Canadian Epidemic Tracker

E-Health Canada – COMP 3004 Deliverable 2

John Vanden Heuvel

Shane Panke

Leigh Pascoe

Sebastian Schneider

Table of Contents

1. Introduction

1.1 Purpose of system

The Canadian Epidemic Tracker is a system designed to streamline the basic tracking of various disease cases in Canada in order to get a better view of the general impact of various disease outbreaks. In addition to creating a simple method to track and view diseases it will also provide tracking and inventory management for the supplies necessary to diagnose and treat them.

Secondarily, the system will provide a basic reporting structure to allow advanced users to create visual charts and graphs depicting various statistics that might be of interest.

1.2 Design Goals

1.2.1 Usability

1. The UI will be designed using a standard GUI paradigm creating an environment that should be familiar to the users based on their experience with other window based systems and their conventions (NFR 2.3.1.1, 2.3.5.2).
    a. Map interface will use Marble for QT4 due to its ease of use and understandability for users
    b. Map interface is to use clear indicators for different datatypes

being concurrently displayed

1.2.2 Stability

1. The server and the client are expected to be relatively independent of each other in maintenance of data allowing for more stability in the event of a client crash (NFR 2.3.2.1)
2. The server is to use a database for file storage allowing for easy persistent data maintenance (NFR 2.3.2.2, 2.3.2.4)
3. Error checking and exception handling is to be built-into the system to prevent improper operation due to improper input by user (NFR 2.3.2.3)

1.2.3 Modularity

1. Communication between the client and server will use sockets, this allows for scalability should the requirements regarding the number of simultaneous connections be changed in the future (NFR 2.3.3.1, 2.3.6.1)
2. Message protocol and data store are to be easily expandable to allow for future data types not currently being tracked (i.e. emergency vehicles in use)
3. Tracking capabilities will be built such that the addition of other countries should be possible.

1.2.4 Concurrency

1. Server will be built to handle a high amount of simultaneous incoming connections via threading, this should contribute to stability as a faulty connection would kill a thread at most leaving the server

1.3 Overview of document

The system design document decomposes the C.E.T. system. The first section discusses the subsystem overview.

We first examine the subsystem decomposition, detailing the links between the subsystems and the objects they contain. Next will be the hardware and software mapping where the links between the individual software components relate to the physical systems they reside. The strategy for persistant data management will also be outlined, explaining the reasoning behind the design decision.

After decomposition and hardware/software mapping is the global control flow. Describing the design choices with respect to decisions regarding event-driven and thread based design in the various components. Along with global design choices is the access control list, describing the interfaces offered by objects and outlining which user types have access to which functions.

We follow this with the boundary conditions, which describe the use-cases

involving exceptional circumstances (i.e. initial server startup). These primarily consist of conditions external to the requirements analysis phase or were missed at that stage.

The last three sections deal with design patterns, subsystem services and message protocol. The design pattern section describes the components of the system that utilize common coding strategies and the reason for their use. Subsystem services entails the descriptions of the services offered by the various subsystems and how they form the inter-relationship between the subsystems. Finally the Message protocol lays out the format and translation method the messages between the client and the server components

2. System Design

2.1 Overview

This section will include the initial decomposition into subsystems, which will be refined to meet all design goals. Design strategies will be established for hardware and software mapping, data management, access control, control flow and boundary conditions. Subsystem services will be shown and will include subsystem interface operations. Lastly, Message Protocol will provide insight on how the client will interact with the server.

Established design strategies will be further developed as follows: Hardware and software mapping will contain off-the-self and legacy software components, hardware configuration, and inter-node communications strategy. Data management will identify persistent data, storage location, and access mechanisms. Access control will handle authentication, authorization while maintaining confidentiality. Boundary conditions will initialize and shutdown the system as well as handling all exception cases.

2.2 Subsystem Decomposition

Figures 1 and 2 are two different depictions of the subsystem breakdowns. Figure 1 shows the inter-relations between the subsystems. Figure 2 shows the individual objects that make up each of the individual subsystem. A description of the subsystems and their objects follows the diagrams

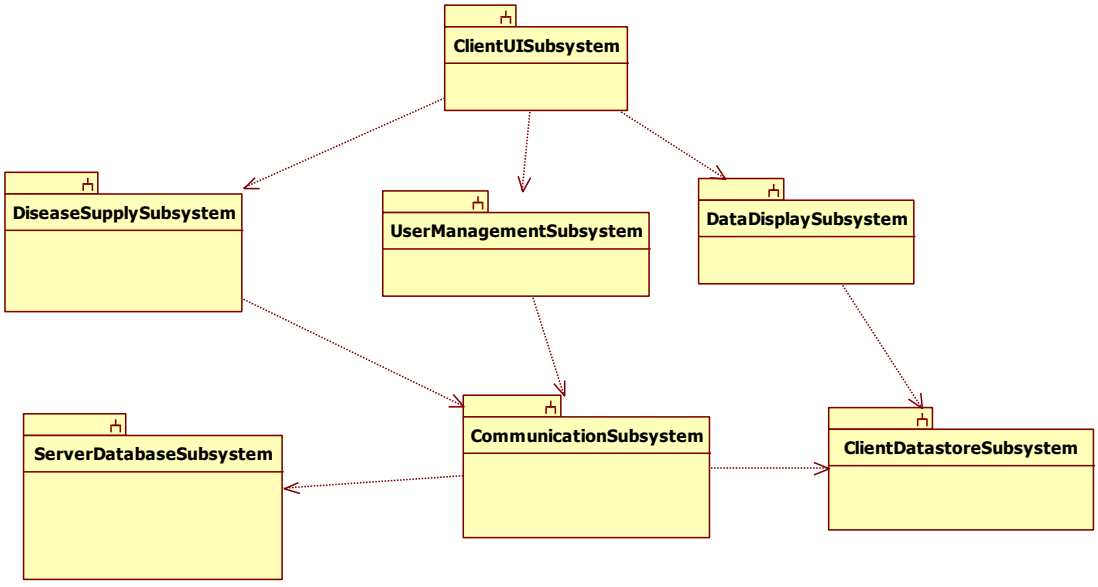


Figure 1 Subsystem Inter-relations

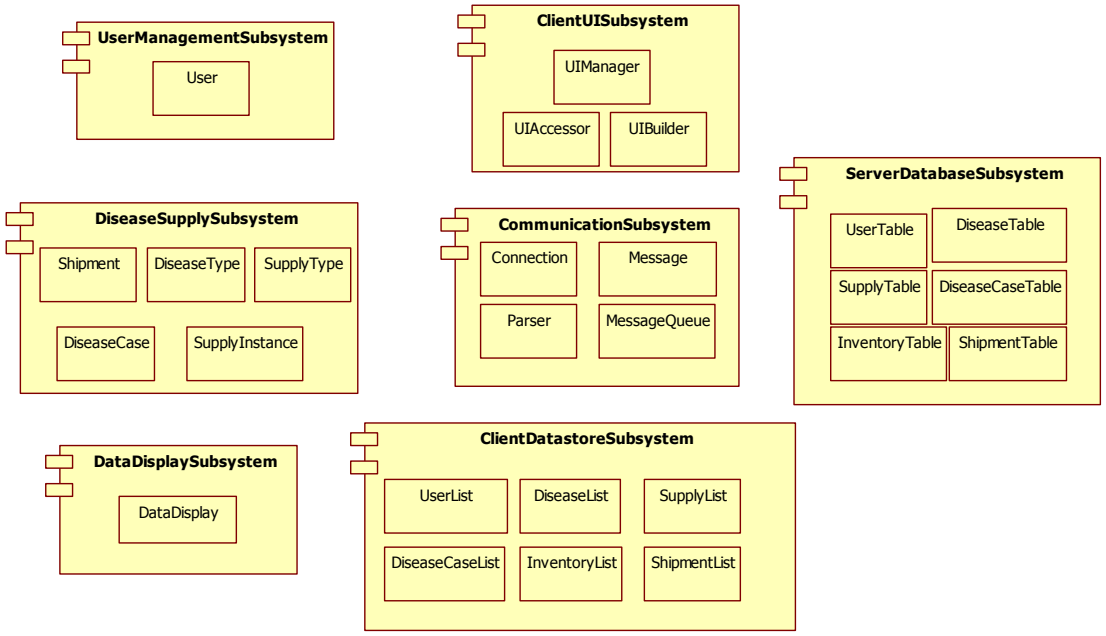


Figure 2 Detailed Subsystem breakdown indicating member objects

Item	Description
ClientUISubsystem	The ClientUISubsystem is responsible for displaying all valid options to the user.
- *UIAccessor*	The UIAccessor represents the interface for the subsystems.
- *UIManager*	The UIManager represents a mediator between input and the other subsystems.
- *UIBuilder*	The UIBuilder represents the factory that builds the UI.
DiseaseSupplySubsystem	The DiseaseSupplySubsystem is responsible for receiving information from the user through the ClientUISubsystem.
- *Shipment*	A Shipment represents an object that has the quantity of a SupplyType and a destination.
- *SupplyType*	A SupplyType represents a supply from the SupplyList.
- *DiseaseType*	A DiseaseType represents a disease from the DiseaseList.
- *SupplyInstance*	The SupplyInstance represents an object that has the quantity of a SupplyType from a given location.
- *DiseaseCase*	The DiseaseCase represents an occurrence of a DiseaseType.

Item	Description
UserManagementSubsystem	The UserManagementSubsystem is responsible for handling the creation, deletion and modification of users through the ClientUISubsystem.
- *User*	A User represents a user from the UserList.
DataDisplaySubsystem	The DataDisplaySubsystem is responsible for displaying current known information through the ClientUISubsystem.
- *DataDisplay*	DataDisplay represents the data to be displayed and sends it to the UIManager.
ServerDatabaseSubsystem	The ServerDatabaseSubsystem is responsible for updating the ClientDatastoreSubsystem through the CommunicationSubsystem.
- *UserTable*	The UserTable represents the User data stored on the server.
- *SupplyTable*	The SupplyTable represents the SupplyType data stored on the server.
- *InventoryTable*	The InventoryTable represents the collection of SupplyInstances
- *DiseaseTable*	The DiseaseTable represents the DiseaseType data stored on the server.
- *DiseaseCaseTable*	The DiseaseCaseTable represents the DiseaseCase data stored on the server.
- *ShipmentTable*	The ShipmentTable represents the Shipment data stored on the server.
CommunicationSubsystem	The CommunicationSubsystem is responsible updating the ServerDatabaseSubsystem through data gathered from the DiseaseSupplySubsystem and UserManagementSubsystem.
- *Connection*	The Connection represents the connection between the ClientUISubsystem and the ServerDatabaseSubsystem.
- *Message*	A Message represents a SupplyType or DiseaseType and its related Shipment, DiseaseCase and SupplyInstance.
- *Parser*	The parse represents an encoder and decoder for Messages.
- *MessageQueue*	The MessageQueue represents a queue of Messages.

Item	Description
ClientDatastoreSubsystem	The ClientDatastoreSubsystem is responsible for retaining data and displaying data upon request through the DataDisplaySubsystem.
- *UserList*	The UserList represents the client's version of the UserTable.
- *DiseaseList*	The DiseaseList represents the client's version of the DiseaseTable.
- *SupplyList*	The SupplyList represents the client's version of the SupplyTable.
- *DiseaseCaseList*	The DiseaseCaseList represents the client's version of the DiseaseCaseTable.
- *InventoryList*	The InventoryList represents the client's version of the InventoryTable.
- *ShipmentList*	The ShipmentList represents the client's version of the ShipmentTable.

2.3 Design strategies

2.3.1 Hardware/Software Mapping

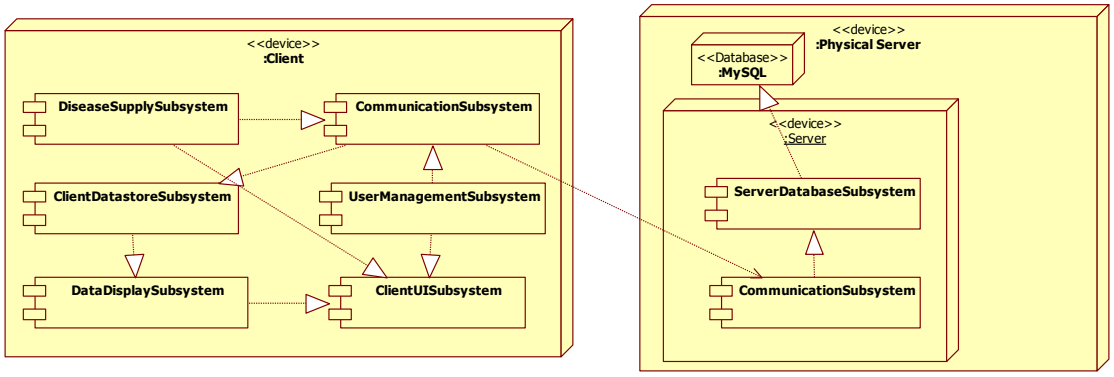


Figure 3 UML component Diagram illustrating relationship between the hardware, subsystems and components

For our software, having a database mapped to each client lets us access information faster as well as giving us control over the information temporarily, but it creates a problem of synchronization between the server and the client as a result we created the *CommunicationSubsystem the client and server.* Having a server and client also creates the problem of uptime. If the server or the connection fails then

then the new data that is created on the client has to be kept until the connection is re-established. To solve this problem we created a message queue to hold the unsent messages.

We needed to implement the design pattern in order to reduce coupling between objects in different subsystems. They then give a simple interface for other objects to use. For example, the *ClientDatabase* can only be accessed through functions such as add(object), and find(criteria).

2.3.2 Persistent Data Management

The goal for persistent data management is to use a MySQL database for all storage. This is for two main reasons. First storing in this manner creates a permanent file that is updated with all database changes. This allows for minimal data-lose should the server accidentally shut down. Second, this allows for a clean and easily searchable method of tracking all objects.

Tables in the database are to include: users, diseases being tracked, supply types being tracked, the actually recorded cases of the diseases, the national supply inventory, and records of supply transfer shipments.

2.3.3 Access control and security

This section details the access differing user types have in accessing the interfaces of various objects in the subsystems. These access rights will be

enforced by the Client UI. When the user logs in the ui will enforce the elements that the user has access to.

	Clerk	Medical Administrator	System Administrator
ListContainer	AddDiseaseCase() AdjustInventory() AddShipment()	AddDiseaseCase() AdjustInventory() AddShipment()	AddUser() AddToDiseaseList() AddSupplyType()
DataDisplay	UpdateMap() Find()	DrawHistogram() DrawPieChart() UpDateMap() Find()	
UIDrawHelper	UITable() UIMapUPdate()	UIHistogram() UIGraph() UITable() UIMapUpdate()	UITable()
ClientParser	Parse() CreateMsg() CreateMsg(DiseaseCase) CreateMsg(Supply) CreateMsg(Shipment)	Parse() CreateMsg() CreateMsg(DiseaseCase) CreateMsg(Supply) CreateMsg(Shipmet	Parse() CreateMsg() CreateMsg(SupplyType) CreateMsg(DiseaseType) CreateMsg(User)
Connection-Management		LoadFile()	

2.3.4 Global Software Control

The client end of the system will be event-driven controlled. The client will wait for an external event from the user. For example, the user chooses to add a

new case to the system by clicking the "Add New Case" button on the client. The client opens a new window to allow this action. The client then awaits the user to both input the correct fields and click "Add" or awaits the "Cancel" button to be clicked. If the "Add" button is clicked, it is dispatched to the appropriate object, based on information associated with the event. If "Cancel" is clicked it merely closes the window and awaits the next selection by the user.

The server end of the system will make use of threads. Each time a client connects to the server, through use of a socket, the server will send the client an updated version of the database. It will also store the client's address in a list so it can update the client if a change is committed, it will also remove them from the list if their connection is terminated or times out. The client will ping the server if there is nothing else to transmit to ensure the connection remains active. Should the user send the server information, whether it is a new case or shipment, the server will assign a thread from the thread poll and process the data on first come first served basis. After an update has been committed the server will send each client the update utilize all the threads and update each client currently connected through the list generated by each connection client/thread to reflect the changes.

2.3.5 Boundary Conditions

Use Case Name	ServerStartup
Participating Actors	Initialized by System Administrator
Flow of Events	1. The system administrator types a command in the console which starts the server in the background 2. The Server checks if there are clients requesting a connection to the server and provides the connection if requested 3. When the connection is established the server uploads the information in the database to the client
Entry Condition	The user is logged in to the computer where the server runs on
Exit Condition	The server started up successfully or did not start up successfully

Use Case Name	ServerShutdown
Participating Actors	Initialized by SystemAdministrator
Flow of Events	1. The System administrator types a command in the console, which stops the server. 2. The Server closes all connections and safely disconnects from the database.
Entry Condition	The System administrator is logged in to the system
Exit Condition	The shutdown of the server was successful or not successful.

Use Case Name	ServerUnexpectedDisconnection
Participating Actors	No actors
Flow of Events	1. All threads are completed. 2. List of servers is reset.
Entry Condition	Server is disconnected
Exit Condition	Server reestablishes its connection

Use Case Name	ServerFirstTimeStartup
Participating Actors	Initialized by System Administrator
Flow of Events	1. Connects to MySQL. 2. Installs the database it will use. 3. The system administrator types a command in the console which starts the server in the background 4. The Server checks if there are clients requesting a connection to the server and provides the connection if requested 5. When the connection is established the server uploads the information in the database to the client
Entry Condition	The user is logged in to the computer where the server runs on for the first time
Exit Condition	The server started up successfully or did not start up successfully

Use Case Name	UnintentionalServerShutdown
Participating Actors	Server Administrator (possibly)
Flow of Events	1. The system administrator boots the server's computer and reinitiates the console 2. The system administrator types a command in the console which starts the server in the background 3. The Server checks if there are clients requesting a connection to the server and provides the connection if requested 4. When the connection is established the server uploads the information in the database to the client
Entry Condition	Server is terminated (eg. Power surge)
Exit Condition	Server is restarted

Use Case Name	ClientStartup
Participating Actors	Initialized by User
Flow of Events	1. The User clicks the icon to start up the Client twice 2. The GUI starts up with the login window shown at first. 3. A file is loaded by the client in the background to load and apply the configurations of the client. 4. When the user logged in successfully it requests a connection at a given server and downloads all the information from the server
Entry Condition	The User is logged in to a computer with the Client software installed
Exit Condition	The User exits the program or logs out of his system

Use Case Name	ClientShutdown
Participating Actors	Initialized by User
Flow of Events	1. The User presses the exit button 2. The Client disconnects the connection to the server and closes the GUI.
Entry Condition	The user is logged in to the Client.
Exit Condition	The client is successfully closed or the client is not successfully closed.

2.3.6 Design Patterns

Our Client subsystem design is based around a Model , View , Controller; where the Model is the *ClientDatabaseSubsystem*, the View Is the *ClientUISubsystem, and DataDisplaySubsystem, and the controller is the CommunicationSubsystem*. This allows a more simple way of keeping the correct flow of data across the system. We implemented the Façade design pattern in order to encapsulated the subsystems and give a compact and easy to understand interface for other subsystems to use. This lets us hid the implementation of the subsystems internal classes.

The *UIBuilder* is going to be responsible for building whatever user

interface object is necessary for the GUI. It will be built using the factory method. There will have many functions to use the one factory to build all of the window components. This also allows us to separate the window into different sections having the map side of the GUI as well as a Tabbed section on the other side of the window.

The management between the GUI and the Client was another issue that came up. To keep the view and the model separate we implemented the mediator design pattern. This controlled the communication between the GUI and the rest of the system. In our ClientUISubsystem we used the *UIManager*, this class is used to take information from the GUI and build objects out of them or pass the data to other objects.

A message is used to communicate between the client and the server; this was a perfect candidate for the interpreter design pattern. This made it so we could pass the *Parser* class an object and command so it can interpret it to a message that the server can understand. The parser can take a message and interpret to an object and command that the client can understand.

2.4 Subsystem services

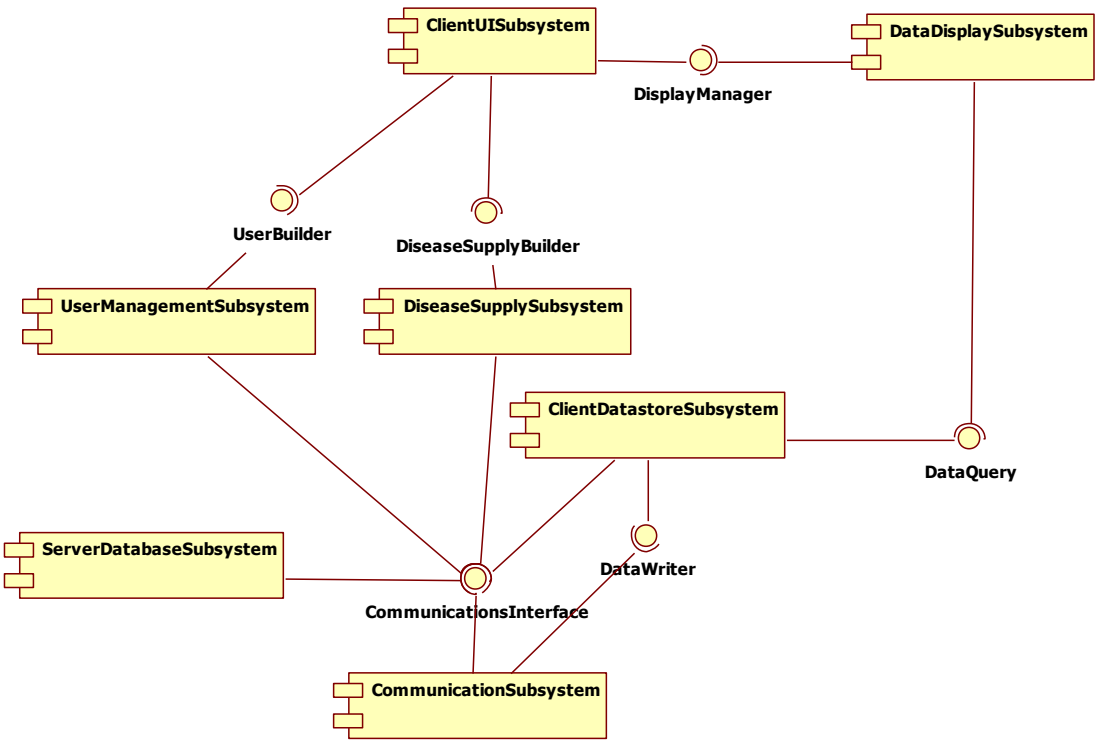


Figure 4 Subsystem component diagram illustrating subsystem services

- *DisplayManager* allows a subsystem to format and display their information in the GUI.
- *UserBuilder* allows a subsystem to create a user based on information provided.
- *DiseaseSupplyBuilder* allows a subsystem to add a case, disease, supply or shipment and enables them to create shipments.
- *DataQuery* allows a subsystem to request using specific criteria and receive the response.
- *DataWriter* allows a subsystem to commit data to the data storage using specific criteria.

- *CommunicationsInterface* allows a subsystem to register with the CommunicationSubsystem, to authenticate and initiate and close connections.

2.5 Message protocol

Our message protocol uses an XML standard. The connection between client and server will be opened when the client wish to connect through sockets over the Lambda servers. When the connection is established the XML string is passed over the network. Once the message is finished sending, the connection is terminated. To see the message protocol please see refer to the attached document following this one.

3. Object Design

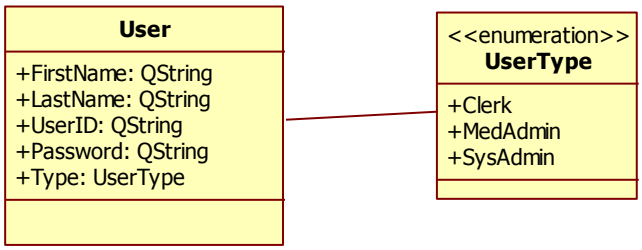
3.1 Overview

This section will specify the solution domain and close the gap between application domain objects and COTS components by identifying solution domain objects. Detailed object models will be created through the use of analysis object models and system design model's subsystem decompositions and system architecture strategies.

The main tasks of detailed Object Design are identifying opportunities for software reuse, which result in additional COTS components and design patterns. Specifying services to reflect interface specifications, restructuring

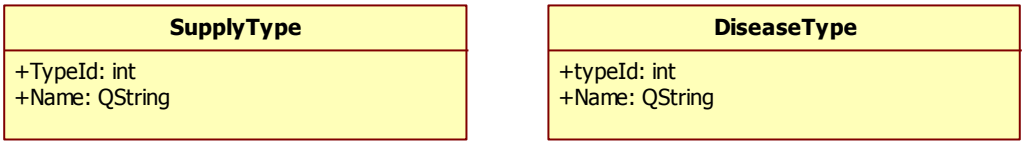
object models for clarity and maintainability, and optimizing object models to meet performance requirements.

3.2 Class interfaces



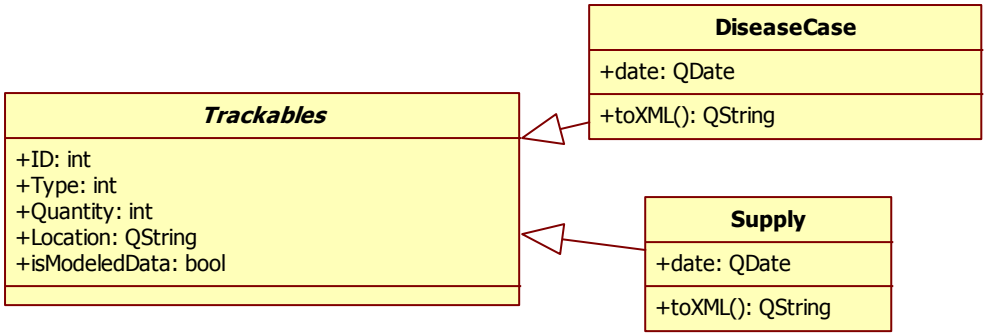
User – object representing the various users of the C.E.T. System composed of:

- *First Name* and *Last Name* – self explanatory
- *UserID* – unique string representing the user
- *Password* – unique string, self-explanatory
- *Type* – enumeration indicating the type of user (Clerk, Medical Administrator, System Administrator)



SupplyType and **DiseaseType** – container objects for self-explanatory data types, consisting of:

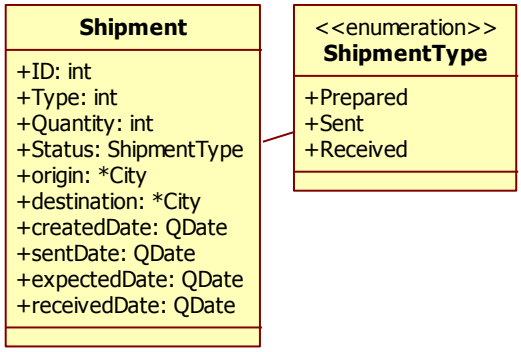
- *Name* – string consisting of the name of the supply or string
- *TypeID* – unique numerical identifier



Trackables, DiseaseCase and **Supply** – trackables is an abstract class that both **DiseaseCase** and **Supply** inherit from, as the name implies they are used

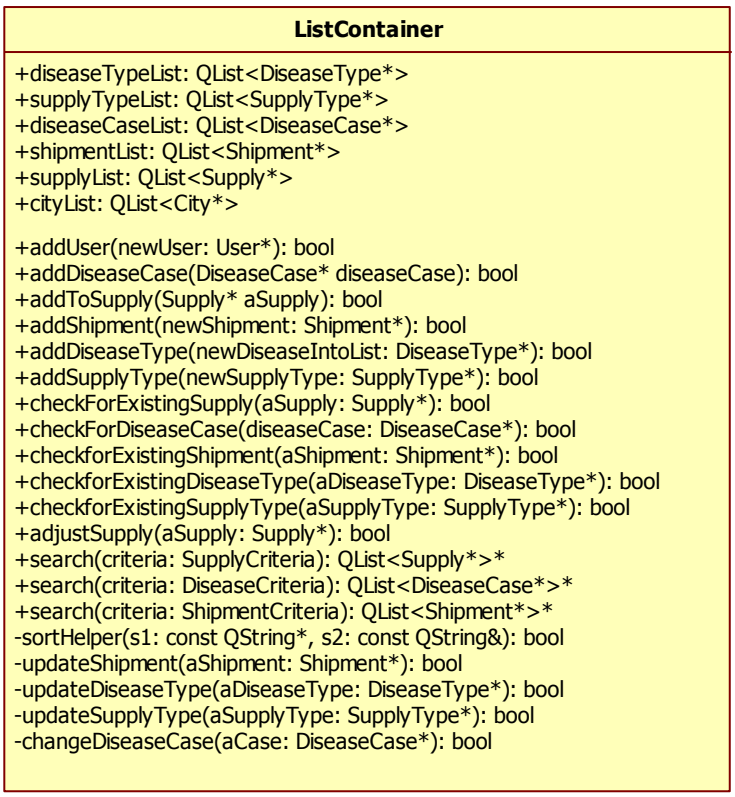
as the basic data object representing to two different data types being tracked by the system. Their basic attributes consist of:

- *ID* – a unique integer identifying the object
- *Type* – an integer consisting of the unique ID of the type being tracked (either disease type or supply type)
- *Quantity* – an integer indicating the number of items in that particular occurrence, either the number of cases in that report or the quantity of the supply at the location
- *Location* – the location of the instance being recorded
- *IsModeledData* – a Boolean indicating if the data record is part of the real data set or part of a modeled data set imported for report generation
- *Date* – self-explanatory, only used in the **DiseaseCase** object to record the date of the reported occurences
- *ToXML* – a QString that contains data converted to XML elements



Shipment is the object containing all the information required to track the transfer of supplies between two locations. It consists of:

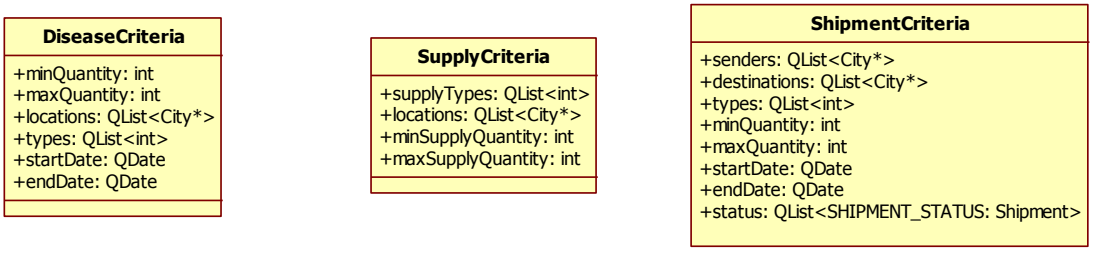
- *ID* – the unique ID identifying that particular shipment
- *Type* – int representing the type of supply involved in the transfer
- *Quantity* – int indicating the number of items transferred
- *Status* – an enumeration indicating whether the shipment is prepared, sent or received
- *Origin* – the originating location of the shipment
- *Destination* – the receiving location of the shipment
- *CreatedDate* – the date(QDate) the shipment was created by the user
- *SentDate* – the date(QDate) the shipment was shipped
- *ExpectedDate* – the date(QDate) the shipment can be expected to arrive on
- *ReceivedDate* – the actual date(QDate) the shipment was received on



ListContainer is the master data structure for the client component of the C.E.T. system, its attributes and operations are:

- *DiseaseTypeList* – *a* QList of pointers to the DiseaseType Objects
- *SupplyTypeList* – a QList of pointers to SupplyType Objects
- *DiseaseCaseList* – a QList of pointers to DiseaseCase Objects
- *ShipmentList* – a QList of pointers to Shipment Objects
- *SupplyList* – a Qlist of pointers to Supply Objects
- *CityList* – a QList of pointers to City Objects
- *addUser* – takes a pointer to a User object and inserts it into the UserList, returns true if successful, false if not
- *AddDiseaseCase* – takes a pointer to a DiseaseCase and inserts it into the DiseaseCaseList and returns true if successful, false if not
- *AddToSupply* – takes a pointer to a Supply and inserts it into the SupplyList, returns true if successful, false if not
- *AddShipment* – takes a pointer to a Shipment and inserts it into the ShipmentList, returns true if successful, false if not
- *AddDiseaseType* – takes a pointer to a DiseaseType and inserts it into the DiseaseTypeList, returns true if successful, false if not
- *AddSupplyType* – takes a pointer to a SupplyType and inserts it into the SupplyTypeList, returns true if successful, false if not

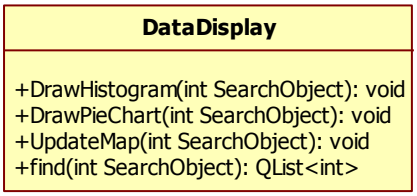
- *CheckForExistingSupply* – takes a pointer to a Supply and checks to see if any results exist, returns true if successful, false if not
- *CheckForDiseaseCase* – takes a pointer to a DiseaseCase and checks to see if any results exist, returns true if successful, false if not
- *CheckForExistingShipment* – takes a pointer to a Shipment and checks to see if any results exist, returns true if successful, false if not
- *CheckForExistingDiseaseType* – takes a pointer to a DiseaseType and checks to see if any results exist, returns true if successful, false if not
- *CheckForExistingSupplyType* – takes a pointer to a SupplyType and checks to see if any results exist, returns true if successful, false if not
- *AdjustSupply* – takes a pointer to a Supply and modifies the value, returns true if successful, false if not
- *Search(SupplyCriteria)* – takes supply criteria and searches to match those specifications, returns a list of Supply
- *Search(DiseaseCriteria)* – takes disease criteria and searches to match those specifications, returns a list of DiseaseCase
- *Search(ShipmentCriteria)* – takes shipment criteria and searches to match those specifications, returns a list of Shipment
- *SortHelper* – takes two QStrings and helps in sorting the list that uses it, returns true if successful, false if not
- *UpdateShipment* – takes a pointer to a shipment and updates its information, returns true if successful, adds it if false
- *UpdateDiseaseType* – takes a pointer to a DiseaseType and updates its information, returns true if successful, adds it if false
- *UpdateSupplyType* – takes a pointer to a SupplyType and updates its information, returns true if successful, adds it if false
- *ChangeDiseaseCase* – takes a pointer to a DiseaseCase and modifies it, returns true if successful, false if not



DiseaseCriteria, SupplyCriteria, and ShipmentCriteria, serve to package the various requirements for searches. The attributes of the classes are:

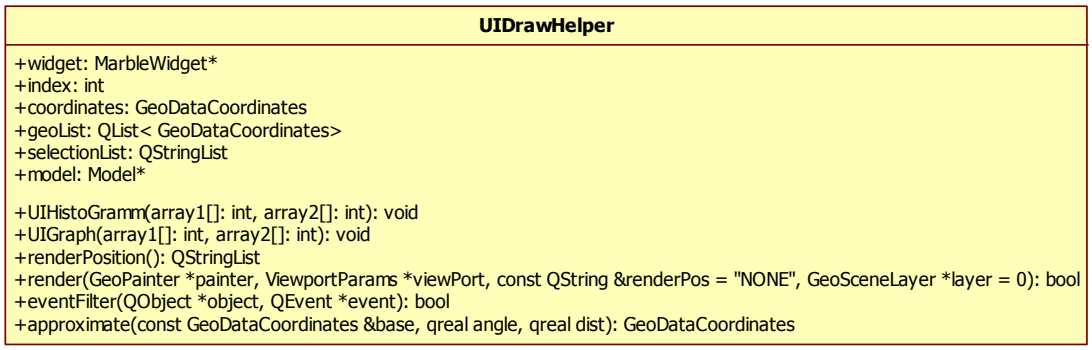
- *Min(Supply)Quantity* – an int that specifies the minimum quantity desired
- *Max(Supply)Quantity* – an int that specifies the maximum quantity desired
- *Locations/Destinations* – a QList of City pointers that specifies the City(s) desired
- *(Supply)Types* – a QList of integers that correspond to a different type
- *StartDate* – a QDate that specifies the lowest occurrence date desired

- *EndDate* – a QDate that specifies the highest occurrence date desired
- *Status* – used by ShipmentCriteria objects, an enum indicating shipment status (prepared, sent, received)



DataDisplay handles all co-ordination between the Client UI and the Client Data store. Its primary function is to gather the information required by the UI for its various display functions, be they graphical or just a table display. It has no attributes of its own however it has a number of operations:

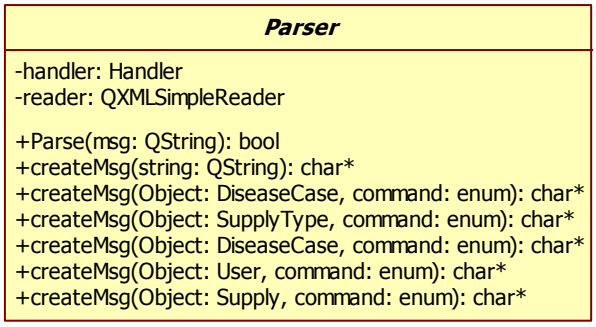
- *DrawHistogram* – takes a search object as a parameter, searches the datastore for the matching entries and packages them in a format usable by the *UIHistogram* function of the **UIDrawHelper** class (defined following)
- *DrawPieChart* – like the Drawhistogram, but the end result is pie chart formatted data, it also has a companion function in **UIDrawHelper**
- *UpdateMap* – again like the DrawHistogram, but its purpose is to supply properly formatted data for its companion **UIDrawHelper** function
- *Find* – Takes a search object and returns an unformatted QList of integers



UIDrawHelper is the class in the Client UI that handles the drawing functions within the GUI, it takes in the properly formatted data from the **DataDisplay** operations. Its operations are:

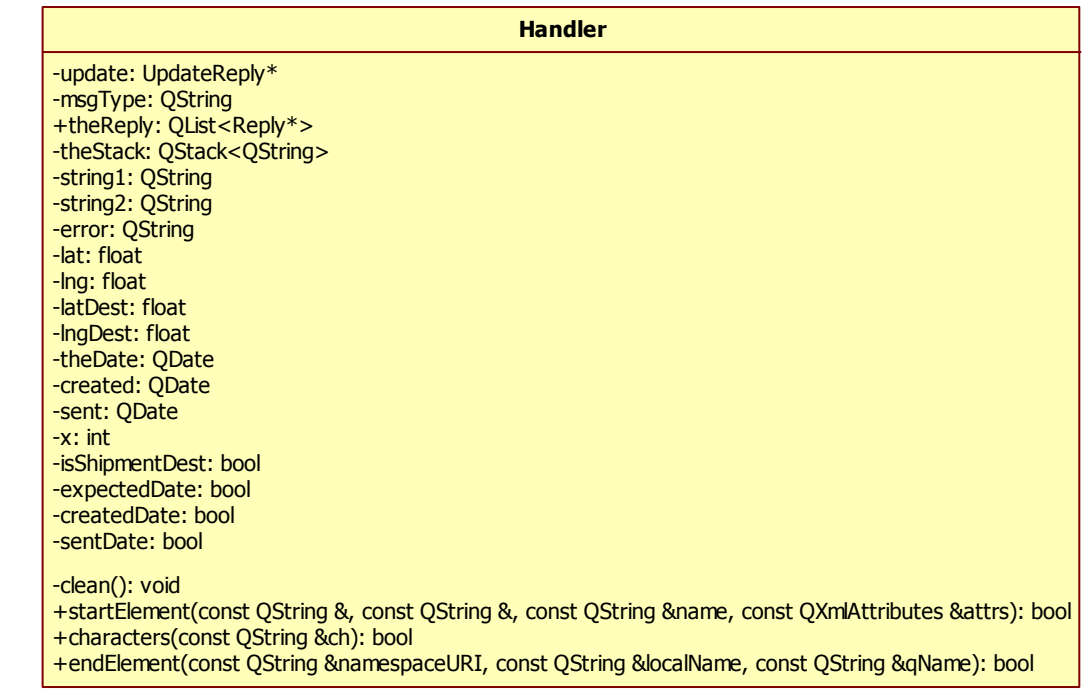
- *Widget* – a pointer to MarbleWidget we is used to load the Marble API
- *Index* – an integer used by renderPosition and the name speaks for itself
- *Coordinates* – GeoDataCoordinates contains longitude and latitude
- *GeoList* – a list of GeoDataCoordinates used with the database
- *SelectionList* – keeps a list of all entries selected
- *Model* – a pointer of Model to access containers

- *UIHistoGramm* – takes in two arrays, the first is the list of entries for the x-axis, the second is the corresponding values for the y-axis
- *UIGraph* – like the UIHistogram function but draws a line graph
- *RenderPosition* – render desired position on the GUI
- *Render* – override function to allowing drawing in the GUI, returns true if successful, false if not
- *EventFilter* – override function for key events
- *Approximate* – creates an approximation for GeoDataCoordinates



The **Parser** class and its children **ClientParser** and **ServerParser** handle the message conversion for the communication systems. The operation signatures are identical, although their implementations will differ due to the differing data storage structures within the client and server. The operations are:

- *Handler* – an object used by QXMLSimpleReader to parse the message
- *Reader* – parses the XML message
- *Parse* – takes the message received from **ConnectionManagement**, and converts it to either the appropriate object and passes it to the appropriate subsystem, or initiates the execution of the command
- *CreateMsg* – takes the object parameter and command and hands it off to the **ConnectionManagement** object for transmission

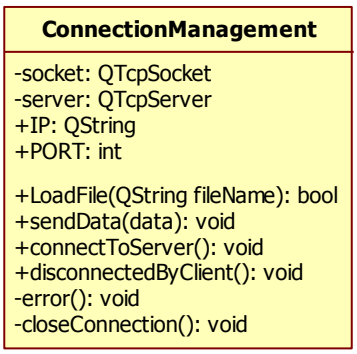


The **Handler** as the name implies handles XML and extracts data from its elements. It consists of:

- *Update* – container that holds objects
- *TheReply* – container that holds the message
- *The remainder of the attributes are helper variables*
- *Clean* – resets the helper variables
- *StartElement* – executes when the parser encounters a start element, returns true if successfully executed, false if not
- *Characters* – executes when the parser encounters a none XML element, returns true if successfully executed, false if not
- *EndElement* – executes when the parser encounters an end element, returns true if successfully executed, false if not

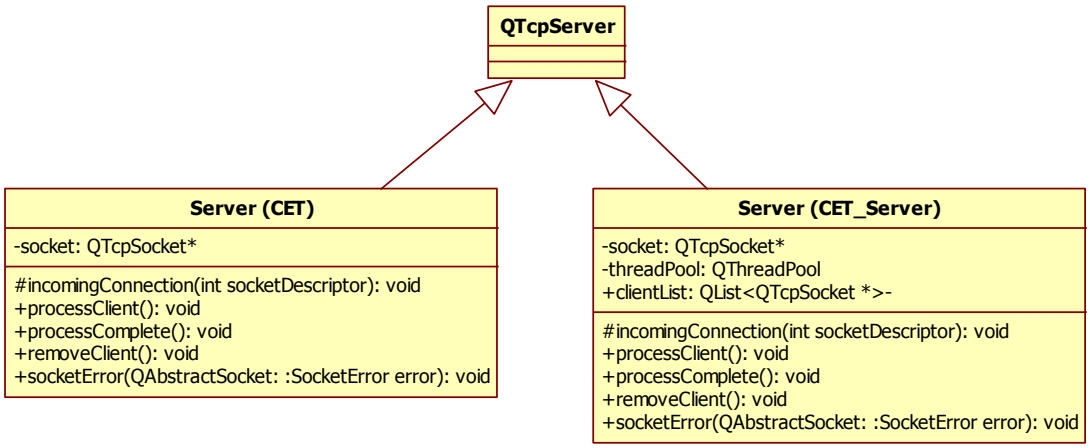
Data storage will be maintained by **data** it consists of:

- *Db* – the database in which all data is stored
- *LatestRev* – is the current revision number of the database
- *Add* – adds a pointer of a City/City QList/SupplyType/SupplyType QList/ DiseaseType/DiseaseType QList/Shipment/Shipment QList/Supply/Supply QList to the database
- *GetCity* – checks both floats as geographical coordinates and returns a city that corresponds to them, null otherwise
- *GetCitiesUpdate* – checks for all Cities added since specified revision number and QList of them
- *GetSupplyTypeUpdate* – checks for all SupplyTypes added since specified revision number and returns a QList of them
- *GetDiseaseTypeUpdate* – checks for all DiseaseTypes added since specified revision number and returns a QList of them
- *GetDiseaseCaseUpdate* – checks for all DiseaseCases added since specified revision number and returns a QList of them
- *GetShipmentUpdate* – checks for all Shipments added since specified revision number and returns a QList of them
- *GetSupplyUpdate* – checks for all Supplies added since specified revision number and returns a QList of them



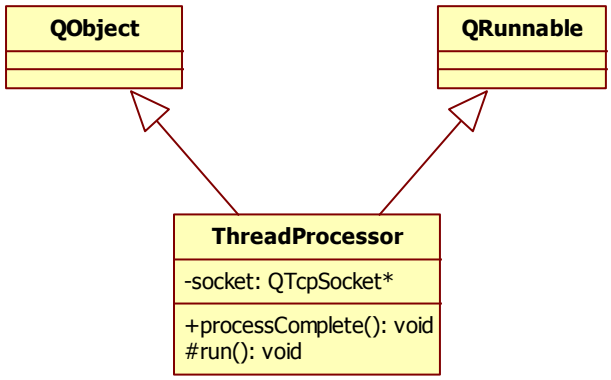
As the name implies **ConnectionManagement** handles the connections and communication between the client and server it consists of:

- *QTcpSocket* – a QT socket implementation that allows the client and server to establish a connection
- *QTcpServer* – part of the QT TCP implementation, allows for listening for incoming connections
- *IP* – An IP address
- *PORT* – A port number
- *LoadFile* – loads a file in for displaying modeled data
- *SendData* – sends the data to the server after converting it to a QByteArray
- *ConnectToServer* – attempts to connect to the IP through the specified PORT
- *DisconnectedByClient* – the client has chosen to disconnect from the server
- *Error* – a socket error exception is thrown
- *CloseConnection* – terminates the connection



Server inherits from QTcpServer allowing use of all its functions and also gives the ability to override some of its functions, it consists of:

- *Socket* – used to help process the client
- *IncomingConnection* – an override of QTcpServer's incomingConection, allows signals to be handled different with custom slots
- *ProcessClient* – used to process the client when the signal is read ready
- *ProcessComplete* – returned when the processing of the client is complete
- RemoveClient – removes the client when the disconnect signal is sent
- SocketError – an exception that is thrown when an issue occur with the socket



ThreadProcessor inherits from both QObject and QRunnable allowing there use to process threads sent from the server, it consists of:

- *Socket* – stores the socket information sent through the constructor
- *ProcessComplete* – a function that exists in Server
- *Run* – a QRunnable function that has been overrided in order to process the threads request