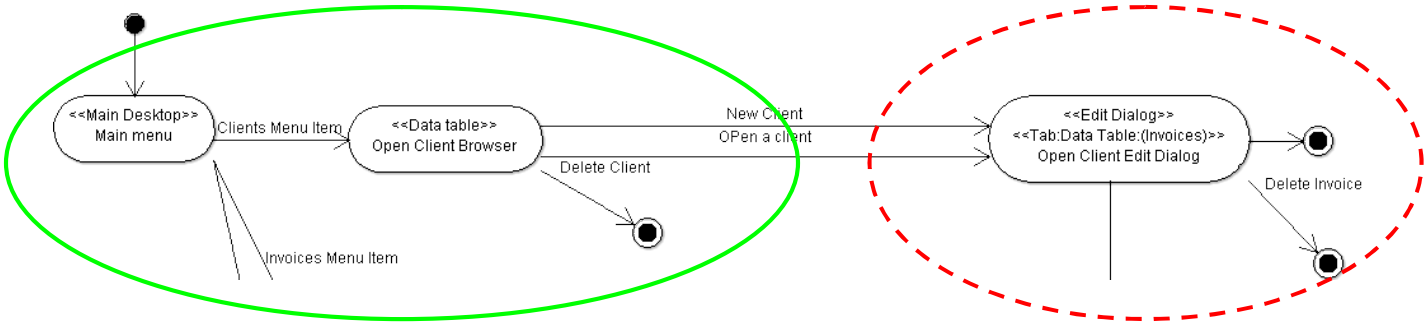


The Framework provides

- Server and client projects
- Security
- Communications
- Automatic sort, filter and print tools, aside from the reporting engine, and much more…

4.3.3 Building a data editor

Following our GUI design, we have now built our domain's entities and our first data table: the client's data table. To finish the remaining GUI elements for the client's entity, we now proceed to build the client's edit window.



First we need to look into the *problemDomainComponent* and see what the attributes of the client entity are:

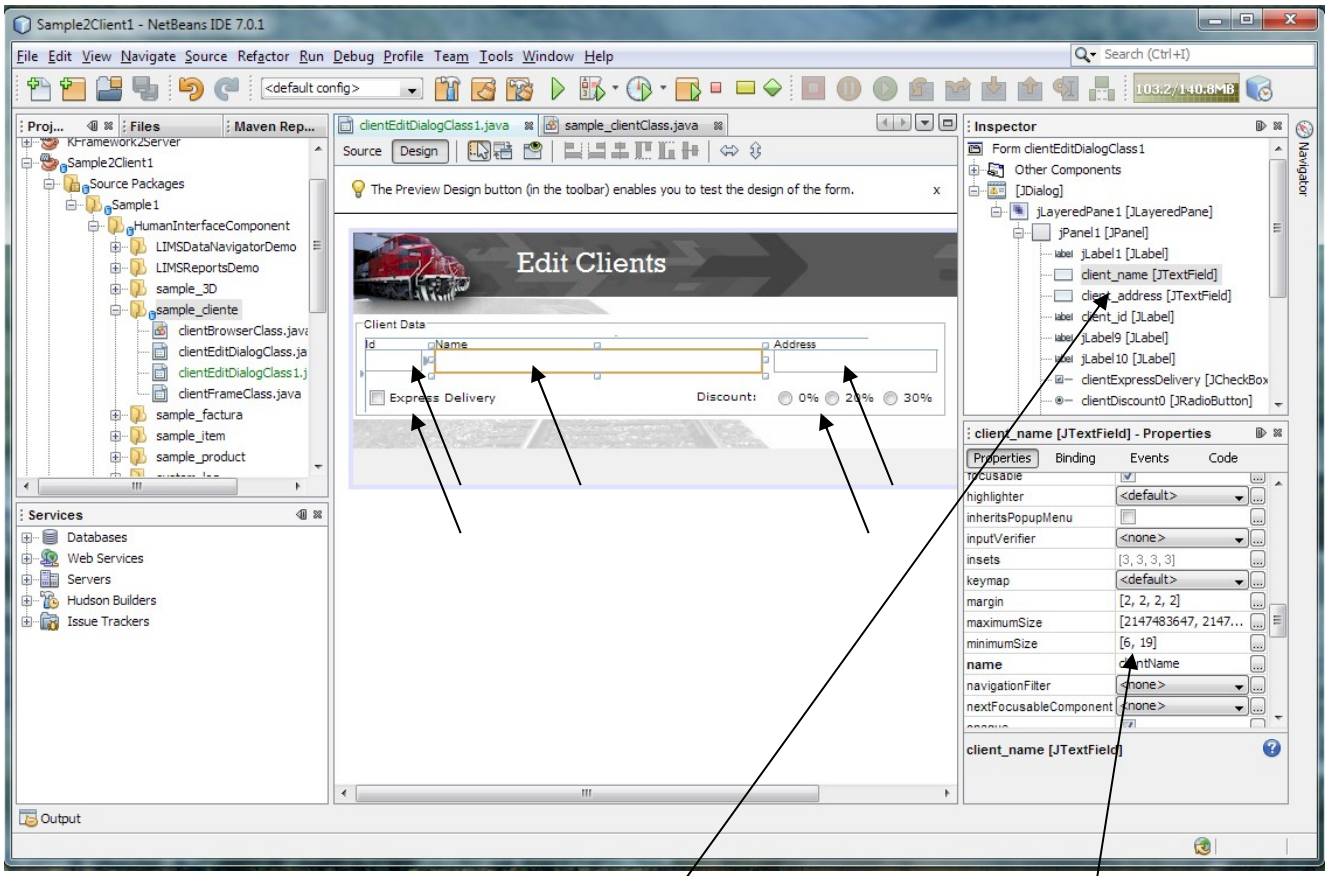
```
@KID
@Id
@GeneratedValue( strategy = javax.persistence.GenerationType.TABLE, generator="KIDGenerator" )
@Column(name = "CLIENT_ID")
private long clientId;


@Column(name = "CLIENT_NAME")
private String clientName;

@Column(name = "CLIENT_ADDRESS")
private String clientAddress;

@Column(name = "CLIENT_EXPRESS_DELIVERY")
private String clientExpressDelivery;

@Column(name = "CLIENT_DISCOUNT")
private String clientDiscount;

@KObjectVersion
@Column(name = "version")
private long version;

public sample_clientClass() throws KExceptionClass {
}
```

Except for the version field, we need a GUI widget for all the fields.

1. ►Open again the *client* package in the *HumanInterfaceComponent* that we used previously, and where we will put all the client GUI artifacts.

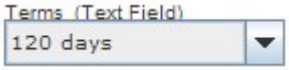
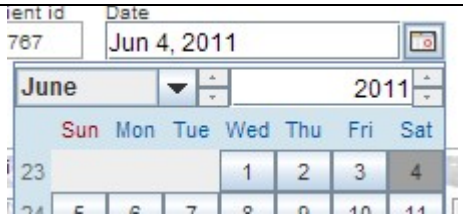
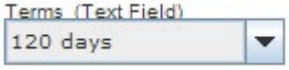
2. ► Create a new Swing JDialog: In Netbeans, on the clients package, click new->Other, and select Swing GUI forms ->JDialog Form.

3. ► Don't worry about code or HTML nor anything like that, just visually paint all the components for the client's attributes. Add some nice logos and design, etc.



Entity fields are mapped automatically to widgets. When the user closes or applies the window, materialize() is called in the business object which will automatically read the changes from the widgets. Finally the object is sent to the server for business logic and persistence. For this to work <u>Name the field for the entity field to be mapped(1), don't confuse with the variable name (2), which is not important.</u>

Widgets supported for data bounding / auto mapping to POJOS are:

 (Saving displayed text in db)	JComboBox (Mapped directly fields)
	JTextComponent an decendants like TextBoxes and TextAreas
Label	JLabel
	JCheckBox (Mapped to db text fields with Y /N)
	JRadioButtons (Make a group and name ALL members as the field to map, the selected member will be assigned to a db text field)
	com.toedter.calendar.JDateChooser (Mapped to db date fields)
 (For foreing key fields, linking a detail table via PK-Fk relationships)	dropDownFillerClass (Combos mapped to tables via Foreign Keys, see the corresponding section in the advanced concepts.) (Provided)
.. and most any other.	Third Party widgets cab be integrated, see advanced functionality

► To make this dialog work we don't need to map fields, but there are a few things needed to tie it to the framework flow:

```
public class clientEditDialogClass1
extends javax.swing.JDialog
implements KDialogInterface
{

    // uses
    private KConfigurationClass             configuration;
    private KLogClass                       log;

    // has defaulted
    private KDialogDirectorClass            KDialogController;
```

1. Implement the KDialogInterface. This will require some methods to be implemented, leave them empty for now.
2. As usual, also add configuration and log as attributes to the class, also as usual this will be passed in the constructor.
3. Finally add a KDialogControllerClass attribute to the dialog class. This is the Frameworks' object which will control the dialog's lifecycle.
4. ► Now we need a constructor with a special signature, so that the framework can open the window automatically:

```
/** Creates new form facturaEditDialogClass */
public clientEditDialogClass1(
    KConfigurationClass configurationParam, KLogClass logParam, java.awt.Window parentWindow )
    throws KExceptionClass
{

    super( parentWindow, java.awt.Dialog.ModalityType.DOCUMENT_MODAL );
    initComponents ();
    pack ();
    setSize( 630, 600 );
    KMetaUtilsClass.centerInScreen( this );

    // uses
    configuration = configurationParam;
    log = logParam;

    // has - defaulted
    KDialogController = new KDialogDirectorClass(
            configuration, log,
            sample_clientClass.class, this, getContentPane() );

}
```

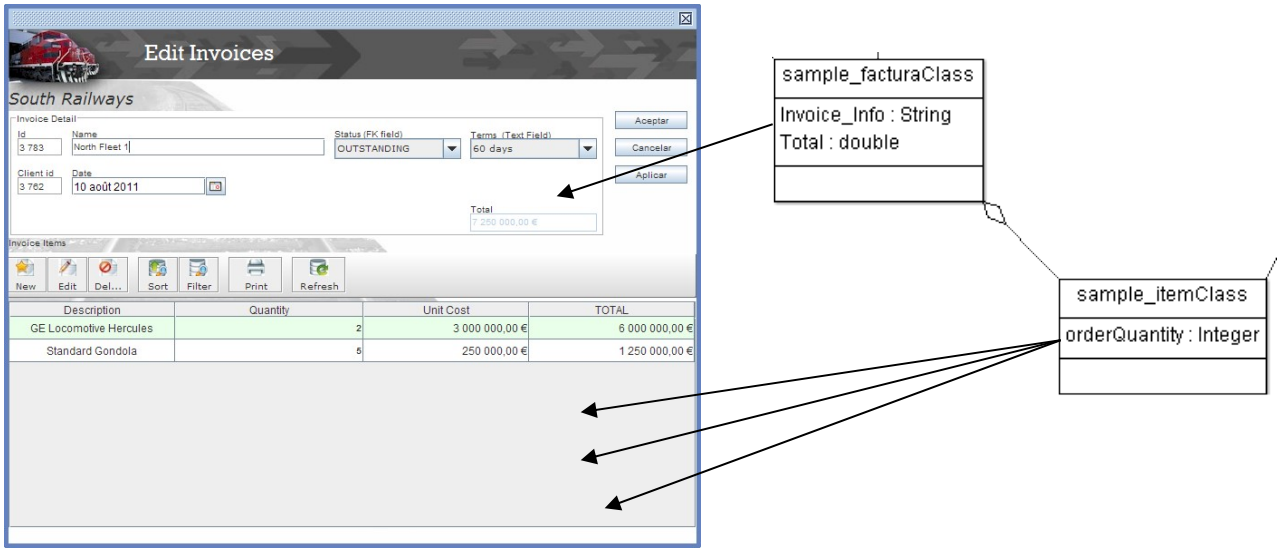
1. As usual, the constructor signature will receive configuration and log, but also the parent window and we need a throws clause for KExceptionClass.
2. Then we call the base constructor, we pass the parent window we received and set the window mode to modal.
3. We call the *initComponents()* which is generated by NetBeans, pack() for widgets to get to place and finally set the window's dimensions.
4. Then we call the framework's *centerInScreen(this),* to center the dialog.
5. We assign the configuration and log
6. Finally, we build the dialog controller, which will control the dialog's lifecycle. What it does is to read the entities' data in and saves it when applicable. We pass the typical configuration and log, plus the class of entity to edit, the parent dialog, and the pane that holds the widgets we draw. If unsure, just copy the call.

5. ► Implement all 3 methods from the *KDialogInterface*

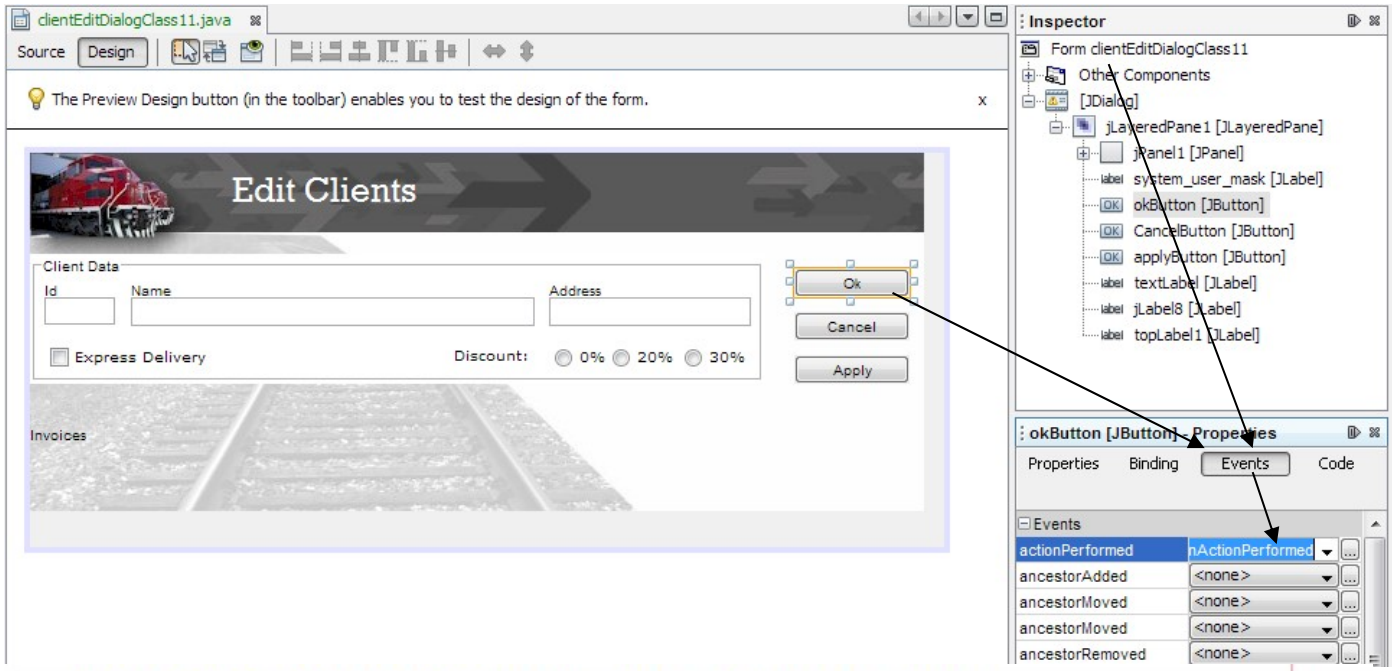
```
@Override
public void initializeDialog(int dialogModeParam, Long ID, Map foreingKeys ) throws KExceptionClass {

    // start
    KDialogController.initializeDialog( dialogModeParam, ID, null  );
}

@Override
public void setupTables( long id )
throws KExceptionClass{
}

@Override
public void saveBrowserChanges() {
}
```

1) The initialize dialog is called just after construction to load the data, which is loaded in the controller's *initializeDialog*, so this method gives you the chance to run some logic just before the entity is loaded and just after it was displayed. For example, just before you might load some data bound combo boxes and, just after load, you might load the parent object to display some additional data in the window.

2) The *setupTables* and *saveBrowserChanges* are called to start any sub table included in the editor or to save any data for writable tables. A sub table is how child objects are chained together with the parent object, for example to tie invoices with their items we do:

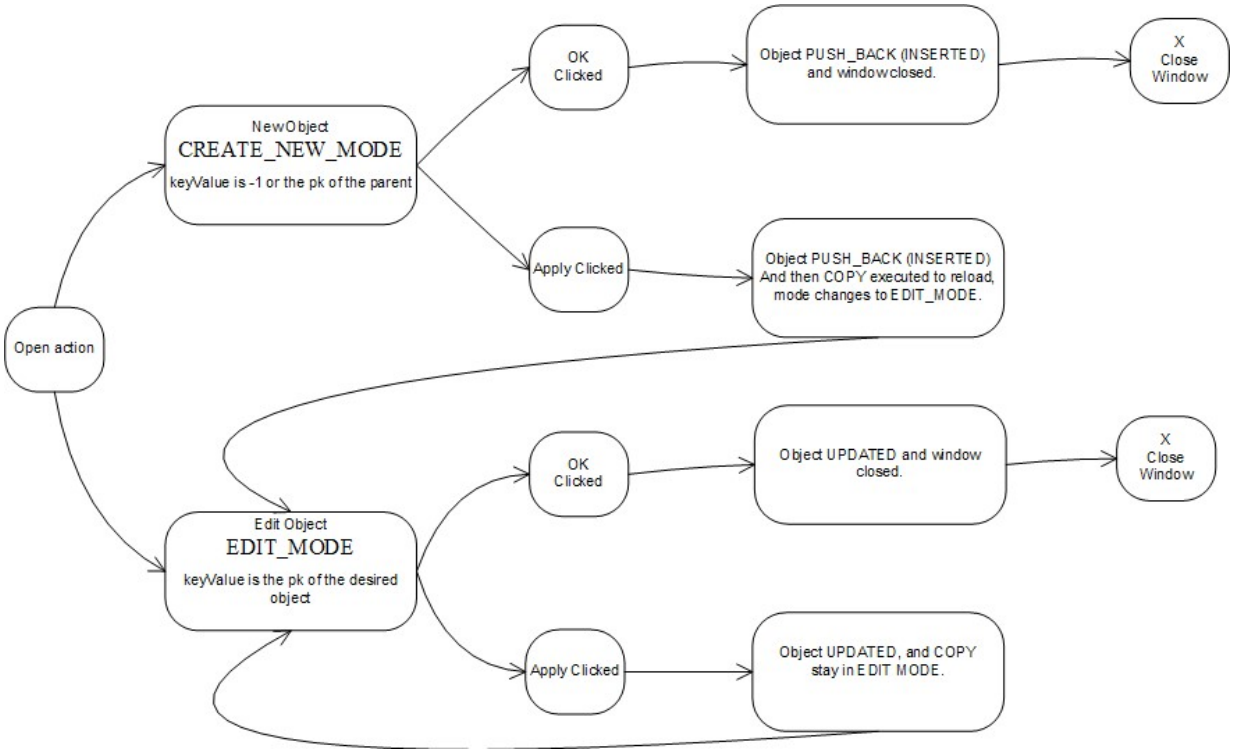


6. ► Finally code the event handlers for window closing, OK and Apply buttons. Select the corresponding object in the GUI builder, then, in the inspector, click "events", then click the action performed handler text box and then hit enter, to create a default handler.

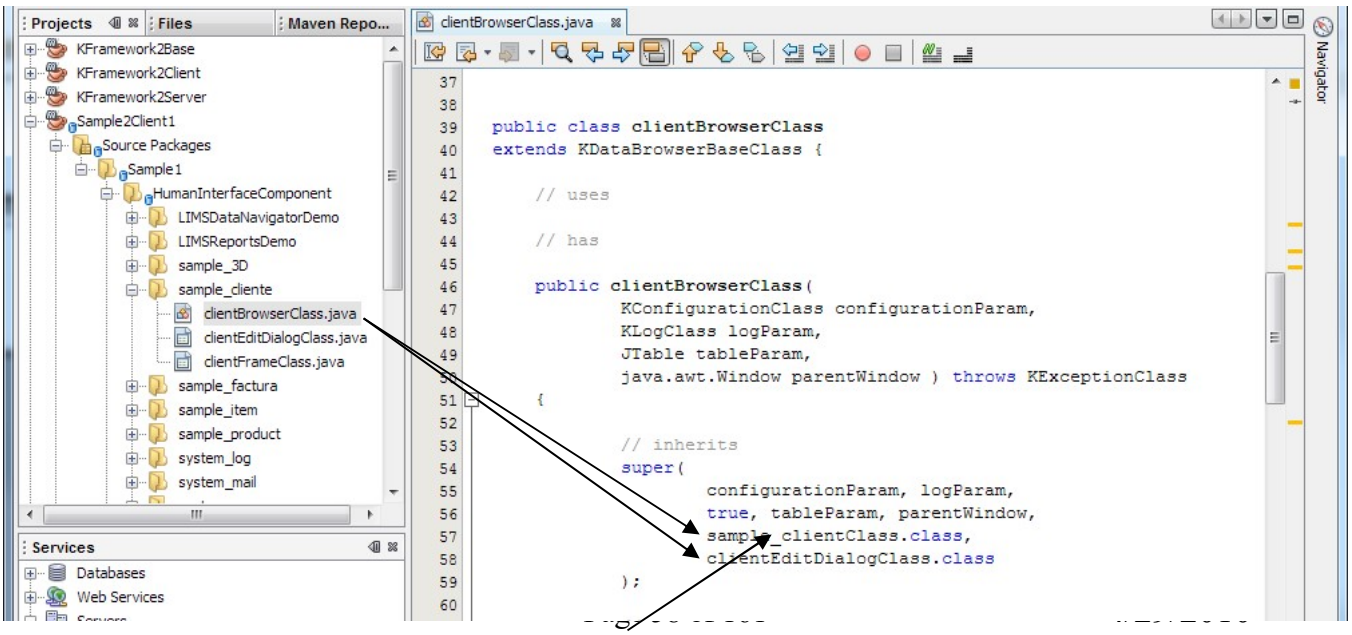



```
    private void closeDialog(java.awt.event.WindowEvent evt) {

        setVisible (false);
        dispose ();
    }

private void applyButtonActionPerformed(java.awt.event.ActionEvent evt) {

        try {

            KDialogController.OKbuttonActionPerformed();

        } catch (KExceptionClass error ) {
              log.log( this, KMetaUtilsClass.getStackTrace( error ) );
              KMetaUtilsClass.showErrorMessageFromException(  getOwner(), error );
        }
}
   private void okButtonActionPerformed(java.awt.event.ActionEvent evt) {

         try {

             KDialogController.OKbuttonActionPerformed();
             setVisible (false);
             dispose ();

         } catch (KExceptionClass error ) {
               log.log( this, KMetaUtilsClass.getStackTrace( error ) );
               KMetaUtilsClass.showErrorMessageFromException( getOwner(), error );
         }
```

Note that the dialogs controller will execute different functions when you click apply and OK depending on the window's state. In the framework all edit dialogs go through the following cycle, depending whether it starts for a new object or to edit an existing object.



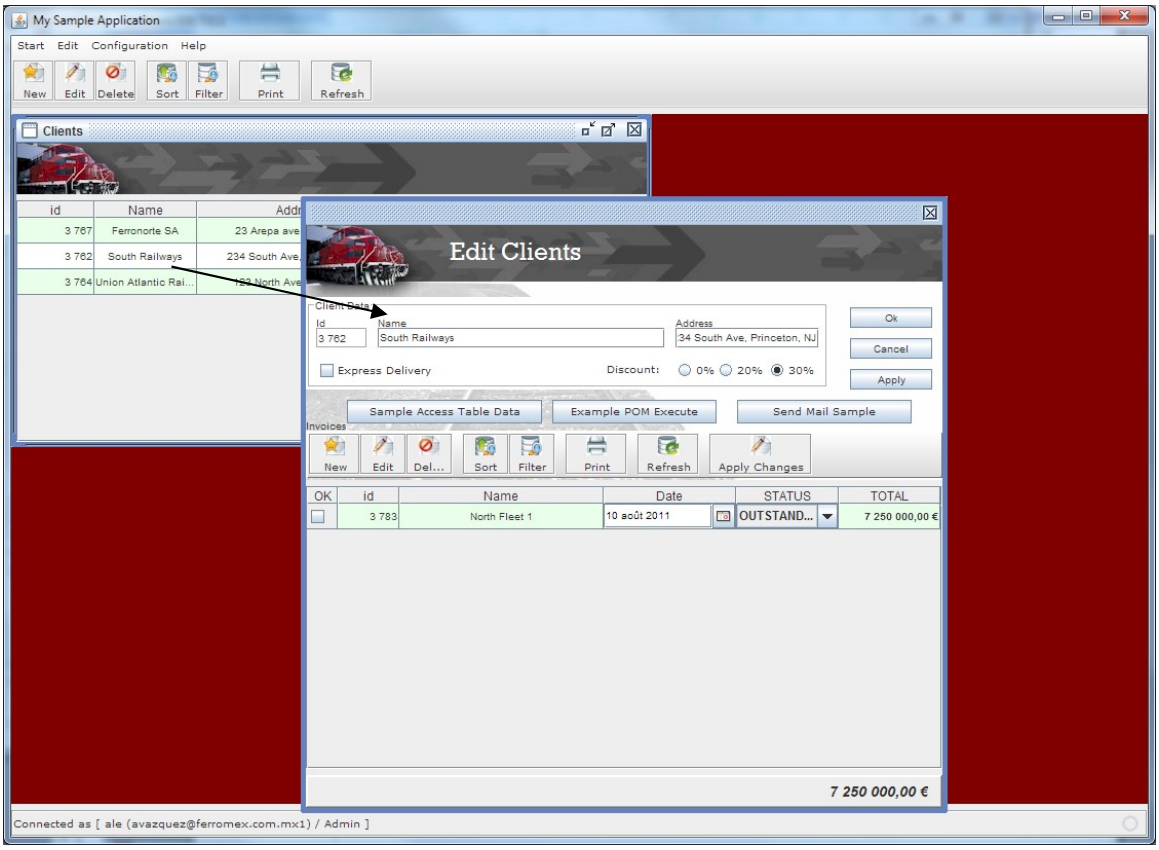
7. ► That's it for the editor window. Editors are always opened in the context of a data table, where you choose the record to edit in the first place. To hook this editor to the application flow, let's open the clients table we created before and declare this editor as the edit window for clients in the constructor, where we left *nulls* before.



1) Set the entity class to be edited and the class of the editor. Note that the class of object is required because in case a user hits "delete" only a confirmation box is displayed, the editor is not opened. You can only delete form tables.

8. ►That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…

Now, when a user clicks edit or double clicks a client row, in any context, our client editor will be opened.



Great you are done with the clients entity. Now we need to repeat the cycle for each entity, until we finish the application. The next section explores how to chain the invoices into the client GUI we just build, as shown above.

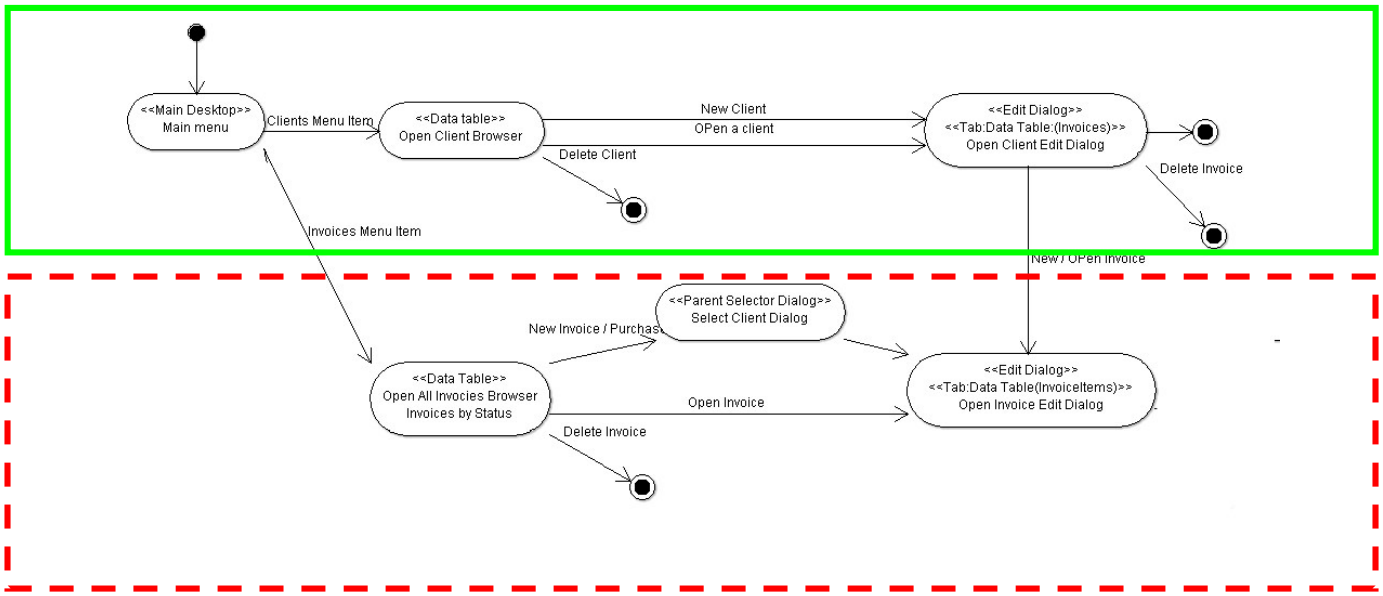
5 Putting it all together

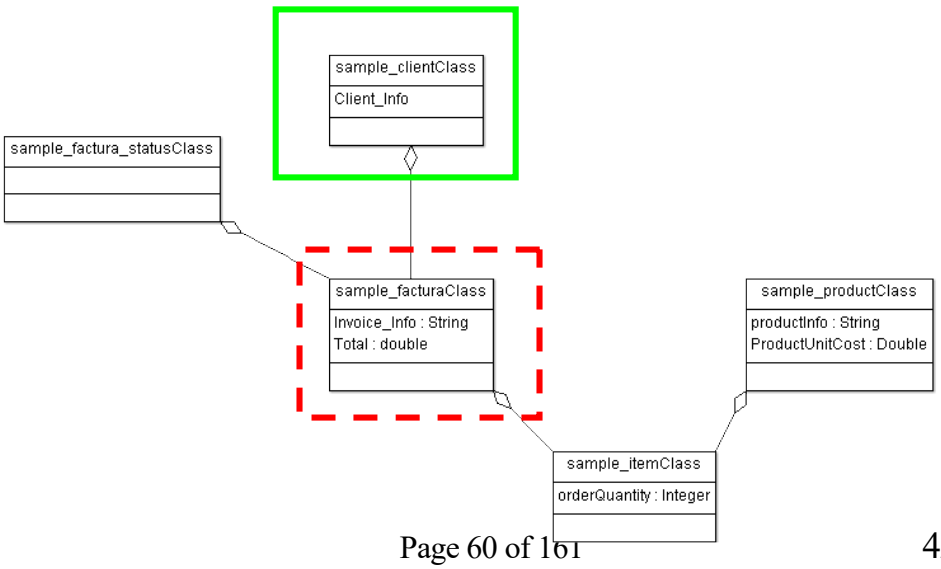
We will now proceed with the demo application and now explore some more advanced topics.

Having a table and an editor for and entity, as described in the previous sections, exemplifies the main GUI elements the application is built from.

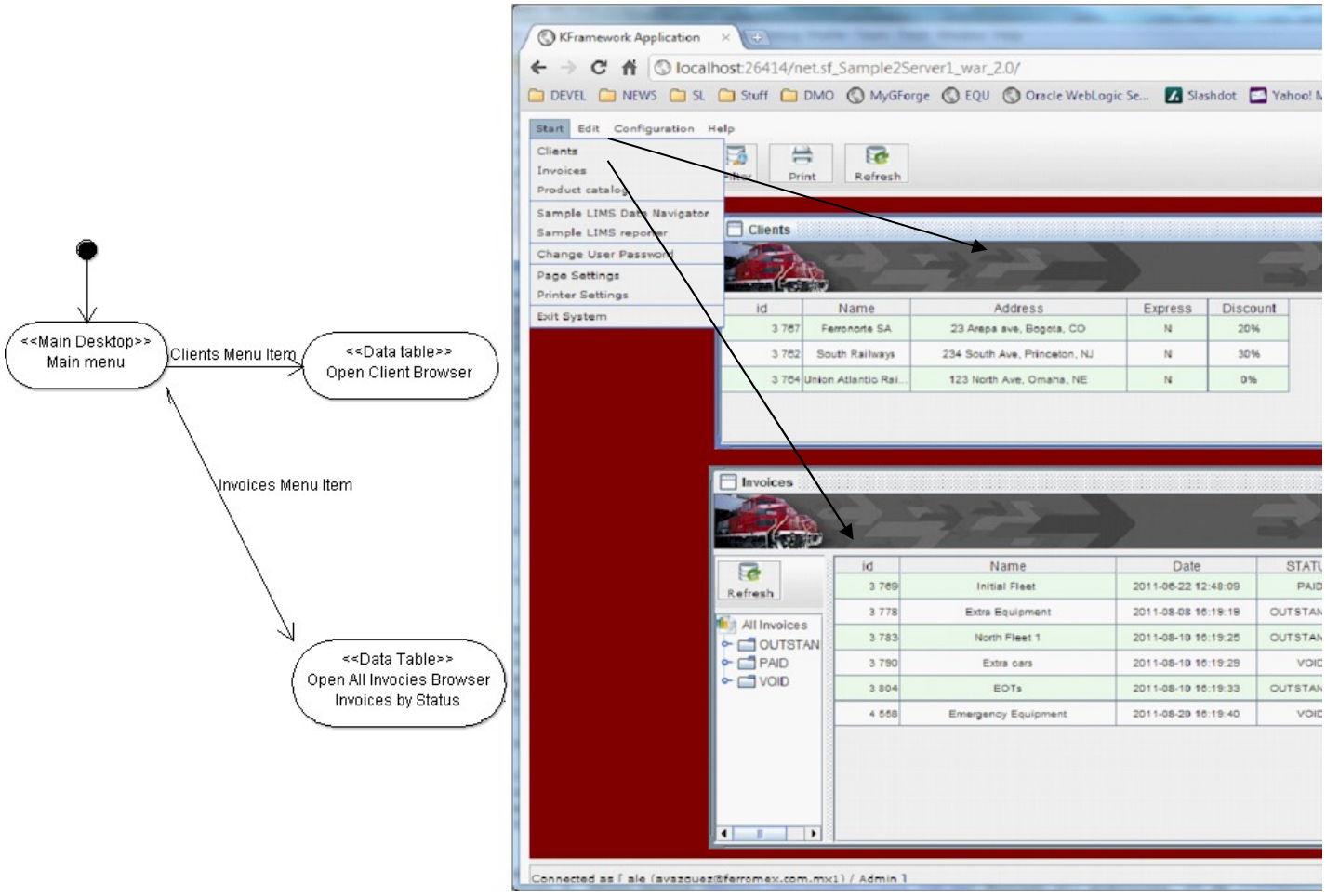
Seldom will an application be so simple. Now we will continue and tie a new entity into the one we created previously. We will go on adding and connecting entities with tables and editors until the whole app is built.

Let's now tie in the invoices for a client so that we continue implementing as agreed in our GUI design:



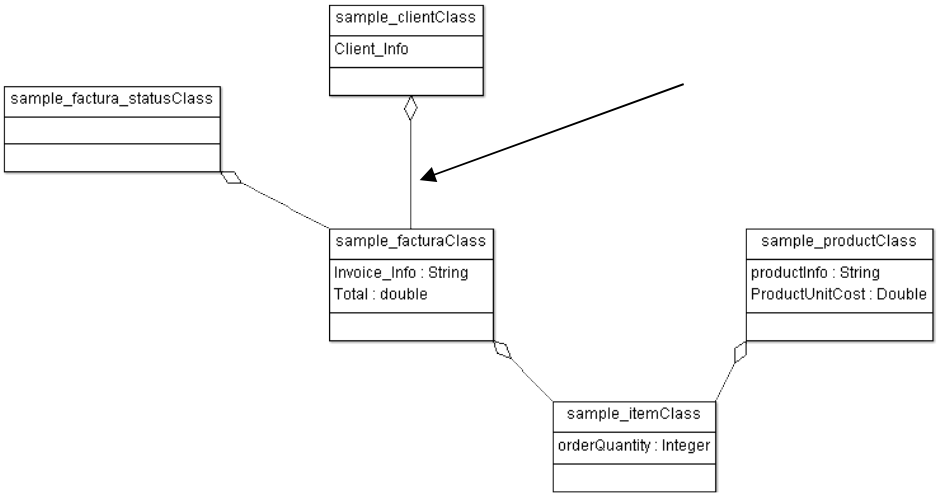


To continue with our demo project proceed and add an entity for the invoices (facturas), and, just like we did previously with the clients, build a data table and an editor. You will end up with 2 tables and 2 editors like this…



Don't worry about the tree view in the picture; we will see how to put that later.

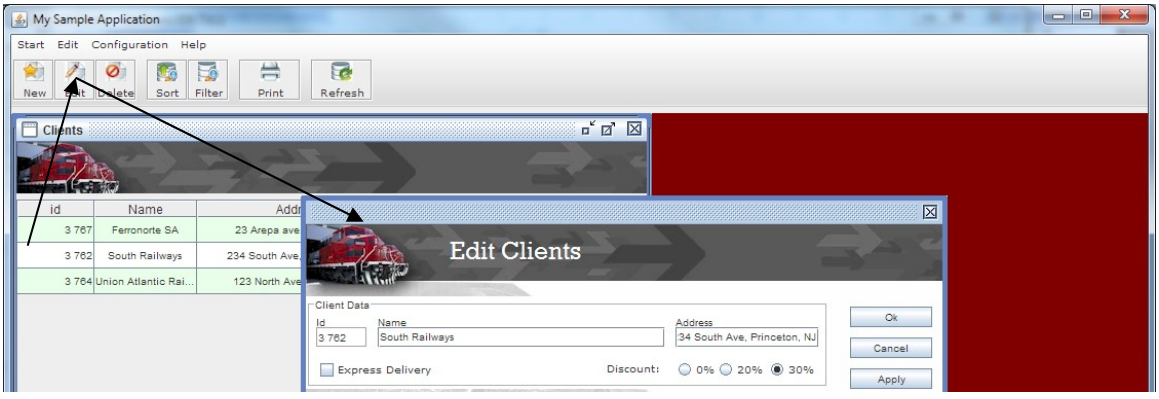
The problem you will face is that when you click new for a new invoice you don't have a value for the foreign key client_id. That's because the design says that invoices always depend from clients, in other words, you need a client to make an invoice, according to the following relation we established before:



So we need to ask for a client first, and then create the invoice for that client. We will never allow orphan records of any type, not even temporarily…

This is done in the invoice table view. If you recall, you make a table view with an initialize method and some SQL. Also remember that, in the constructor of the table, you declared the class of the edit window, because the actions of New/Edit/Delete are always acted over a selected record in a table.

These works because the table super class, the KDataBrowserBaseClass, implements the action handlers for those actions, and since you declared the EDITOR class in the constructor, it knows what edit window to open for the selected row in the table, with out you having to code anything.



5.1 Overriding and extending a table view event handlers

Now we need to pop up a tool to select a client just before we open the invoice editor, and be able to pass the selected key to the invoice so that we can create a proper invoice record. To do this we will override the handler for the NEW action, pop the select client tool, get the key and then call the super implementation to continue and open the invoice editor normally.

The KDataBrowserBaseClass, the super class of all table views, is also an action listener and it provides several action handling methods that you can override. These methods are called when the table is registered as listener of buttons and other sources of events, and the source is set with one of these action commands:

```
static public final String NEW_ACTION = "new";
static public final String EDIT_ACTION = "edit";
static public final String DELETE_ACTION = "delete";
static public final String SAVE_ACTION = "save";
static public final String COPY_ACTION = "copy";
static public final String SORT_ACTION = "sort";
static public final String FILTER_ACTION = "filter";
static public final String REFRESH_ACTION = "refresh";
static public final String PRINT_ACTION = "print";
static public final String MOUSE_DBL_CLICK = "double_click";
```

The corresponding overridable event handlers are:

```
newButtonActionPerformed();
editButtonActionPerformed();
deleteButtonActionPerformed();
saveButtonActionPerformed();
copyButtonActionPerformed();
sortButtonActionPerformed();
filterButtonActionPerformed();
refreshButtonActionPerformed();
printButtonActionPerformed();
```

To fire any of these methods, paint a button, set the corresponding action command and register the table view as listener.

5.2 Using the provided parent object selector dialog

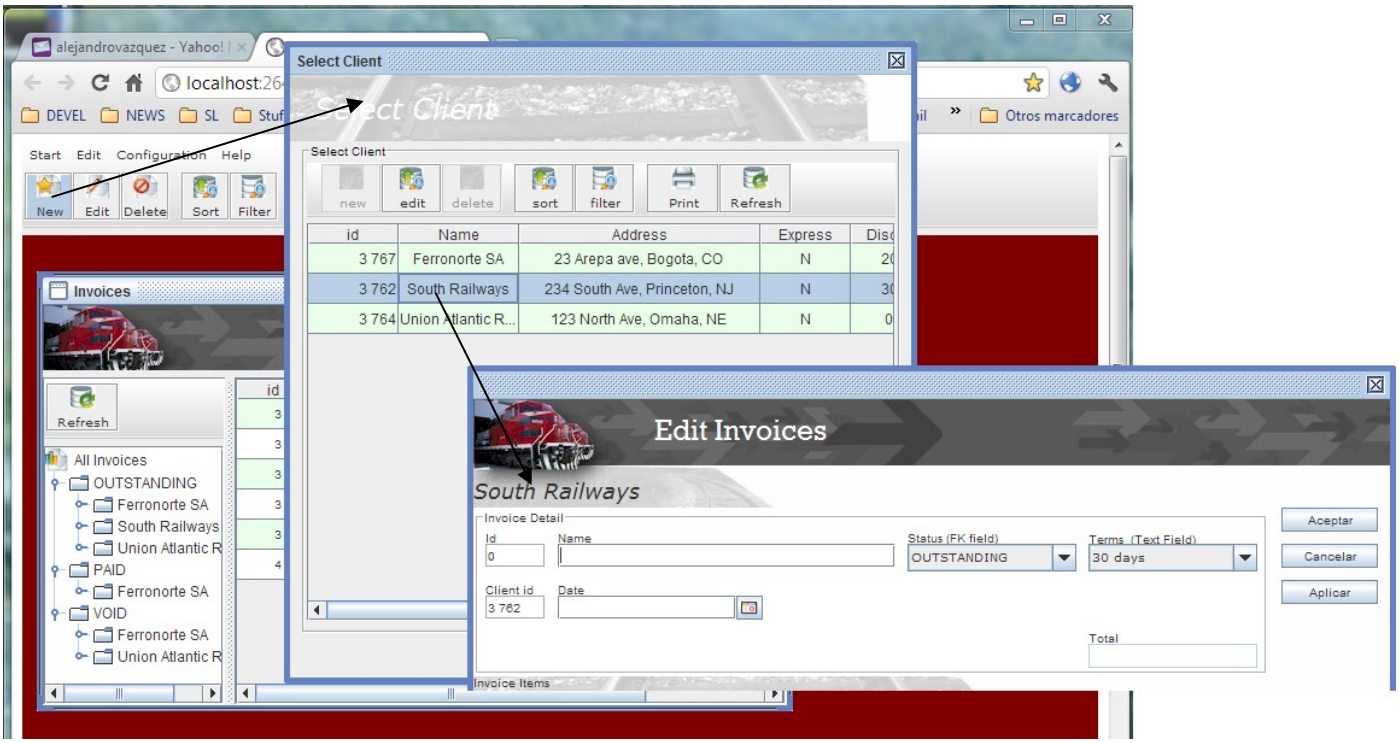
In this case we will override the handler that inserts new records in the table the NEW action. See the previous section on handler overriding.

1. ►Open the invoice table (facturaBrowserClass) and add the following method:

```
@Override
public void newButtonActionPerformed() // need override default to set foreing keys to parent
{
        try{

        // build a client browser
1       clientBrowserClass clientBrowser = new clientBrowserClass(
                configuration, log, new javax.swing.JTable(), parentWindow );
        clientBrowser.initializeTable();

2       selectDialogClass selector = new selectDialogClass(
               configuration, log, parentWindow, clientBrowser, "Select Client" );

        // dont want to allow this, for example
        selector.getNewButton().setEnabled(false);
        selector.getDeleteButton().setEnabled(false);

3       parentID = selector.showDialog();
        if( parentID == -1 ) throw new KExceptionClass( "You must select a client for the invoice!",

         HashMap foreingKeys = new HashMap();
         foreingKeys.put( "clientId", parentID );
4        super.newButtonActionPerformed( foreingKeys );

        }
      catch( Exception error  ){
         log.log( this, KMetaUtilsClass.getStackTrace( error ) );
         KMetaUtilsClass.showErrorMessageFromException( parentWindow, error );
      }
```

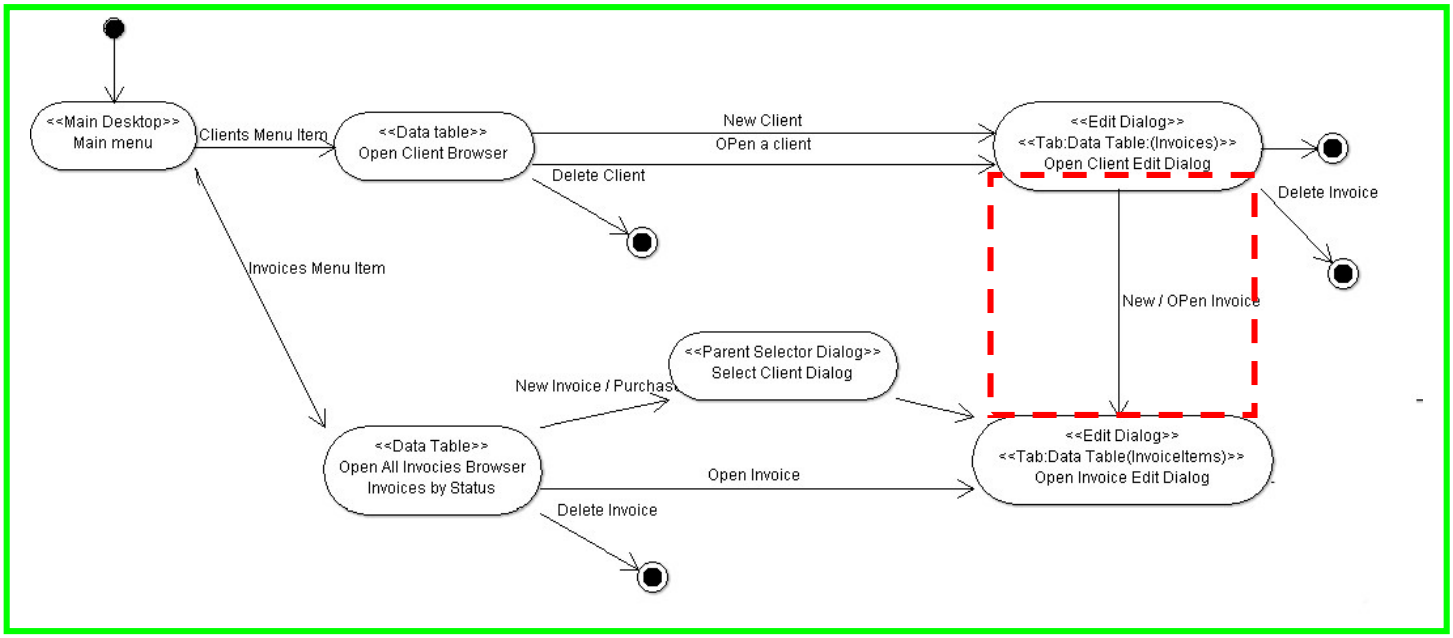
1. First we need a table view of the clients for the user to select. We built one as the first step of this project, we will reuse it. We will not only reuse all the sql and column customization but even the event handlers, just instantiate the client table we build before. Note that we don't have a JTable painted, just pass a new anonymous JTable. The parentWindow is a member of the superclass, just pass it.

2. Now open the generic selector dialog and pass the table we want displayed and set the title of the dialog. The parentWindow is a member of the superclass, just pass it. In this case I want to disable the NEW and EDIT buttons so that users can only choose from existing clients, because in this case the user making invoices can not create clients, you might have a different requirement.
3. Now we pop the window and wait for the user to select a client and get the id. Add a parent_id attribute to the class to store the id, we will need it at that scope later.
4. Finally I proceed and call the super implementation of the NEW which will open an edit window for my brand new invoice. Note how I have to make a MAP to pass the selected client. In this case I need to pass the client_id, but you might need more, put all of them here. The framework will find all this fields in the invoice by name and set the foreign key value.
5. That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…



Here we see how the parent object selector pops when you click new invoice to ask for the client, and then the normal new invoice edit window appears, already set for the previously selected client.

You have now successfully completed the following parts of our demo application:

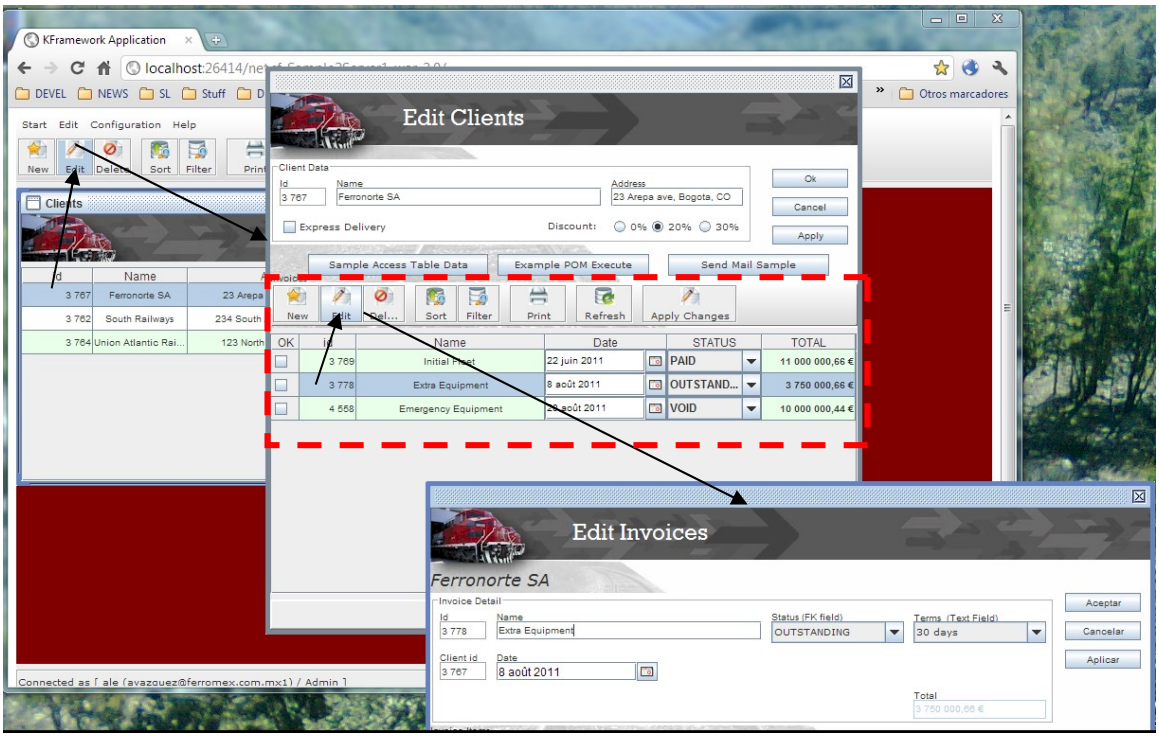


We now can browse and edit clients and invoices. We can event insert invoices and ask for the parent client to maintain data integrity all the time. But the navigation flow is disconnected; from a selected client we can not see the invoices of the client. This might be ok; it depends on your use cases. But in this case we want both, the person managing the invoices to be able to view all invoices, and the managers managing clients to see the invoices for the client they are working with. The Domain Diagram permits it, and the framework makes it natural to do it. Lest see how:

5.3 Linking an entity / sub table in a edit window

Now we want to be able to navigate from the context of a client and see /edit the invoices for that client, apart from being able to see the all invoices view.

If you followed the tutorial up to here, you have table views and edit dialogs for clients and invoices. As you might guess, we will now reuse the all invoices table and insert it into the client edit window, to display the invoices belonging to the client, and to be able to navigate from client to invoice.



5.4 Multiple views in data tables

First we need to filter the invoices table view to show only the invoices for a client.

We reuse the invoice editor so:

1. ►Open the invoice table (facturaBrowser.class) Change the constructor to take two more parameters: The browser mode, either all invoices or client specific, and the id for the client, in case client specific is required. Also, add the corresponding attributes in the class to save the mode and the client_id.

```
// modes
static public final int ALL_INVOICES = 0;
static public final int INVOICES_BY_CLIENT = 1;

// uses
private int                                 mode;
private long                                parentID;
```

```
/** Creates new viajeBrowserClass */
public facturaBrowserClass(
        KConfigurationClass configurationParam,
        KLogClass logParam,
        JTable tableParam,
        int modeParam, long parentIDparam,
        java.awt.Window parentWindow ) throws KExceptionClass
{
```

2. ►Now, in the initializeTable() add the following lines, just **before** the call to super.initializeTable(); These lines will filter the SQL in case we are inside a client context, otherwise it will display both. You can have as many different views as you want.

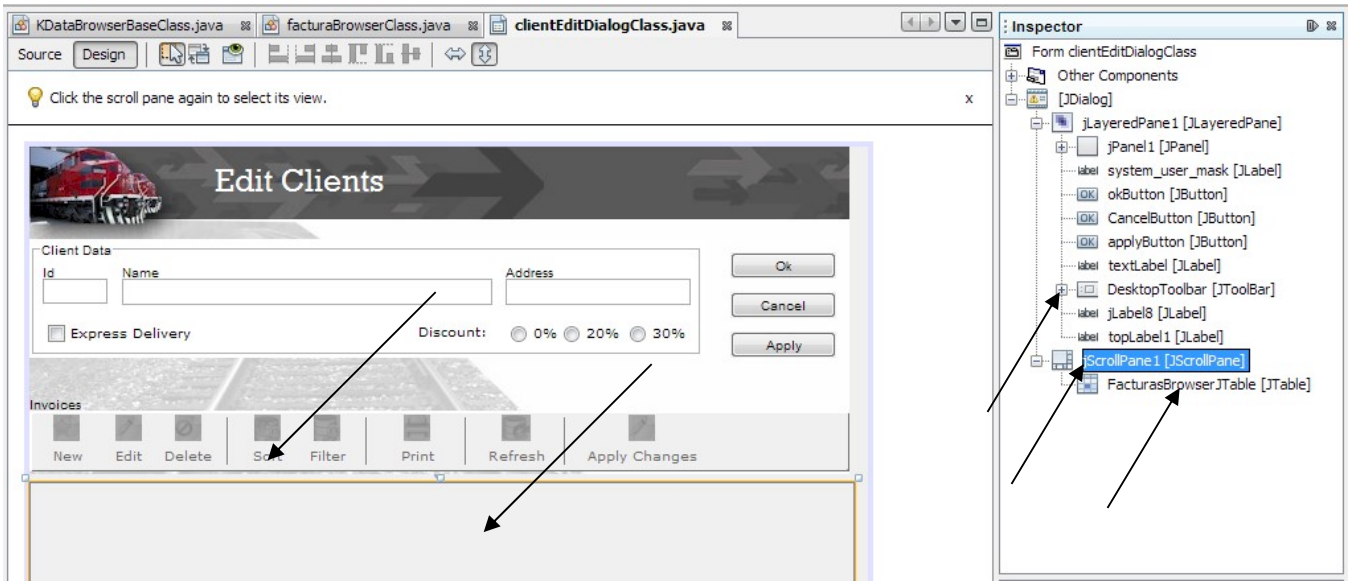
```
setColumnNames( "FAC", "FAC_DATE", "Date", true );
setColumnNames( "STATUS", "FACSTATUS_STATUS", "STATUS", true );
setColumnNames( "FAC", "FAC_TOTAL", "TOTAL", true );

// replace criteria
if( mode == INVOICES_BY_CLIENT ){
    setDefaultCriteria( " client_id = ? " );
    bindDefaultParameter1( ":client_id",  parentID  );
}
```

3. ►Finally we need to change the NEW action handler we override before, to make an IF since, in the context of a client, we don't need to ask for the client id, it will be passed on construction. Of course, only in client mode, leave for all invoices mode.

```
@Override
public void newButtonActionPerformed() // need override default to set foreing keys to parent

{

	try{

		if( mode == ALL_INVOICES ){

		    // build a client browser
			clientBrowserClass clientBrowser = new clientBrowserClass(
					configuration, log, new javax.swing.JTable(), parentWindow );

			clientBrowser.initializeTable();


			selectDialogClass selector = new selectDialogClass(
				configuration, log, parentWindow, clientBrowser, "Select Client" )

			// dont want to allow this, for example
			selector.getNewButton().setEnabled(false);
			selector.getDeleteButton().setEnabled(false);

			parentID = selector.showDialog();

			if( parentID == -1 ) throw new KExceptionClass( "You must select a client
		}

		HashMap foreingKeys = new HashMap();
		foreingKeys.put( "clientId", parentID );
		super.newButtonActionPerformed( foreingKeys );

	}
	catch( Exception error  ){
		log.log( this, KMetaUtilsClass.getStackTrace( error ) );
		KMetaUtilsClass.showErrorMessageFromException( parentWindow, error );
	}
```

Great we now have a table view for invoices, filtered by client. Let's now go and add it inside the client edit window.

1. ►First make some space in the client edit window and add a tab. Inside the tab put a scroll pane and a JTable. Also add a nice toolbar for the actions you will need. New Edit etc. Just paint the buttons and set the action command to any of the ones described in section: *Overriding and extending a table view event handlers.* You can also just copy the toolbar from the main desktop.



2. ►Switch to source code view and add the following attribute in the class:

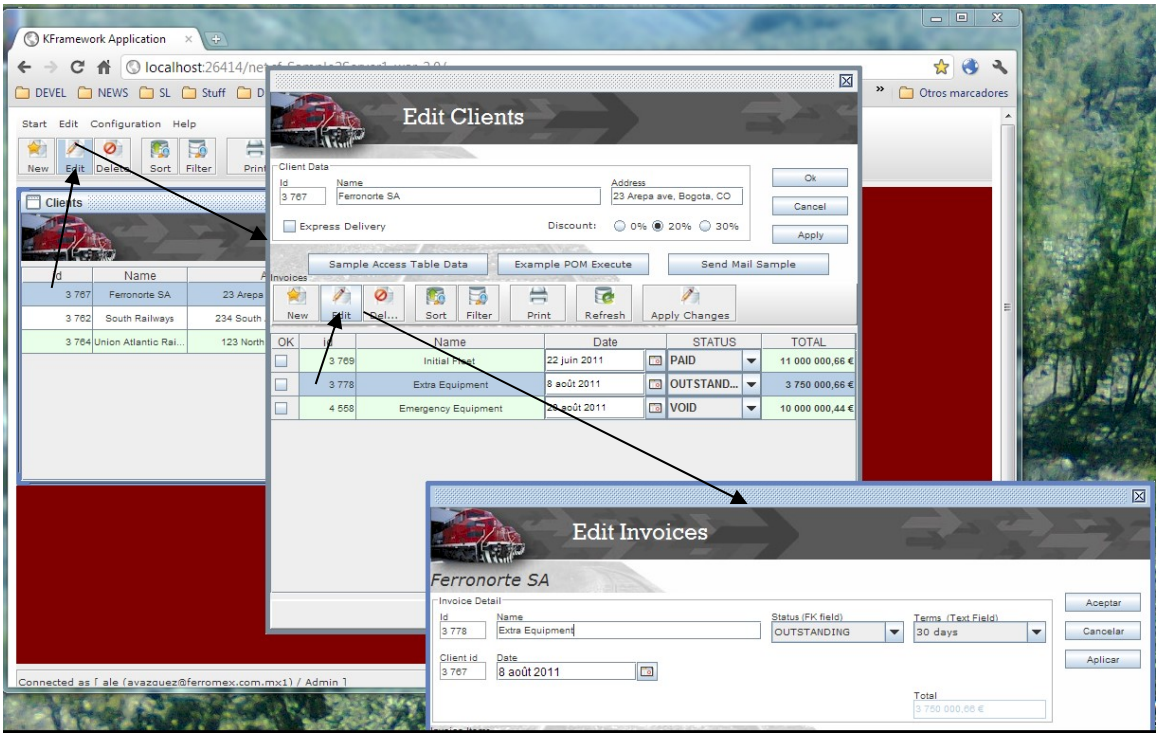
```
private facturaBrowserClass                browser;
```

3. ►Now find the setuTables() method that was left empty before, and add the following code:

```
@Override
public void setupTables( long id )
throws KExceptionClass
{


    browser = new facturaBrowserClass(
            configuration, log, FacturasBrowserJTable,
            facturaBrowserClass.INVOICES_BY_CLIENT, id, this );

    browser.initializeTable();

    //setup container button
    newButton.addActionListener( browser );
    deleteButton.addActionListener( browser );
    editButton.addActionListener( browser );
    sortButton.addActionListener( browser );
    filterButton.addActionListener( browser );
    printButton.addActionListener( browser );
    refreshButton.addActionListener( browser );
    saveChangesButton1.addActionListener( browser );

    saveChangesButton1.setEnabled( true);
    newButton.setEnabled( true);
    deleteButton.setEnabled( true);
    editButton.setEnabled( true);
    sortButton.setEnabled( true);
    filterButton.setEnabled( true);
    printButton.setEnabled( true);
    refreshButton.setEnabled( true);


}
```

Basically, just construct the browser we just customized. You are automatically passed the id of the current client, so you just need to pass it along with the mode and the parent window: this.

6. That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…



Note that for new clients, the invoices table is NOT displayed until after you click "apply", since we don't have a client_id until then.

Also note that you can add and edit invoices automatically, reusing all the code we did before, and, in this context, the new invoice will take the current client as parent, with out asking for a client.

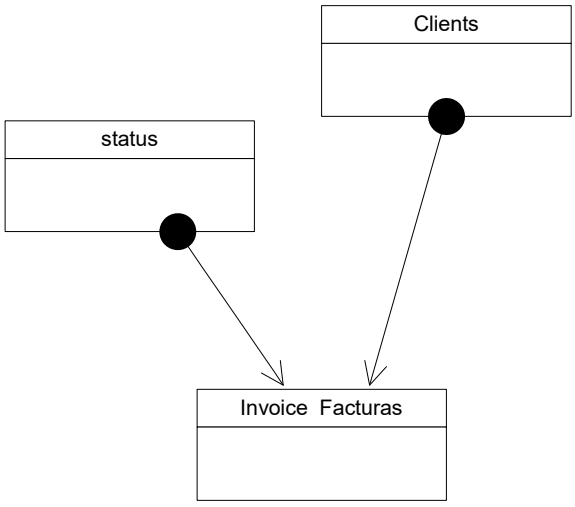
5.5 Using data bound combo boxes

A databound combo displays data based on a table that changes. Others that are hardcoded at design time or with free text input are added like any other widget and mapped automatically by the framework with just setting the name property to a valid entity attribute name.

The next example assumes that the combo will be filled from a catalogue and referential integrity is required. In this case we make a domain object to register for integrity control, and for later use in a catalogue editing UI. If you just need a combo from SQL and you don't require a catalogue and don't need integrity.

Let's say we want to add a status combo box for the invoice objects in the sample. (This example is delivered in the sample project) We assume you already have the invoice edit dialog and browser.

1) ►Make a design



Note that the arrow goes from status to the invoices. Invoices do not have status, but status have invoices. In other words, you need a status first, and then the invoice, there are no invoices with out a status, and erasing a status will be blocked if there are dependant invoices. There are no invoices with more than 1 status, etc. Don't fall in this pitfall.

2) ►Create the corresponding DB table for the Invoices statuses.

The framework will enforce referential integrity to the catalogue, preventing erasing of values selected in objects, or erasing the corresponding dependant objects automatically. Also make sure you follow the PK-FK naming conventions where Foreing Keys are always named as the Primary Key they point to.

3) ►Create the corresponding PDC object. This object is necessary for the automatic integrity to work, and for you, to later on add a catalogue management interface.

```
@Entity
@Table(name = "SAMPLE_FACTURA_STATUS")
public class sample_factura_status
extends KFrameWork.Base.KBusinessObject
implements Serializable {

    @KID
    @Id
    @GeneratedValue( strategy = javax.persistence.GenerationType.TABLE )
    @Column(name = "FACSTATUS_ID")
    private long facstatusId;
    @Column(name = "FACSTATUS_STATUS")
    private String facstatusStatus;

    public sample_factura_status()
    throws KExceptionClass{
    };

    // getter setter -----------------------------------------------------------------
    public long getFacstatusId() { return facstatusId;}
    public void setFacstatusId(long facstatusId) { this.facstatusId = facstatusId;}
    public String getFacstatusStatus() {return facstatusStatus;}
    public void setFacstatusStatus(String facstatusStatus) {this.facstatusStatus = facstatusStatus;}
```

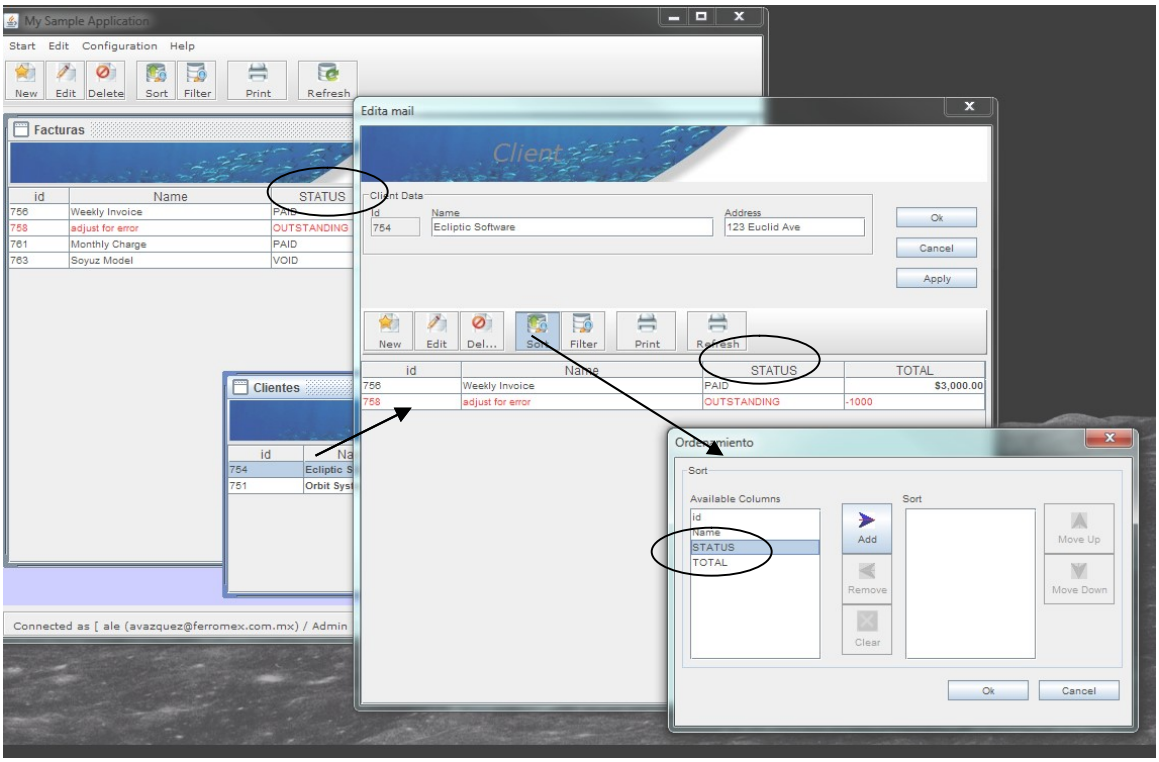
4) ►Register the table in the userBusinessLogicComponentClass at the server for referential integrity handling:

```
// ---------------------------------------------------------------------------
@Override
public void initialize( KLogClass log ) throws KExceptionClass{

    // has
    persistentObjectManager = new KPersistentObjectManagerClass( configuration );

    persistentObjectManager.setCascadeDeletionLogicHandler( this );

    persistentObjectManager.subscribePDCforIntegrityEnforcement( sample_clientClass.class.getName() );
    persistentObjectManager.subscribePDCforIntegrityEnforcement( sample_facturaClass.class.getName()  );
    persistentObjectManager.subscribePDCforIntegrityEnforcement( sample_factura_statusClass.class.getName()  );
    persistentObjectManager.subscribePDCforIntegrityEnforcement( sample_itemClass.class.getName()  );
    persistentObjectManager.subscribePDCforIntegrityEnforcement( sample_productClass.class.getName()  );



}
```

5) ►To make the sample complete, let's fix our invoice browser to display invoice status. See the corresponding section on how to build a browser.

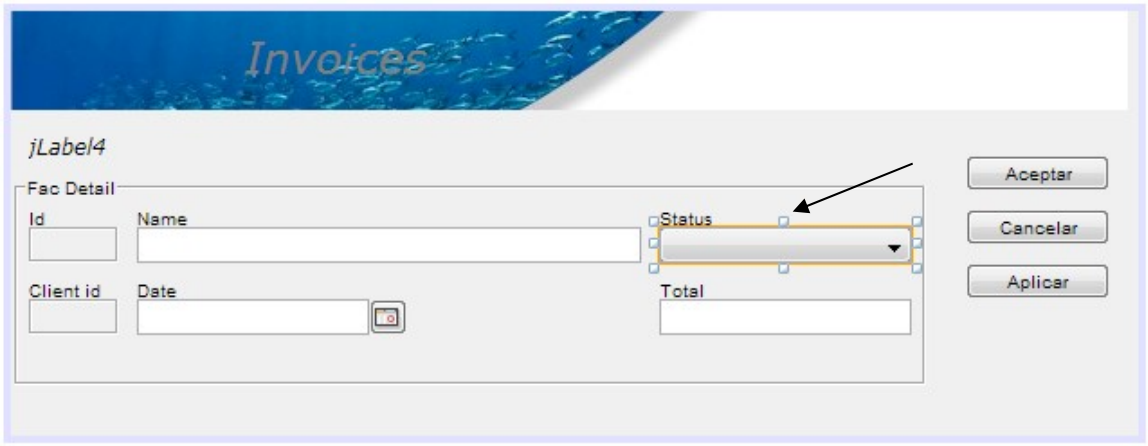
```
public void initializeTable()
throws KExceptionClass
{

        super.initializeSQLQuery(

            // 1 fields
            " fac.fac_id , fac.fac_name, status.facstatus_status, fac.fac_total ",

            // 2 tables and joins
            " sample_factura fac " +
            " left join sample_factura_status status on status.facstatus_id = fac.facstatus_id  ",

            // 3 key of primary PDC object
            "FAC_ID"
                );

        // mayusculas
        setColumnNames( "FAC", "FAC_ID", "id" );
        setColumnNames( "FAC", "FAC_NAME", "Name" );
        setColumnNames( "STATUS", "FACSTATUS_STATUS", "STATUS" );
        setColumnNames( "FAC", "FAC_TOTAL", "TOTAL" );
```

With this single change in one browser the column is available everywhere:



6) ►Now lets setup the combo: Open the invoice edit dialog and:

a) Paint a swing combo box using a regular swing component, and name it as you want, but <u>DO NOT</u> name it as the status field. The field will be mapped to the KFrameWork's dropDownFillerClass.



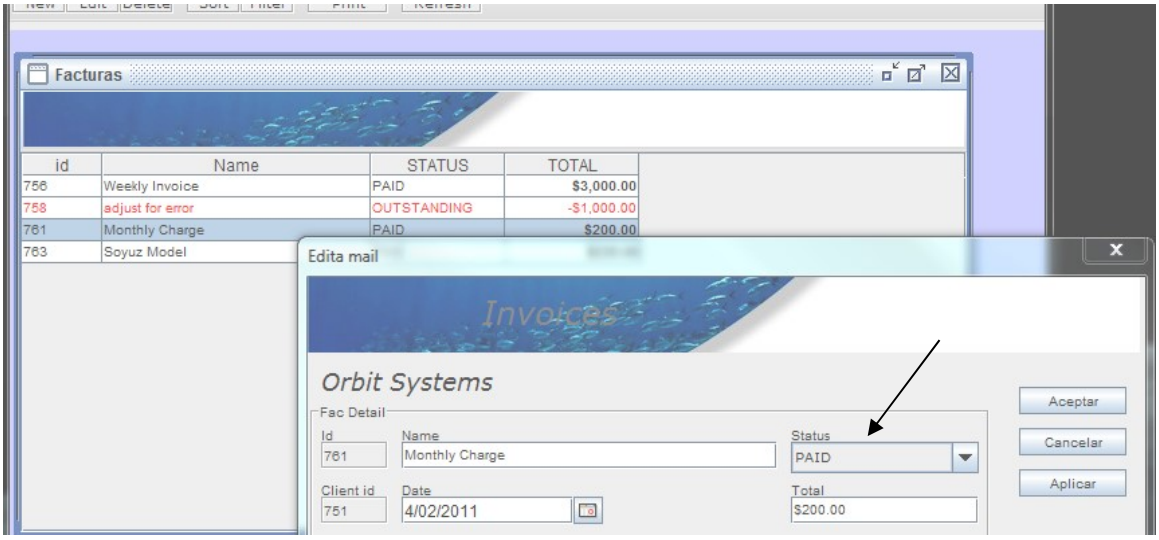
b) In the **initializeDialog() of the dialog**, setup the controller for the combo.

```
public void initializeDialog(int dialogModeParam, Long ID, Map foreingKeys ) throws KExceptionClass {


        //-------------------------------------------------------------
        // status combo, to catalogue FK handeled automatically..
        dropDownFillerClass statusComboFiller = new dropDownFillerClass(
                configuration, log,
                //SQL, might have parameters and where clause or order by
                " select FACSTATUS_ID,  FACSTATUS_STATUS from sample_factura_status ",
                "FACSTATUS_ID", statusCombo, "facstatusId"
                );
        // statusComboFiller.bind( x,  x) bind here any param necessary
        statusComboFiller.load();
        KDialogController.addNonVisibleWidget( statusComboFiller );  // to map it
        //-------------------------------------------------------------
```

Parameters on the constructor of the dropDownFiller are: configuration, log, sql to get the data, PK to bind to the object, the visual combo to fill, and the name of the component, so that the automatic mapper will find it and control it. In other words the field in the PDC object we need to map to.

Note that , at the end, we register the combo controller with the DialogController, so that its added to the dialog lyfecicle control.

7. ►That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…

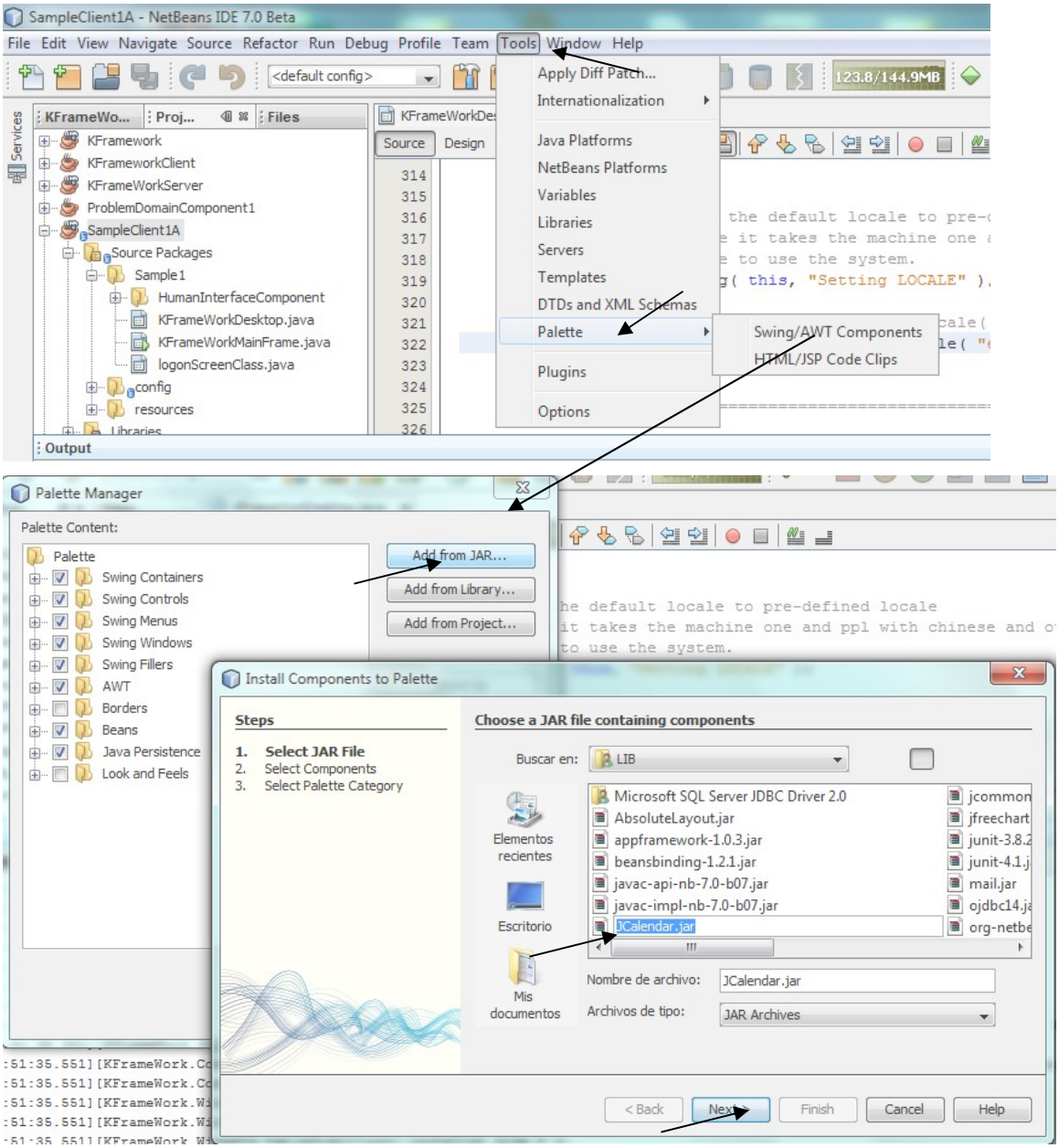


5.6 Integrating a Calendar Control in edit dialogs, or any other 3rd party widget

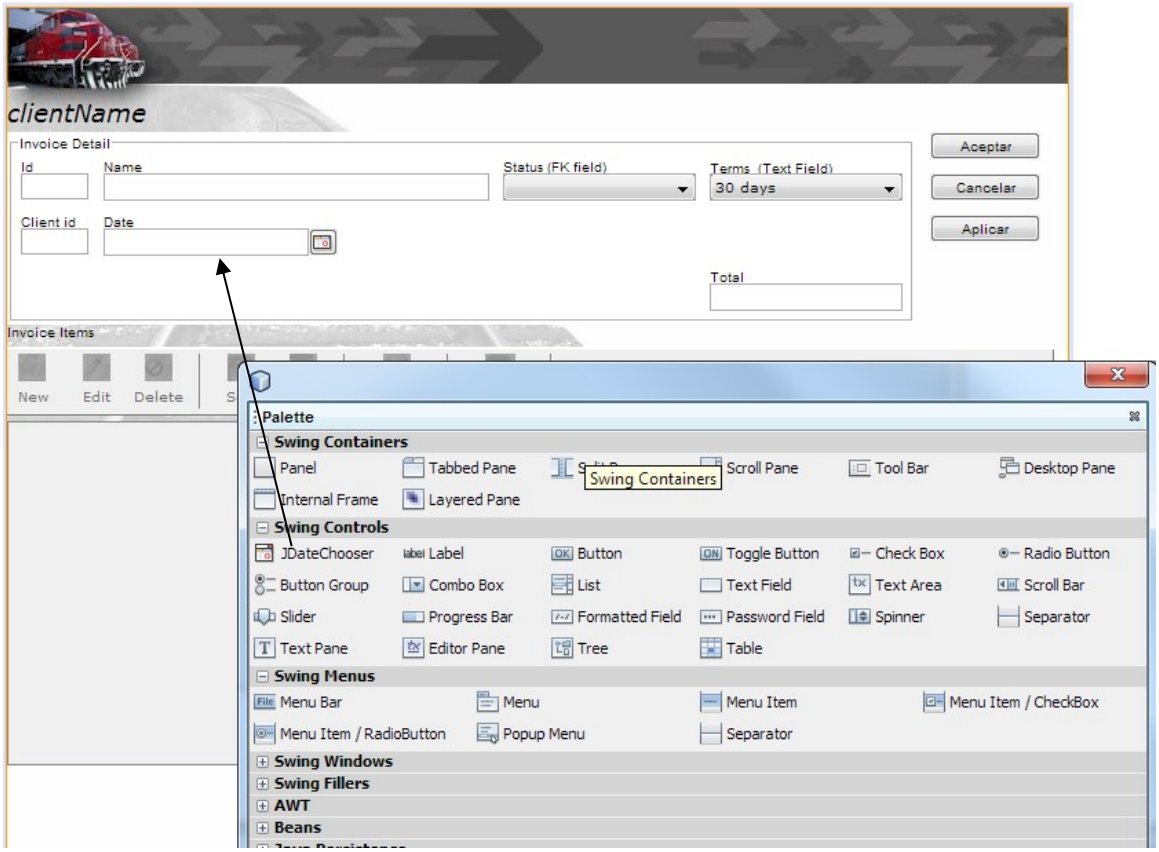
Controls are delivered in JARs. The first thing is to install then into netbeans.

1. ►Install the control's jar.

Installing a third party widget



2. ►Add the control to the Edit Window and DO NOT name if for the field to map to.



Now you need to make a wrapper object that wraps your control and presents a recognizable API to the framework.

In this case we want to integrate the *com.toedter.calendar.JDateChooser* control. So we make a class that will take a jDateChooser and present the API to the framework.

3. ►Make the wrapper class

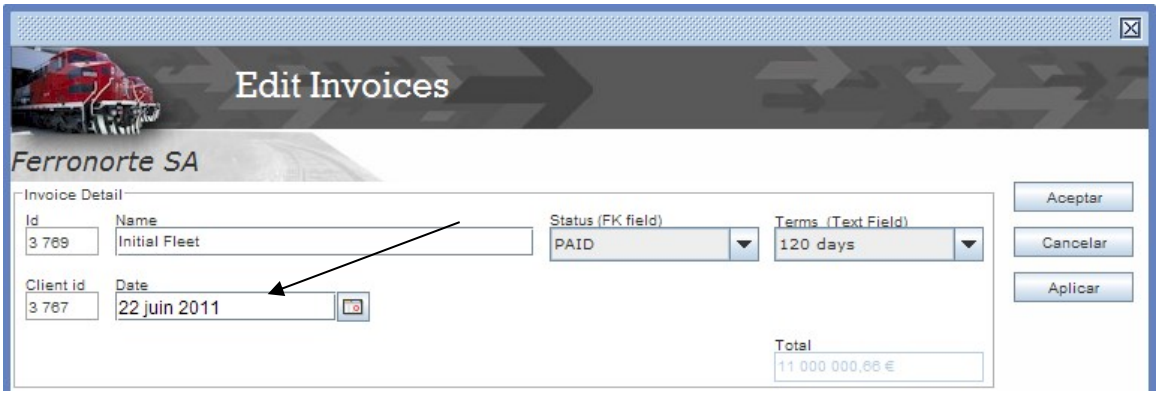
```
public class JCalendarDemoWidgetAdapterClass
implements KCustomWidgetIntegrationIntrerface
{

    // uses
    JDateChooser dateChooser;
    String fieldName;

    public JCalendarDemoWidgetAdapterClass( JDateChooser widgetParam, String fieldNameParam ){

        // uses
        dateChooser = widgetParam;
        fieldName = fieldNameParam;
    }

    public String getName() throws KExceptionClass {
        return( fieldName );
    }
    public void setValue(String newValue) throws KExceptionClass {
        if( newValue == null) return;
        if( newValue.equals("") ) return;
        dateChooser.setDate( KMetaUtilsClass.stringToDate(
                KMetaUtilsClass.KDEFAULT_LONG_DATE_TIME_FORMAT, newValue) );
    }
    public String getValue() throws KExceptionClass {
        return(
            KMetaUtilsClass.dateToString(
                KMetaUtilsClass.KDEFAULT_LONG_DATE_TIME_FORMAT, dateChooser.getDate() )
                );
    }
    public void enable() throws KExceptionClass {
        dateChooser.setEnabled(true);
    }
    public void disable() throws KExceptionClass {
        dateChooser.setEnabled(false);
    }
```

1. First we make a constructor that simple takes the Control and the name to of the field to map to the entity class.
2. getName() returns the name of the field to map to.
3. setValue() is called by the framework at run time to set the initial value of the field, according to the entity being displayed.
4. getValue() will retrieve the current displayed value of the control to map to the entity to be saved.
5. Finally enable() and disable() are called to signal that the control should be grayed out or enabled.

4. ►Finally, in the edit window, find the initializeDialog() method and register your wrapper for mapping to an object's field. Here we pass the control we draw visually and the name of the field to map its value to. This links the drawn control to the editor lifecycle.

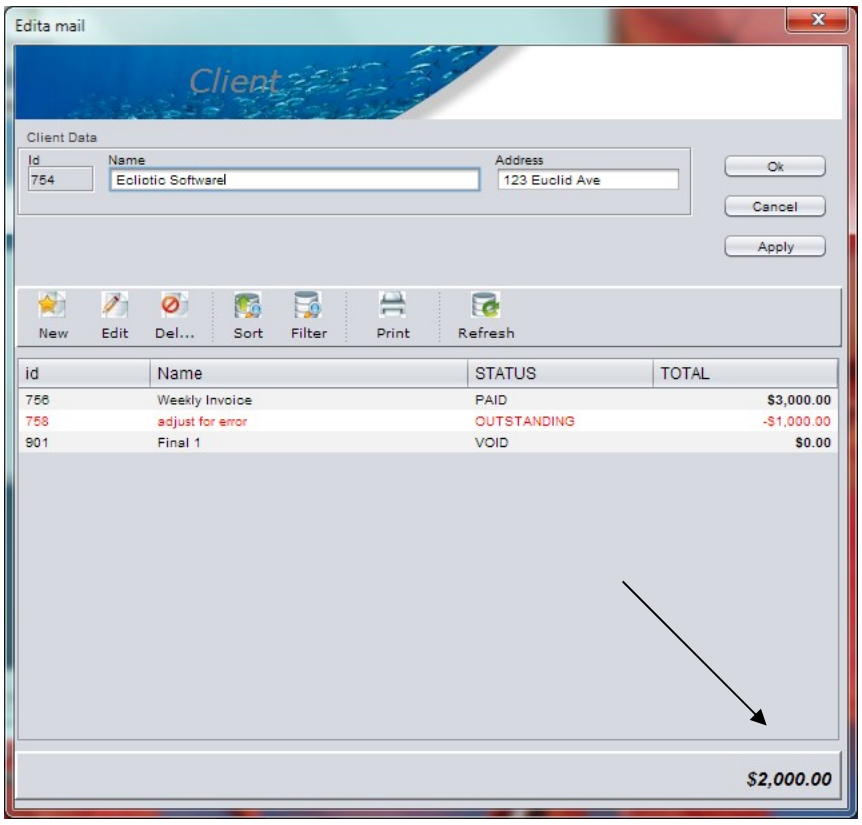
```
public void initializeDialog(int dialogModeParam, Long ID, Map foreingKeys ) throws KExceptionClass {


        //-------------------------------------------------------------
        // status combo, to catalogue FK handeled automatically..
        dropDownFillerClass statusComboFiller = new dropDownFillerClass(
                configuration, log,
                //SQL, might have parameters and where clause or order by
                " select FACSTATUS_ID,  FACSTATUS_STATUS from sample_factura_status ",
                "FACSTATUS_ID", statusCombo, "facstatusId"
                );
        // statusComboFiller.bind( x,  x) bind here any param necessary
        statusComboFiller.load();
        KDialogController.addNonVisibleWidget( statusComboFiller );  // to map it
        //-------------------------------------------------------------


        //-------------------------------------------------------------
        //DEMO cuastom swing component integration with adapter
        // dont forget to visually set the "name" property to the  field you want to map
        JCalendarDemoWidgetAdapterClass dateChooserAdapter =
                new JCalendarDemoWidgetAdapterClass( jDateChooser1, "facDate" );
        KDialogController.includeCustomWidgetForMapping( dateChooserAdapter );
        //-------------------------------------------------------------

    // start
    KDialogController.initializeDialog( dialogModeParam, ID, foreingKeys );
```

5. ►That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…



5.7 Auto calculated fields in forms

Some times, along with a table, you need calculations: A total below the list of invoices, a count of samples, etc. You can assign a formula to a label and make it so that the browser calculates the formula every time data is displayed. For example say you want this:



You don't need to code the logic, just notify the browser to calculate the field. Locate the setupTables function in your edit dialogs and add a saveSQLOperation call:

```
browser = new facturaBrowserClass(
        configuration, log, FacturasBrowserJTable,
        facturaBrowserClass.INVOICES_BY_CLIENT, id, this );

browser.saveSQLOperation(
        totalLabel, " sum( fac.fac_total ) ", facturaBrowserClass.CURRENCY, true );

browser.initializeTable();
```

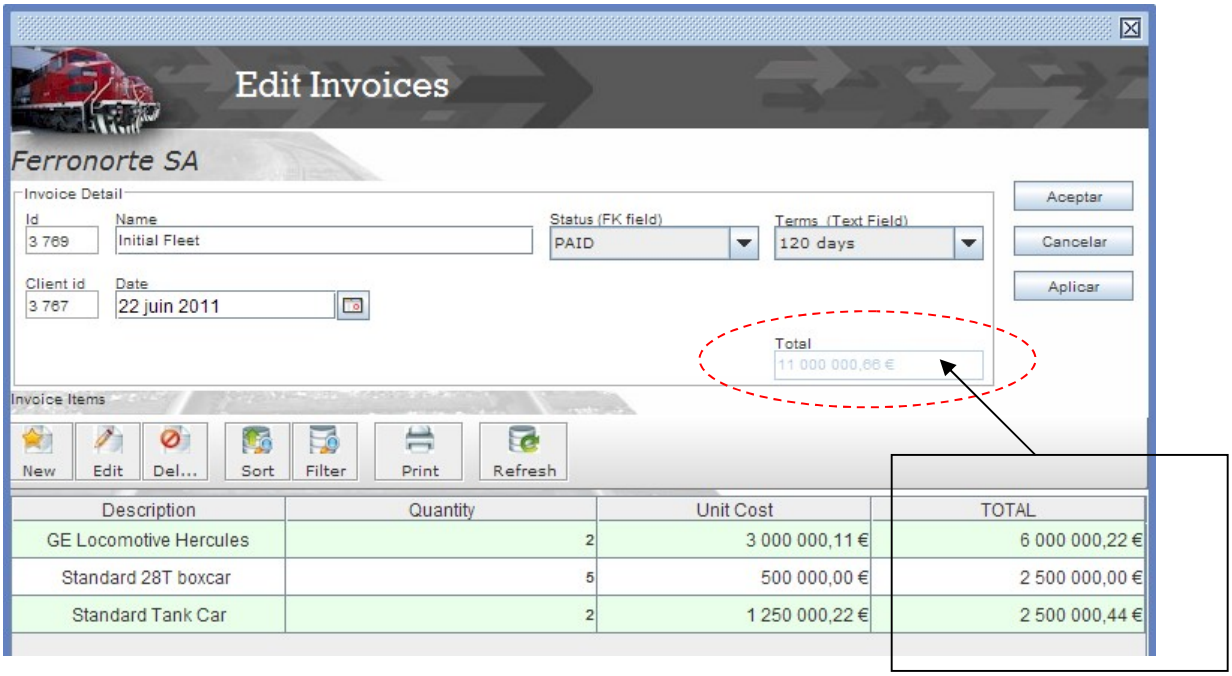
You need to pass the label where to put the result, the SQL formula, the type for formatting and whether you want the runtime user filter to apply.

Any valid SQL is accepted, as long as it can be executed using the browser's SQL. The browser will update any formulas as it loads or refreshes.

You can have as many calculations as you want, but note that too many calculations can take a while to display because the SQL is re-executed for each one. The value is calculated by the database, not the framework.

5.8 Implementing server side business rules / logic and transaction control

Now for our demo project we have a special problem, we have a great invoice edit screen and, after following the previous sections, we integrated a table for all the invoice items as follows:

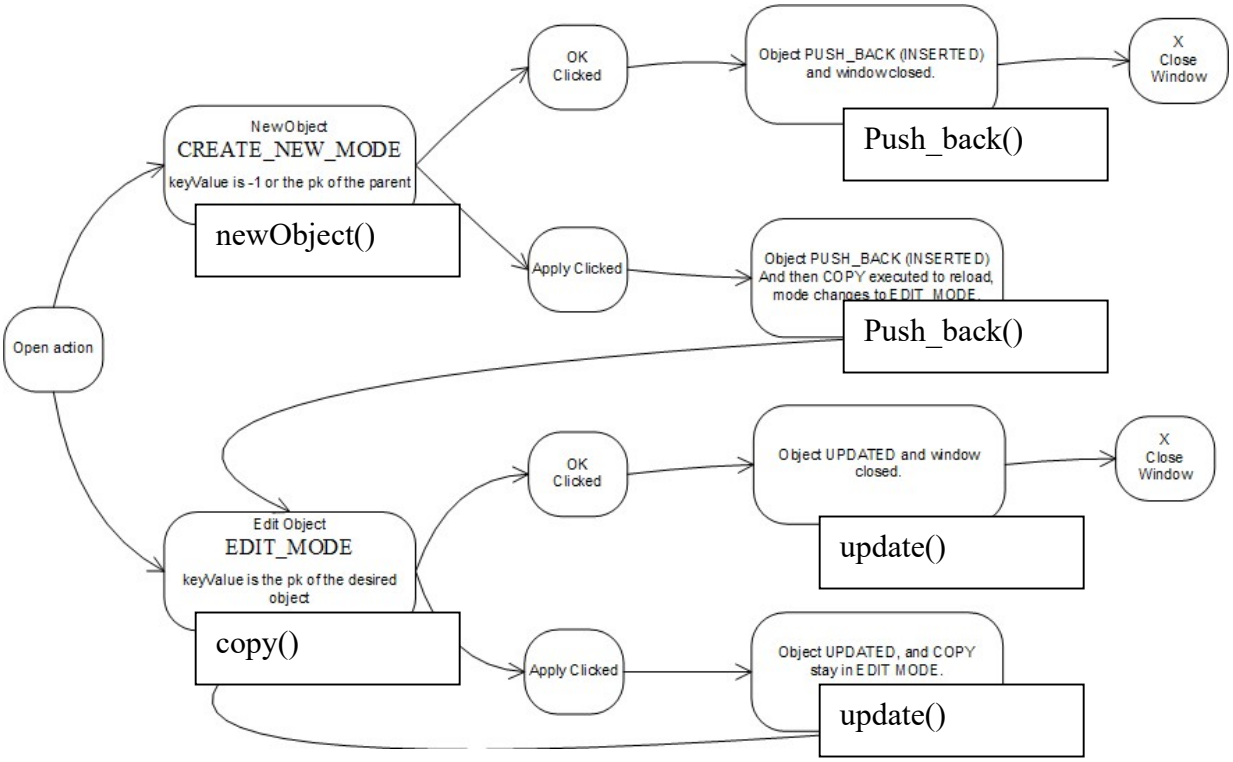


Now we need to put the total for the selected items into the invoice object. Evidently this has to change every time an item is changed. This could be done in the dialog, but then, will we have to call that logic everywhere and item is changed. What if we overlook a place in the GUI where an item can change? The total will change if an item is added and removed, but also if the item is changed. It is clear that this is a server side task.

Thanks to domain drive design the scope of this is delimited to a single class, the invoice items, we don't need to go finding functions every where. Now, thanks to the framework the item's operations are executed in one place: The Server's userBusinessLogicComponentClass. The role of the server is only to handle business logic.

The userBusinessLogicComponentClass is just the main webservice method that dispatched the actions from the UI.

Remember the flow of a dialog, it is mirrored in the userBusinessLogicComponentClass:



So, to achieve the logic to update a client every time an item is added or modified we simply go to the server's userBusinessLogicComponentClass and add our logic, at the corresponding handler in the method processObject Request:

```
// ---------------------------------------------------------------------------
@Override
public KBusinessObjectClass processObjectRequest(
                String action, String objClass,
                KBusinessObjectClass ProblemDomainObject,
                long objID, KSessionClass session, KLogClass log )
                throws KExceptionClass{

    log.log( this, ">>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>Processing request -- " + action );

    KBusinessObjectClass result = null;

    // in this sample one transaction to one db, you might have more than one to many dbs...
    KTransactionClass transaction = new KTransactionClass(
            configuration, log, session.getID(), configuration.getField( "db_pu" ) );

    try{


            //*****************************************************************************************
            if( action.equals( ActionEnum.NEWOBJECT.action ) ){

                result = buildBusinessObject(objClass, log, transaction );

            }else


            //*****************************************************************************************
            if( action.equals( ActionEnum.PUSHOBJECT.action ) ){
```

..etc

1. ►First we will make a method that, generically, updates an invoice totals. Open the persistentObjectManager in the server and add the following method at the end

```
public  void RULEupdateInvoiceTotal( long invoice_id )
throws KExceptionClass
{
    try{

        log.log( this, "Updating invoice -> " +  invoice_id );

        // load invoice (locks by saving the version number, as it checks it on save )
        sample_facturaClass invoice = (sample_facturaClass) copy(
                sample_facturaClass.class.getName(), invoice_id );

        // re-calc total
        Query query = session.getEntityQuery(
                " SELECT SUM( item.itemQuantity * item.itemCost ) "
                + "FROM sample_itemClass item WHERE item.facId = :invoice_id " );

        // set parameter
        query.setParameter( "invoice_id" , invoice_id );

        // execute!
        java.lang.Number returnObject = (java.lang.Number) query.getSingleResult();

        // get result
        Double result = 0.0;
        if( returnObject != null ){ result = returnObject.doubleValue(); }

        // Upadte invoice ..
        // update invoice, checks for verion id not changed during operation}
        // for optimistic locking...
        log.log( this, "New Invoice total:" + result );
        invoice.setFacTotal( result );
        update( invoice );

    }catch( Exception error ){

        // this will rollback the transaction automatically....
        throw new KExceptionClass("Error, could not update invoice " + invoice_id, error );

    }
```

   a. Note how we use the persistent obejct's copy function to get the invoice affected.

   b. Also note how a query is obtained from the session object.

   c. Finally, note that we also use the managers update method to save the invoice.

2. ►Ok, now we go to the processObjectRequest and we hook the method:

In case of insert of item….

```
//*************************************************************************************
if( action.equals( ActionEnum.PUSHOBJECT.action ) ){


        // make sure pk is 0 for auto asign
        ProblemDomainObject.setField(ProblemDomainObject.OIDfield, 0 ) ;

        // execute push
        persistentObjectManager.push_back( transaction, log, ProblemDomainObject );

                //  ------------------------------------------------
                // BUSINESS RULES PUSH OBJECT -------------------

                    // keep invoice totals updated
                    if( ProblemDomainObject instanceof sample_itemClass  ){

                        RULEupdateInvoiceTotal( transaction, log, ((sample_itemClass)ProblemDomainObject).getFacIc
                    }

                // BUSINESS RULES PUSH OBJECT -------------------
                //  ------------------------------------------------

        result =  ProblemDomainObject;

}else
```

In case update of item….

```
//*************************************************************************************
if( action.equals( ActionEnum.UPDATEOBJECT.action ) ){


    // update
    persistentObjectManager.update(  transaction, log, ProblemDomainObject  );

                //  ------------------------------------------------
                // BUSINESS RULES update OBJECT -------------------

                    // keep invoice totals updated
                    if( ProblemDomainObject instanceof sample_itemClass  ){

                        RULEupdateInvoiceTotal( transaction, log, ((sample_itemClass)ProblemDomainObject).getFa

                    }


                // BUSINESS RULES update OBJECT -------------------
                //  ------------------------------------------------

   result =  ProblemDomainObject;

}else
```

For delete the code is slightly different, because we don't get the object only the desired ID to be deleted, so we need to materialize the item first, to get the invoice id, delete the item, and then ran the recalc.

The code in the delete

```
//*****************************************************************************
if( action.equals( ActionEnum.DELETEOBJECT.action ) ){


                //  ---------------------------------------------
                // BUSINESS RULES update OBJECT -------------------

                        // keep invoice totals updated
                        if( objClass.equals( sample_itemClass.class.getName() ) ){

                            // materialize, to see what invoice we are talking about
                            sample_itemClass itemToDelete = (sample_itemClass) persistentObjectManager.copy(
                                    transaction, log, sample_itemClass.class.getName(), objID );

                            // do delete
                            persistentObjectManager.delete( transaction, log, objClass, objID );

                            // update
                            RULEupdateInvoiceTotal( transaction, log, itemToDelete.getFacId() );

                // BUSINESS RULES update OBJECT -------------------
                //  ---------------------------------------------


                        }else{

                            persistentObjectManager.delete( transaction, log, objClass, objID );

                        }
```

Note that, all server actions occur inside a transaction. Every call to a server is naturally an atomic transaction that is protected as a whole. No matter what you do inside any of these handlers, as long as you use the provided session for entity object management and queries all you need to do in case of error is to throw and exception. Everything will be rolled back.

Having one transaction per operation also means that they have to be very fast and you can not tie two GUI actions in one transaction, since we would have to hold the transaction. This is not a limitation, but a way of programming, try never to hold a transaction but for atomic tasks.

That's it for the server logic, the invoice will be updated every time an item is changed. But there is a problem, the edit screen does not update and if you click OK and IF you enabled version control, which you should, it will complaint that you can not continue because the underlying object has changed. We need to update the invoice screen every time something changed in the invoice item browser. Here is how:

3. ►Open the invoice edit window and implement the following interface: *tableToolbarActionPerformedNotificationInterface*.
4. ►You will then have to implement the interface's only method, do like this:

```
// called when a browser has a row changed ...
public void tableToolbarButtonClickedNotification( String action ) {

    try {

            //  .. since the total might have changed, do a reload of the invoice ->
            KDialogController.edit();
            // .. to load the new total which is controlled by the server
            // ... and avoid a record version changed error, for optimitic locking.

    } catch (KExceptionClass error ) {

          // log error
          log.log( this, KMetaUtilsClass.getStackTrace( error ) );
          // show error message
          KMetaUtilsClass.showErrorMessageFromException( getOwner(), error );
```

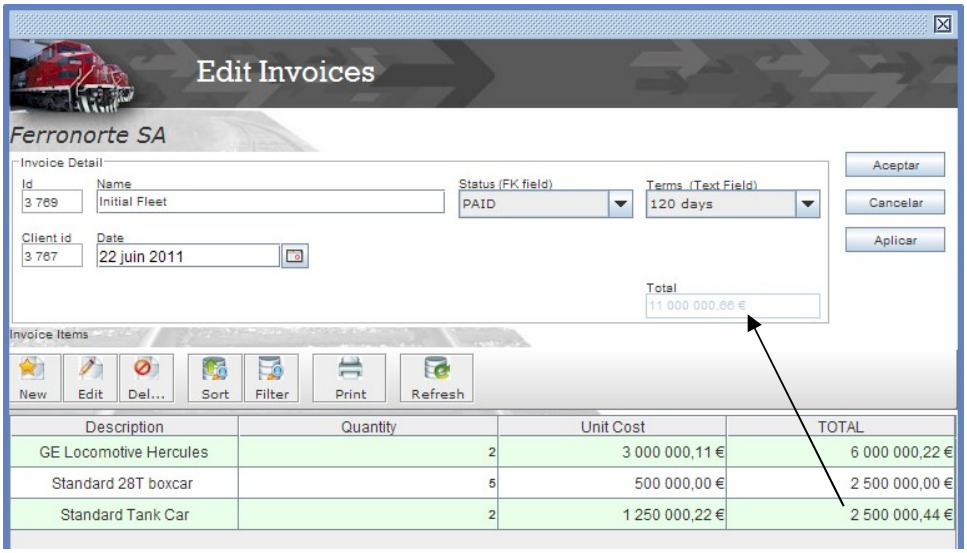
This method will be called by the table view to notify that a change was made. We can do what ever necessary, in this case, we just need to reload the object. We use the dialog's controller to do that.

5. ►Finally find the setupTables() method where tables are built and register the edit window as listener for changes on the invoice items table.

```
@Override
public void setupTables(long businessObjectOID) throws KExceptionClass {

    browser = new itemBrowserClass(
            configuration, log,
            itemBrowserClass.ITEM_BY_INVOICE, businessObjectOID, FacturasBrowse

    browser.initializeTable();

    // register me to be notified for actions on rows, see implement above -> t
    browser.addButtonActionListener( this );
```

6. ►That's it; click run  to test inside Netbeans, or recompile the client, ran the install script, recompile the server and deploy, as explained in the *Setting up and trying the sample project* section, to test from a browser…

Note how, no matter what you or where you do it, any change to an invoice item updates the invoice total. You can do the same to validate STATUS transitions etc.



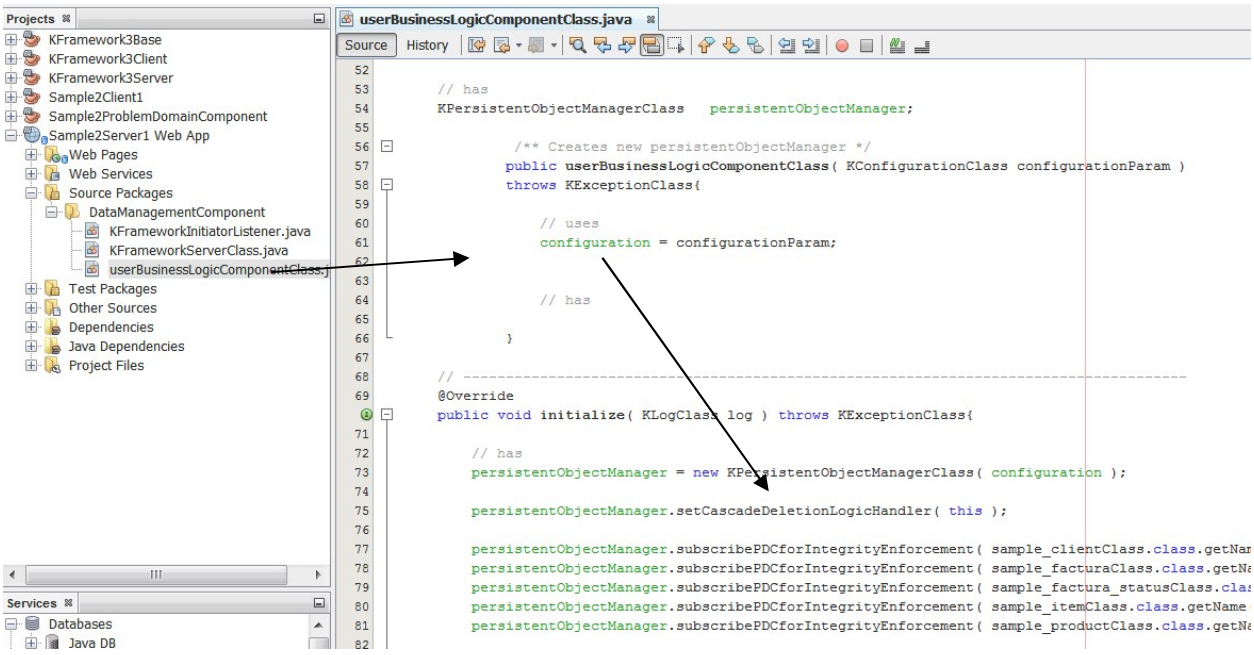
Note that we disabled the total field for user edit, which was done when we created the Problem Domain Entity. See the section on problem domain entities and look into the invoice entity class code to see how to disable fields. This way, no matter where an invoice is displayed, the field is always read only, you don't need to remember that every time you make a new window.

5.9 Configuring the dynamic referential integrity handling

The KFramework's server can control referential integrity for you. That is making sure a delete checks for dependant objects which might be rendered orphan. This is checked at as many levels of relationship as found with no limits.

Unlike the database's integrity control, this is not fixed but dynamic. This means that the rule applied to each object can vary depending on each object's data or any other criteria at runtime, thus allowing erasing a hierarchy of objects when they are flagged void, but preventing when they are flagged as payed, for example.

To beign with, declare the classes for which you want integrity control.



The default for referential integrity enforcement is cascade delete. That is, when an object is deleted all it's decendants are delted.

If this is not appropriate for a relationship, change the rule for that relationship. Referential integrity rules are dynamic and not hardcoded in the K-Framework. Every time an object is deleted, its children are looked for and integrity maintained.

The function called is runCascadeDeletionBusinessLogic in the server's persistentObjectManager. This function is called for every object to be deleted, either by the user or by the automatic referential integrity, for the system to make a decision at runtime. For example, some items might be allowed deletion as long as some status has not been updated, etc.

```
// ------------------------------------------------------------------------------

@Override
public void runCascadeDeletionBusinessLogic(
        KBusinessObjectClass  deletee, String OIDname, long OID,
        KBusinessObjectClass child
        ) throws integrityExceptionClass, KExceptionClass{


    try{


        if( child.getClass() == sample_itemClass.class   ){

            throw new integrityExceptionClass(
                    " No allowed to erase " + deletee.getClass().getName() + " if " + child.getClass().getName() +
                    " persent "
                    );
        }
```

In this simple example, itemClass are not allowed to be deleted, from any parent. This will prevent the invoices to be deleted IF it they have items, otherwise the whole hierarchy is deleted automatically. This will also apply even if the invoice deletion is caused by delete of the client or from any depth of hierarchy.

5.10 Record locking on multiuser environments

By default records are not locked while editing. This means that the database will keep the data from the last person who clicked OK on an editor, for example, and it will not prevent a user from accessing a record that another user is editing.

Normally this is not a problem for most applications but for others, specially the ones that keep monetary values, a problem can occur when user A opens an editor and changes field B, while user X changes field Y in the same record. If user A takes too long he will overwrite value Y with the old value, losing X changes, since he opened the editor before X changes, but saved the data AFTER user X.

There are two ways to prevent this problem and avoid ANY data loss:

5.10.1 Pessimistic Locking Strategy:

Under this strategy all records are locked as a user access them and remain locked until the user releases the records. This ensures that changes will always stay and will never conflict with another user. Problem is that, in practice, a full hierarchy of objects might need to be locked for very long periods of time. Careless users can prevent other users or batch processes to run while they forget to close a window.

Since other users can not access the data, this could severely impact the system's usability. Given the distributed and multiuser nature of the web, this extreme method is seldom used and it is NOT supported by the framework by default, though you can implement it using the persistent user manager and the session.

5.10.2 Optimistic Locking Strategy

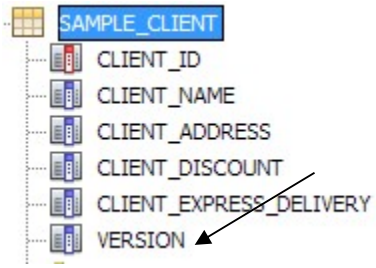
Under this strategy records are not locked as the users access them, hoping that a conflict will not normally occur. Nevertheless every time a user makes a change in a record a version number is advanced on each record. If a user edits a record the current version is noted and, if the record has been changed during the edit as evidenced by a new version number, an error is shown. In this situation he needs to cancel his change, open the editor again with the updated record and try it again. This is the method implemented by the framework.

5.10.3 Using Optimistic Locking Strategy / Automatic Object Version Control

Using optimistic record locking is very easy. You need to do 2 steps for each Problem Domain Object you want to protect. Normally, if you will be using version numbers, you want to do this for ALL problem domain objects.

Note that, unlike other frameworks, there is no last changed timestamp or user id in the KFramework for each PDC object. Remember that the KFramework keeps an Audit Trial (log) of all changes for all objects that includes the full change history for every record. Refer to this log if you need to know who changed what and when. See the corresponding section in this manual.

a) Add a simple numeric field to all your Problem Domain Object's tables.



This field will keep the version numbers.

b) Add the field to your entity, like any other field, and annotate it with the *"@KObjectVersion"* annotation. Don't forget to add the getter and setters.

```
@Entity
@Table(name = "SAMPLE_CLIENT")
public class sample_clientClass
extends KBusinessObjectClass implements Serializable
{

    @KID
    @Id
    @GeneratedValue( strategy = javax.persistence.GenerationType.TABLE )
    @Column(name = "CLIENT_ID")
    private long clientId;


    @Column(name = "CLIENT_NAME")
    private String clientName;

    @Column(name = "CLIENT_ADDRESS")
    private String clientAddress;

    @Column(name = "CLIENT_EXPRESS_DELIVERY")
    private String clientExpressDelivery;

    @Column(name = "CLIENT_DISCOUNT")
    private String clientDiscount;

    @KObjectVersion
    @Column(name = "version")
    private long version;
```

That's it. The framework will now keep and verify the versions and will prevent any data overlap.

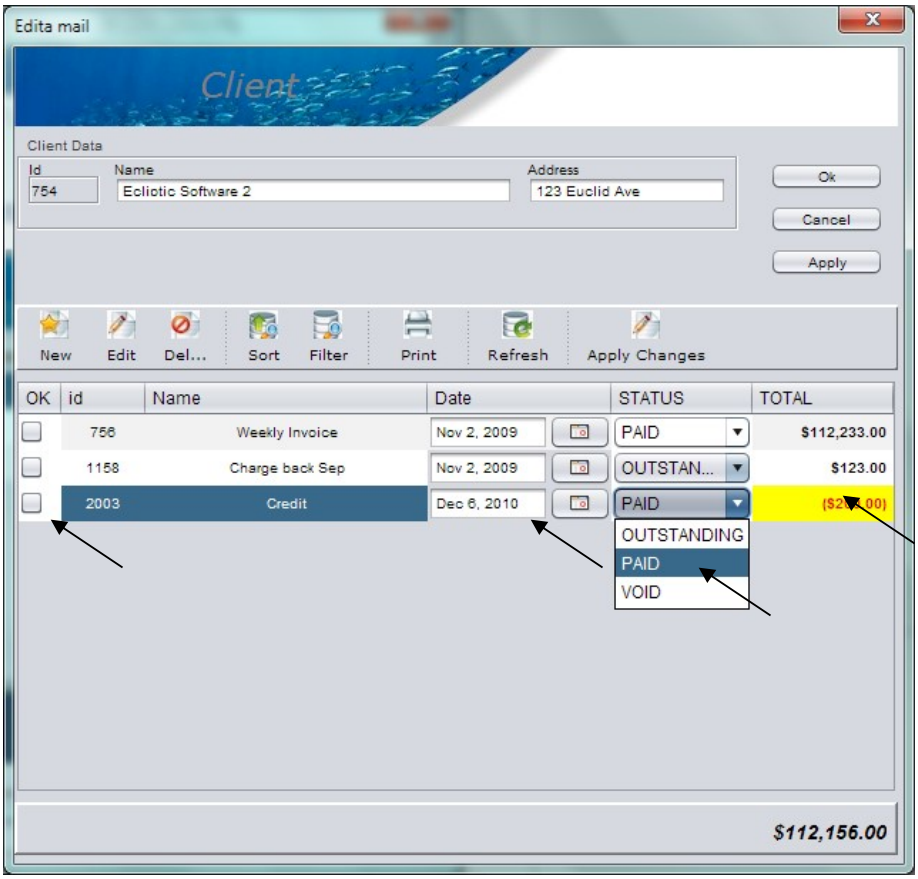

Note that only one field can be annotated as the version number.


Also, this field is now controlled by the framework internally. You DO NOT need a label or box in your dialogs to keep it. A field is automatically generated and added to the materialization's and visualization's extra fields list.

5.11 Data entry tables and custom cell rendering

Browsers can be activated to receive input and display special widgets for cell editing and display, like calendars, combo boxes etc. Additionally, you can customize each cell's display at runtime based on data, to change the cell's foreground color for example.

Table Browser enabled for data entry and custom cell widgets



5.11.1 Receiving input in browsers

This assumes you already know how to setup a regular browser, if not, check the corresponding section.

To recap a little, in a regular read only table browser the main setup function looks like this:

```
public void initializeTable()
throws KExceptionClass
{

        // set the SQL
        super.initializeSQLQuery(

            // 1 campos
            " client_id , client_name, client_address ",

            // 2 tablas and joins
            " sample_client cli ",

            // 3 llave principal (mayusculas!)
            "CLIENT_ID"

                );

        // define column settings
        setColumnNames( "cli", "CLIENT_ID", "id" );
        setColumnNames( "cli", "CLIENT_NAME", "Name" );
        setColumnNames( "cli", "CLIENT_ADDRESS", "Address" );

        setDefaultOrder( "  client_name " );

         // load data
         super.initializeTable();

         // some customization
         adjustColumnWidth( "Name", 100 );
         adjustColumnWidth( "Address", 200 );
         //adjustColumnFont(  "Name",  new Font( "arial", Font.BOLD, 10  ) );
         //adjustColumnForegroundColor( "Name", Color.BLUE  );

 }
```

You need to set the SQL, define the column display settings, initialize and finally do some customization like setting fonts and colors.

To make a cell available for input, just call the setColumnNames as shown above, and set the read/write parameter to true. The actual signature of the setColumnNames method is:

```
setColumnNames( {table.alias}, {db_fieldName}, {DisplayName}, {isEditable} );
```

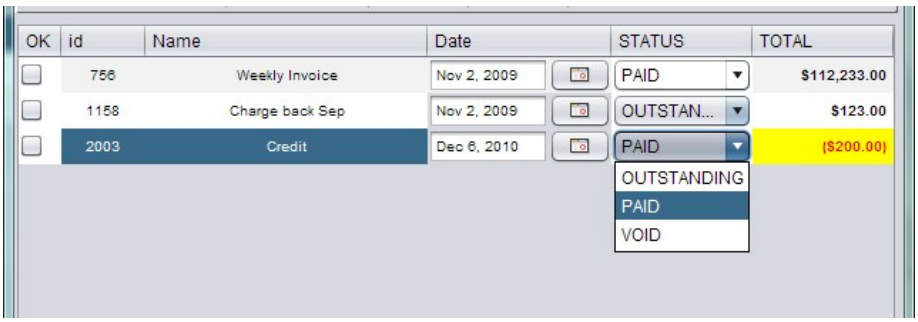
The version in the code example above is in reality an overload that sets the isEditable parameter to false by default, so for example, to make client_name available for input just call setColumnNames like this:

```
setColumnNames( "cli", "CLIENT_NAME", "Name", true );
```

At runtime, when you click on the name field, a text editor will appear for the user to enter data.

The browser can take a complex query with several joins and sub queries, to avoid limiting this functionality and to integrate it to the framework's PDC object transport, the entered data is no saved automatically, but sent to a user method for processing.

For example, to save the data of the following browser (The example depicted here can be found in the facturaBrowserClass of the KFrameworks delivery in the sampleClient project) :



We want the browser class itself to process the data so:

1) Implement the *KBrowserDataWriterInterface* in the browser.

```
public class facturaBrowserClass
extends KDataBrowserBaseClass
implements cellRenderingHookInterface,  // to customize the data at runtime
KBrowserDataWriterInterface // to make it RW
{
```

2) Implement the *save* method:

```
@Override
public void save( java.util.List< String > fieldNames, java.util.List< recordClass > data ){

    try {

       // for you to see
        Iterator dataRowChanged = data.iterator();
        while( dataRowChanged.hasNext() ){

            recordClass currentRow = (recordClass) dataRowChanged.next();

            // materialize object
            sample_factura factura = new sample_factura();

            persistentObjectManagerClass pom = new persistentObjectManagerClass(configuration, log)
            pom.copy( Long.parseLong(  currentRow.getValueAt(6) ), factura  );
```

The save method is fired when the browser receives the save event. See below for more details on that. It will pass two arguments: A list of fields in the table for your reference and a List of records, each containing a full row of data. You receive only rows that where changed.

So, in the example above we iterate the rows, or the data list of records. Then, to get a specific column's value we do:

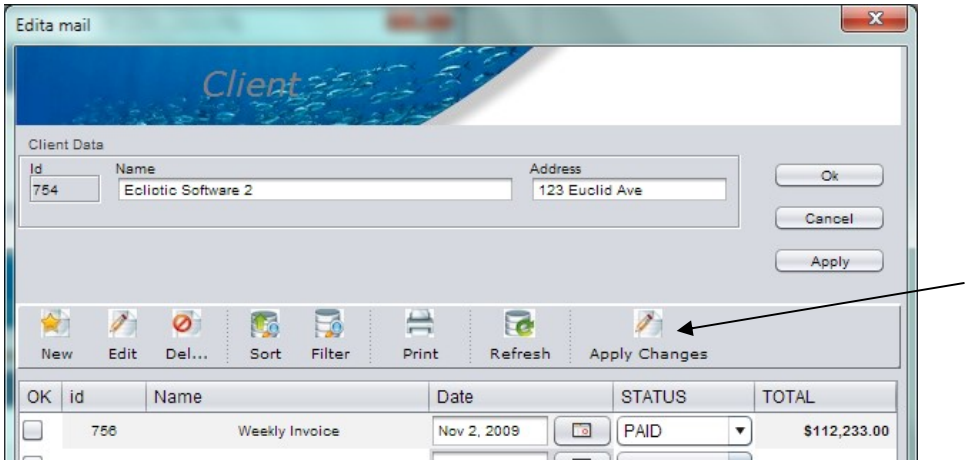
```
Record.getValueAt( {ColumnIndex} );
```

The data is always returned as a string, even for numbers. Dates are returned as ISO yyyymmdd and Booleans as true or false.

All browsers have a hidden column at the end, where the primary key of the current row is stored. In the example above we have 0-5 fields, we access the 6th field to get the primary key of the invoice and materialize the object for edit and save. See the code example. But you might as well use the data and call a webservice or whatever else you might need.

5.11.2 Browser's save event

All browsers have a save event handler just like the new, edit, delete, sort or print, so you can just put a button, set its action command to "save" , and set the browser as a listener.



For example, in the above example where a client editor handles the facturaBrowser, we paint a button and code in the *setupTables*:

```
private void setupTables( long id )
throws KExceptionClass
{

    //-------------------------------------------------------------


    browser = new facturaBrowserClass(
            configuration, log, FacturasBrowserJTable,
            facturaBrowserClass.INVOICES_BY_CLIENT, id, this );

    browser.saveSQLOperation(
            totalLabel, " sum( fac.fac_total ) ", facturaBrowserClass.CURRENCY, true );

    browser.initializeTable();

    //setup container button
    newButton.addActionListener( browser );
    deleteButton.addActionListener( browser );
    editButton.addActionListener( browser );
    sortButton.addActionListener( browser );
    filterButton.addActionListener( browser );
    printButton.addActionListener( browser );
    refreshButton.addActionListener( browser );
    saveChangesButton1.addActionListener( browser );
```

5.11.3 Provided widgets for table cells

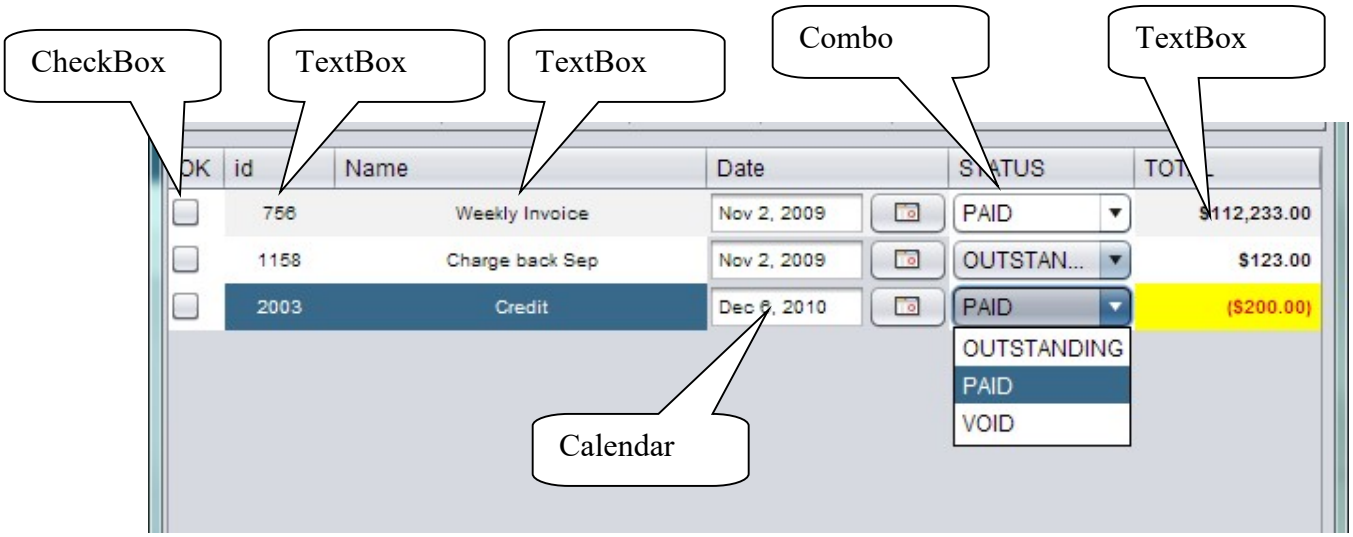
If you follow the above instructions to enable a column to be editable, you will always get a textbox for editing, but you can change that. You can define a special widget from the framework's library or make your own.

The framework provides the following widgets for cell data display:

1. CalendarCellRendererClass
2. CheckBoxCellRendererClass
3. ComboCellRendererClass
4. TextBoxCellRendererClass

.. and the following for data entry:

1. CalendarCellEditorClass
2. CheckBoxCellEditorClass
3. ComboCellEditorClass
4. TextBoxCellEditorClass



5.11.4 Using a Calendar Widget for Date Table Cells

To use a calendar, like in the above example, set the renderer for the column in the browser's main initialize method (See the example in facturaBrowserClass.java):

```
super.initializeTable();

adjustColumnWidth( "OK", 30 );
adjustColumnWidth( "id", 60 );
adjustColumnWidth( "Name", 200 );
adjustColumnWidth( "Date", 130 );
adjustColumnWidth( "STATUS", 100 );
adjustColumnWidth( "TOTAL", 100 );

adjustColumnType( "Date", DATE );

if( mode == INVOICES_BY_CLIENT ){

    //DATE
    setColumnRenderer("Date", new CalendarCellRendererClass(tableModel, log)); //<-- no n
    setColumnEditor("Date", new CalendarCellEditorClass(tableModel, log));
```

You need to make sure you add the renderer or editor after the call to super.initializeTable(). The renderers need the tableModel and log parameters of the browser, pass them on, you do not need to configure anything else.

The calendar control only recognizes dates as strings in a specific format, so make sure you code accordingly:

For oracle:

```
to_CHAR( fac.fac_date, 'yyyy-mm-dd HH24:MI:SS' )
```

For SQL Sever:

```
CONVERT( varchar, fac.fac_date, 120 )
```

.. or the equivalent for your database.

5.11.5 Table's Check Box Widget

To use a checkBox, like in the above example, set the renderer for the column in the browser's main initialize method (See the example in facturaBrowserClass.java):

```
// OK box
setColumnRenderer("OK", new CheckBoxCellRendererClass(tableModel, log) );
setColumnEditor("OK", new CheckBoxCellEditorClass(tableModel, log) );
```

You need to make sure you add the renderer or editor after the call to super.initializeTable(). The renderers need the tableModel and log parameters of the browser, pass them on, you do not need to configure anything else.

The checkbox control only understands values for the column as Strings with the words "Y" or "N", so make your SQL accordingly.

5.11.6 Table's Combo widget

To use a comboBox, like in the above example, set the renderer for the column in the browser's main initialize method (See the example in facturaBrowserClass.java):

```
// STATUS
Vector< String > options = new Vector< String >();
options.add("OUTSTANDING");
options.add("PAID");
options.add("VOID");
setColumnEditor("STATUS", new ComboCellEditorClass( options,  tableModel, log, true ) );
setColumnRenderer("STATUS", new ComboCellRendererClass( options, tableModel, log) );
```

You need to make sure you add the renderer or editor after the call to super.initializeTable(). The renderers need the tableModel and log parameters of the browser, pass them on. Additionally, you need to pass a List with all possible values and whether the field can be freely written on or whether only listed values are allowed.

If not editable, or "false", make sure you pass all possible values.

If you set a combo editor to editable, then do not set the renderer, and let the default renderer to paint the field, or you might not see anything for values not in the list.

5.11.7 Making your own or using 3rd party widgets for cell rendering in tables

If you want to make your own cell renderer or editor, you can make your own by extending the *KTableCellEditorBaseClass* or the *KTableCellRendererBaseClass*.

All you need to implement are the *getTableCellEditor/RendererComponent()*, *getCellEditorValue() and getColumnType().*

Optionally, if it applies, you might implement:

```
public int getColumnAligment() {
public void setColumnAligment(int columnAligment) {
public Font getColumnFont() {
public void setColumnFont(Font columnFont) {
```

These are called by the browser when the developer sets the corresponding column's properties.

See any of the provided widget's code for details on how to code a custom renderer.

5.11.8 Dynamic cell's rendering on data at runtime

Some times we need to customize the cells appearance at runtime, depending on the data. To color a negative value in red, for example:

Each time a cell is painted or edited, a callback is called which passes the prepared renderer or editor just before it is used for display, for the user to customize it, or replace it all together. This call back also passes the entire row's data for you to make decisions. You might also use the callback for other business logic.

To receive the callback:

1) Implement the *cellRenderingHookInterface* in your browser.

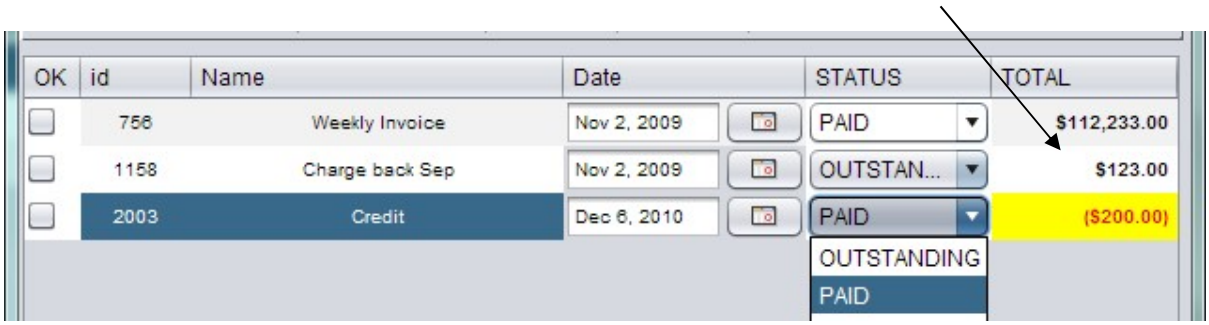
```
public class facturaBrowserClass
extends KDataBrowserBaseClass
implements cellRenderingHookInterface,  // to customize the data at runtime
KBrowserDataWriterInterface // to make it RW
{
```

2) Code the call back and set your customization, for the above example:

```
public void cellEditHook(int row, int col, boolean isSelected, Component editor, String
        // not customizing the editor
}

public void cellRenderingHook(
    int row, int col, // what cell are we executing for
    boolean isSelected, // is it currently highlited ?
    Component renderer, // here is the renderer, change it, or replaze it altogether
    String ColumnName, String value,  // data
    recordClass record, // the whole row data
    KLogClass log ) // the log used
    throws KExceptionClass
{
    boolean updateRenderer = false;

    // ------------------------------------------------------------------------
    if( ColumnName.equals( "TOTAL" ) ) {

        if( Double.parseDouble( value ) < 0 ){

            renderer.setForeground( Color.red );

            if( isSelected  )renderer.setBackground( Color.yellow );

            updateRenderer = true;
        }

    }
    // ------------------------------------------------------------------------

}
```

You are passed the coordinates of the cell being painted, whether the row is selected, the current renderer or editor ready to paint, the column's name, the cell's value, a record with the full row's data and the log for you to report any issues.

In this case, if we are painting the column named "Total", we check the value, if negative, set the text color to red.



6 Advanced Functionality

6.1 Advanced topics: Using the KTreeViewerClass

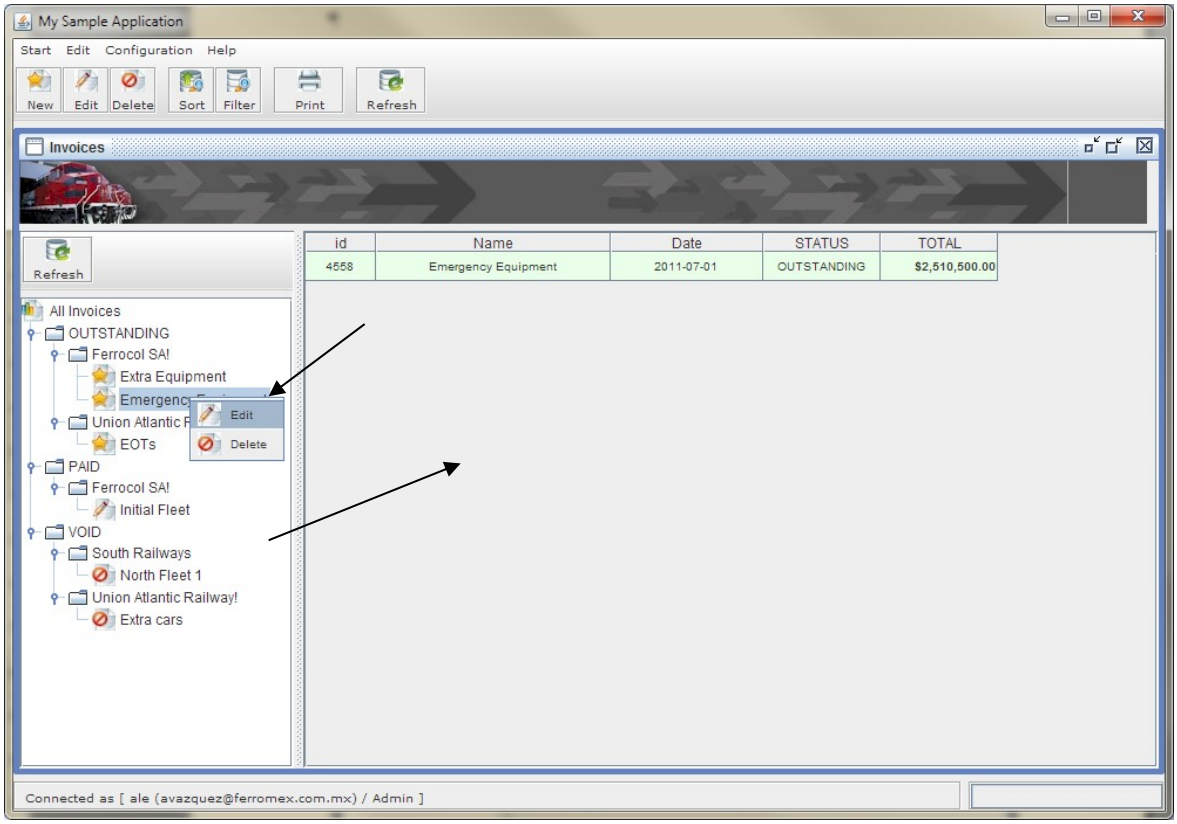
The tree viewer component allows you to build navigable tree views of data. A tree view is formed by a KTreeViewerClass object, to which you add levelClass objects for each level on the tree.

You can listen for event click events and add a standard New/edit/Menu or a customizable menu.

Additionally you can customize the icons on each leaf or node.

For this tutorial we will setup a simple tree view to display inside a frame, which will automatically filter the browser to the right, depending on the selected node in the tree.

The tree view example

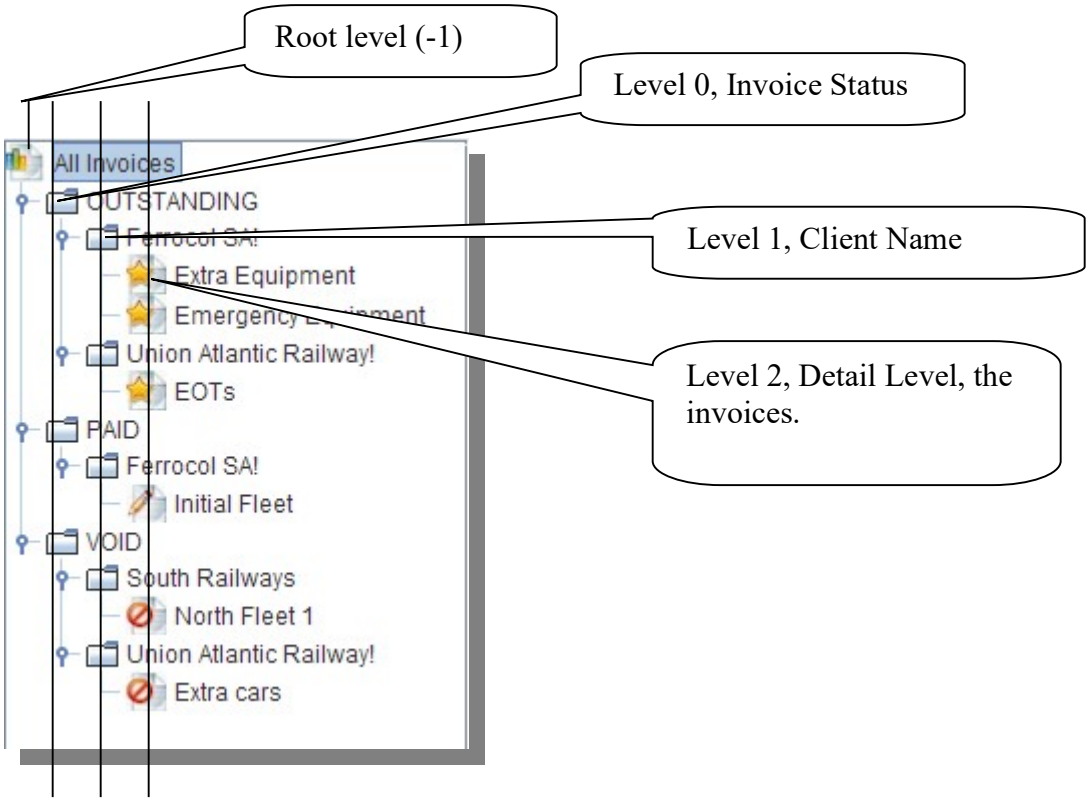


First we will create a new object for the tree object. This object will only have to fields, the JTree and the Browser, and will inherit from KTreeViewerClass.

The facturaTreeViewerClass

```
public class facturaTreeViewerClass
extends KTreeViewerClass
{

    // uses

    private JTree                       tree;
    private facturaBrowserClass         browser;

    // has - defaulted

    // -------------------------------------------------------
    /** Creates new applicationTreeViewClass */
    public facturaTreeViewerClass(
```

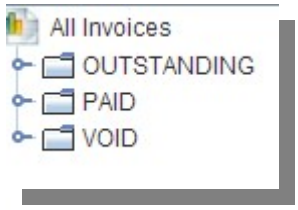
Then we need to create the levels:



Each level object requires a dbTransaction with the SQL required to get ALL the data for the level.

First, Root is provided, so we start by defining level 0, to show all invoice status:

```
{
dbTransactionClientClass factStatusLevel = new dbTransactionClientClass( configuration, log );

factStatusLevel.prepare(
     //                       V--  PK          V-- label
    " select distinct fac.FACSTATUS_ID, facstatus.FACSTATUS_STATUS " +
    " from SAMPLE_FACTURA fac" +
    " join SAMPLE_FACTURA_STATUS facstatus on fac.FACSTATUS_ID = facstatus.FACSTATUS_ID " +
    " order by facstatus.FACSTATUS_STATUS "
    );

// bind here if necessary
// factStatusLevel.bind( xx , xx )

addLevel(
    // PK name, node label, trx, load mode, icon
    "FACSTATUS_ID", "FACSTATUS_STATUS", factStatusLevel ,
    KTreeViewerClass.NODE_MODE_AUTO, //-- load all data on display
    null // new ImageIcon( systemResources.getImage( "application.jpg" ) )
    );
}
```



1. We start by creating a dbTransaction. For more information on dbTransactions, or how to access data directly from the client, see: *3.4.1 Advanced Using dbTransactions (Direct SQL) in the client.*
   In this case we make a SQLto get all invoice status as the first level. Note that we only need to prepare and bind, the framework will execute and fetch at runtime.
   The SQL needs to return 3 items: The keyfield of the object displayed in the nodes, the text to be displayed and the foreign keys to all previous levels, to tie each node of the tree. In this case, no FKs are required since it's the first level.
   Finally, also note that no specific order by is required.

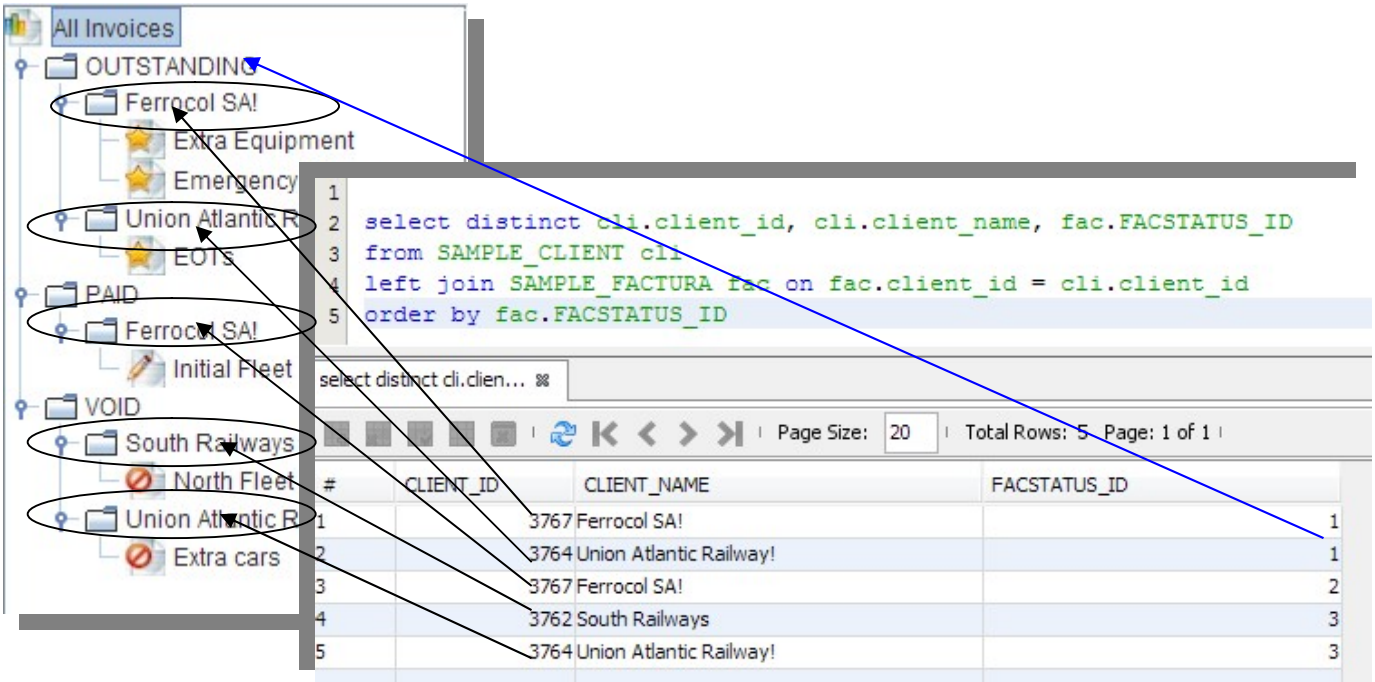
2. Secondly, we add the level to the KTree. Parameters required are the name of the key field, the name of the text field to label each node, the dbTransaction, the node mode (see blelow) and, optionally, and icon for the nodes,

That's it, now for the next level, the clients with invoices on each status:

```
{
dbTransactionClientClass cliStatusLevel = new dbTransactionClientClass( configuration, log );

cliStatusLevel.prepare(
     //                      V--  PK         V-- label    V-- FK to parent above
    " select distinct cli.client_id, cli.client_name, fac.FACSTATUS_ID " +
    " from SAMPLE_CLIENT cli " +
    " left join SAMPLE_FACTURA fac on fac.client_id = cli.client_id " +
    " order by cli.client_name "
    );

// bind here if necessary
// factStatusLevel.bind( xx , xx )

addLevel(
    // PK name, node label, trx, load mode, icon
    "CLIENT_ID", "CLIENT_NAME", cliStatusLevel ,
    KTreeViewerClass.NODE_MODE_AUTO, //-- load all data on display
    null // new ImageIcon( systemResources.getImage( "application.jpg" ) )
    );
}
```

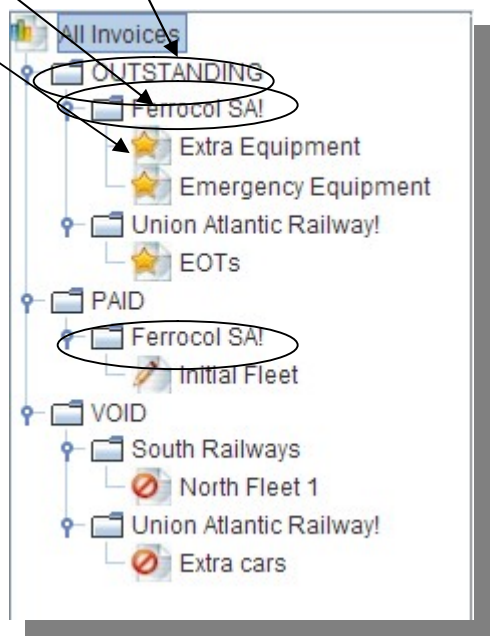
The next level is pretty much the same: we make an SQL to get the clients. Each record is a leaf node in the tree, below the status. Since we want the clients attached to each status for which they might have an invoice, we join in the invoices for each client. We need to return the PK of the client, the client name to display, and the status of the invoices, in other words the FK to use to tie each record to the correct parent node.

Again, but graphically:



Finally we add the last level: the invoice detail. Note that as you go down the tree you need to return all the FKs for ALL the previous levels, since a node can repeat for a different parent node. So, we not only need to return the invoice client id, but the corresponding status, since a client might appear with many statuses.

```
{
dbTransactionClientClass factLevel = new dbTransactionClientClass( configuration, log );

factLevel.prepare(
     //                V--  PK V-- label  V-- FK to parent1 V-- FK to parent 2
    " select distinct FAC_ID, FAC_NAME, CLIENT_ID, FACSTATUS_ID " +
    " from SAMPLE_FACTURA  " +
```



You do not declare the name of the FKs in the addLevel method, the framework will expect you to return the corresponding columns with the same name as the PK they point to in the previous levels.

6.1.1 Node Loading Modes

We will now use this last level to explain the node loading modes. As explained up to here, all data will be loaded at once and the FULL tree built, but for trees of a few thousand nodes, it can take a while to load all the data and assemble the tree. So for the last levels or the last detail level, you set the node mode for ON_DEMAND. In this mode, the KTree will not load the data for the marked level from the start, but only for the corresponding nodes as you click on them at runtime.

Note in the example project provided, in the LIMS Data Navigator, how the final analysis and sample levels are retrieved on demand to allow for fast loading.

In the current example we use this to load the invoices ON_DEMAND.

```
{
dbTransactionClientClass factLevel = new dbTransactionClientClass( configuration, log );

factLevel.prepare(
     //               V--  PK V-- label  V-- FK to parent1 V-- FK to parent 2
    " select distinct FAC_ID, FAC_NAME, CLIENT_ID, FACSTATUS_ID " +
    " from SAMPLE_FACTURA  " +

    // on demand, then ad where with FKs, they will be bound automatically ---
    " where CLIENT_ID = ? and FACSTATUS_ID = ?"  // <- ONLY REQUIRED IF ONDEMAND - binding automati
    );

// bind here if necessary
// factStatusLevel.bind( xx , xx )

addLevel(
    // PK name, node label, trx, load mode, icon
    "FAC_ID", "FAC_NAME", factLevel ,
    KTreeViewerClass.NODE_MODE_ONDEMAND, //-- DEFER LOAD UNTIL CLICK
    null // new ImageIcon( systemResources.getImage( "application.jpg" ) )
    );
}
```

2

1

For ON_DEMAND nodes you need two changes:

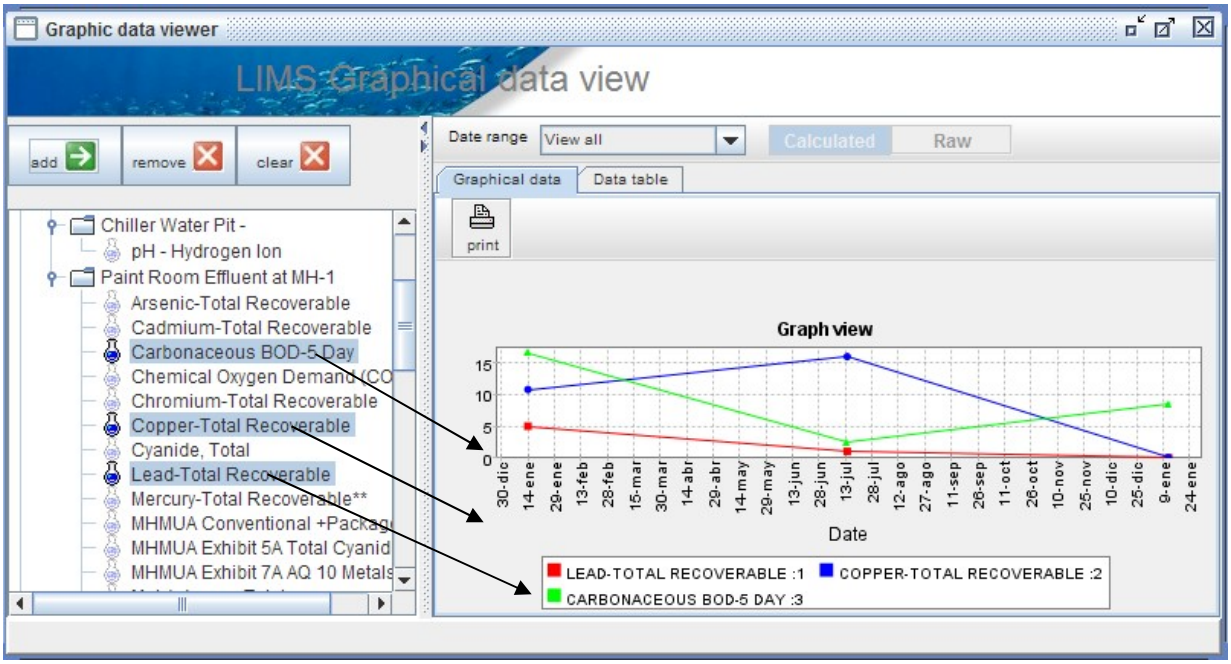
1. Declare NODE_MODE_ONDEMAND in the addLevel method and
2. add a where clause for each FK. The framework will automatically execute this SQL for each leaf and bind with the correct keys at runtime.

Finally we initialize the tree view and define right click-pop up menus for each level. Note that we use the provided KtreeNodePopupMenuClass, which provides New/Edit/Delete. If you want to use your own swing pop menu, just don't forget to send the NEW, EDIT and DELETE events to the treeView's actionPerformed function for the tree to update properly.

To see how to handle the events from the menus, see *"Using TreeView Events*", below.

```
initialize();

setLevelPopMenu( -1, null );   // root level
setLevelPopMenu( 0, new KtreeNodePopupMenuClass( true, false, false ) ); // ok new, no edit, n
setLevelPopMenu( 1, new KtreeNodePopupMenuClass( false, true, true ) ); // no new, ok edit, ok

// set select mode
treeParam.getSelectionModel().setSelectionMode( TreeSelectionModel.SINGLE_TREE_SELECTION );
```

6.1.2 Adding dynamic icons for each node.



This is achieved using a callback from the KTreeViewer that is called as it node is rendered. You are given the data, and then you can choose, for each node, what icon to display or to leave the default. For the example we want invoice nodes to change based on the invoice status:

Adding a dynamic custom node Icon

```
@Override
public Component getNodeRenderer(
        String ID, int level, String label, treeNodeClass selectedNode,
        defaultNodeRendererClass defaultNodeRenderer  ){

    if( level == 2 ){ // if we are in factura level

       if( selectedNode.parent.parent.label.equals( "OUTSTANDING" ) ){
           defaultNodeRenderer.
                   setIcon(new javax.swing.ImageIcon(getClass().getResource("/resources/new1.jpg")));
       }else
       if( selectedNode.parent.parent.label.equals( "VOID" ) ){
            defaultNodeRenderer.
                    setIcon(new javax.swing.ImageIcon(getClass().getResource("/resources/delete1.jpg")));
       }
       else
       // otherwise
       defaultNodeRenderer.setIcon(new javax.swing.ImageIcon(getClass().getResource("/resources/edit1.jpg")
    }

    return( defaultNodeRenderer );
```


Note how resources stored in the Resource Directory can be accessed

6.1.3 treeNodeClass

The important tools here are the treeNodeClass and the defaultNodeRenderer. The treeNodeClass gives you access to the current node and the whole three, you can navigate the tree using the nodes properties: Parent and Children, which returns the corresponding tree nodes which you can subsequently navigate. Note also the properties Label and RowData to access the data of the node.

6.1.4 defaultNodeRendererClass

Finally we access the current renderer, which will paint the icon. This class inherits from Swing's DefaultTreeCellRenderer, and you can replace the whole renderer with a custom one, or use the setIcon() or .addPopUpMenu() functions to customize. See the example above:

6.1.5 Using TreeView Events

Finally we might want to override the provided event handlers. Following are the functions called for event handling. If you do not use the provided PopMenus, you only can use the: mouseClickPerformed and the mouseRightClickPerformed.

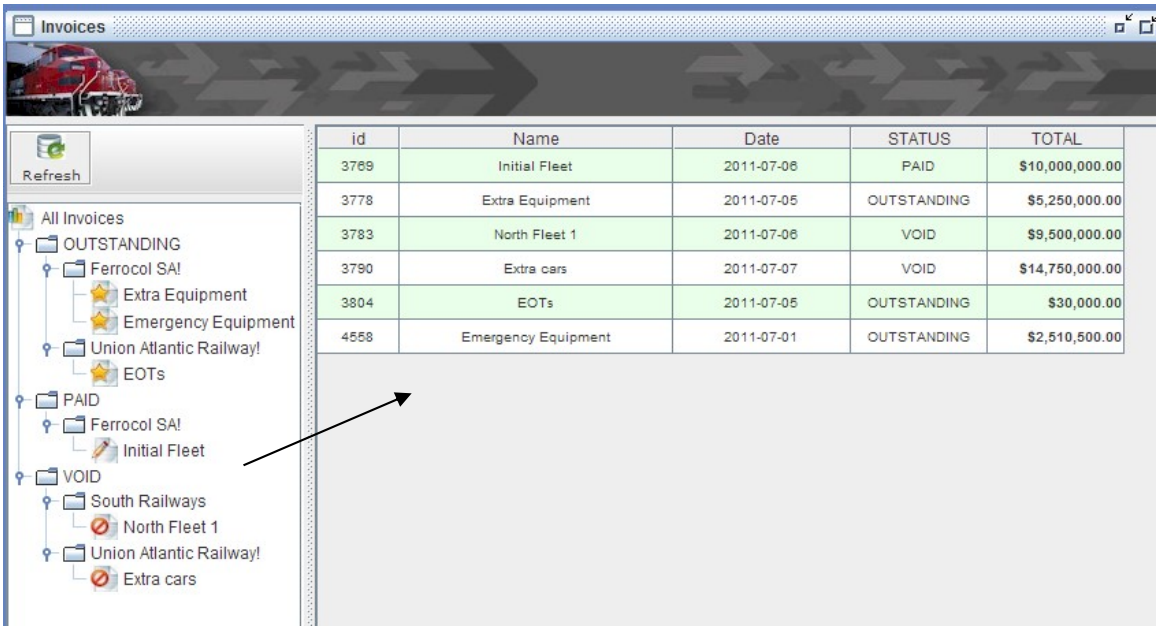
```
// for user to override
public abstract void mouseClickPerformed( String ID, int level, String label, treeNodeClass selectedNode );

public abstract void mouseRightClickPerformed( String ID, int level, String label,
        treeNodeClass selectedNode, JPopupMenu popMenu, java.awt.event.MouseEvent event );

public abstract void newButtonActionPerformed( String ID, int level, String label, treeNodeClass selectedNode );

public abstract void editButtonActionPerformed( String ID, int level, String label, treeNodeClass selectedNode );

public abstract void deleteButtonActionPerformed( String ID, int level, String label, treeNodeClass selectedNode );

// optional
public void refreshButtonActionPerformed(){
    try{
        initialize();
    } catch( Exception error ){
            // log error
            log.log( this, metaUtilsClass.getStackTrace( error ) );
            // show error message
            metaUtilsClass.showErrorMessage( component, error.toString() );
    };
}
```

These are called at runtime to notify for the corresponding event. The provided pop up menu implementation fires one function for all levels, and the level id is given. You might add your own pop up menu for each level and add a specialized event listener.

For this example we want the browser to the right to change the contents based on the selected leaf:

```
public void mouseClickPerformed( String ID, int level, String label, treeNodeClass selectedNode ){

    try{


        if( level == -1 ){ //root is selected...
            browser.clearDefaultCriteria();
            browser.refresh();
        }

        if( level == 0 ) { // status level
            browser.clearDefaultCriteria();
            browser.setDefaultCriteria( " status.facstatus_id = ? " );
            browser.bindDefaultParameter( "1", selectedNode.ID );
            browser.refresh();
        }

        if( level == 1 ) { // status level
            browser.clearDefaultCriteria();
            browser.setDefaultCriteria( " status.facstatus_id = ? and fac.client_id = ? " );
            browser.bindDefaultParameter( "2", selectedNode.parent.ID );
            browser.bindDefaultParameter( "1", selectedNode.ID );
            browser.refresh();
        }

        if( level == 2 ) { // status level
            browser.clearDefaultCriteria();
            browser.setDefaultCriteria( " status.facstatus_id = ? and fac.client_id = ? and fac_id = ?" );
            browser.bindDefaultParameter( "3", selectedNode.parent.parent.ID );
            browser.bindDefaultParameter( "2", selectedNode.parent.ID );
            browser.bindDefaultParameter( "1", selectedNode.ID );
            browser.refresh();
        }
```

Clicking on the tree updates the browser to the right

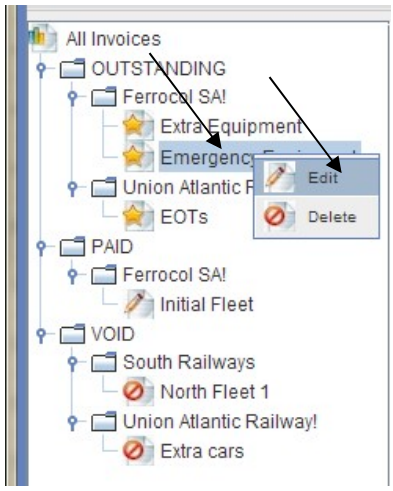


6.1.6 New, Edit and Delete from a Tree

Finally let's code the New Edit and Delete events. This is easy, because we just rely the event to the browsers, reusing code to the maximum.

Note that Edit and Delete is executed on a selected invoice, but new is only available at the parent node level. This was setted up in the constructor:

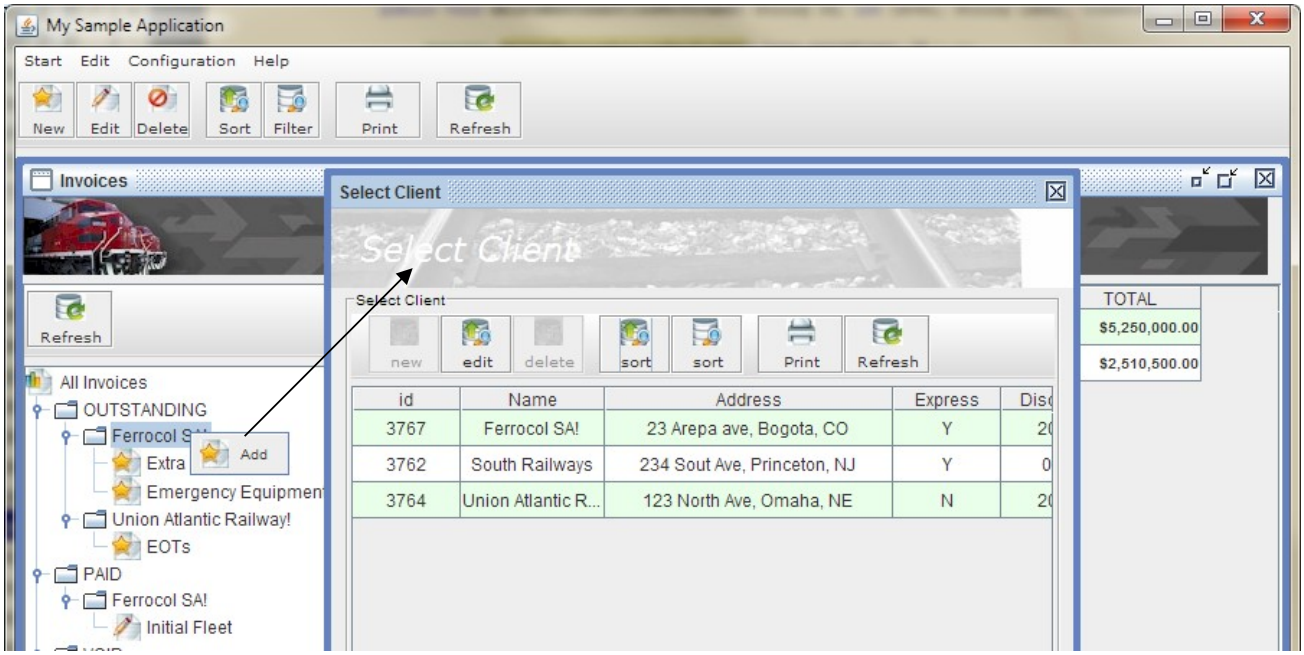
```
initialize();

setLevelPopMenu( -1, null );    // root level
setLevelPopMenu( 0, new KtreeNodePopupMenuClass( true, false, false ) ); // ok new, no edit, n
setLevelPopMenu( 1, new KtreeNodePopupMenuClass( false, true, true ) ); // no new, ok edit, ok

// set select mode
treeParam.getSelectionModel().setSelectionMode( TreeSelectionModel.SINGLE_TREE_SELECTION );
```



The code:

```
@Override
public void newButtonActionPerformed( String ID, int level, String label, treeNodeClass selectedNod

    browser.newButtonActionPerformed();
}
@Override
public void editButtonActionPerformed( String ID, int level, String label, treeNodeClass selectedNo

     browser.editButtonActionPerformed( Long.parseLong( ID ) );
}
@Override
public void deleteButtonActionPerformed( String ID, int level, String label, treeNodeClass selected

     browser.deleteButtonActionPerformed( Long.parseLong( ID ) );
}
@Override
public void mouseRightClickPerformed(String ID, int level, String label, treeNodeClass selectedNode
        JPopupMenu popMenu, MouseEvent event) { }
```

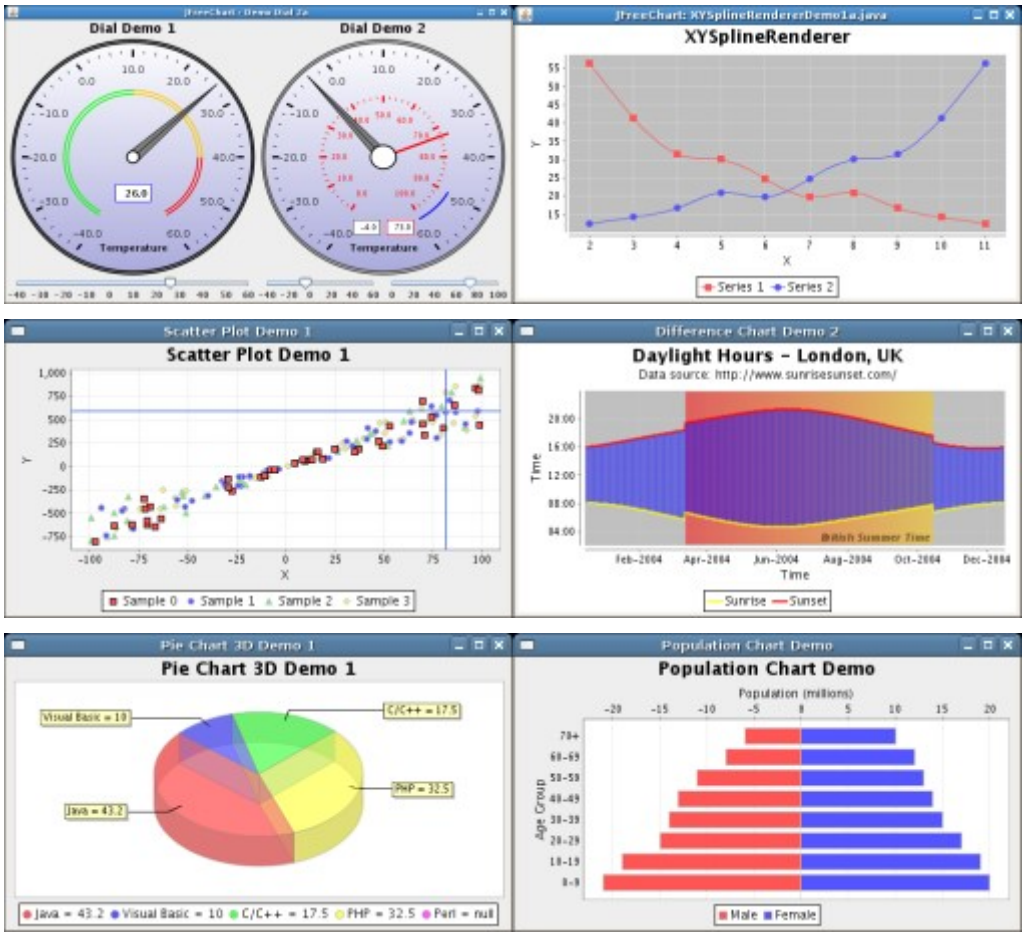
Clicking new in the node brings the select client dialog automatically, because we reuse the code written before for this event in the browser:


My Sample Application
Start Edit Configuration Help
New Edit Delete Sort Filter Print Refresh
Invoices
Refresh
All Invoices
OUTSTANDING
Ferrocol S
Extra
Emergency Equipmen
Union Atlantic Railway!
EOTs
PAID
Ferrocol SA!
Initial Fleet
Add
Select Client
Select Client
Select Client
new edit delete sort sort Print Refresh
id Name Address Express Disc
3767 Ferrocol SA! 23 Arepa ave, Bogota, CO Y 2(
3762 South Railways 234 Sout Ave, Princeton, NJ Y 0
3764 Union Atlantic R... 123 North Ave, Omaha, NE N 2(
TOTAL
$5,250,000.00
$2,510,500.00

6.2 Advanced Topics: Drawing Dynamic Graphics with the JFreeChart

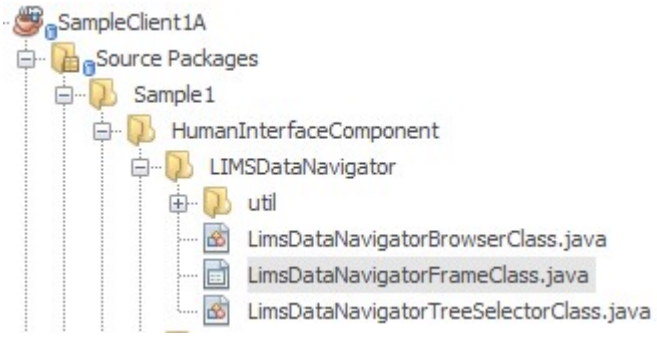
To draw graphics we use the JFreeChart component. This component is also open source and can be downloaded from here: http://www.jfree.org/.

The following example explains how to integrate it into a KFramework Application; it is not the scope to explain how the JFree works. The Jfree is founded by selling the manual, so please buy one.



The sample project includes the module LIMSDataNavigator. This module is a production module used by a LIMS (Laboratory Data Management Component) system built using the KFramework. This module implements a simple graphic using the JFreeChart.

Open the LIMSDataNavigator and open the LimsDataNavigatorFrameClass.java, locate the drawChart method for an example.



To make a graph we need data series. A data series is what makes one line or a pie, or a dial. A data series is just a list of data points. Depending on the type of chart these data points can be of one or multiple values. For a line chart we need two datapoints, X and Y. For a Pie you only need X. Additionally, if you want multiple lines, we need multiple dataseries.

More specifically, for this example, we need a line chart where one axis is a value and the other is a Date. Look into the Jfree documentation for more information on other types of datasets.

1) First we instantiate the dataset, which will contain all our series, in this case a TimeSeriesCollection. This will handle ALL our data.

```
TimeSeriesCollection mainDataset = new TimeSeriesCollection();
```

2) Now we build each data serie, in other words, each line.

```
TimeSeries dataByDateSerie = new TimeSeries( "serie name",
Day.class );
```

Note that, in this case, the time series is at the "day" granularity.

3) We fill the series with data

```
For( … ) dataByDateSerie.add( {date}, {value});
```

4) We add it to the dataset

```
mainDataset.addSeries( dataByDateSerie );
```

5) Great, let's build the chart.

```
JFreeChart chart =  ChartFactory.createTimeSeriesChart(
            "Results by Date", "Date", "Result", mainDataset,
             true, false, false);
```

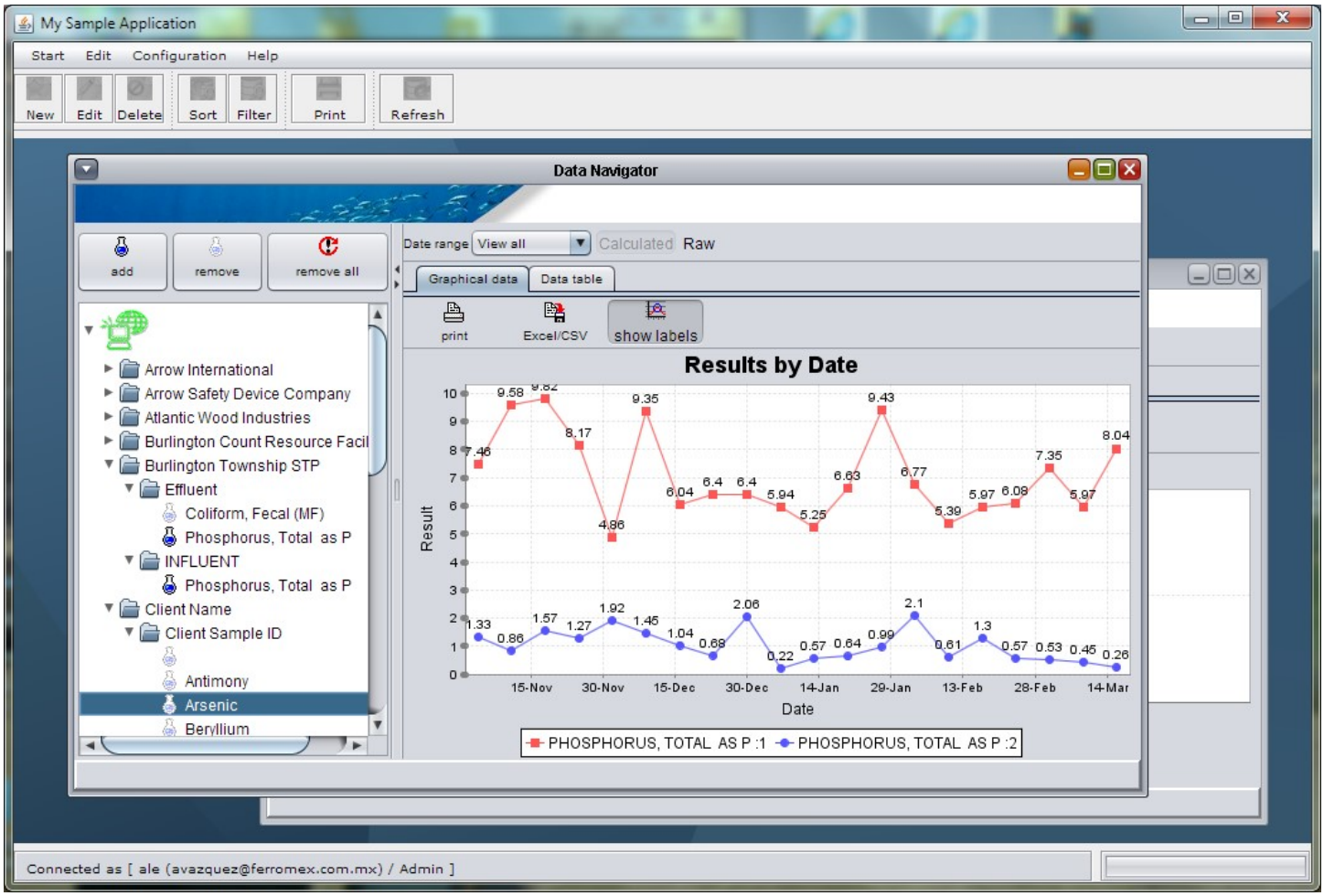
6) We customize the chart:

```
chart.setAntiAlias( true );
XYPlot plot = chart.getXYPlot();

plot.getRangeAxis().setTickMarksVisible(true);
plot.getRangeAxis().setTickMarkStroke( new BasicStroke( 5,
BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND ) );
plot.getRangeAxis().setAutoTickUnitSelection(true);
plot.getRangeAxis().setTickLabelsVisible(true);

final XYLineAndShapeRenderer renderer = new
XYLineAndShapeRenderer();

renderer.setShapesVisible( true );

if( showTickLabels ){
    renderer.setBaseItemLabelsVisible(true);
    renderer.setBaseItemLabelGenerator( new
StandardXYItemLabelGenerator() );
}


plot.setRenderer( renderer );
```

7) In the example we have a regular JPanel called "graphPanel". We get the chartPanel and just add it to the JPanel and revalidate. That's it.

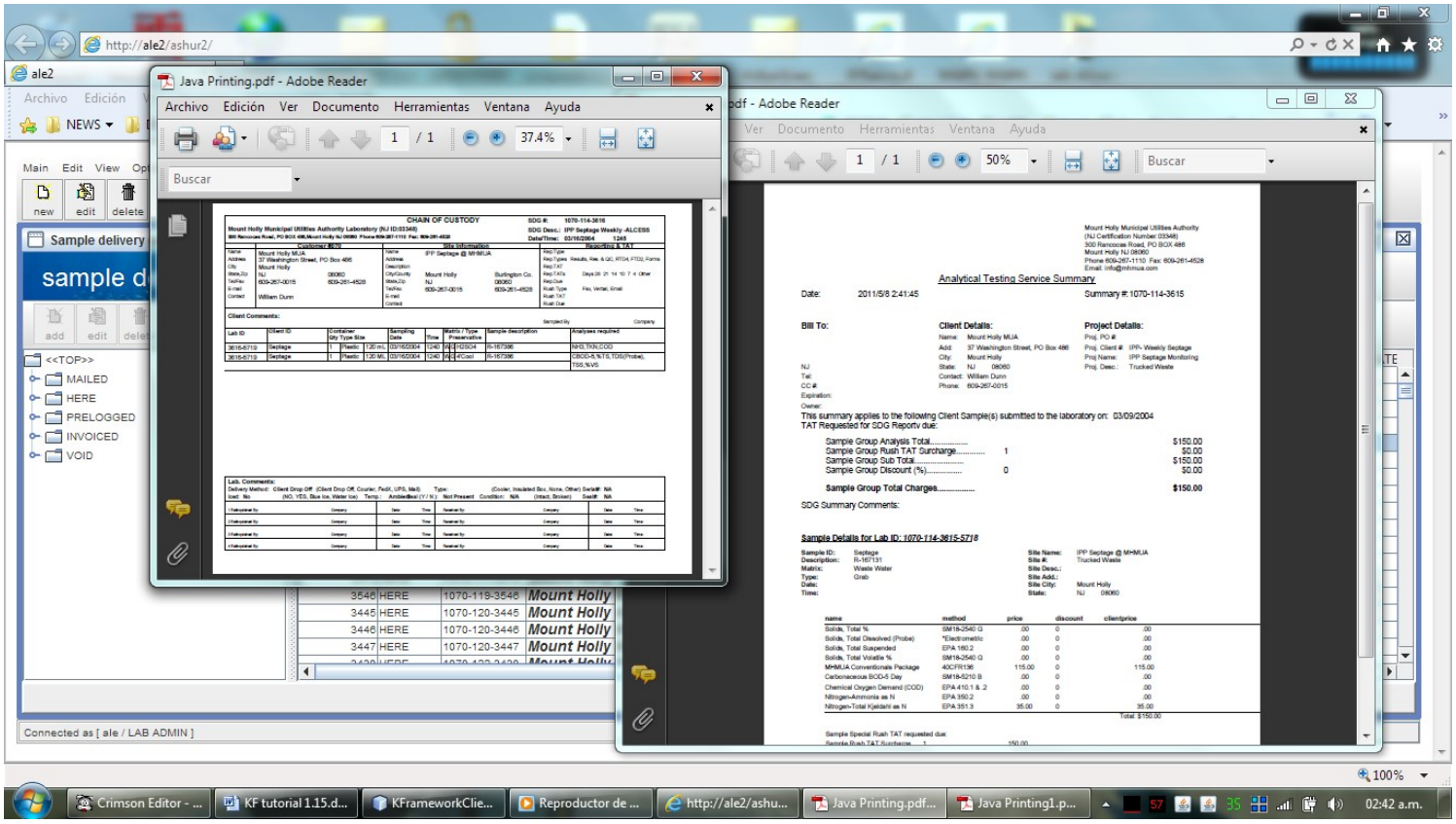
```
chartPanel = new ChartPanel(chart);
graphPanel.add( chartPanel, BorderLayout.CENTER );
graphPanel.revalidate();
```

Note that you can rebuild the chart dynamically as many times as you want, to add or remove data for example, or you can switch from line to pie at runtime with out needing to call the server.



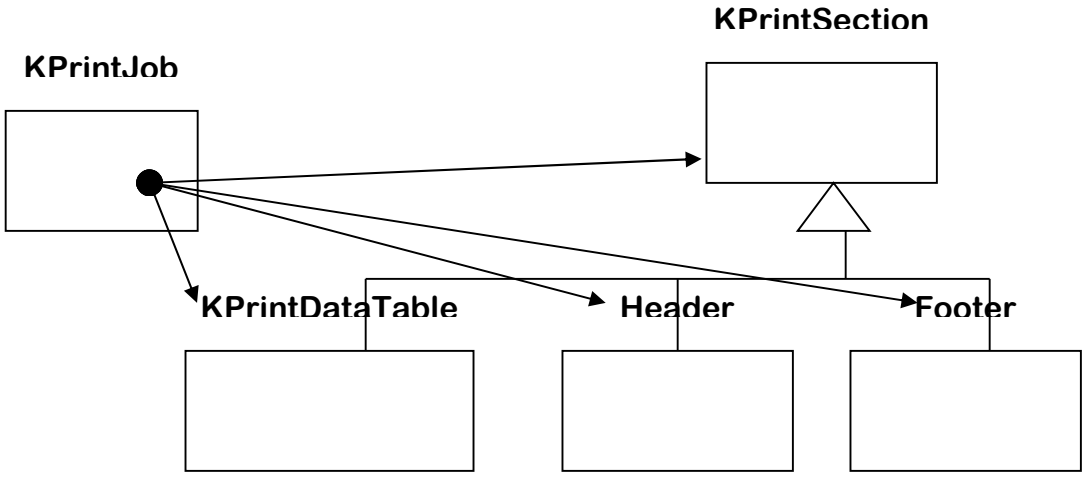
6.3 Advanced Topics: Using the reports engine

The framework provides a full reports framework. These reports are "real" reports that communicate to the printer, no web pages. They are paginated with headers, footers, group sections etc.

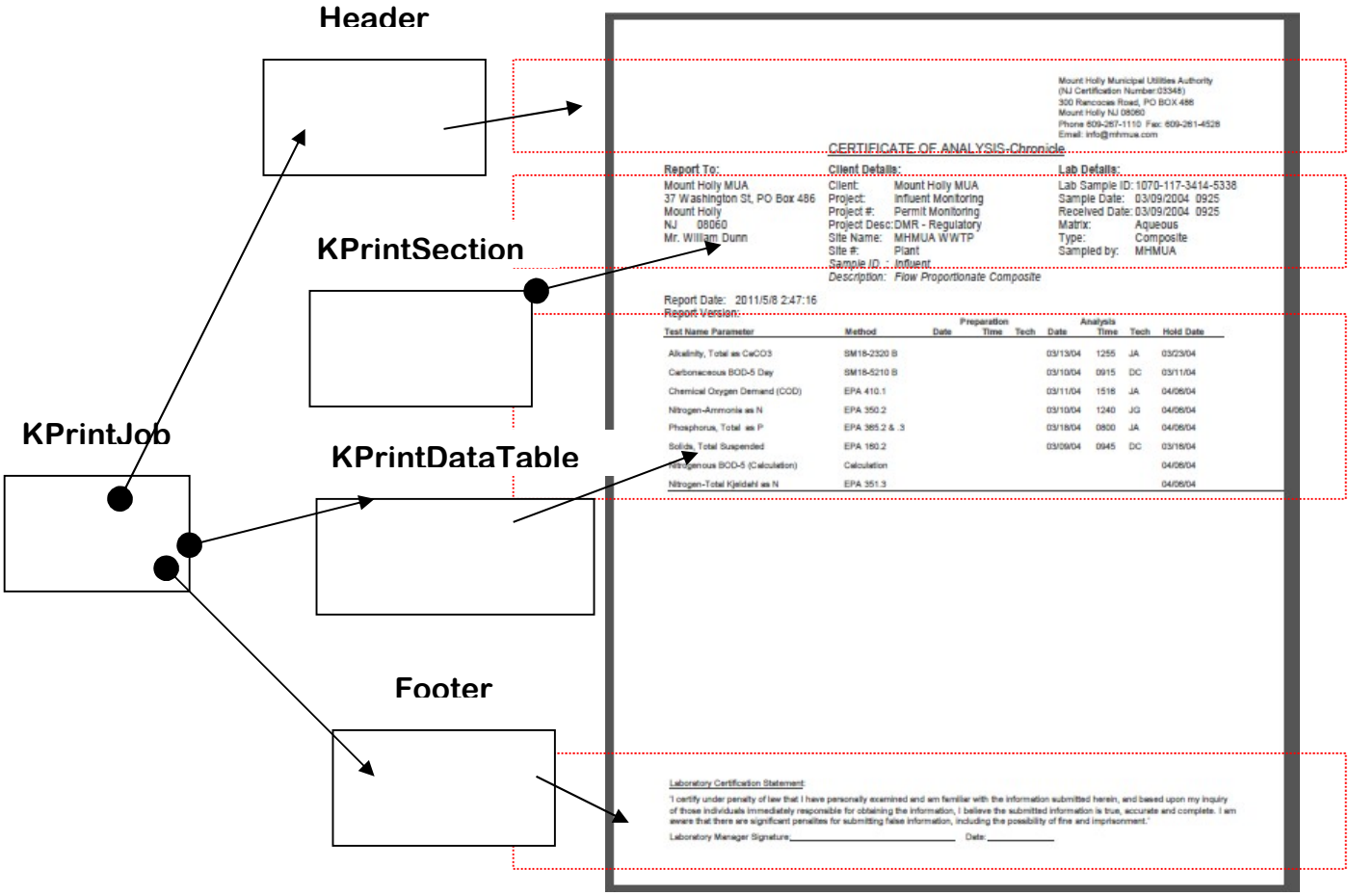


A report is built using a KPrintJobClass to which you add KPrintSectionClasses. The framework handles the paging, headers and footers. Summary Sections, headers, footers, etc are all descendants from KPrintSectionClass. You can select a printer and page setup, or let the user use the KFramework tools to configure those at runtime. As a programmer you just need to add sections and tables to the report.

The class diagram is like this



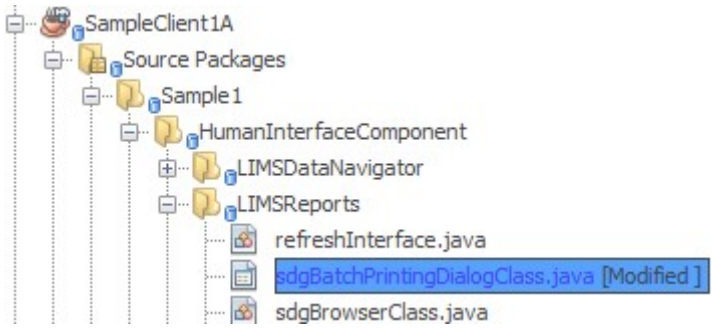
For example for a simple report:



In this case you build each section, pass SQL to the table and add to the KPrintJob. At runtime the framework will read the data and add pages as necessary, automatically setting the headers and footers until all the data is printed.

You can define anything inside a section. Graphics, Text, Lines etc, and then you add them to the KPrintJob, but you can not define where a section is printed. Sections are added one after the next and their size might change at runtime. The framework will print them one by one, adding pages as necessary.

For a full example coded see the " sdgBatchPrintingDialogClass ", in the SampleClient1A project, in method: ***printInvoice().***



That is a fairly complex example that tries to cover all aspects. Your reports might be much simpler than this, like in the following example.

6.3.1 Building a report

1) First let's start our print job.

```
KPrintJobClass printJob =
    new KPrintJobClass( configuration, log,
        KMetaUtilsClass.getParentFrame( caller ) );
```

The sample framework provides page and printer settings dialogs for the user to configure. A printJob declared like this will take these defaults, but you can force some defaults, say to use a specific printer or page orientation.

Current defaults are always available in :

```
KPrintJobClass.defaultPageFormat
KPrintJobClass.defaultPrinterJob
```

To change them, just reassign them. If you want some settings only for the current print job, say to force portrait:

```
PageFormat fixedPageFormat = ( PageFormat ) KPrintJobClass.defaultPageFormat.clone();
fixedPageFormat.setOrientation( PageFormat.PORTRAIT );

printJob.useSpecificPrintingDefaults(
    KPrintJobClass.defaultPrinterJob, fixedPageFormat );
```

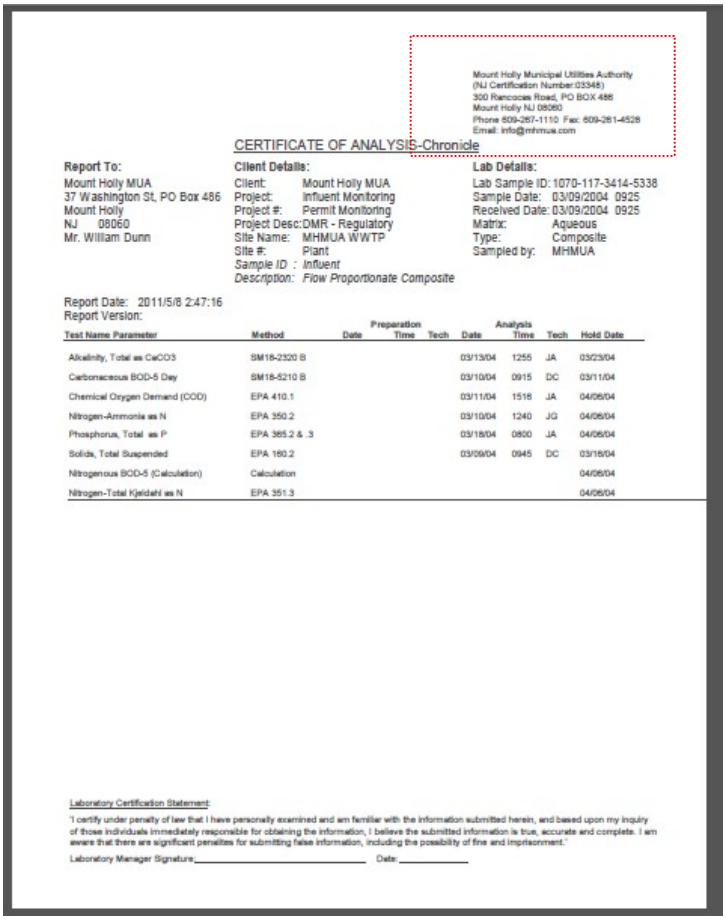
2) Now lets set some printJob defaults:

```
// start job
printJob.setDefaultFont( new Font("Arial", Font.PLAIN, 10) );
printJob.setLeftMargin( 50 );
printJob.setBottomMargin( 40 );
```

3) Now we materialize all business objects required for the report:

```
persistentObjectManagerClass persistentObjectManager =
        new persistentObjectManagerClass( configuration, log );

//materialize the sdg
ASHURSampleDeliveryGroupClass sdg = new ASHURSampleDeliveryGroupClass();
persistentObjectManager.copy( sdg_id, sdg );

//materialize the project
ASHURProjectClass project = new ASHURProjectClass();
persistentObjectManager.copy( sdg.getProjectId(), project );

//materialize the client
ASHURClientClass client = new ASHURClientClass();
persistentObjectManager.copy( project.getClientId(), client );
```

4) Now we make the header that will repeat in all pages




```
// ------------------------------------------------------------------------------
// ------------------------------------------------------------------------------
//Header section
KPrintSectionClass headerSection = new KPrintSectionClass(configuration, log, 520, 140);

// Header: Address "Mount Holly"
headerSection.setFont( new Font( "arial", Font.PLAIN,                    8) );
headerSection.printText("Kframework Municipal Utilities Authority ",   350, 40 );
headerSection.printText("(Certification Number:1234)",                350, 50 );
headerSection.printText("300 Saturn Road, PO BOX 486",                  350, 60 );
headerSection.printText("Mount Everest KA 08060",                        350, 70 );
headerSection.printText("Phone 609-123-1223  Fax: 609-123-1234",        350, 80 );
headerSection.printText("Email: avazqueznj@users.sourceforge.net",
```

A section is made just by declaring its size. It's like a rectangle added to the page that pushes all subsequent sections down. Headers and footers are special cases, because they are printed in a fixed position and repeated on every page.

The section class presents several tools to add text, lines and other constructs to the section. You can only print in a section; you can not put anything in an arbitrary location

in the page, all coordinates are relative to the current section. At runtime they will be printed and the location in the page decided.

`public void setFont( Font fontParam )` Sets the font, covers from the next printObjects until its reset again.
`public void setColor( Color colorParam )` Sets the current fore color until next call
`public void printText( String text, int x, int y )` Prints a free form text. Up to you to make sure it does not overlap with something else.
`public void printGraphicJFreeChart( JFreeChart chart, int width, int height, int x, int y )` Prints a JFreeChart, given the size.

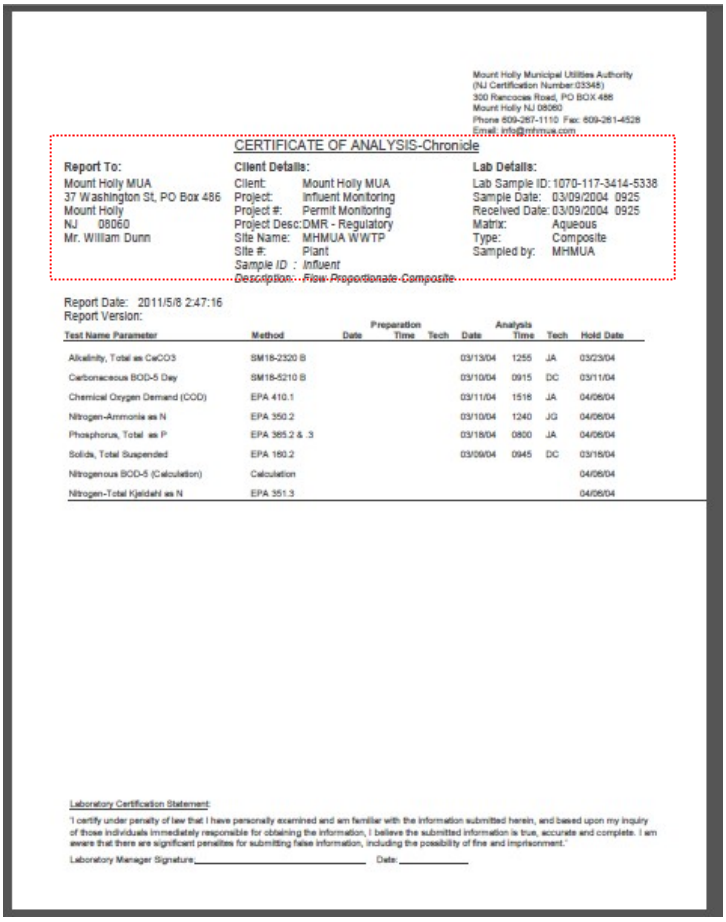
`public void printField( String text, int x, int y, int fieldWidth, int alignment )`
 	
`public void printField( String text, int x, int y, int fieldWidth )`
 	
`public void printField( long numberString, int x, int y, int fieldWidth )`
Prints text given a clip area and aligment, in some cases. Usefull for number columns.
Use for align:
`public static final int LEFT = 0;`
`public static final int RIGHT = 1;`
`public static final int CENTER = 2;`

`public void printLine( int x1, int y1, int x2, int y2 )` Draws a line
`public void printImage( Image image, int x, int y )` Print an image, useful for logos etc.

5) We add the header to the printJob

```
printJob.SetHeader( headerSection, KPrintJobClass.CENTER );
```

6) Now we add the page address section. This section is added as the first thing in the report so it's printed first, just below the first header, but not repeated on every subsequent page.



Mount Holly Municipal Utilities Authority
(NJ Certification Number:03348)
300 Rancocas Road, PO BOX 488
Mount Holly NJ 08060
Phone 609-267-1110 Fax: 609-261-4528
Email: info@mhmua.com

CERTIFICATE OF ANALYSIS-Chronicle

Report To:
Mount Holly MUA
37 Washington St, PO Box 486
Mount Holly
NJ 08060
Mr. William Dunn

Client Details:
Client: Mount Holly MUA
Project: Influent Monitoring
Project #: Permit Monitoring
Project Desc: DMR - Regulatory
Site Name: MHMUA WWTP
Site #: Plant
Sample ID : Influent
Description: Flow Proportionate Composite

Lab Details:
Lab Sample ID: 1070-117-3414-5338
Sample Date: 03/09/2004 0925
Received Date: 03/09/2004 0925
Matrix: Aqueous
Type: Composite
Sampled by: MHMUA

Report Date: 2011/5/8 2:47:16
Report Version:

Test Name Parameter	Method	Preparation Date	Preparation Time	Preparation Tech	Analysis Date	Analysis Time	Analysis Tech	Hold Date
Alkalinity, Total as CaCO3	SM18-2320 B				03/13/04	1255	JA	03/23/04
Carbonaceous BOD-5 Day	SM18-5210 B				03/10/04	0915	DC	03/11/04
Chemical Oxygen Demand (COD)	EPA 410.1				03/11/04	1516	JA	04/06/04
Nitrogen-Ammonia as N	EPA 350.2				03/10/04	1240	JG	04/06/04
Phosphorus, Total as P	EPA 365.2 & .3				03/18/04	0800	JA	04/06/04
Solids, Total Suspended	EPA 160.2				03/09/04	0945	DC	03/16/04
Nitrogenous BOD-5 (Calculation)	Calculation							04/06/04
Nitrogen-Total Kjeldahl as N	EPA 351.3							04/06/04

Laboratory Certification Statement:
'I certify under penalty of law that I have personally examined and am familiar with the information submitted herein, and based upon my inquiry of those individuals immediately responsible for obtaining the information, I believe the submitted information is true, accurate and complete. I am aware that there are significant penalties for submitting false information, including the possibility of fine and imprisonment.'
Laboratory Manager Signature:_____ Date:_____

```
// Address section
KPrintSectionClass addressSection = new KPrintSectionClass(configuration, log, 520, 130 );
addressSection.setFont( new Font( "arial", Font.BOLD, 10) );
addressSection.printText("Bill To:",                                    0, 20 );
addressSection.printText("Client Details:",                           170, 20 );
addressSection.printText("Project Details:",                          350, 20 );

// Address "Report-To"
addressSection.setFont( new Font( "arial", Font.PLAIN, 8) );
addressSection.printField(billTo.getBilltoName(),                      0, 35, 150 );
addressSection.printField(billTo.getBilltoAddress1(),                  0, 47, 150 );
addressSection.printField(billTo.getBilltoCity(),                      0, 59, 150 );
addressSection.printText(billTo.getBilltoState(),                      0, 71 );
addressSection.printText(billTo.getBilltoZip(),                       30, 71 );
addressSection.printText("Tel:",                                    0, 83 );
addressSection.printField(billTo.getBilltoPhone(),                    50, 83, 150 );
addressSection.printText("CC #:",                                   0, 95 );
addressSection.printField(billTo.getBilltoCardNumber(),               50, 95, 150 );
```

Note how the Business Object fields are printed in the section.

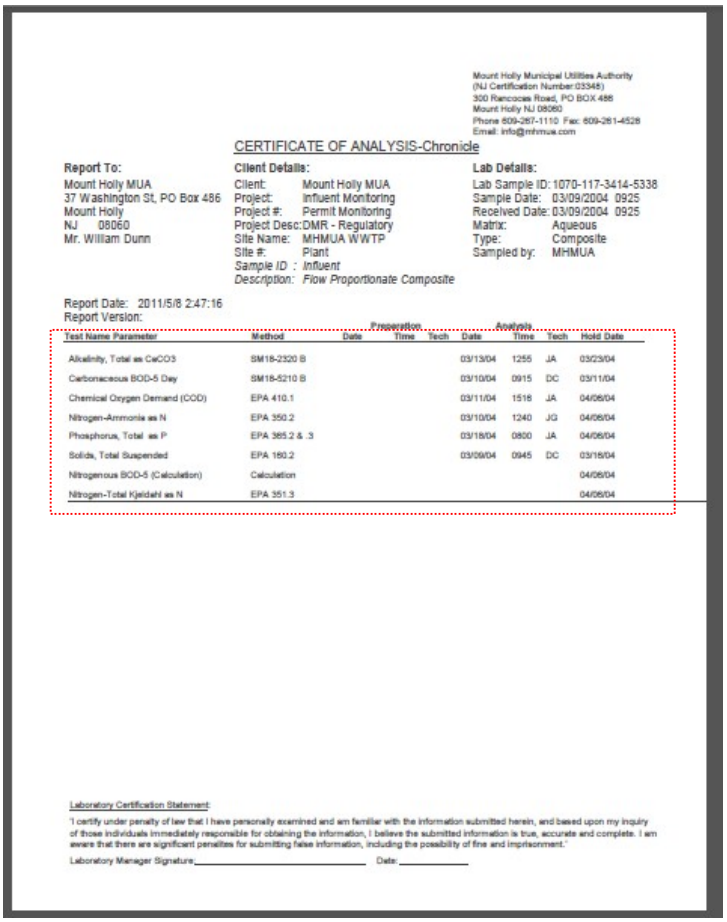
```
printJob.printSection( addressSection, KPrintJobClass.CENTER );
```

7) Now we add a data table, first we need to get the data from somewhere. You have two options:

a) Make a regular SQL dbTransaction, see the corresponding section "Using dbTransactions", or

b) taking the current SQL of a displaying browser at runtime, including the current sort and filters the user might be using.

In this case we make a dbTransaction and pass SQL.



Mount Holly Municipal Utilities Authority
(NJ Certification Number 03348)
300 Rancocas Road, PO BOX 486
Mount Holly NJ 08060
Phone 609-267-1110 Fax: 609-261-4528
Email: info@mhmua.com

CERTIFICATE OF ANALYSIS-Chronicle

Report To:
Mount Holly MUA
37 Washington St, PO Box 486
Mount Holly
NJ 08060
Mr. William Dunn

Client Details:
Client: Mount Holly MUA
Project: Influent Monitoring
Project #: Permit Monitoring
Project Desc: DMR - Regulatory
Site Name: MHMUA WWTP
Site #: Plant
Sample ID : Influent
Description: Flow Proportionate Composite

Lab Details:
Lab Sample ID: 1070-117-3414-5338
Sample Date: 03/09/2004 0925
Received Date: 03/09/2004 0925
Matrix: Aqueous
Type: Composite
Sampled by: MHMUA

Report Date: 2011/5/8 2:47:16
Report Version:

Test Name Parameter	Method	Preparation Date	Preparation Time	Preparation Tech	Analysis Date	Analysis Time	Analysis Tech	Hold Date
Alkalinity, Total as CaCO3	SM18-2320 B				03/13/04	1255	JA	03/23/04
Carbonaceous BOD-5 Day	SM18-5210 B				03/10/04	0915	DC	03/11/04
Chemical Oxygen Demand (COD)	EPA 410.1				03/11/04	1516	JA	04/06/04
Nitrogen-Ammonia as N	EPA 350.2				03/10/04	1240	JG	04/06/04
Phosphorus, Total as P	EPA 365.2 & .3				03/18/04	0800	JA	04/06/04
Solids, Total Suspended	EPA 160.2				03/09/04	0945	DC	03/16/04
Nitrogenous BOD-5 (Calculation)	Calculation							04/06/04
Nitrogen-Total Kjeldahl as N	EPA 351.3							04/06/04

Laboratory Certification Statement:
'I certify under penalty of law that I have personally examined and am familiar with the information submitted herein, and based upon my inquiry of those individuals immediately responsible for obtaining the information, I believe the submitted information is true, accurate and complete. I am aware that there are significant penalties for submitting false information, including the possibility of fine and imprisonment.'
Laboratory Manager Signature:_____ Date:_____

```
//to get analysis info
 dbTransactionClientClass AnalysisdbTransaction =
     new dbTransactionClientClass(
         configuration, log,
         configuration.getField( "server_address" ),
         configuration.getField( "SESSION" ),

AnalysisdbTransaction.prepare(

    " select " +
         " analysis_link.ANALYSIS_NAME , " +
         " analysis_link.ANALYSIS_METHOD,"+
         " TO_CHAR( analysis_link.ANALYSIS_PRICE, '999999.99') AS ANALYSIS
         " analysis_link.ANALYSIS_DISCOUNT," +
         " TO_CHAR( analysis_link.ANALYSIS_CLIENT_PRICE, '999999.99') AS A
         " analysis_link.SAMPLE_ID  " +
         //" ROUND( sample.SAMPLE_TOTAL, 2 ) AS SAMPLE_TOTAL, "+
         //" TO_CHAR(sample.SAMPLE_TAT,'99999.99') AS SAMPLE_TAT,"+

     // End Fields used in the receipt

 AnalysisdbTransaction.bind( ":v1", sampleID );

 log.log( caller, "Loading ashur analysis data" );
 AnalysisdbTransaction.executeQuery( 0, 655356 );
 log.log( caller, "Loading complete." );
```

8) We have data. Now we can loop the data and make very fancy sections as records, or get a KTable and let the framework do the work, lets use a KTable:

9) First let's make the KTable and pass the dbTransaction we just made as an argument.

```
// setup the DB printer
KPrintDataTableClass dbTable = new KPrintDataTableClass(
            configuration, log,
            AnalysisdbTransaction, printJob, 0, 655356 );
```

… you can set from what row to what row to print.

10) The SQL might have been copied from somewhere else and not all fields might be printed and not in the SQL order, so we set the fields, the display names, column sizes and, optionally, the column alignment:

```
//print anasys info
dbTable.addField( "ANALYSIS_NAME",         "NAME",         125 );
dbTable.addField( "ANALYSIS_METHOD",       "METHOD",        60 );
dbTable.addField( "ANALYSIS_PRICE",        "PRICE",         40 );
dbTable.addField( "ANALYSIS_DISCOUNT",     "DISCOUNT",      40 );
dbTable.addField( "ANALYSIS_CLIENT_PRICE","CLIENT PRICE", 100, KPrintJobClass.RIGHT );
//DBPrinter.addField( "SAMPLE_ID",             "SAMPLE ID",    30 );
```

11) We might reuse this table and switch fields on and off, lets turn all on:

```
dbTable.setPrintingField( "NAME" );
dbTable.setPrintingField( "METHOD" );
dbTable.setPrintingField( "PRICE" );
dbTable.setPrintingField( "DISCOUNT" );
dbTable.setPrintingField( "CLIENT PRICE" );
//DBPrinter.setPrintingField( "SAMPLE ID" );
```

12) Lets add a summary total

```
dbTable.addSummary( "CLIENT PRICE", KPrintDataTableClass.SUM, "Total: $", null, 2 );
```

We set the column to use, the summary required, the prefix if any, suffix and precision.

13) Let's also add a column header row. A table might not have a header row, in case you want to put it yourself, or it might be a page header, or it might be on the top of the table, and repeated on every page. Lets have a header inline with the table:

```
dbTable.setHeadersMode( dbTable.HEADER_TYPE_INLINEHEADER );
```

14) Done, lets add the table to the printJob:

```
dbTable.print();
```

15) The last section of our report is the footer. Headers and Footers can be added at any time, before the printJob is submitted. You don't need to declare the footer at the end like here. Footers are added just like headers, but with the addFooter method, so we will skip that here.

16) The final step is to begin printing. At that point the KFramework will start building each page, adding sections and adding subsequent pages as required by the data and page design.

```
printJob.submitPrintJob();
```

6.4 Advanced Topics: Using the Audit Trial

The system can log ALL actions to an audit trail. Every update, delete or insert will trigger the function. The before and after are compared and a log entry saved.

To enable the audit trial, make sure you have the table system_log in your database. To enable the audit trial:

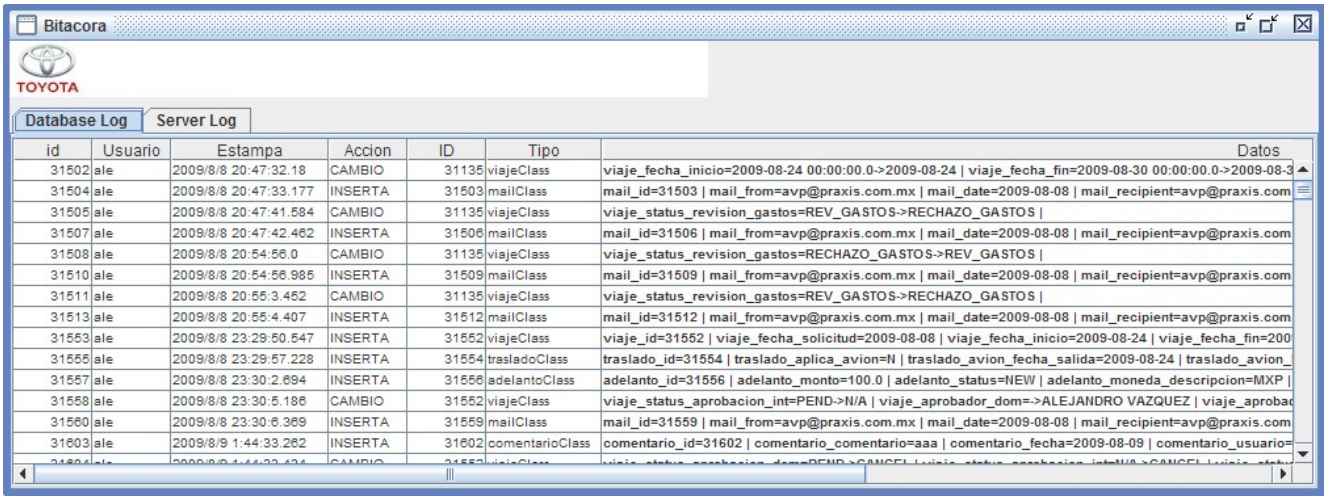
1) Enable the audit trail in the server's magnus.conf, don't confuse with the normal log.

```
auditrail=yes
```

You might notice that the framework provides a systemLogClass to read and write audit trial entries, like any other business object.

Entries are stored in database table "SYSTEMLOGCLASS"

Check the sample client for the provided audit trial browser and editor.



6.5 Advanced Topics: Using the Mail Engine

Mails are coded at the client, but always sent by the server. So you might have a client running through the internet in china, he can still send mails. You need to insert mailClass objects into the database, like using any other regular object.

The server implements standard SMTP and it can be authenticated. This will run from windows or UNIX.

For example, in the client do:

```
        persistentObjectManagerClass persistentObjectManager
                = new persistentObjectManagerClass(
configuration, log );

        mailClass mail = new mailClass();
        persistentObjectManager.createNew( mail );

        mail.mail_from = from;
        mail.mail_recipient = to;
        mail.mail_subject = subject;
        mail.mail_data = message;
        mail.mail_data_type = message_type;  // text_plain
        mail.mail_data += comments;

        persistentObjectManager.push_back( mail );
```

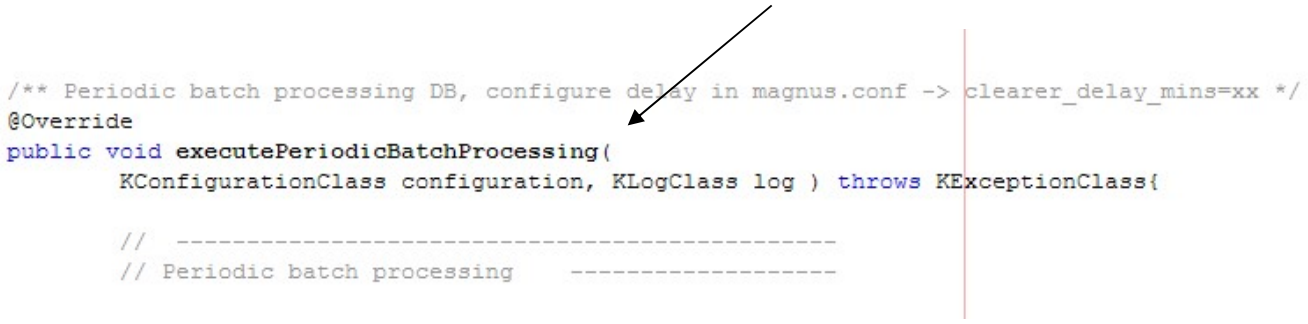
You need to configure the server, in the main configuration fix:

```
// mailer
mail_enabled=true
mail_server_address=smtp.xxxxx.com
mail_server_user=xxxx
mail_server_password=xxxxx
mail.smtp.auth=false
```

6.6 Advanced Topics: Using the Batch Process Engine

The server fires a function periodically, which you can use to fire tasks on a schedule.

The method is in the persistentObjectManager.



```
/** Periodic batch processing DB, configure delay in magnus.conf -> clearer_delay_mins=xx */
@Override
public void executePeriodicBatchProcessing(
        KConfigurationClass configuration, KLogClass log ) throws KExceptionClass{

        //  ------------------------------------------------
        // Periodic batch processing    --------------------
```

To configure how often the service runs, you need to code to check the time. The services will fire several times per hour, depending on the configuration file. The server also schedules internal tasks, so don't change the frequency time. Again, your code needs to check when it needs to run, every time the function is fired.

Configuration:

```
clearer_delay_mins=1
```

By default the functions is called every minute. This is also used to check for idle sessions to expire and send mails.

6.7 Advanced Topics: Doing Authentication and Authorization

Authentication is customizable to what ever you require. The framework provides a simple schema to authenticate using passwords and users in a table in the database, but you can change this to fit your needs.

The POM executes the authentication in function ***authenticateUser***.

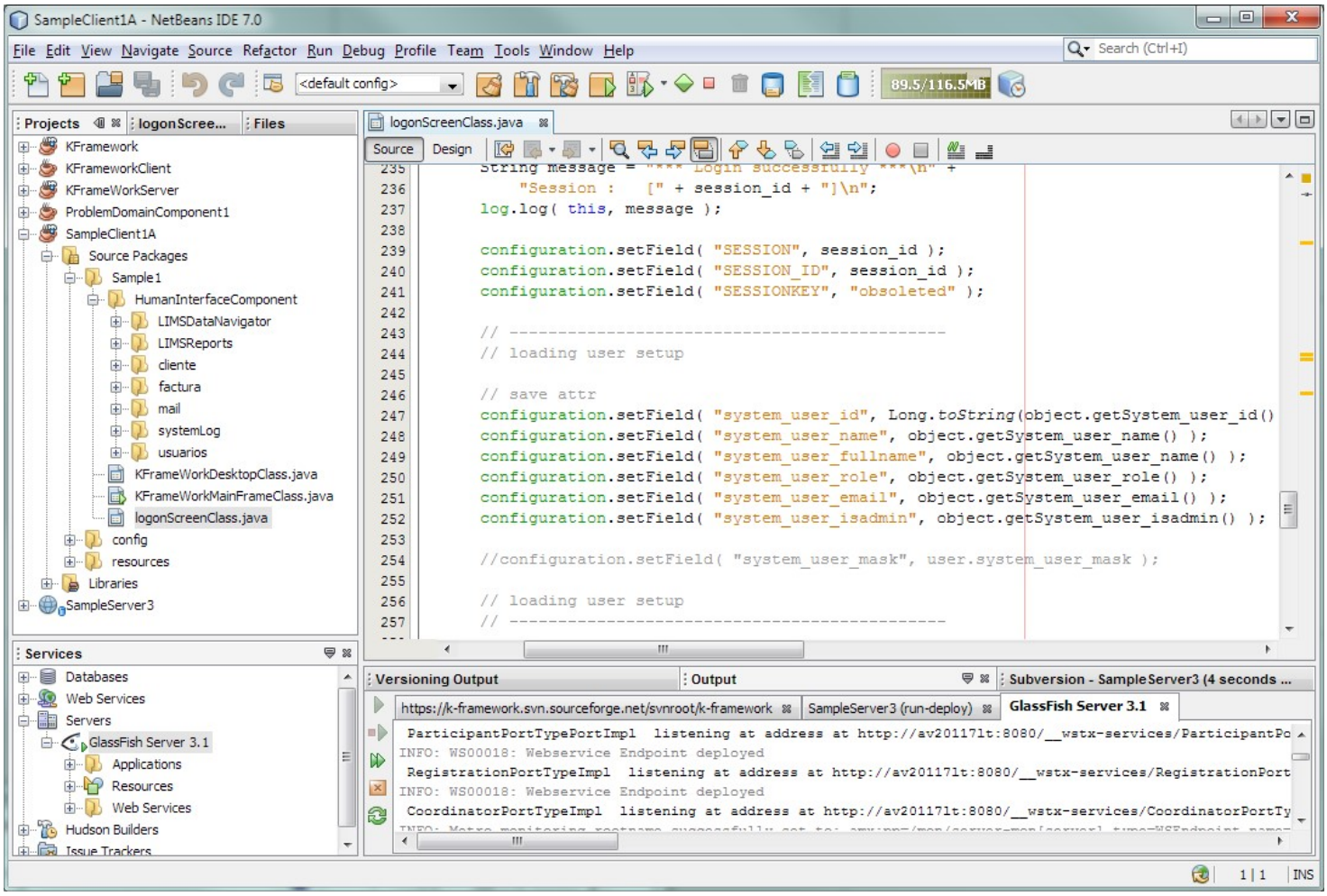
All you are required to do is to return any user object; the framework does not care how you get the object.

For authorization, just add any properties to the configuration object, that way those rights are available every where in the server. See the example code ->

```
public class accessToApplicationDeniedExceptionClass
extends KExceptionClass{
    public accessToApplicationDeniedExceptionClass(){
        super( "Invalid credentials, Access denied.", null );
    }
}

@Override
public  KBusinessObjectClass authenticateUser( systemUserClass userTryingToLogin, KSessionClass session, KLogClass log
throws KExceptionClass
{

    try{
            // in this sample one transaction to one db, you might have more than one to many dbs...
            KTransactionClass transaction = new KTransactionClass(
                    configuration, log, "SERVER_LOGIN", configuration.getField( "db_pu" ) );


        try{
                String user_name = userTryingToLogin.getFieldasString("system_user_name");
                log.log( this, " User [" + user_name + "] is logging in..." );

                // ------------------------------------------------------------
                //Try to find the user in DB ... you can do it any other way ...
                EntityManager entityTransaction = transaction.getEntityManager();
                String SQL =
                        " SELECT x FROM systemUserClass x WHERE " +
```

For the client, note that the user object is returned to the client. Likewise in the server, you can add any user rights to the client's configuration object for them to be available everywhere.

See the corresponding code in the login Dialog code:



The systemUserClass is a regular PDC object included in the Problem Domain Component that you can customize.

7 KFramework Advanced Server Cluster

For high availability and load balancing the new KFramework 2 is now cluster enabled. With his functionality a single application can be hosted by 2 or more servers.

The KFramework Cluster has many advantages over traditional HTML based applications, greatly reducing costs and increasing availability:

*** No need of synchronization of servers via UDP or any other method**

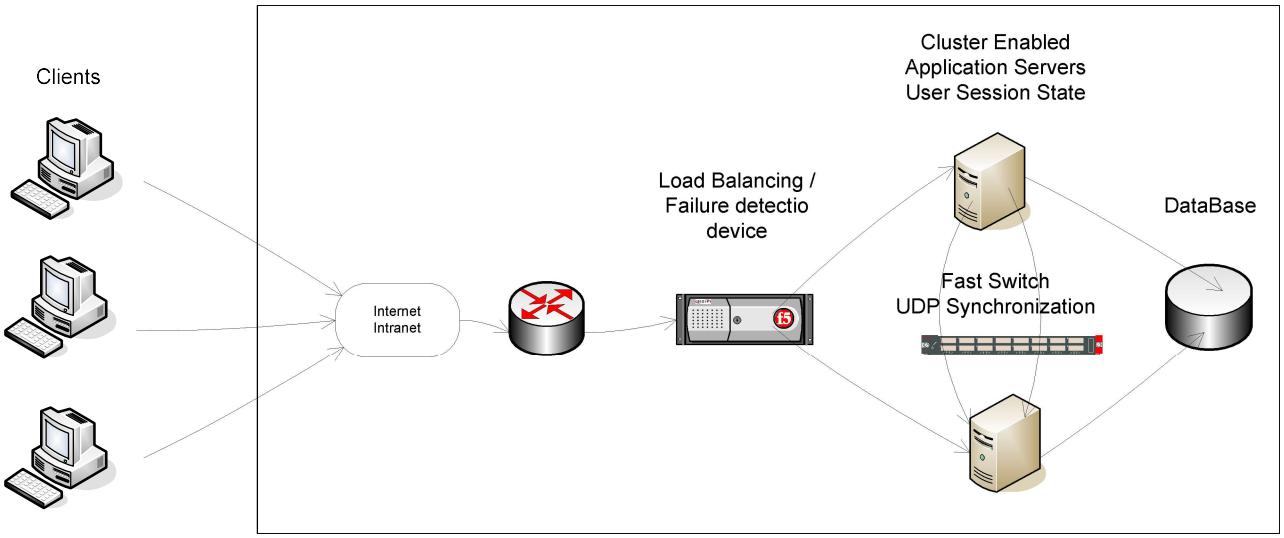
*** Servers can be on different locales as long as the can "see" the same database**

*** Application functionality is not lost even if ALL servers are shutdown and reset**

*** You can have a cluster of servers with no need of server's cluster support; you can even have nodes of different server vendors**

*** Balancing and failure detection is handled by the clients; there is no need for F5 or other load balancing / failure detection devices.**

Under a traditional cluster setup:

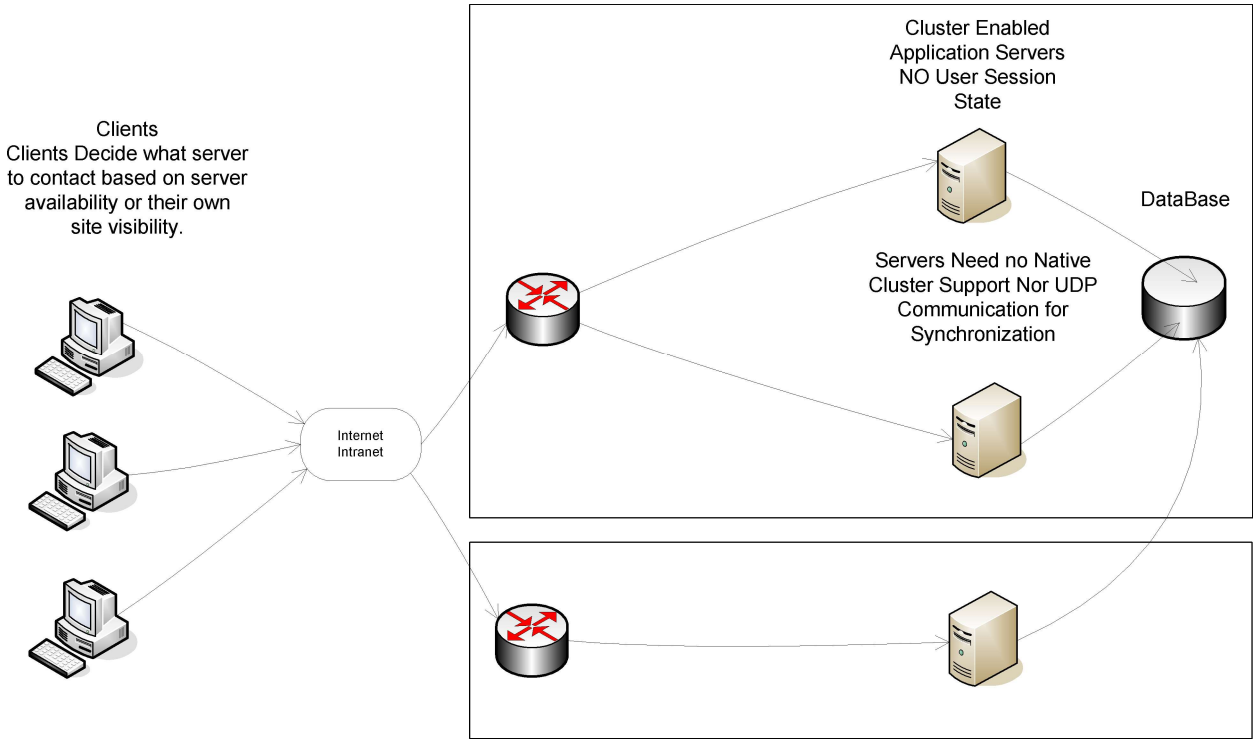


* Server Cluster Suppoert and Licencing $$

* Load Balancing Failure Detection Device $$

* Fast UDP synchronization hardware $$

With the Framework model, and using the advantages of Client maintained Session state, we can greatly simplify the setup, reducing costs and risks:

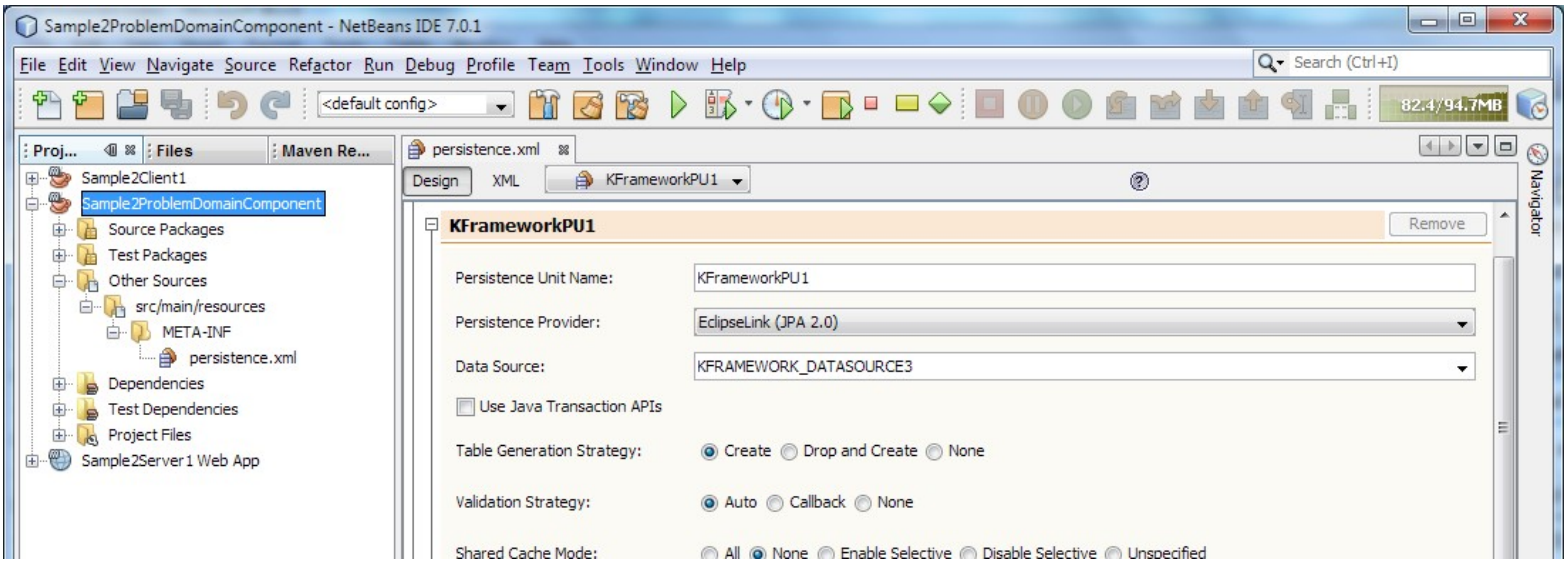


Using the KFramework, clients control the session state, allowing contacting any server. Load Balancing and Failure Detection is handled by the clients, they could even switch to different sites based on their own visibility to the server sites. No F5 or similar device is required. No cluster support and licensing are required from the servers, and you can even mix server vendors.

7.1 Configuring a Cluster

The delivered packages are configured assuming a cluster setup. You only need to configure the clients. In the other hand if you wish to increase the performance of a single server setup, you need to enable the JPA's caches which are disabled by default. Check JPA's documentation and the persistent.xml to enable cache.

The persistent.xml is in the *Sample2ProblemDomainComponent* module as follows:



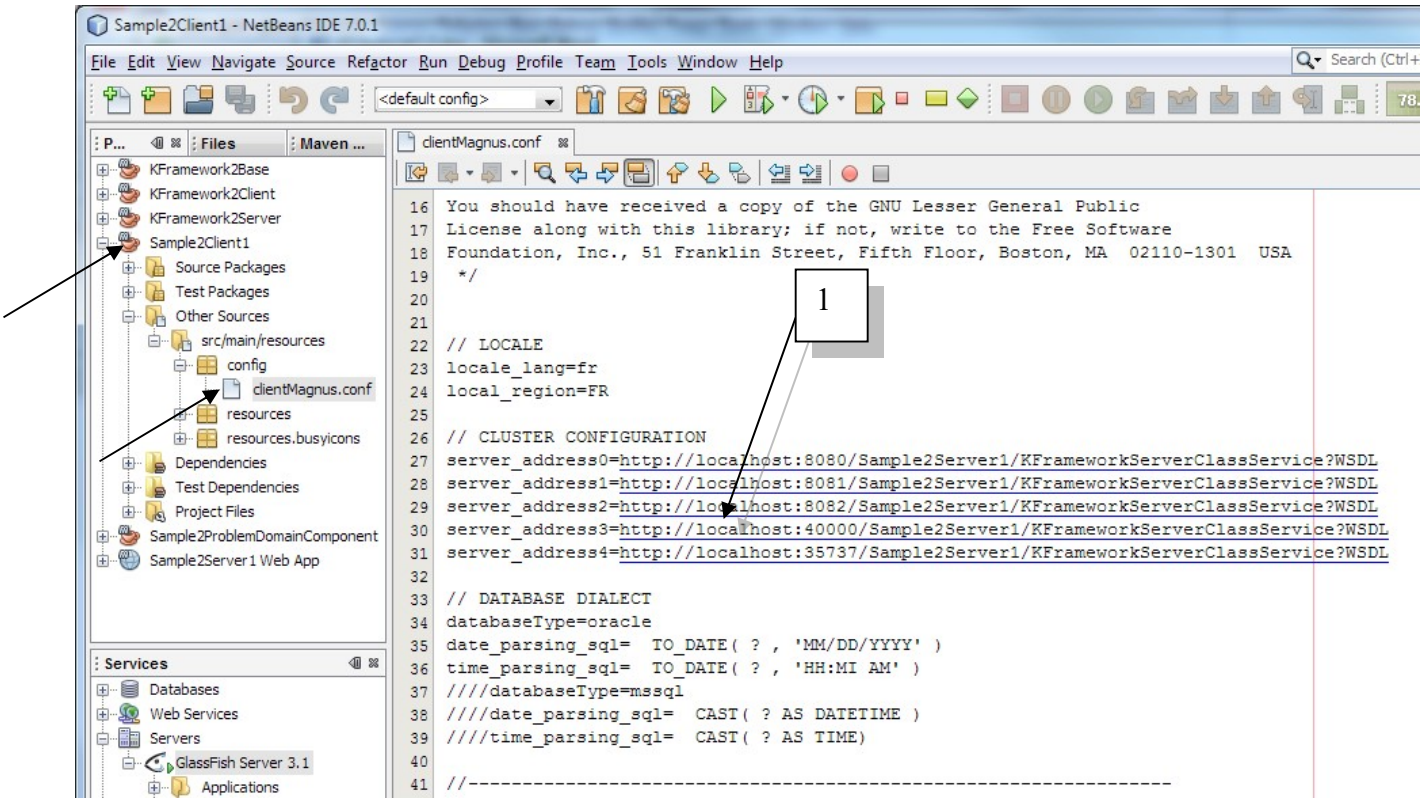
Note that if you enable JPA cache, changes to objects will not be immediately saved to the database leading to stale objects in other nodes and object version errors, or worst, loosing changes. Only enable JPA cache for a SINGLE server setup.

To configure the cluster:

1. Deploy all servers, just like in a single setup server

2. Register the servers in the client's magnus.conf

3. Open de clients configuration file, and configure user options:

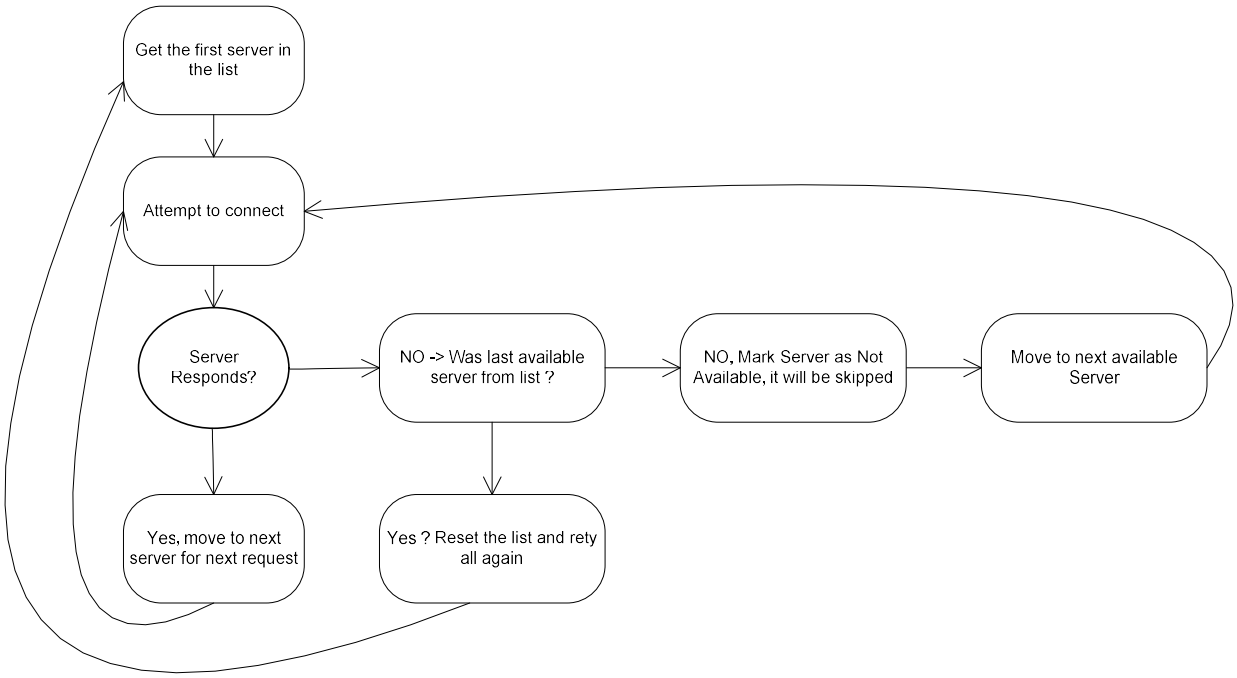


4. Set the servers addresses. You can have stand by servers, or even add entries for future setups.
5. Recompile the client, run the install script, recompile the server, and deploy. You can test it from netbeans.

7.2 How does the cluster work?

The cluster works because it is the client application that actively seeks the servers and keeps the application state, also, by keeping the communication to the server in atomic transactions any server can fulfill any request.

Once the client is downloaded and starts it begins the connection process, the connection process is as follows:



In summary the server will round robin all servers, and skip any server that is not reached, so that no time wasted contacting a failed server. The list is reset every time the application loads and all servers are tried again. If the application fails to contact any server it will then show an error message and reset the list. If the user retries all servers will be checked again and the application will continue if at least 1 server responds.

Since servers do not keep session state, servers can be shutdown or reset and still service clients with sessions already started.

Internally the server DOES keep a session, but these are restored in case they expire or a new client starts a connection started in any other server.

8 Appendix A: KDataBrowserBaseClass reference

Constructor Summary
DataBrowserBaseClass(KFrameWork.Base.configurationClass configurationParam, KFrameWork.Base.logClass logParam, boolean showKeyFieldParam, javax.swing.JTable tableParam, java.awt.Component componentParam) This constructor uses the standard configuration, log an optional key parameter used for dialog based browsers, the JTable to control and a visual parameter, normally (this), to be used as parent of dialog boxes.

Method Summary	
void	actionPerformed(java.awt.event.ActionEvent event) This is fired for all actions handled. New, Edit, Delete, Filter, Sort and Print, plus any extra you define be registering this browser as listener in a control.
void	addCustomCriteria(java.util.List filters) Used to add a fixed and not changeable filters.
void	addCustomCriteria(java.lang.String filter) Used to add a fixed and not changeable filters.
void	adjustColumnBackgroundColor(java.lang.String columnName, java.awt.Color bgColor) Sets the background color for the column This method should be called after initializeTable method called
void	adjustColumnEditor(int column_index, javax.swing.table.TableCellEditor CellEditor) Using a JTextField, JCheckBox, or JComboBox as an Editor If you are setting the editor for a single column of cells (using the TableColumn setCellEditor method), you specify the editor using an argument that adheres to the TableCellEditor interface.
void	adjustColumnFont(java.lang.String columnName, java.awt.Font font) Set the Font for the column This method should be called after initializeTable method called
void	adjustColumnForegroundColor(java.lang.String columnName, java.awt.Color fgColor) Sets the foreground color for the column This method should be called after initializeTable method called

void	adjustColumnJustification(java.lang.String columnName, int alignment)
	Sets the justification for the column This method should be called after initializeTable method called. Aligments are: * LEFT SwingConstants.LEFT * CENTER SwingConstants.CENTER * RIGHT SwingConstants.RIGHT * LEADING SwingConstants.LEADING * TRAILING SwingConstants.TRAILING
void	adjustColumnType(java.lang.String columnName, int type)
	Set the data type for the column This method should be called after initializeTable method called. Types are used to format the column. Valid types are: public static final int CHARACTER = 0; public static final int NUMERIC = 1; public static final int NUMERIC2 = 2; public static final int CURRENCY = 3; public static final int DATE = 4; public static final int TIME = 5;
void	adjustColumnWidth(java.lang.String columnName, int width)
	Sets the width for the column This method should be called after initializeTable method called
void	**adjustHeaderRenderer**(tableHeaderRendererClass renderer) Save renderers and subscribe operations. Se below for a tutorial on headers and calculated fields.
void	bindCustomParameter(java.lang.String parameterName, long parameterValue) Adds a bind parameter for the SQL used for the dynamic SQL at runtime.
void	bindCustomParameter(java.lang.String parameterName, java.lang.String parameterValue) Adds a bind parameter for the SQL used for the dynamic SQL at runtime.

void	bindDefaultParameter(java.lang.String parameterName, java.lang.String parameterValue) Adds a bind parameter for the SQL used for the fixed filter that will not change at runtime.
void	clearCustomCriteria() Release all custom filters and bound arguments.
void	clearDefaultOrder() Clears the order by clause.
void	copyButtonActionPerformed() Called to handle actions. User can override to handle on his own.
long	dataBaseRowCount(boolean applyCustomFilters) Executes a count on the recordset. You can decide to apply the current filters, or execute over the default SQL.
void	deleteButtonActionPerformed() Called to handle actions. User can override to handle on his own.
void	displayRefresh() Reload the view
void	editButtonActionPerformed() Called to handle actions. User can override to handle on his own.
java.util.AbstractMap	evaluateOperation(java.lang.String SQLformula, boolean applyCustomFilters) Used to execute a SQL operation, like "sum(total)". The browser adds the rest. You can operate over the default SQL, or over the current filters defined by the user at runtime.
void	filterButtonActionPerformed() Called to handle actions. User can override to handle on his own.
java.util.Vector	getCheckSelectedRowKeys(int column_index) Return the key field values of current multi selected rows marked by Check Box.
void	getColumnNames(java.util.List nameList) Gets the table column names This method should be called after initializeTable.
int	getColumnType(java.lang.String columnName) Gets the table column type via column name This method should be called after initializeTable.
java.lang.String[][]	GetCustomCriteriaRowData() Returns the current where clauses as configured by user at runtime.
java.util.List	GetCustomOrderData()

	Returns the current sort as configured by user at runtime.
java.lang.String	getDefaultCriteria() Returns the current where clauses that are fixed.
java.util.List	getDefaultParameters() Get the bound parameters used by the default / fixed criteria.
javax.swing.JTable	getJTable()
java.util.Vector	getMultiSelectedRowKeys() Return the key field values of current multi selected table rows.
java.util.Vector	getMultiSelectedRowKeysAsStrings() Return the key field values of current multi selected table rows.
recordClass	getRecord(long OID) Returns in a recordClass the record corresponding to the given OID. Only for displaying records.
java.lang.String	getSelectedColumnVisualHeader() Return the visual header of current selected table column.
java.lang.String	getSelectedFieldValue(java.lang.String ColumnName) Return the table value at the selected row under the column name
long	getSelectedRowKey() Return the key field value of current selected table row.
java.lang.String	getTableDataAsHtmlTable() Get an HTM view of the current table
java.lang.String	getTableDataAsString() Get an String view of the current table
java.util.List	getTableDataHeaders() Get the columna headers
void	initializeSQLQuery(java.lang.String SQLSelect, java.lang.String DBTable, java.lang.String keyFieldParam) Initialize SQL statement This method is called only once after constructor, and before any other method.
void	initializeTable() Load the data into table
boolean	isLoaded() Flag indicating whether the initializeTable() has been

	called.
void	markChanged(long OID, java.lang.String changedParam)
	Called to handle actions. User can override to handle on his own.
void	mouseClicked(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseClickPerformed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseDoubleClickPerformed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseEntered(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseEnteredPerformed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseExited(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseExitedPerformed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mousePressed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mousePressedPerformed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseReleased(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	mouseReleasedPerformed(java.awt.event.MouseEvent event)
	Called to handle actions. User can override to handle on his own.
void	newButtonActionPerformed()
	Called to handle actions. User can override to handle on his own.
void	notifyListeners(java.lang.String actionParam)
	Used to fire events manually.

void	**prepareCustomFieldsDBTransaction**(java.lang.String customFields, dbTransactionClientClass dbTransaction, boolean reflectCustomFilter)
	Gets the current SQL and sets a dbTransaction for query. This method should be called after initializeTable.
void	**prepareDefaultDBTransactionForTable**(dbTransactionClientClass dbTransaction)
	Gets the current SQL and sets a dbTransaction for query. This method should be called after initializeTable.
void	**prepareDefaultDBTransactionForTable**(dbTransactionClientClass dbTransaction, java.lang.String orderBy)
	Gets the current SQL and sets a dbTransaction for query. This method should be called after initializeTable.
void	**prepareTransactionWithBrowserSQL**(dbTransactionClientClass dbTransaction)
	Gets the current SQL and sets a dbTransaction for query. This method should be called after initializeTable.
void	print(java.lang.String report_name, java.lang.String report_owner)
void	printButtonActionPerformed()
	Called to handle actions. User can override to handle on his own.
void	refresh()
	Reload the table as the new setting applied.
void	refreshButtonActionPerformed()
	Called to handle actions. User can override to handle on his own.
void	**registerListener**(DataBrowserBaseClass.tableFillerListenerInterface listenerParam)
	Register to by notified for: NEW_ACTION EDIT_ACTION DELETE_ACTION SAVE_ACTION COPY_ACTION SORT_ACTION FILTER_ACTION

	REFRESH_ACTION PRINT_ACTION MOUSE_DBL_CLICK Used to fire a reload or a calculation.
void	resetToDefaults() Reload the table, clearing all custom filters.

void	**saveBrowserChanges**(tableFillerDataWriterInterface data Writer)
	Called for Read/Write Browsers. A read/write browser has some columns setup with controls that allow writing. This returns the changes to an implementor of:
	`import KFrameWork.Base.*;`
	`public interface tableFillerDataWriterInterface {`
	`public void save( java.util.List fieldNames, java.util.List data ) throws KExceptionClass;`
	`}`

void	saveButtonActionPerformed() Action handler for the user. Not implemented by browser.
void	saveSQLOperation(java.lang.Object visualComponent, java.lang.String sqlOperation, boolean reflectCustomFilter) Used to BIND widgets. Like a total under the table. Or a count used for another process. On every reload the browser will execute the SQL on the browser and update the control. Valid controls are JLabel or JTextField.
void	saveSQLOperation(java.lang.Object visualComponent, java.lang.String sqlOperation, int dataType, boolean reflectCustomFilter) Used to BIND widgets. Like a total under the table. Or a count used for another process. On every reload the browser will execute the SQL on the browser and update the control. Valid controls are JLabel or JTextField. Use browser datatypes for formatting.
void	setBrowserReadWrite(boolean flag, java.lang.String tableAlias) Set the read write flag to true
void	setCacheSize(int size)

	Set the size of the record cache in records.
void	**setCellDisplayHook**(cellRenderingHookInterface cellDisplayHookParam) Register a cell render hook for rendering customization cell by cell and at runtime.
void	**setCellWriter**(cellWriterInterface cellWriterParam)
	Receive the writing events for the cell, validate and store the data. The browser does not cache changes.
void	setColumnNames(java.lang.String aliasName, java.lang.String fieldName, java.lang.String headerName) Set the visual column name from a column. Note that you need to pass the corresponding table ALIAS, to avoid ambiguity with to tables having a cell named the same.
void	setColumnNames(java.lang.String aliasName, java.lang.String fieldName, java.lang.String headerName, boolean colEditable)
	Set the visual column name from a column. Note that you need to pass the corresponding table ALIAS, to avoid ambiguity with to tables having a cell named the same. Also indicate if the cell is editable.
void	setColumnNames(java.lang.String aliasName, java.lang.String fieldName, java.lang.String headerName, boolean colEditable, boolean defaultRender)
	Set the visual column name from a column. Note that you need to pass the corresponding table ALIAS, to avoid ambiguity with to tables having a cell named the same. Also indicate if the cell is editable.
void	setCustomCriteriaRowData(java.lang.String[][] data)
	Add custom SQL where criteria.
void	setCustomOrder(java.util.List orderList)
	Set the order of the browser
void	setDefaultCriteria(java.lang.String criteria)
	Add custom SQL where criteria.
void	setDefaultOrder(java.lang.String order)
	Set the initial order of the browser
void	setDefaultParameters(java.util.List parameters)
	Set the parameters to bind to the default criteria.
void	setDoubleClickEnabled(boolean doubleClickEnabledParam)
	Indicate whether double click event is enabled. Usefull when browsers are read only.
void	setTableFont(java.awt.Font font)
	Sets the Font for the table This method can be

	called before or after initializeTable method called
void	setVisibleColumnCount(int visibleColumnCountParam)
	To hide columns, add them at the end and define here how many are visible.
void	softRefresh()
	Reload with current cache data, no access to server.
void	sortButtonActionPerformed()
	Action handler, the user can override.

9 Appendix B: GNU License

The KFramework is
Copyright © 2001 Alejandro Vazquez , Ke Li
(avazqueznj@users.sourceforge.net)

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts as the successor of the GNU Library Public License, version 2, hence the version number 2.1.]

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages--typically libraries--of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes
a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the

users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE
TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion

of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) The modified work must itself be a software library.

b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.

c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.

d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for

that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the

Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.

c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.

d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.

e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on

the Library and of the other library facilities is otherwise
permitted, and provided that you do these two things:

a) Accompany the combined library with a copy of the same work
based on the Library, uncombined with any other library
facilities. This must be distributed under the terms of the
Sections above.

b) Give prominent notice with the combined library of the fact
that part of it is a work based on the Library, and explaining
where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute
the Library except as expressly provided under this License. Any
attempt otherwise to copy, modify, sublicense, link with, or
distribute the Library is void, and will automatically terminate your
rights under this License. However, parties who have received copies,
or rights, from you under this License will not have their licenses
terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not
signed it. However, nothing else grants you permission to modify or
distribute the Library or its derivative works. These actions are
prohibited by law if you do not accept this License. Therefore, by
modifying or distributing the Library (or any work based on the
Library), you indicate your acceptance of this License to do so, and
all its terms and conditions for copying, distributing or modifying
the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the
Library), the recipient automatically receives a license from the
original licensor to copy, distribute, link with or modify the Library
subject to these terms and conditions. You may not impose any further
restrictions on the recipients' exercise of the rights granted herein.
You are not responsible for enforcing compliance by third parties with
this License.

11. If, as a consequence of a court judgment or allegation of patent
infringement or for any other reason (not limited to patent issues),
conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not
excuse you from the conditions of this License. If you cannot
distribute so as to satisfy simultaneously your obligations under this
License and any other pertinent obligations, then as a consequence you
may not distribute the Library at all. For example, if a patent
license would not permit royalty-free redistribution of the Library by
all those who receive copies directly or indirectly through you, then
the only way you could satisfy both it and this License would be to
refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under
any
particular circumstance, the balance of the section is intended to
apply,
and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any
patents or other property right claims or to contest validity of any

such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add
an explicit geographical distribution limitation excluding those countries,
so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY

AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS