Yahoo! Project Code Name: MirrorWeave

Charles Durham, Immanuel Alexander, Leo Tanady, Andrew Prasetyo Jo, AJ Arora cs307b@cs.purdue.edu

Introduction

Document Scope

The purpose of this document is to provide a design overview of features of the Yahoo data quality widget project. Its coverage ranges from the high level deployment view to descriptions of the functions necessary to be describing at this point in the design process. This is however not a complete description of the low level details as this was not our intention, we instead feature the most important functions as to not distract the reader from the main design aspects of this product.

This document first features a quintet of diagrams which are all very related to each other but vary in depth. These diagrams start with the high-level deployment view and each proceeding diagram is a zoomed in version of the previous. This quintet of diagrams are the high-level deployment view, the high-level component view, the detailed component view, the logical view of the "Widget GUI" component, and the logical view of the "Data Access" component. The next diagram is a UI flow diagram followed by the process view and finally the "Use Case" diagram with some potentially important action/response use cases.

Intended audience

The intended audience for this document is the Yahoo SDS Data Quality team, including any sub team(s) directly related to the Yahoo SDS Data Quality team.

Project Overview

The main goal of the project is in building a graphical user interface that interacts with the data quality database to obtain data and provide graphical representation of the data. In addition, the project will allow the user to provide feedback about a bug or anomaly. Access to the data quality database is not restricted since it is in the Yahoo internal network, so there is not much concern about how to access the database and issues in security. For additional information see the previous requirements document.

High-Level Design

Goals and Guidelines

The main goal of the design is to have a Yahoo widget with a streamlined GUI design. We will feature two tabs, with the first tab containing property views and the second tab containing issues regarding the individual properties. We have two tabs in order to properly separate the different requirements of our

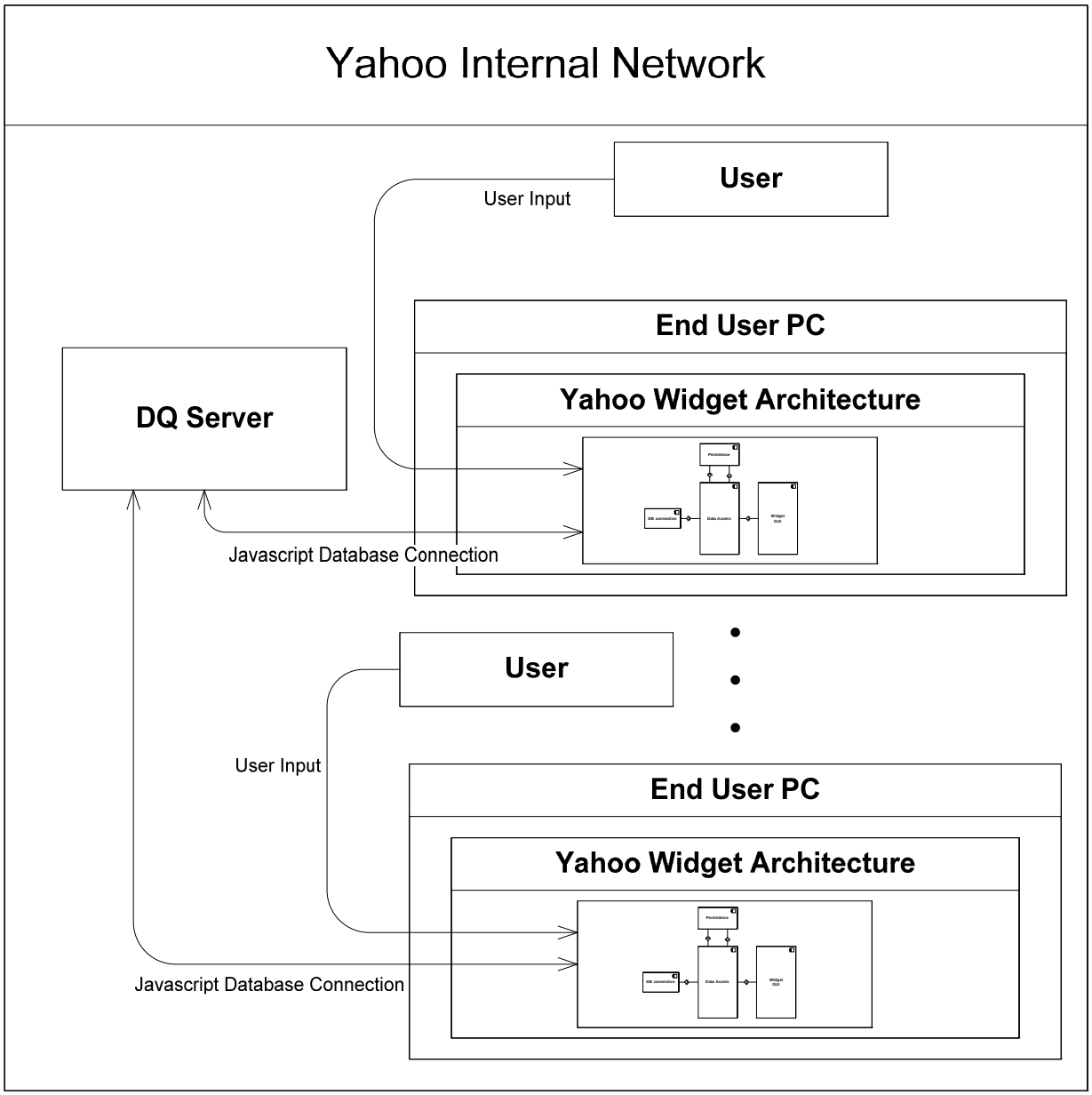
project in order to increase the user's productivity in his/her interaction with our product. The property view will be meant for analysis of data and the data issues view will be fulfilling the user's desire to inspect and manipulate reported issues in regards to their requested properties and regions.

Architectural Strategies

One of the design decisions includes providing a user friendly interface by using tabs to separate different components, data and issues. The first thing the widget does is to obtain the user login information and then connect to the database for authorization. We decided to provide a separate functionality to access the Data Quality Database and also use SQL Lite that already exists in the Yahoo widget architecture to cache data obtained from the Data Quality Database. Keeping with the mentality of the design document's simplicity, we decided to leave out details regarding SQL Lite at the moment as this brings up further implementation issues which will be addressed later. As such, the complicated part of the design is displaying the obtained data to the user interface. We feel that we have solved these complicated issues and have provided the proper amount of information to any one tab to minimize desktop space used while providing enough information in a tab to remain useful given the data's context.

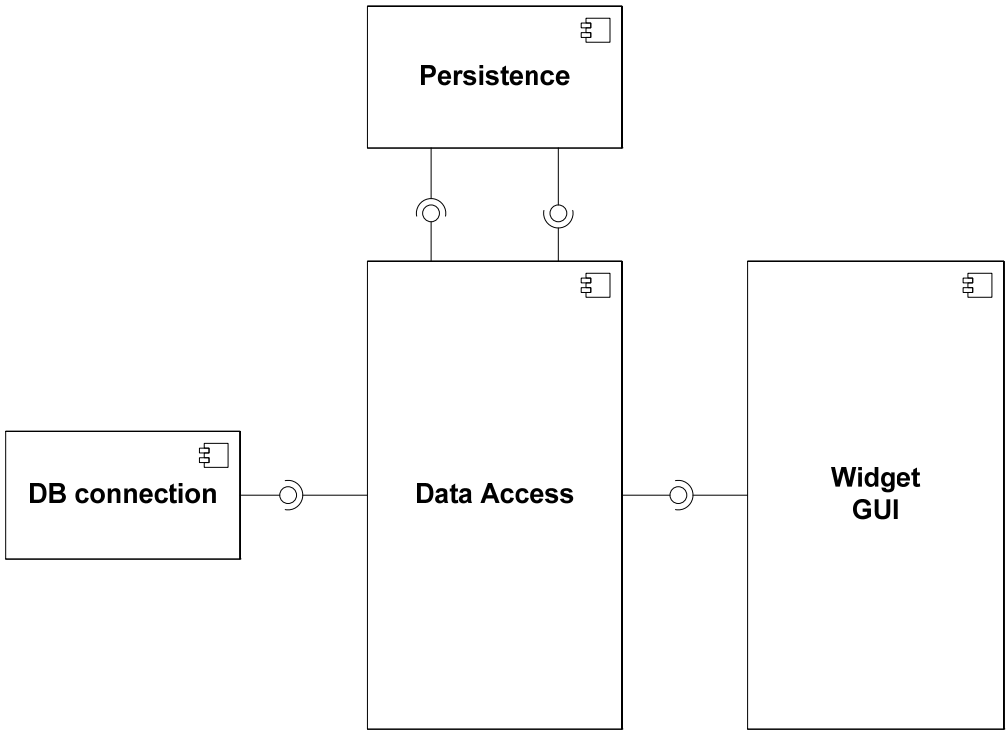
To reiterate from above, from an architectural standpoint, the information worth putting in the diagrams concerns two things, manipulating the database and providing relatively complex GUI functionality. Other smaller issues will be addressed as the corresponding diagram makes it apparent.

High-Level Deployment View



This diagram is pretty straightforward. We feature the diagram sitting inside the yahoo network with each end user pc containing the necessary architecture to run the widget that we are developing. Each widget that is running will have a connection to the DQ server through the JavaScript COM model and there can potentially be many connections to the database as there will be multiple deployments of our software.

High-Level Component View



DB connection

- Holds the connection to the DQdb

Persistence

- Persists all data read from DQdb. Data is read from and written to this component

Data Access

- Accesses the DQdb using the DB connection and formats data for use in the Widget GUI

Widget GUI

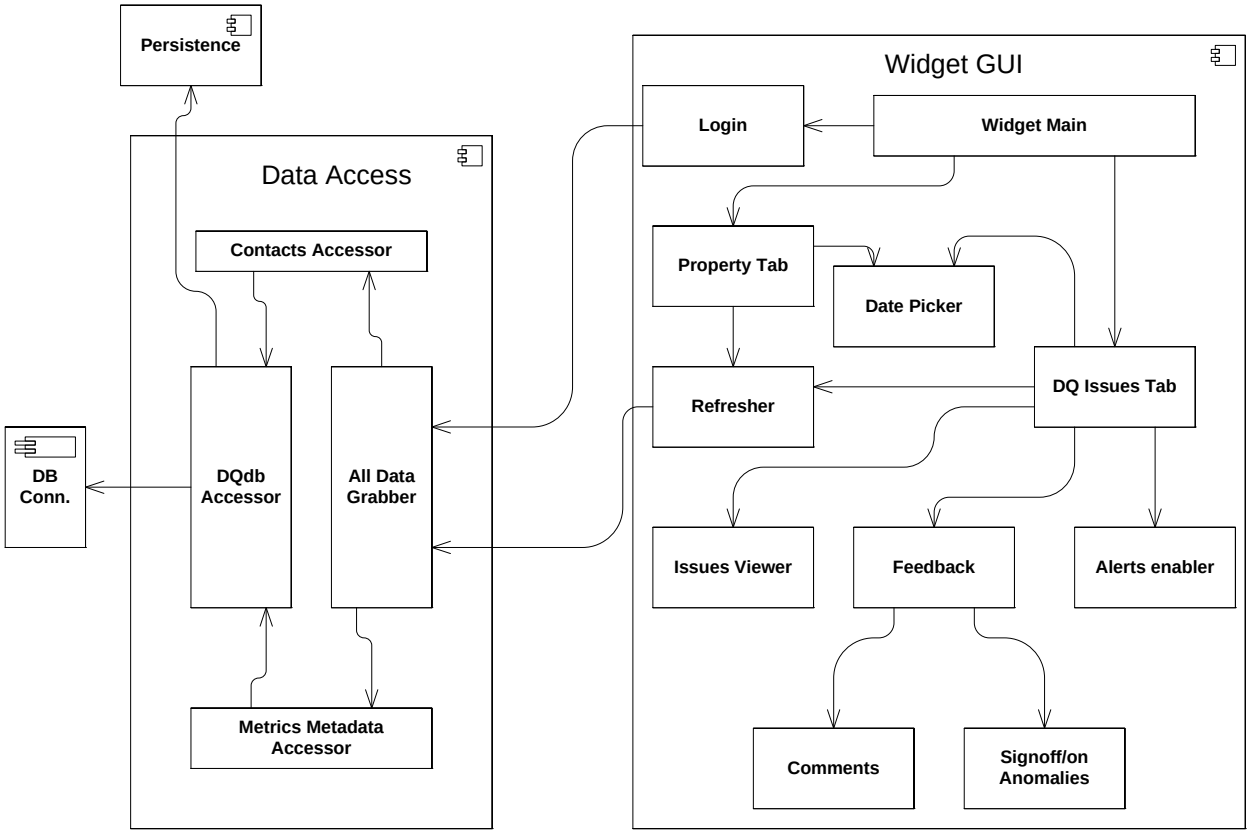
- Handles the appearance of the data and is responsible for the user manipulation of the data as well

We chose this decomposition from the perspective of how the widget should obtain information and how it should display that information. As discussed in the Architectural Strategies section, we felt that there were two main issues at hand; dealing with data and providing the user with a complex GUI. In addition, we felt that a separate component to handle persistence and the DB connection would help streamline the development process as they are simple tasks that are easily broken away from the complexities of the Data Access component and the Widget GUI component.

Detailed Design

We now move on to the more detailed aspects of our diagrams which continue to follow the quintet of diagrams discussed above in the Document Scope section. In addition, in this section we will provide our process view, UI flow diagram, use case diagram, and individual use cases.

Detailed Component View

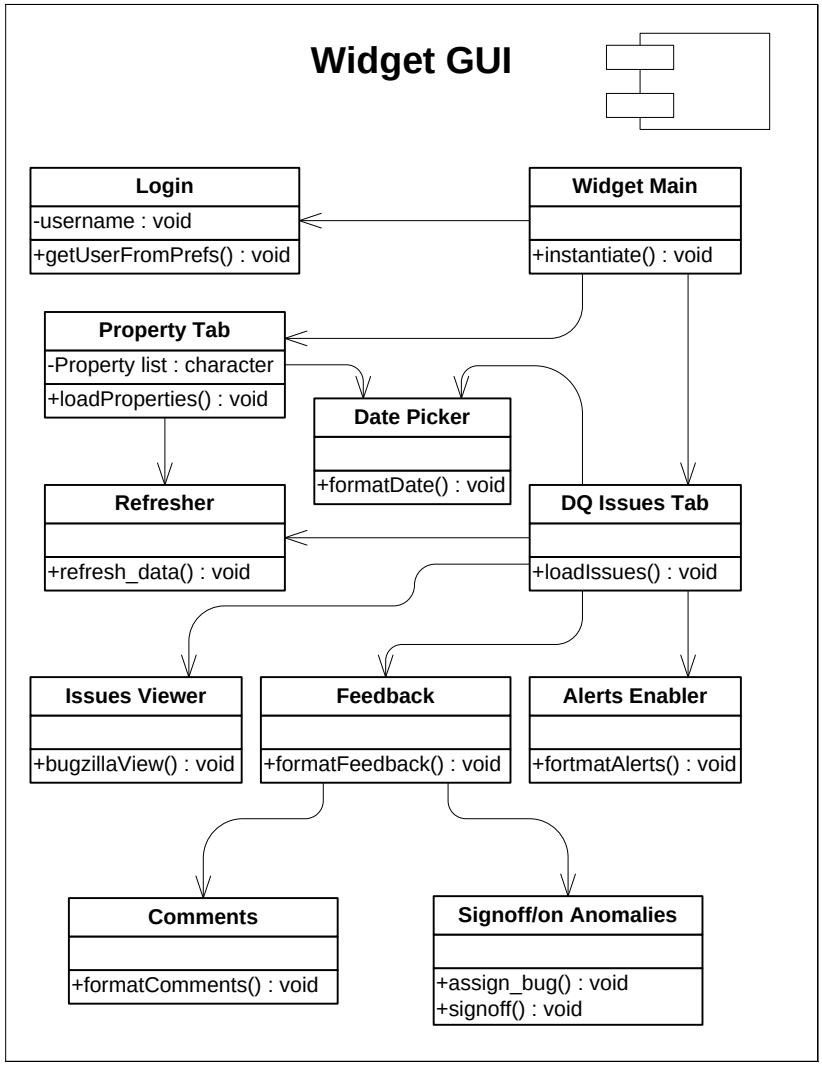


This diagram is obviously a look underneath the more complex Data Access component and the Widget GUI component. Starting with the Data Access component, we felt it was necessary to have a generic data grabber that can be interfaced with from the Widget GUI to provide a smoother connection to the Data Access component. This role is fulfilled by the “All Data Grabber”. This object when interfaced with decides which accessor to use, either the “Contacts Accessor” or the “Metrics Metadata Accessor”. The “Contacts Accessor” and “Metrics Metadata Accessor” interface with the DQdb Accessor to get data from the server. They then format the data appropriately and pass it back to the “All Data Grabber”.

The whole widget starts with “Widget Main” inside the Widget GUI component. The user then logs in and the “Login” class verifies his/her username by a request through the “All Data Grabber”. The user is then presented with two tabs, the “Property Tab” and the “DQ Issues Tab”. The “Property Tab” is pretty straightforward and features a “Date Picker” to determine the date with which the data should be represented from. The property tab is then populated by a call to the “Refresher” class which in turn calls the “All Data Grabber” functions. Similarly, the “DQ Issues Tab” uses the “Date Picker” to determine the date with which the corresponding issues are displayed. The “DQ Issues Tab” also relies

on the refresher to populate its lists. The “DQ Issues Tab” can view issues, can enable/disable alerts, and can provide feedback by either adding comments or by signing off on anomalies to upgrade them to bugs or to change bugs back down to anomalies.

Widget GUI Logical View



Corresponding to the description above for the Detailed Component View, we will provide descriptions of the diagrams classes above in list form.

Widget Main

- Has instantiate function which initializes the tabs in the GUI and loads the Login screen if a previous user is not cached on the Widget. Instantiates the backbone of the GUI with which everything will sit on top of.

Login

- Contains the username variable to store while the widget is running. Very first thing login does is check to see if a previous username is store in preferences. As above, but not shown, login negotiates directly with the Data Access component.

Property Tab

- Contains the list of properties owned by the user. Originally calls load_properties to set list of properties inside its class.

DQ Issues Tab

- Loads issues relevant to the user.

Date Picker

- Formats the date for use of the Property Tab and the DQ Issues Tab so the user can request that his/her data correspond to a certain date.

Refresher

- Refreshes the current data. Will check GUI to see which tab user is on in order to refresh appropriate data.

Issues Viewer

- Initializes the pane which will contain the issues formatted for usability and navigation

Alerts Enabler

- Formats alerts based on input to function and allows users to hide alerts from the Issues Viewer

Feedback

- Contains function to format any type of feedback left on a selected bug or anomaly.

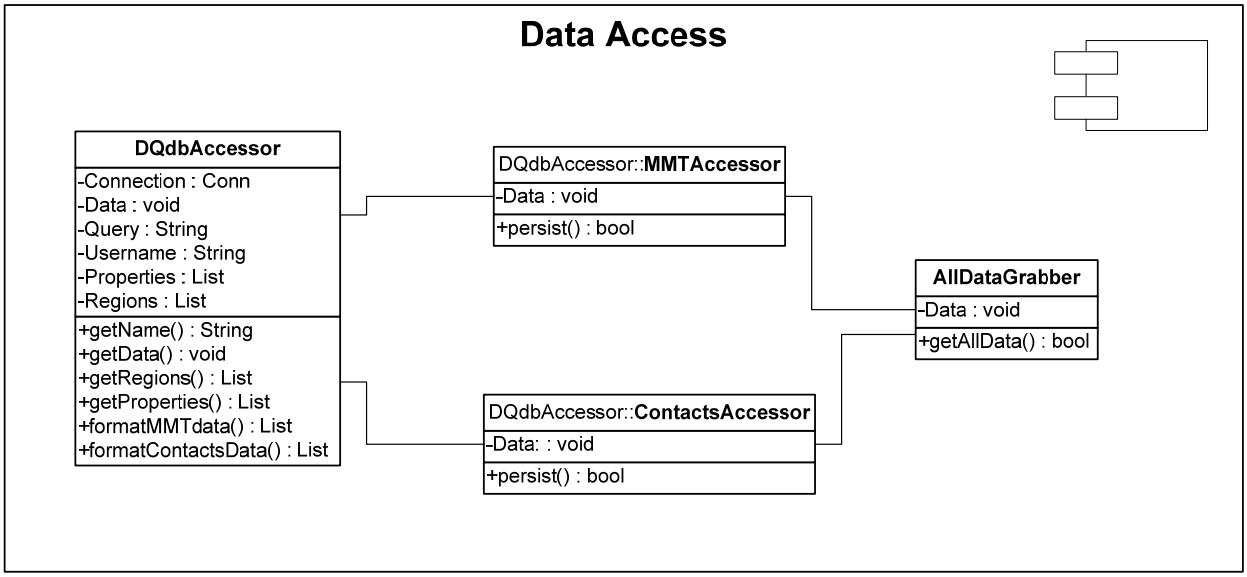
Comments

- Formats a comment for placement in the database

Signoff/on Anomalies

- Contains function to sign off on a bug and a function to assign a bug number to a newly upgraded bug.

Data Access Logical View



Following the spirit of the last Logical view, we are going to proceed to list off the functions of each class.

AllDataGrabber

- Has function that gets all data. Depending on arguments passed to function, will request data from either the MMTaccessor or the Contacts Accessor. When data is received, the data is filled in the data field. The data is then tentatively fed formatted into a corresponding SQL Lite table and the function called returns true to assure that the data was written to the table

MMTAccessor

- Has a function that persists the database to update the list of Metadata Table entries. When persist function returns true, implies that the data was successfully retrieved and the Data variable now contains formatted data for the ALLDataGrabber to add to the SQL Lite table.

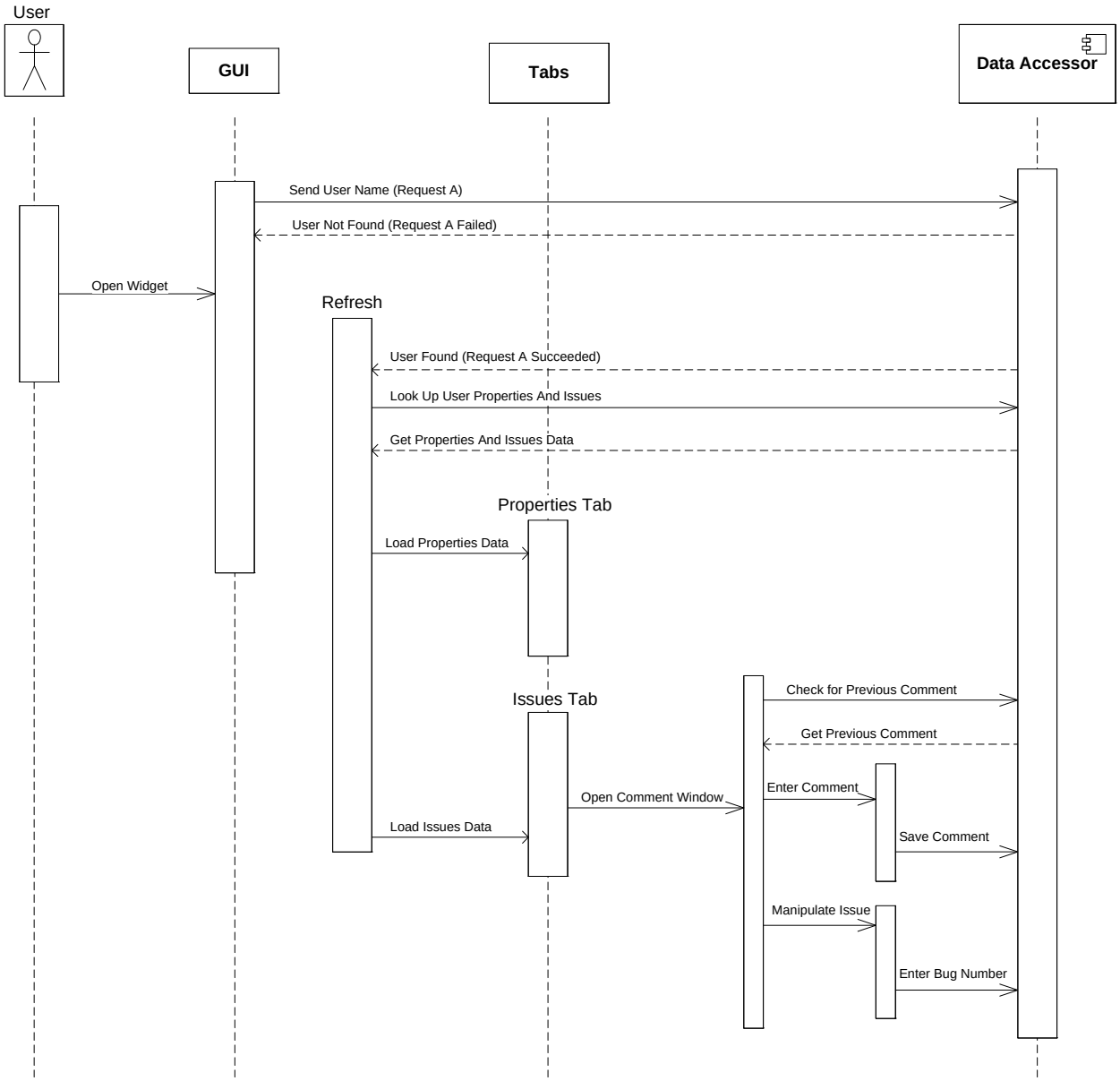
ContactsAccessor

- Contains the same functionality as MMTAccessor but provides different methods of formatting retrieved data.

DQdbAccessor

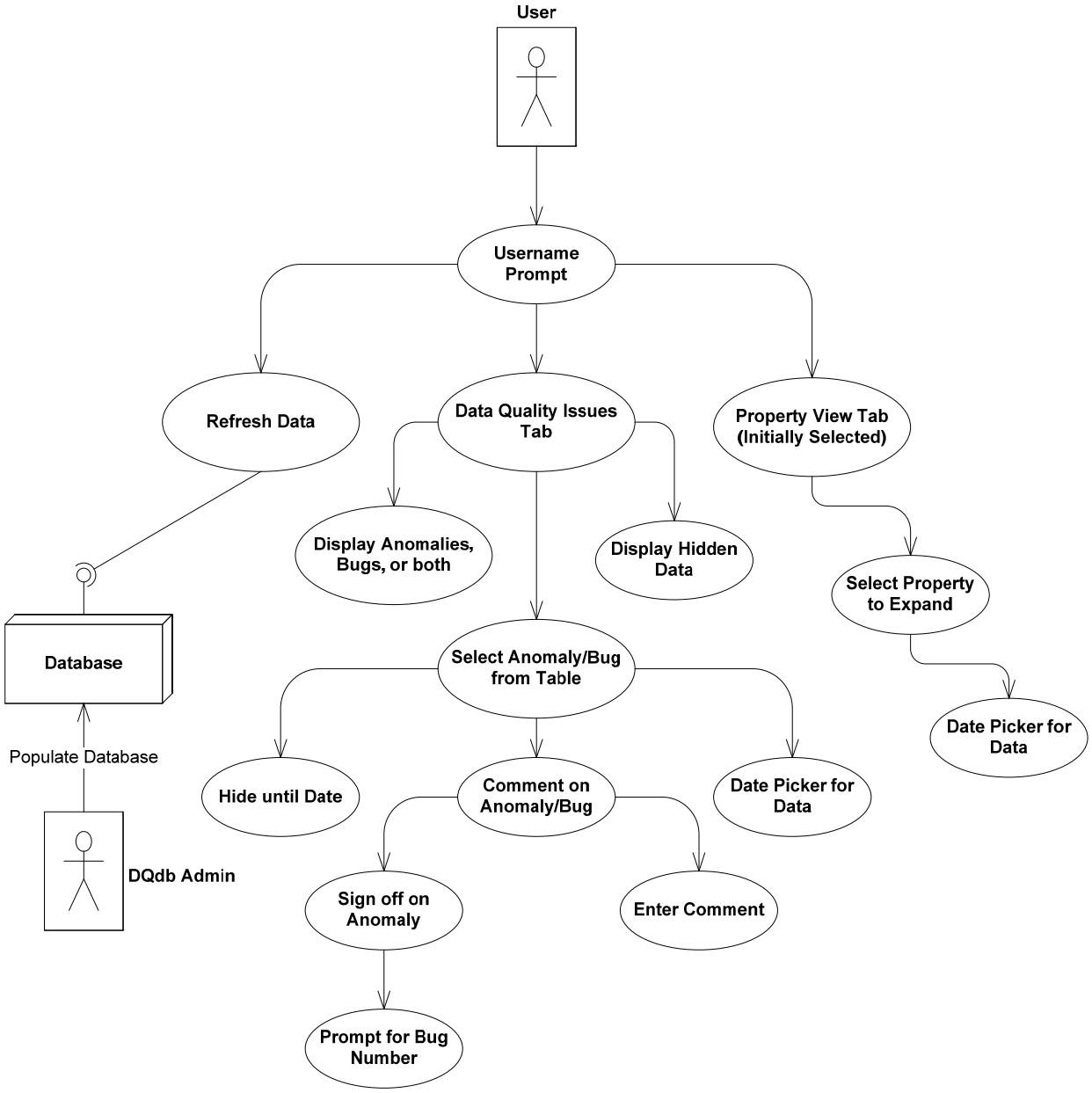
- Maintains the connection and a vast array of functions and fields to accommodate MMTAccessor and ContactsAccessor. Serves as a butler for the two accessors.

Process View



This diagram illustrates an interpretation of the layers of our product. The first layer is the User, which at the highest level opens the widget. At the GUI level, the user is able to verify their username if it isn't stored from previous sessions. The GUI sends a request to the Data Accessor component to find the user name. If the user name is not found then the user remains at that level, if the username is found, then the user can go to the GUI level of refreshing data to be displayed. The user is then able to navigate either the properties tab or the issues tab. Each time a refresh happens, the user properties and issues are again loaded to populate the properties and issues tabs. While in the Issues tab, the user may open a comment window and make comments or manipulate bugs. The diagram provides the necessary illustrations of these phases in regards to the database accesses.

Use Case Diagram



We decided to feature the Use Case Diagram before the UI Flow Model in order to get the reader acquainted with the simple ideas of the GUI design. The only item of note in this diagram is that the Database is actively being populated by another agent. In order to explain this diagram, we provided some use cases relevant to the above use case diagram.

Use Cases

Property View Use Case

Actor Actions	*System Responses*
1. Click on 'Property View' tab	2. If not already displayed, 'Property View' is displayed
3. Click on a given property to expand	4. Property is expanded to display usage statistics
5. Use date picker to select usage statistics date	6. Usage statistic data is updated to contain usage statistic data for that date

Data Quality Issues Use Case view data issue

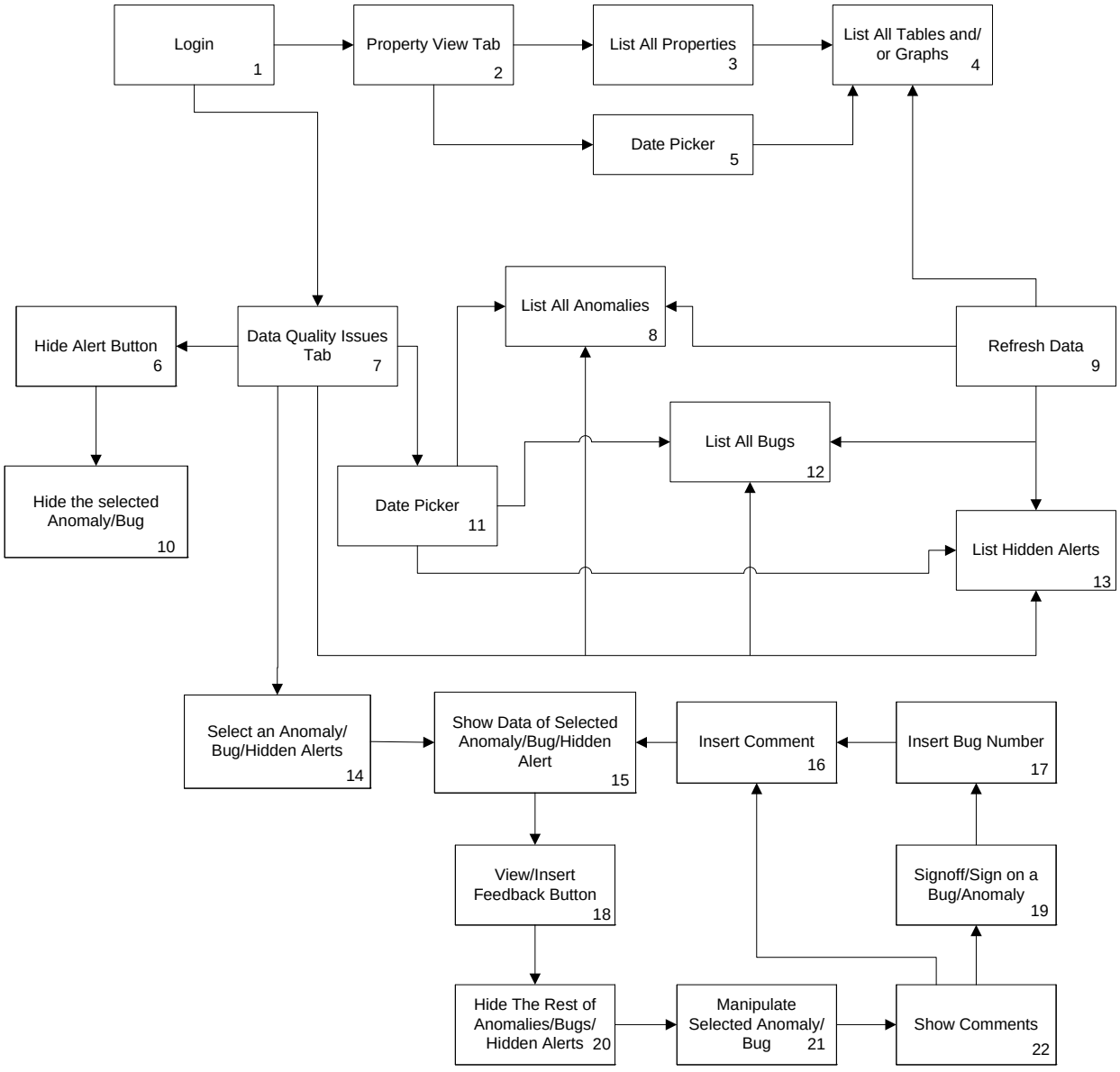
Actor Actions	*System Responses*
1. Click on 'Data Quality Issues' tab	2. If not already displayed, Data Quality Issues are displayed
3. Selects radio button to display Anomalies, Bugs, or both	4. Issues table is updated to contain only data types that are specified
5. Hits 'Display Hidden Data' toggle button	6.a. If toggle button is toggled on, then hidden data is displayed in table. Otherwise hidden data is not displayed in table
	6.b. Data is refreshed
1. Select anomaly or bug from table	2. Issue report is displayed on selected issue. Data is displayed below table. Report contains data pertaining to the anomaly or bug selected

Data Quality Issues Use Case comment on issue (starting from view data issue)

Actor Actions	*System Responses*
1. User selects 'Comment' button located in Issue report	2. Issue report section replaces Issues table. Comment section appears underneath issue report section
3. User enters comments	4. Comments are entered into database
5. If issue is an anomaly, user hits 'Signs off on Anomaly' button	6. User is prompted with the bug number
7. User enters bug number	8. Anomaly is updated to bug status

It is recommended for the reader to read the use cases while following the use case diagram in order to get more acquainted with the different phases in our GUI style. We now present the more complex UI Flow Model.

User Interface Flow Model



This model should only be a more refined model of the user interface as opposed to the diagram and use cases presented above. The main thing of note is that we included numbering of each state in order to more easily reference the states for outlining future plans of our UI layout. We also present this flow model with the stipulation that a user name was found by the database during the login phase. If a username was not found, it would be relatively impossible to move to further states in the model without requesting every piece of data contained in the Metrics Metadata Table.

Conclusion

We have presented our diagrams in a way we saw practical for the understanding of our products design. There are many more details to iron out, but these are the details that our team found relevant to the design implications of our earlier requirements and the high level goals set at the beginning of this document. As such, many of the following documentation will certainly reference this document for the implementation constraints that were committed to in this document.