APSP[1]

Abraham James[2], Christopher Rebstock[3], Kareemullah Quadir[4],
Roshan Rammohan[5], Vinod Doshi[6], Yaroslav Halchenko[7]
http://apsp.onerussian.com

April 4, 2002

[1]Automated Personal Software Process
[2]ajames@cs.unm.edu
[3]chrisr@cs.unm.edu
[4]quadir@cs.unm.edu
[5]roshan@cs.unm.edu
[6]vdoshi@cs.unm.edu
[7]yoh@cs.unm.edu

Contents

1 Introduction **2**

2 Architecture Design **3**

2.1 System Design . . . 4
2.2 CVS Wrapper . . . 4
2.3 Database design . . . 4
2.4 OO Design (Classes) . . . 10
2.4.1 Client . . . 12
2.4.2 CDir . . . 16
2.4.3 CDirInterface . . . 18
2.4.4 Server . . . 19
2.4.5 File . . . 21
2.4.6 Dir . . . 23
2.4.7 Project . . . 24
2.4.8 DB . . . 26
2.4.9 Phase . . . 27
2.4.10 PPS . . . 28
2.4.11 PP . . . 30
2.4.12 PIP . . . 32
2.4.13 Defect . . . 33
2.5 Sequence Diagram . . . 36
2.6 Data Flow Diagram . . . 36

Chapter 1

Introduction

In this paper we present design for our project:

- general architectural design
- database schema
- OO design (classes)
- example of sequence diagram on specific use case
- example of data flow diagrams on specific use case.

Chapter 2

Architecture Design

Its basic structure is a 3-tier architecture and it consists of 4 main components: Client, Server, Database and the Web Application [1]. Each of the components are described below.

We tried to decrease any coupling between business logic and user interface elements of the design. So UI on client side is separate class from real client, so UI is not aware of Client/Server organization of the system at all. It just delegates necessary functionality to the client.

The GUI of the client (class **MainWindow**) will be the main interface between system and the user. GUI invokes methods of **Client** which communicates with the **Server**. The main task of the **Client** is to gather data from the user through UI and to grab statistics on project files and then to send them to the **Server**. Information about changes on files will be drawn from comparison of files in a directory and files in *./APSP*[2] subdirectory where we would keep snapshot of files from previous comparison. After comparison is done, files with changes from current directory are copied over files in *./APSP* subdirectory, so when next time monitoring is done we will have only new changes in files.

On the other hand **Client** receives some statistical information from the **Server** in the form of forms (**PIP**, **PPS** and **DRL**). No validation is performed on the client-side, which makes it simple and compact. Loose coupling between UI and **Client** gives the flexibility to change, upgrade and add new features to UI in future versions with no or minimal change in the "business" logic. Almost all validations and business logic is applied at the **Server** and corresponding server-side classes. The **Client** monitors the projects and the corresponding files for lines added, modified and deleted. It also keeps track of the times periods when these changes occur, records and sends the time spent in each phase of the project. It also allows the user to switch between projects. Both UI and Client can in no way communicate with the database thus securing the system. **Client** takes input from the UI and files' statistics and sends requests/updates to the **Server**. It also receives information and data from the **Server** and displays it to the user.

All mentioned components will be implemented using Java technology. Client, Server and web-interface would use Java RMI serializable objects and method calls from remote objects to provide intercommunication.

The **Server** is the gateway between the **Client** and the database (basic interactions with database covered by class **DB**). It allows the user to be registered with the system. Client requests server to authenticate the user and allows him to add and edit forms, specify project information, change user information once authenticated. If the user forgets the password, it emails the password to him/her. All the validations are performed in the Server and the business logic too is applied at the **Server**. When **Server** receives requests from the **Client** it queries the database, performs validations and sends corresponding information to the **Client**. It also receives data from the **Client** and updates the database. The **Server** uses MySql JDBC driver to connect to the MySQL database which would store all the information.

The Database stores the information received from the server. The database would have tables to store information

[1] There is 5th component of the system - CVSwrapper, which will be involved when the user invoke "cvs update" command which can bring code which doesn't belong to the user, so we need to avoid getting statistics from it

[2] So each directory which deals with CVS and has *./CVS* subdirectory also would have *./APSP* subdirectory for our purposes

pertaining to Users, Projects, Phases, Files, Directories, Defects, PIP forms, PPS forms, Reports. It would have relationships between the tables [3].

The Web Application allows users to view their graphs, PIP forms, PPS forms and DRL forms. The Web Application implements user authentication to secure user information in case when user tries to get information without activating client's UI elements[4]. Multiple users can login simultaneously and view their projects' information. This would allow users to have access to their projects even when they dont have the Client on their machine thus giving him/her universal accessibility.

The Web Application does not allow the user to change any information in the database regarding his/her projects, it only allows the user to view the information. If the user wishes to change information, then he must use the Client. This would make the system more secure. The user can view graphs through the project ranges to within the project. The resolution of the graphs would be the refresh time that he has set. He can even view graphs on a single file or directories. He can also view PIP, PPS and DRL forms for each project[5]. Web Application would be implemented using Java Servlets. It would also RMI to get information from the server, so we wouldn't need to open another DB connection to our database and will have just single unified interface of the server's classes to get any necessary information.

2.1 System Design

2.2 CVS Wrapper

CVS wrapper is just a simple bash script which will be aliased against cvs. So when user needs to run *cvs* he needs to run our wrapper (lets name it *apspcvs*. The only duty of the wrapper is to react on "update" command to CVS because it can bring a lot of modifications to the code of the program which are not introduced by current user. The wrapper is necessary to prevent these lines of code which the user didn't write from being counted.

So *apspcvs*, when "update" command is asked will do next: for each directory which is subdirectory of directory where cvs command is invoked, it will copy files which are monitored to directory *./APSP/STAMPZZ/BEFORECVS/*[6], run "cvs update" with parameters which were specified by user, then make a copy of all files which were updated to the directory *./APSP/STAMPZZ/AFTERCVS/* and remove unmodified files from directory *./APSP/STAMPZZ/BEFORECVS/*.

After this the Client will monitor files for changes it can find that there are directories *./APSP/STAMPZZ* somewhere on its path. So it will track changes considering that changes between files in *./APSP/STAMPZZ/AFTERCVS/* and *./APSP/STAMPZZ/BEFORECVS/* are not valid. After client monitors all changes it removes *./APSP/STAMPZZ* directories.

Locking mechanism will be used to prevent any modification on files when either client or CVS wrapper working on them. We will use simply file *./APSP/lock* which would symbolize that somebody (client or cvs wrapper) is working on files currently in this directory, so the other process should wait till this file will be removed and then proceed further.

2.3 Database design

Our database has a set of 14 Tables enumerated thus. (See Fig. 2.2);

[3] More detailed description of the database will be provided in the next chapter

[4] When user activates menus/buttons within client, no authentication would be required because client has information about current opened session, so it can tell specify web interface hash value for current opened session

[5] This capability wasn't mentioned in UI document, but that one will be updated to reflect this

[6] "ZZ" here in directory names just next available number. So if there was already directory *STAMP01*, then *STAMP02* will be created for current case

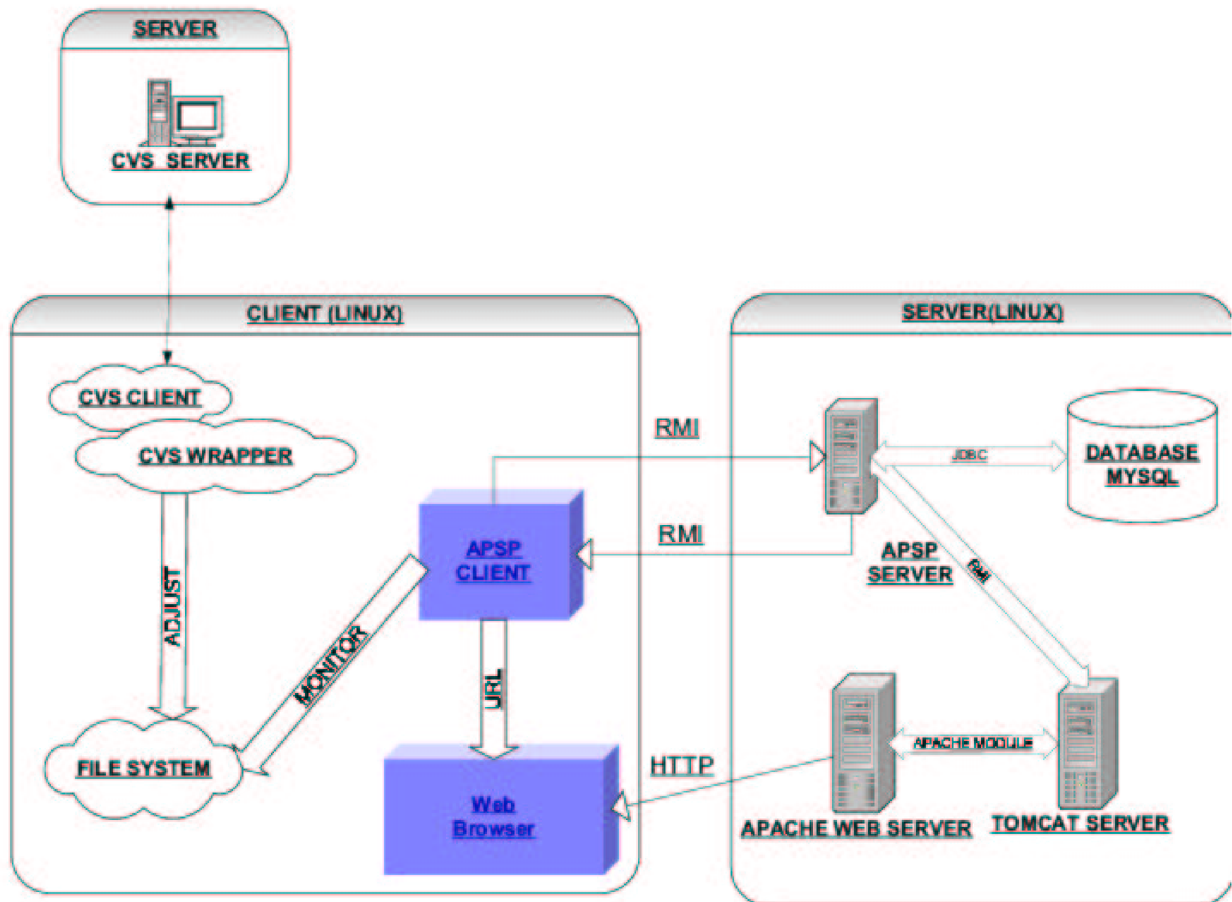

Figure 2.1: System Design Diagram

1. USERS
   The Users table has the following fields.
   - USR_ID : int A unique key field for this table.
   - LOGIN : string Holds User's Login name.
   - EMAIL:string User's Email ID.
   - PASSW_HASH : bytes User's encrypted password information.
   - LOGGED_IN : bool Status of User, If the user is currently logged in or not.
   - SESSION_HASH : bytes
2. FILE_CHANGES
   The FILE_CHANGES table has the following fields.
   - FL_CH_ID : int A unique key field for this table.

- USR_USR_ID : int Stores User Id of User who owns/operates in this file.
- FL_FL_ID:int Stores File Id of the associated file.
- LINES_ADDED: int Stores the number of lines added in this particular file change.
- LINES_REMOVED: int Stores the number of lines removed in this particular file change.
- LINES_MODIFIED: int Stores the number of lines modified in this particular file change.

3. FILES
   The FILES table has the following fields.
   - FL_ID : int A unique key field for this table.
   - PRJ_PRJ_ID : int Stores the Project Id of the project that this file is associated with.
   - FILENAME:string Stores Filename
   - PATH_IN_PRJ: string Stores relative path of the file from root directory of project.
4. PHASE
   The PHASE table has the following fields.
   - PHS_ID : int A unique key field for this table.
   - NAME : string Stores the name of the Phase.
5. TREE
   The TREE table has the following fields.
   - FL_R_ID : int A unique key field for this table.
   - FL_R_PARENT_ID : int Stores the key of the parent directory.
   - DEPTH : int Stores the depth from the root directory.
6. PLACE
   The PLACE table has the following fields.
   - PL_ID : int A unique key field for this table.
   - NAME : string Stores the name of the place where the user works from.
7. LOGINS
   The LOGINS table has the following fields.
   - LG_ID : int A unique key field for this table.
   - PLC_PLC_ID : int Stores the PLC_ID of the place where user logged in from for which session a record of this table was created.
   - USR_USR_ID : int Stores the PLC_ID of the place where user logged in from for which session a record of this table was created.
   - TIME : time Stores the logon time.
8. PHASE_CHANGES
   The PHASE_CHANGES table has the following fields.
   - PSCH_ID : int A unique key field for this table.
   - USR_USR_ID : int Stores the User's ID to whom a record of phase change is assocated with.
   - FL_R_ID : int Stores the ID of the file in tree to associate it to the directory in which user was working when changing the phase.

- PHS_PHS_ID : int Stores the Phase ID of the phase that was changed.
- TIME_START : time Stores the time when work began on this phase.
- TIME_END : time Stores the time when work ended on this phase.

9. PIP
   The PIP table has the following fields.
   - PIP_ID : int A unique key field for this table.
   - USR_USR_ID : int Stores the User's ID to whom a record of PIP is assocated with.
   - PRJ_PRJ_ID : int Stores the ID of Project which is assocaited with this PIP.
   - COMMENTS : string Stores user comments in reference to the proposal.
10. PROBLEM_PROPOSAL
   The PROBLEM_PROPOSAL table has the following fields.
   - PP_ID : int A unique key field for this table.
   - PIP_ID : int Stores PIP to which this PP belongs.
   - PROBLEM : string Stores the Problem description.
   - PROPOSAL : string Stores the Proposal description.
11. DEFECT_REPORTS
   The DEFECT_REPORTS table has the following fields.
   - DFCT_ID : int A unique key field for this table.
   - PRJ_PRJ_ID : int Stores Project ID to which this defect report belongs to.
   - USR_USR_ID : int Stores User ID to whom this defect report is associated with.
   - PHS_PHS_ID_INJECTED : int Stores Phase id of the phase when defect was injected.
   - TIME_INJECTED : time Stores Phase id of the phase when defect was injected.
   - PHS_PHS_ID_REMOVED : int Stores Phase id of phase when defect was removed.
   - TIME_REMOVED : time Stores Phase id of phase when defect was removed.
   - DESCRIPTION : string Stores the description of the defect.
   - FIX_TIME: int Stores the time taken to fix defect in minutes.
12. PROJECTS
   The PROJECTS table has the following fields.
   - PRJ_ID : int A unique key field for this table.
   - CVS_PATH : int Stores the path of root directory of the project as is in the CVS server.
13. PPS
   The PPS table has the following fields.
   - PPS_ID : int A unique key field for this table.
   - PRJ_PRJ_ID : int Stores project id of the project to which this PPS belongs to.
   - USR_USR_ID : int Stores user id of the user to whom this PPS belongs to
   - LOC_HR : int Stores the Lines of code per hour. (Estimated Value)
   - TOTAL_LOC : int Stores the Total number of lines of code. (Estimated Value)
   - TOTAL_TIME : int Stores the total time taken for the project summary.

14. PPS_ITEMS
    The PPS_ITEMS table has the following fields.

    - PPS_PPS_ID : int Stores the PPS id of the PPS that this item is associated with.
    - PHS_PHS_ID : int Stores the Phase id of the phase for which we note parameters.
    - DEFECTS_INJ : int Stores the number of defects injected in one phase.
    - DEFECTS_REM : int Stores the number of defects removed in one phase.
    - TIME : int Stores the time spent in one phase.

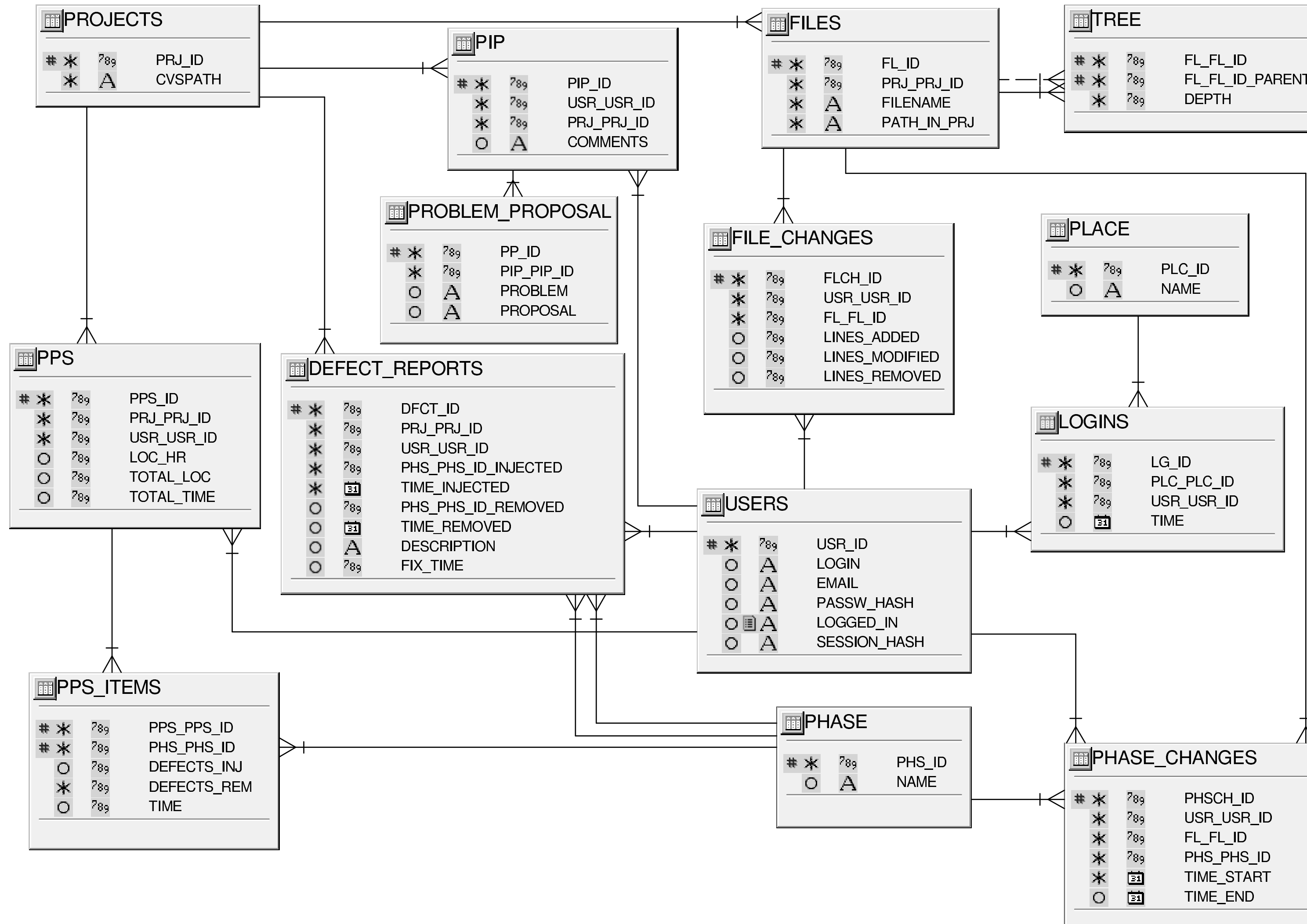

Figure 2.2: Database design.

2.4 OO Design (Classes)

In this sections we present classes we designed for our system. Some classes are server-side RMI classes so they need corresponding interface declaration based on methods they implement.

These interfaces can be drawn directly from corresponding classes that is why they are not presented in this paper.

There are two major parts of classes in our design. We did strong decoupling of UI from any logic, so UI interface (class **MainWindow** and its members - menus, buttons etc) delegates actual actions to be implemented by **Client**.

UI classes have direct corresponding to designed UI that is why there is no much sense in detailed description of each of them. You can see general UI classes diagram on Fig. 2.3

Further we will present classes for client and server which we would have in our system. Communication between client and server will be implemented using Java RMI, some classes will be not serializable but server-side so we can communicate to the database invoking on client side their methods on server. So class **Client** has not actually instances of class **Project** but it has handler to it. We don't distinguish between class and handler in name of type in diagrams. Either it is handler or not can be decided on type serializable/server-size of the class and place of current class (client/server)

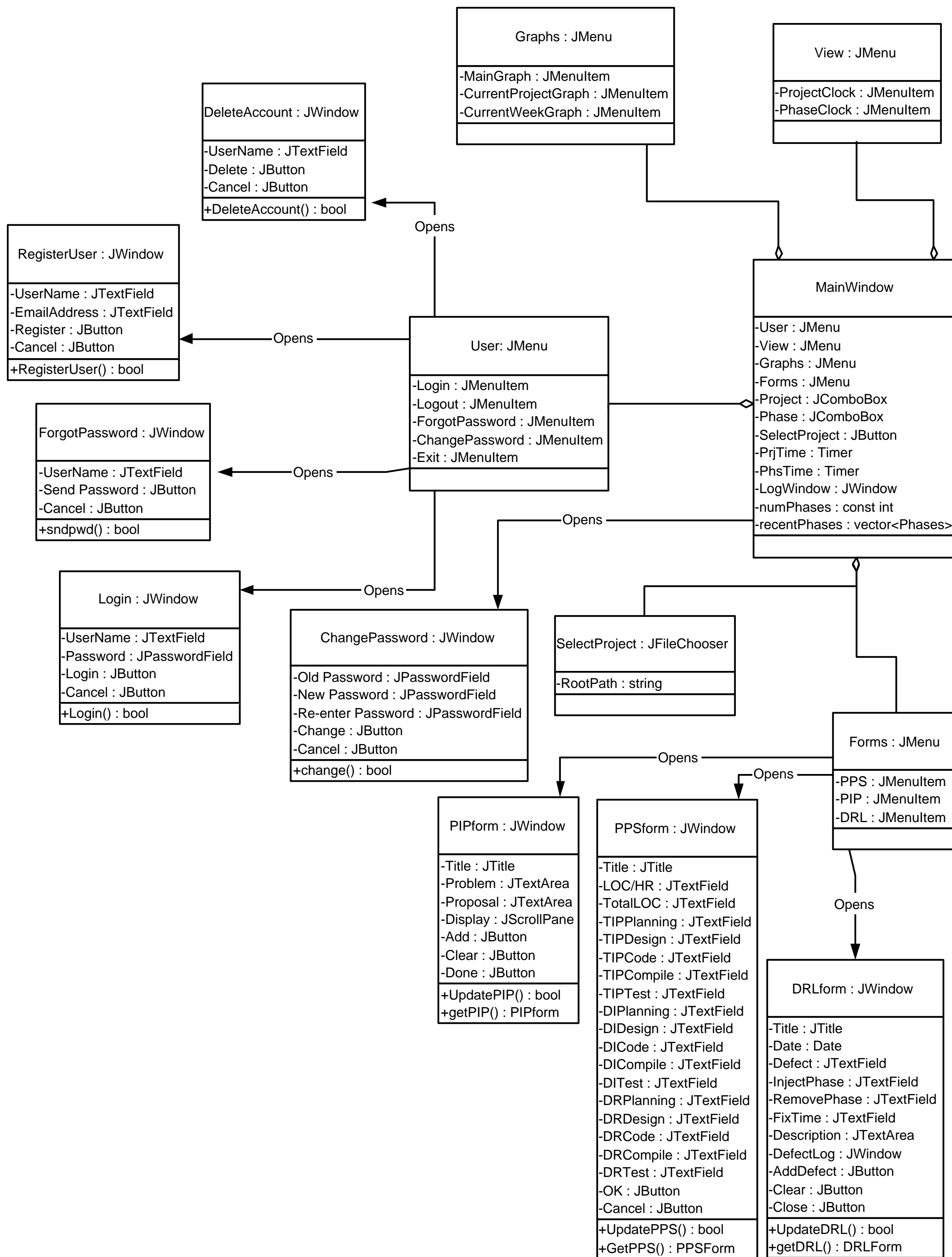

Figure 2.3: UI classes.

2.4.1 Client

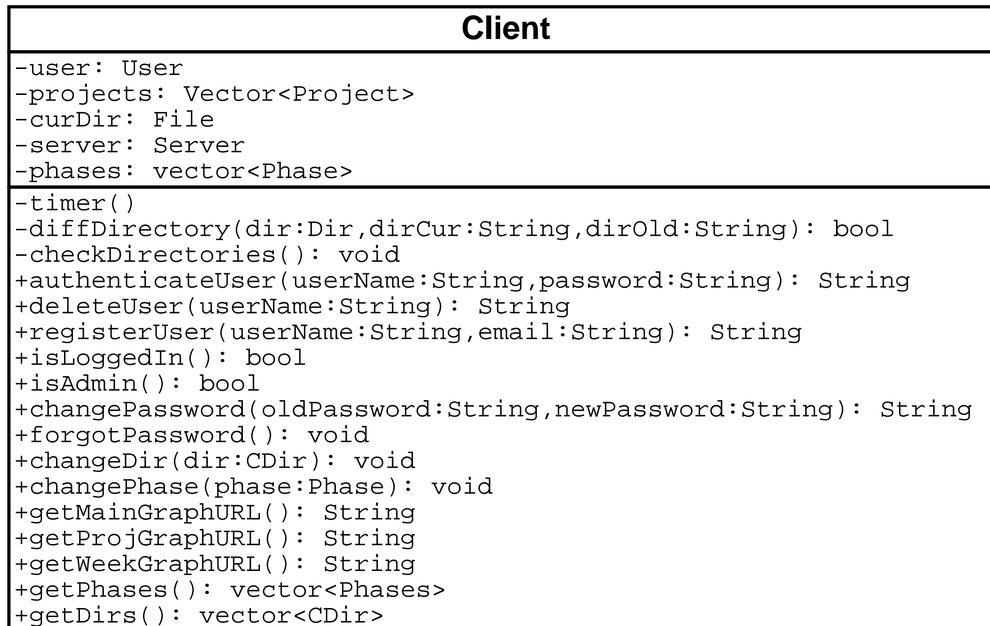

Figure 2.4: Class Client

Attributes

- **Private:**

  user:User Specifies handler for server side object of type **User**.

  projects:Vector{Project} List of projects in which user currently involved so between which he will be able to switch. Current list is built based on list of directories found in configuration file.

  curDir:File Specifies handler for server side object of type **File** which is current selected file/directory on which user is working on.

  server:Server Specifies handler for server side object of type **Server** which corresponds to the server main object. It is obtained through RMI mechanism at the time when client runs and reads address of the server from configuration file.

  phase:Vector{Phases} Just a container for objects which describe possible phases. This list is obtained from the server after connection to it set up.

Methods

- **Private:**

  ---

  timer(...)

  Thread which runs in background and runs checkDirectories() in specified amount of time.

---

checkDirectories(...)

Goes through all directories of current project and checks for changes it find there using diffDirectory() method.

---

diffDirectory(...)

Compares files in two directories which is current and the one with files from time when it was previously checked. If it finds changes in any of files, it reports them to the server and copies them into shapshot directory so we can keep track later on new changes.

Arguments:

- **dir:Dir** server side class which correspond to this directory
- **dirCur:String** directory with files which can be updated
- **dirOld:String** it is directory with snapshot which was made before

Returns (type **bool**): returns true if there were files changed in current directory (not recursive check here).

- **Public:**

---

authenticateUser(...)

Arguments:

- **userName:String** name of the user provided by user interface to the client
- **password:String** password provided by UI for mentioned **userName**

Returns (type **String**): returns empty string in case when user was properly authentication, otherwise returns string with error message to be displayed to the user.

---

deleteUser(...)

Current method performs any operation just in case when isAdmin() function returns true.

Arguments:

- **userName:String** name of the user provided by user interface to the client

Returns (type **String**): returns empty string in case when user deletion was without any errors, otherwise returns string with error message to be displayed to the user (like if userName doesn't correspond to any user registered within the system).

---

registerUser(...)

Current method performs any operation just in case when isAdmin() function returns true.

Arguments:

- **userName:String** name of the user provided by user interface to the client
- **email:String** email of the user, where generated random password will be sent, so user can change it to the new one.

Returns (type **String**): returns empty string in case when user registration was without any errors, otherwise returns string with error message to be displayed to the user (like if userName is already registered within the system).

---

isLoggedIn(...)

Returns (type **bool**): returns "true" when user is logged in with the server.

---

isAdmin(...)

Returns (type **bool**): returns "true" if current user is administrator so UI can create specific for Administrator UI items in the menu..

---

changePassword(...)

Just a proxy function to call method on server side **user** object.

Arguments:

- **oldPassword,newPassword:String** old and new passwords accordingly.

Returns (type **String**): empty string in case if password was successfully changed, otherwise returns error message.

---

forgotPassword(...)

Just a proxy function to call method on server side **user** object.

---

changeDir(...)

Mostly proxy function to call methods on server side to change current project.

Arguments:

- **dir:CDir** directory to which user wish to change.

---

changePhase(...)

Proxy function to call method on server side **project** object to change current phase of the project.

Arguments:

- **phase:Phase** phase to change to

---

getPhases(...)

Gives UI possibility to obtain list of available **Phases**. Because **Phase** is just simple serializable class we can pass all the way to the UI.

Returns (type **Vector**{**Phases**}): vector of phases currently used in the system.

---

getDirs(...)

Provides UI with vector of directories between which user can switch.

Returns (type **Vector**{**CDir**}): vector of directories of projects user currently working on. This list obtained from configuration file on local hard drive.

---

getMainGraphURL,getProjGraphURL,getWeekGraphURL(...)

Also client runs web browser, path to which provided in *.apsprc* file and opens a page from web server with statistics about projects of current logged in user.

Returns (type **String**): these methods return URL to the web server by which graphs for current user and accordingly general, for project and for current week graphs can be shown (in case that UI can provide web browser capability and no external browser necessary).

2.4.2 CDir

Class **CDir** serves as a messenger between client and UI. We can't just give UI handlers to a server side **Dir** objects, that is why we create an interface **CDirInterface** which has basic functions to keep tree structure. Manipulating with it UI can specify which directory use as next active one. On other hand each **CDir** instance will have also information about corresponding object on server-side.

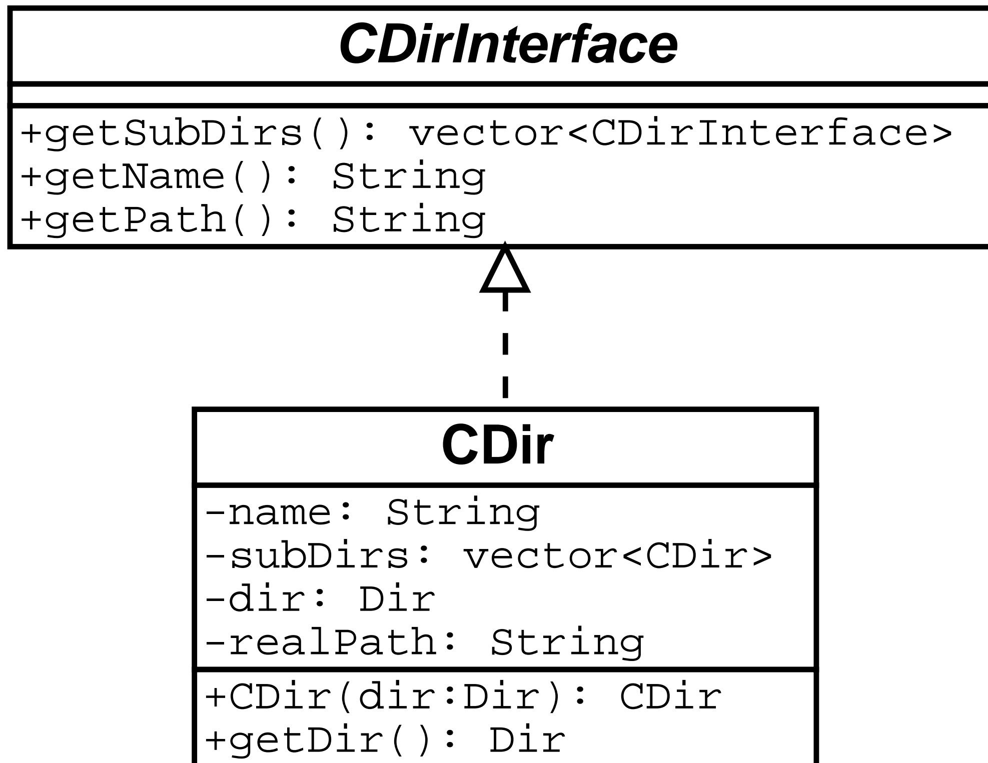

Figure 2.5: Class CDir

Attributes

- **Private:**

  name:String Name of directory.

  subDir:vector¡Dir¿ Stores the list of subdirectories.

  dir:Dir realPath:String

Methods

- **Public:**

  CDir(...)

  Arguments:

– **dir:Dir** Id of the phase.

Returns (type **Cdir**): Constructor, so returns object of same type.

---

getDir(...)

Returns (type **Dir**): Directory

2.4.3 CDirInterface

Methods

- **Public:** ———————

  getSubDirs(...)

  Returns (type **vector¡CDirInterface¿**): Gets list of all subdirectories for display.

  ---

  getName(...)

  Returns (type **String**): Returns name.

  ---

  getPath(...)

  Returns (type **String**): Returns Path

2.4.4 Server

```
Server
-db: DB
-openedProjects: container<Project>
+authenticateUser(login:string,password:string,place:String): User
+getUser(session_hash:string): User
+getPhases(): vector<Phases>
+getProject(cvspath:String): project
```

Figure 2.6: Class Server

Server-side class interactions with which would be implemented through RMI remote-method invocations. This class performs the duty of initial access to corresponding class as **Phase**, **Project** and **User**.

Attributes

- **Private:**

  db:DB This is an object of the type **DB**.

  openedProjects:container This is some kind of container(a vector or a hash table) containing objects of the class **Project** so we dont have to create separate objects for inquiries to the same project.

Methods

- **Public:**

  ---

  authenticateUser(...)

  Arguments:

  - **userName:String** name of the user provided by client
  - **password:String** password for the user

  Returns (type **String**): returns empty string in case when user was properly authenticated, otherwise returns string with error message to be displayed to the user.

  ---

  getPhases(...)

  Returns (type **vector(Phase)**): returns a vector containing all the phases defined in the system.

  ---

  getProject(...)

  Finds a project for specific CVS path, so just by looking in real physical directory on hard drive and *CVS/Root*[7] there we can find necessary object on server side.

  Arguments:

  - **cvspath:String** the path in CVS where the directory resides

[7] It can look like *:pserver:yoh@earwax.cs.unm.edu:/home/yoh/progr/cs460/cvs.data*

Returns (type **Project**): returns the **Project** object (NULL object if no such directory found)

---

getUser(...)

Arguments:

- **sessionHash:String** sessionHash is used to check if the user is alreasy logged in

Returns (type **User**): returns the User object

2.4.5 File

The File class will reside on the server so will be a server-side class so it can interact with **db**. There is a constructor for file where we provide all its private attributes to initialize.

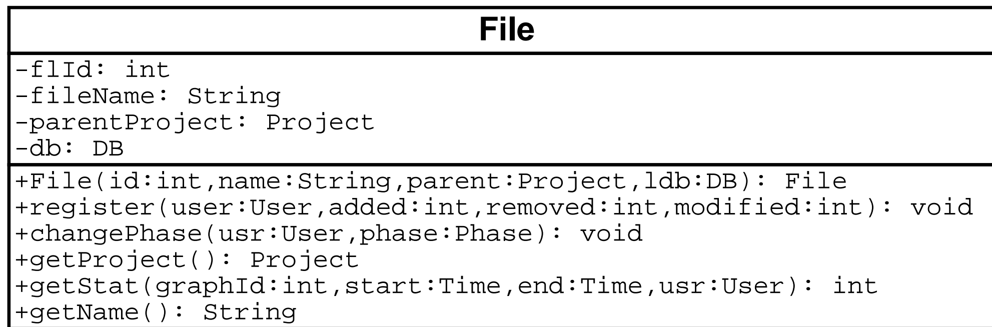

Figure 2.7: Class File

Attributes

- **Private:**

  flId:int An integer handler which would be a unique id for each File.

  fileName:String Name of the file without path.

  parentProject:Project The Project to which this file belongs.

  db:DB The DataBase.

Methods

- **Public:**

  ---

  register(...)

  Registers changes on the file in **db**

  Arguments:

  - **usr:User** The User with which this class is associated.
  - **added:int** Lines added.
  - **removed:int** Lines removed.
  - **modified:int** Lines modified.

  Returns (type **void**): Just registers a change in the file status.

  ---

  changePhase(...)

  Arguments:

  - **usr:User** The User with which this class is associated.

– **phs:Phase** The phase that was changes to.

Returns (type **void**): Just registers a change in the phase status.

---

getProject(...)

Returns (type **Project**): Returns the Project object to which the file belongs.

---

getName(...)

Returns (type **String**): Returns file's name.

---

getStat(...)

Arguments:

– **graphId:int** A handler for the type of graph that needs to be displayed.
– **start:Time** Start time.
– **end:Time** End time.
– **usr:User** The user specific to whom graphs need o be displayed.

Returns (type **int**): return a value for mentioned type of graph.

2.4.6 Dir

The **Dir** class will reside on the server. The **Dir** class inherits all attributes from the **File** class and overrides getStat() method because statistics on directory is composed from statistics on files and directories which belong to it.

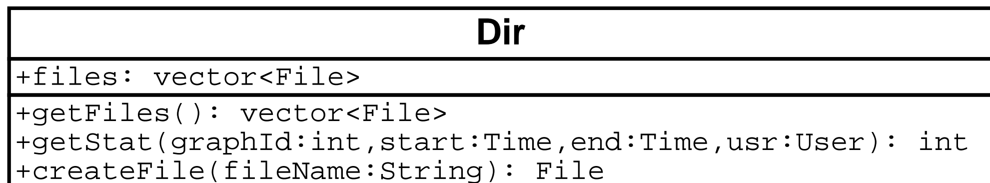

Figure 2.8: Class Dir

Attributes

- **Private:**
  files:Vector{File} This would contain the objects corresponding to all the files in this directory.

Methods

- **Public:**

  ---

  getFiles(...)

  Returns (type **Vector{File}**): returns all the Files in this directory.

  ---

  createFile(...)

  Creates a file in this directory. This function is necessary when there is no file in **db** yet.

  Arguments:

  - **filename:String** the name of the file to be created

  Returns (type **String**): creates a file and returns the **File** object

  ---

  getStat(...)

  (overoaded) Arguments:

  - **graphId:int** A handler for the type of graph that needs to be displayed.
  - **start:Time** Start time.
  - **end:Time** End time.
  - **usr:User** The user specific to whom graphs need o be displayed.

  Returns (type **int**): return a value for mentioned type of graph.

2.4.7 Project

The Project class too will reside on the server but will be serializable.

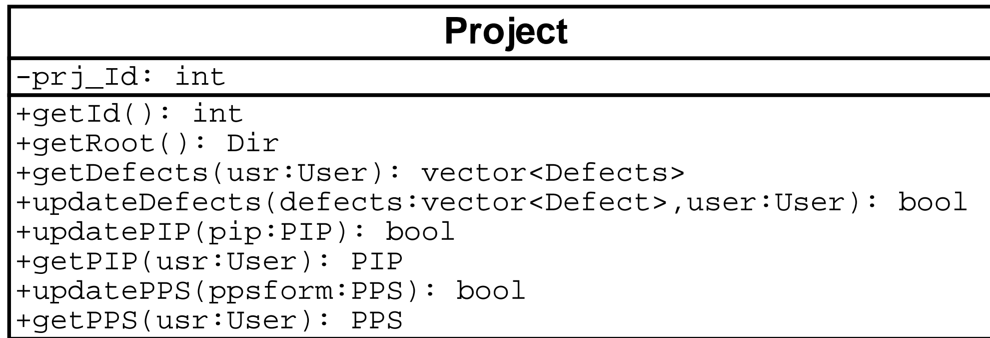

Figure 2.9: Class Project

Attributes

- **Private: prjId:int** A unique handler for each project in the system.

Methods

- **Public:**

  ---

  getID(...)

  Returns (type **int**): Returns the projectID.

  ---

  getRoot(...)

  Returns (type **Dir**): Returns the root directory of the project.

  ---

  getDefects(...)

  Arguments:

  – **usr:User** The user whose Defect log is needed.

  Returns (type **Vector{Defect}**): Returns a collection of all the defects in the log for that project.

  ---

  updateDefects(...)

  Arguments:

  – **usr:User** The user whose Defect log is needed.

  – **defects:Vector{Defect}** a collection of all the defects which were modified or added, so they should be updated in **db**.

Returns (type **bool**): Returns true in case of a successful update or false otherwise.

---

updatePIP(...)

Arguments:

- **PIP** An object of type PIP which contains process improvement proposal inofrmation.

Returns (type **bool**): Returns true in case of a successful update or false otherwise.

---

getPIP(...)

Arguments:

- **usr:User** The user whose PIP form is needed.

Returns (type **PIP**): Returns an object of type PIP with all information about user's Process Improvement Proposal.

---

updatePPS(...)

Arguments:

- **PPS** An object of type PPS which contains project plan summary information.

Returns (type **bool**): Returns true in case of a successful update or false otherwise.

---

getPPS(...)

Arguments:

- **usr:User** The user whose PIP form is needed.

Returns (type **PPS**): Returns an object of type PPS with all information about user's Project Plan Summary.

2.4.8 DB

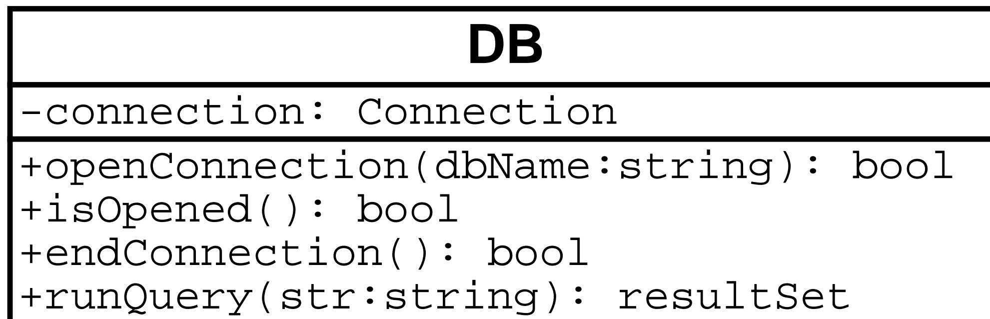

Figure 2.10: Class DB

Attributes

- **Private:**
  connection:Connection This is an object of the type Connection.

Methods

- **Public:**

  ---

  openConnection(...)
  Arguments:
  - **dbName:String** name of the database

  Returns (type **bool**): returns true if connection is opened, else returns false.

  ---

  endConnection(...)
  Returns (type **bool**): returns true if connection is closed, else returns false.

  ---

  isOpened(...)
  Returns (type **bool**): returns true if connection is open, else returns false.

  ---

  runQuery(...)
  Arguments:
  - **str:String** this string has the query to be run on the database

  Returns (type **Resultset**): returns the Resultset returned when the query is executed.

2.4.9 Phase

This class is used by the other classes to get the phase for a specific project. This is an RMI serializable class , hence its definition will exist on both client and server.

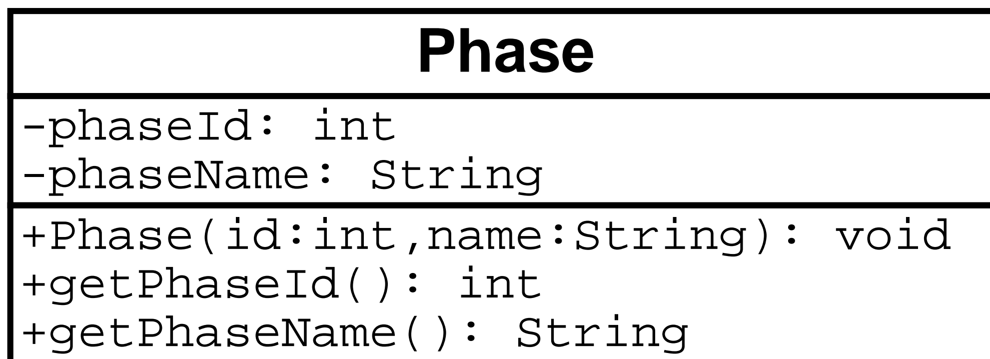

Figure 2.11: Class Phase

Attributes

- **Private:**

  phaseId:int Unique identifier for each phase.

  phaseName:String Stores the name of the phase.

Methods

- **Public:**

  Phase(...)

  Arguments:

  - **id:String** Id of the phase.
  - **name:String** Name of the phase.

  Returns (type **void**): Sets phaseId and phaseName to the arguments passed.

  ---

  getPhaseId(...)

  Returns (type **int**): Returns phaseId.

  ---

  getPhaseName(...)

  Returns (type **String**): Returns phaseName.

2.4.10 PPS

This class stores the information of the estimated, actual and to date values of time and defects in phases. This is an RMI serializable class , hence its definition will exist on both client and server.

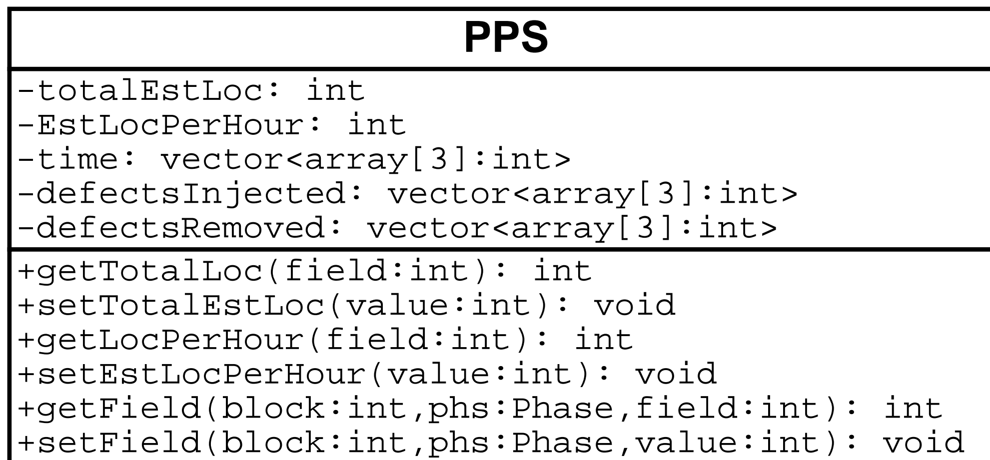

Figure 2.12: Class PPS

Attributes

- **Private:**

  totalEstLoc:int Stores the total lines of code for a specific project.

  EstLocPerHour:int Stores the lines of code per hour for a specific project.

  time:vector(array[3]:int) Stores the estimated, actual and to date times for all the phases of the project.

  defectsInjected:vector(array[3]:int) Stores the estimated, actual and to date defects injected for each the phases of the project.

  defectsRemoved:vector(array[3]:int) Stores the estimated, actual and to date defects removed for each the phases of the project.

Methods

- **Public:**

  ---

  getTotalLoc(...)

  Arguments:

  - **field:int** Possible values for field are 1 for estimated, 2 for actual and 3 for to date.

  Returns (type **int**): Returns totalEstLoc if field is 1, computes the actual total lines of code for the specified project and user if field is 2 and computes the to date total lines of code for all the projects till the current project for the specified user if field is 3.

---

setTotalEstLoc(...)

Arguments:

- **value:int** Total estimated lines of code for the project.

Returns (type **void**): Sets totalEstLoc to the argument passed.

---

getLocPerHour(...)

Arguments:

- **field:int** Possible values for field are 1 for estimated, 2 for actual and 3 for to date.

Returns (type **int**): Returns EstLocPerHour if field is 1, computes the actual lines of code per hour for the specified project and user if field is 2 and computes the to date lines of code per hour for all the projects till the current project for the specified user if field is 3.

---

setEstLocPerHour(...)

Arguments:

- **value:int** Estimated lines of code per hour for the project.

Returns (type **void**): Sets EstLocPerHour to the argument passed.

---

getField(...)

Arguments:

- **block:int** Determines the vector. 1 for Possible values are 1 for time, 2 for defectsInjected and 3 for defectsRemoved.
- **phs:Phase** Phase of the project.
- **field:int** Possible values for field are 1 for estimated, 2 for actual and 3 for to date.

Returns (type **int**): Computes and returns the specified value depending on the arguments.

---

setField(...)

Arguments:

- **block:int** Determines the vector. 1 for Possible values are 1 for time, 2 for defectsInjected and 3 for defectsRemoved.
- **phs:Phase** Phase of the project.
- **value:int** Value of the specified field determined by block and phase.

Returns (type **void**): Sets the value of the first index of the array of the specified block and phase to the argument value passed. The first index of the array is an estimated value, the scond is actual and the third is the to date value.

2.4.11 PP

This class is used by the user to store problem and proposal that is linked to a particular PIP form. This is an RMI serializable class , hence its definition will exist on both client and server.

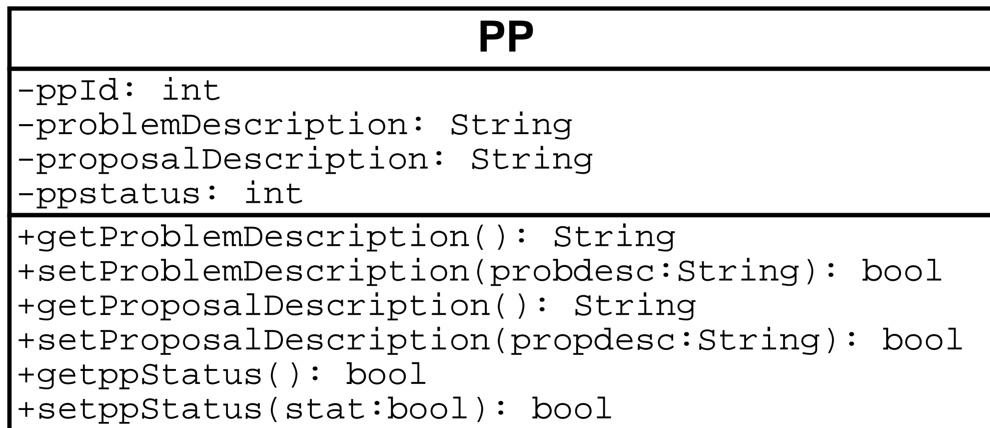

Figure 2.13: Class PP

Attributes

- **Private:**

  ppId:int Unique identifier for each PP.

  problemDescription:String Stores the description of the problem.

  proposalDescriptionType: String Stores the description of the proposal.

  ppStatus:int Stores the status indicating the operation to be performed. False is for adding the problem-proposal to the database and true is for updating the PP (problem-proposal) in the database.

Methods

- **Public:**

  ---

  getProblemDescription(...)

  Returns (type **String**): Returns problemDescription.

  ---

  setProblemDescription(...)

  Arguments:

  - **probdesc:String** Description of the problem.

  Returns (type **bool**): Sets problemDescription to the String argument passed to it and returns true if successful and false if failed.

  ---

  getProposalDescription(...)

Returns (type **String**): Returns proposalDescription.

---

setProposalDescription(...)

Arguments:

- **propdesc:String** Description of the proposal.

Returns (type **bool**): Sets proposalDescription to the String argument passed to it and returns true if successful and false if failed.

---

getppStatus(...)

Returns (type **bool**): Returns ppStatus.

---

setppStatus(...)

Arguments:

- **stat:bool** Status of the operation whether being added or edited.

Returns (type **bool**): Sets ppStatus to the bool argument and returns true if successful and false if failed.

2.4.12 PIP

This class is a container of all the problem-proposals for a specific project and user. This is an RMI serializable class , hence its definition will exist on both client and server.

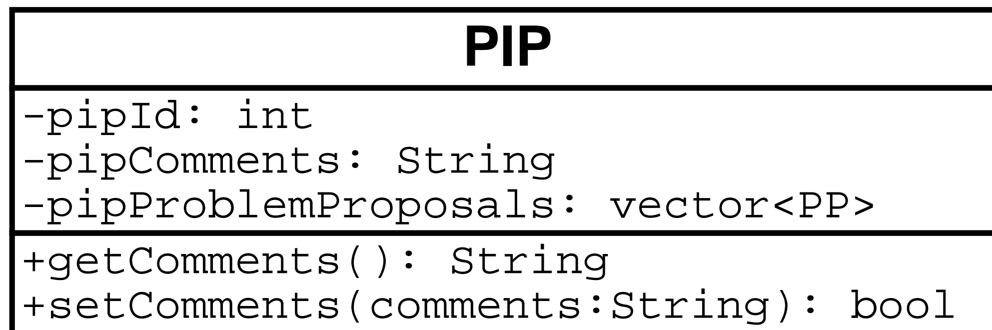

Figure 2.14: Class PIP

Attributes

- **Private:**

  pipId:int Stores unique Id for each PIP.

  pipComments:String Stores comments and notes for the PIP.

  pipProblemProposals:vector(ProblemProposal) Stores the vector of all the PPs for the specific user and project.

Methods

- **Public:**

  ---

  getComments(...)

  Returns (type **String**): Returns pipComments.

  ---

  setComments(...)

  Arguments:

  – **comments:String** Comments and notes for the PIP.

  Returns (type **bool**): Sets pipComments to the String argument passed to it and returns true if successful and false if failed.

2.4.13 Defect

This class is used to store the defects added by the user. This is an RMI serializable class, hence the class resides both on the Client and the Server.

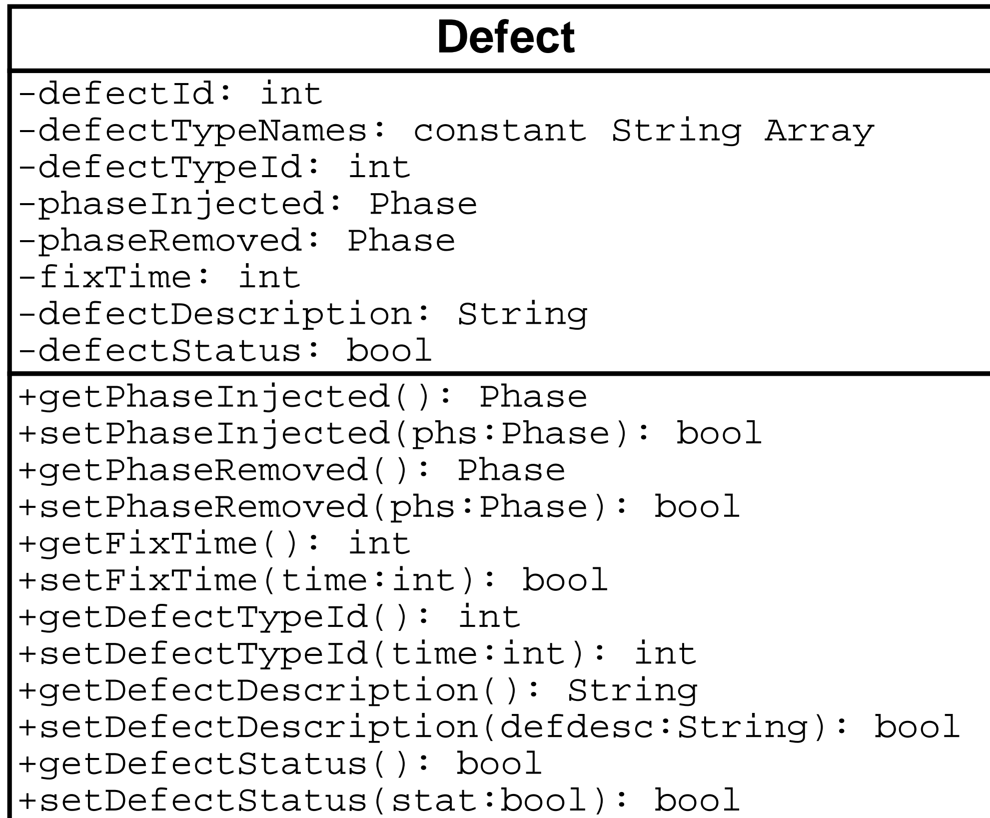

Figure 2.15: Class Defect

Attributes

- **Private:**

  defectId:int Unique identifier for each defect.

  defectTypeNames:constant String Array Stores the string names for defect types.

  defectTypeId:int Integer code for the defect type and also acts as an index for defectTypes string array to get the name of the defect type.

  phaseInjected:Phase Stores the phase in which the defect was injected.

  phaseRemoved:Phase Stores the phase in which the phase was removed.

  fixTime:int Stores the time taken to fix the defect.

  defectDescription:String Stores the description of the defect.

  defectStatus:bool Stores the status indicating the operation to be performed. False is for adding the defect to the database and true is for updating the defect in the database.

Methods

- **Public:**

---

getPhaseInjected(...)

Returns (type **phs:Phase**): Returns phaseInjected.

---

setPhaseInjected(...)

Arguments:

– **phs:Phase** The Phase in which the defect was injected.

Returns (type **bool**): Sets phaseInjected to the Phase passed as argument and returns true if successful and false if failed.

---

getPhaseRemoved(...)

Returns (type **Phase**): Returns the phaseRemoved.

---

setPhaseRemoved(...)

Arguments:

– **phs:Phase** Phase in which the defect was removed.

Returns (type **bool**): Sets phaseRemoved to the Phase passed as argument and returns true if successful and false if failed.

---

getFixTime(...)

Returns (type **int**): Returns fixTime.

---

setFixTime(...)

Arguments:

– **time:int** The time taken to fix the defect.

Returns (type **bool**): Sets fixTime to the int passed as argument and returns true if successful and false if failed.

---

getDefectTypeId(...)

Returns (type **int**): Returns defectTypeId.

---

setDefectTypeId(...)

Arguments:

– **time:int** The id of the defect type.

Returns (type **bool**): Sets defectTypeID to the int passed as argument and returns true if successful and false if failed.

---

getDefectDescription(...)

Returns (type **String**): Returns defectDescription.

---

setDefectDescription(...)

Arguments:

- **defdesc:String** Description of the defect.

Returns (type **bool**): Sets defectDescription to the string passed as argument and returns true if successful and false if failed.

---

getDefectStatus(...)

Returns (type **bool**): Returns defectStatus.

---

setDefectStatus(...)

Arguments:

- **stat:bool** Status of the operation whether being added or edited.

Returns (type **bool**): Sets defectStatus to the bool passed as argument and returns true if successful and false if failed.

2.5 Sequence Diagram

This particular sequence diagram is based on the user requesting a PIP (Process Improvement Proposal) form to edit/modify. There is corresponding serializable class **PIP**, so we can get whole PIP from the **Server** to the **Client**.

The process begins with the user clicking on the forms menu, in the main window of our program and then clicking on the PIP menu item. This will trigger the function, PIPform(), which will open a window of type **PIPform**. This window will immediately tell the logical client to request the **PIP** form from the server for specific user, through the function getPIP(). The logical **Client** (a separate entity from the UI) will then call the remote method getPIP(User) from **curProj** object of **Project**, which is an instance of the class on the server, but not on the client. The result of this will be the **Project** class making a query to the **DB** by calling the function runQuery("Select"). The **DB** will then respond to this query with appropriate information - serializable object of class **PIP**.

After the user has completed modification of this PIP form, he/she has two options. Option one, he can close the PIP form window without saving any changes by clicking on the X button on the right top corner. Option two, which is modeled in the lower portion of our diagram, is to save the changes he made by clicking on the Done button.

The save process begins when the user clicks the Done button. This action will trigger a function call, UpdatePIP(PIPform). This is a function call to the logical **Client**, which will then make a request to save data to the **Server** through a function called UpdatePIP(user,PIP). Since **PIP** is a serializable class, we will use RMI to transfer the class in its entirety to the server. Once the class has arrived (assuming no transfer errors occurred), the **Project** class on the server will initiate a save request to the database **DB** through the function, runQuery("Update").

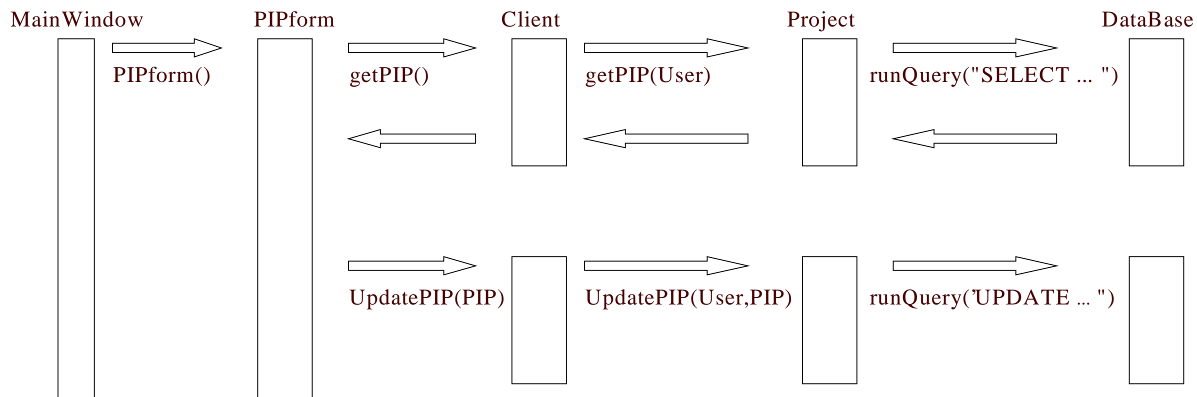

Figure 2.16: Sequence Diagram (PIPform use-case).

2.6 Data Flow Diagram

This data flow diagram begins with the user at the Login Window. Then the user enters his username and password. Once the user clicks on the Login button, the username and password is sent to the client through the Login() method in the UI. This method then calls AuthenticateUser(), a remote method on the server through the logical client. In other words, the logical client will invoke a remote method on the server, which will query the database for that particular userName.

Database will return a hash value of the password and the server compares this hash value to the hash value it computes for the password entered by the user, thus verifying the user. Also database returns parameter isLoggedIn which will state that user is already logged so he can't login now, so authentication fails. If the user is authenticated, the server updates the database that the user is logged in. The AuthenticateUser method returns an instance of serializable **User**

class with appropriate attributes set both for the valid and invalid user. This information is then stored on the logical client. Depending on values of the attributes(isLoggedIn) in the User class, client will provide UI with message if there is a problem with login or directly gives him access to the different main window menu items.

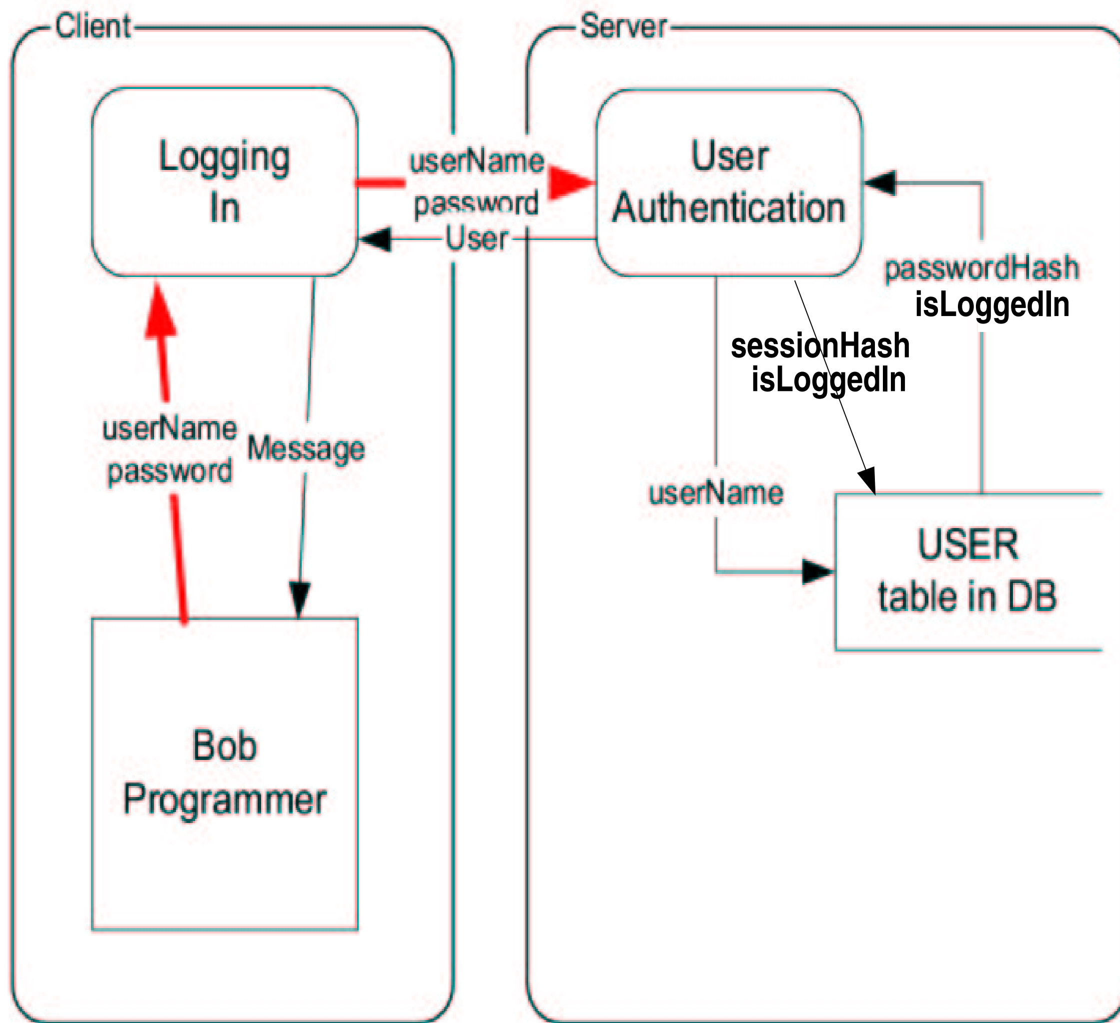

Figure 2.17: Data Flow Diagram for Login use case.

References