Carleton University

cuCare

System Design Update

Team: The Four Puppeteers
Sergey Vershinin (Team Lead)
David Carson
Devin Denis
Mike Yuill

Submitted to:
Dr. Christine Laurendeau
COMP 3004 Object-Oriented Software Engineering
School of Computer Science
Carleton University

FINAL
2012-12-03

Table of Contents

1 Introduction 1
2 Subsystem Decomposition 2
2.2 System Decomposition 2
3 Design Strategies 4
3.1 Persistent Data Management 4
4 Subsystem Services 5
5 Class Interfaces 6

1 Introduction

The Four Puppeteers Team proudly presents this cuCare system design update. This document describes the differences between the system design outlined in the system design document submitted for your review on November 22, 2012, and the actual implementation of the cuCare Prototype #2.

The changes are described in their respective sections, titled in the same manner as the original system design document. If the change was placed in a sub-section of the original document, the section hierarchy is preserved.

To improve readability each of the changes is presented in a table that contains two sections, titled **OLD** and **NEW**. The **OLD** section contains the relevant description taken from the original system design document, while the **NEW** section contains an up-to-date description.

2 Subsystem Decomposition

2.2 System Decomposition

System Decomposition Diagrams

OLD

The diagram titled "Figure 4 – cuCare – Classes" on page 8 of the original system design document contained a list of the classes contained in the Model library.

NEW

The Model library implemented in Prototype #2 also includes the following classes:

- Property
- IntProperty
- StringProperty
- Utility

The Property class and its child classes IntProperty and StringProperty enable implementation of data member reflection for Model classes. Data member reflection is used by the Client Data Storage, Object Interpreter, and Storage subsystems.

The Utility class provides integer-to-string and string-to-integer conversion functions used throughout the system. This class also provides a function to print a stack trace used for testing purposes.

The GUI, Access Control, Patient Data Management, Client Data Storage, Object Interpreter, and Storage subsystems make use of classes contained by the Model library.

Subsystem Descriptions

GUI

OLD

Any operation requiring information to be stored on or retrieved from the server is handled by the WindowController class that uses services from other subsystems to send or request the relevant data.

The WindowController also uses the Access Control subsystem to check the privileges of the current user and select between the PatientViewWindow and the SysAdminViewWindow, as well as enable or disable GUI elements as appropriate.

NEW

It was determined that creating an additional layer of abstraction was unnecessary as the services used by the GUI subsystem are sufficient for that purpose. As such, the Access Control and Patient Data services are accessed directly by the window and dialog classes of the GUI subsystem.

Communications

NEW

Several classes in Communications are now abstract (where they weren't before). They are abstract because they have only static member functions (utilities used to perform data and object conversion in the Communications subsystem). There's no point in being allowed to instantiate a class that only has static utilities. For this change, two classes were renamed:

- AbstractNetworkMessenger -> BaseNetworkMessenger
- AbstractNetworkTranslator -> BaseNetworkTranslator

DESIGN NOTES:

1. The QJson library is used by the Communications subsystem for serialization purposes.
2. During implementation of the message packing functionality, it became apparent that Qt did not provide as much automation as was initially assumed. As the result, a number of new functions needed to be defined. The resulting changes could have been implemented using Request and Reply classes. However, it was deemed better to remain consistent with the original system design where possible.

Storage

NEW

The Storage subsystem now includes two new classes: StorageObject and RepoModel.

The StorageObject class is a simple storage wrapper that facilitates communication between the Server Controller (Controller class) and Storage (Repository class) subsystems.

The RepoModel class, used by the Repository class, implements the part of the Storage subsystem functionality that makes use of classes in the Model library. RepoModel is used to set-up the database tables during the first time the server process is executed. This class was introduced to encapsulate functionality that is dependent on the Model library from the rest of the Storage subsystem.

3 Design Strategies

3.1 Persistent Data Management

OLD

Figures 8 (ER diagram) and 9 (Table Schema) on page 21 of the original system design document show an implementation using vertical inheritance.

NEW

The cuCare Prototype #2 was implemented using horizontal inheritance. This design change was made because creating separate tables for child classes for user and follow-ups would have unnecessarily increased implementation complexity without an obvious benefit.

Switching to horizontal inheritance does mean that the USERS and FOLLOWUPS tables will contain null values for unused attributes (fields). However, this does not decrease performance nor substantially increase the volume of information stored in the database.

4 Subsystem Services

This section provides descriptions of operations implemented in cuCare Prototype #2, that enables functionality not accounted for in the original system design document.

NOTE: Operation description tables include a pseudo code prototype to provide additional information about the operation. Actual function declarations are found in the Class Interfaces section.

Access Control

Login Verification

NEW	
Subsystem	Access Control
Class	AccessControl
Service	Login Verification
Operation	Log Out
Prototype	*(void) AccessControl::Log Out ()*
Description	This operation allows the GUI to request that the current user be logged out of the system. The operation causes the subsystem to set access control status to "logged out".
Traceability	UC-110

Client Data Storage

Retrieval

NEW	
Subsystem	Client Data Storage
Class	ClientData
Service	Retrieval
Operation	Pull Patients by Follow-up Status
Prototype	*(List<int>) ClientData::Pull Patients by Follow-up Status (FollowupType)*
Description	This operation is a specialized version of the Pull Model Objects operation. This operation refills the warehouse with Patient objects, filtered by follow-up type, returned from communications when an SQL query with the specified parameters is run on the database. This operation always uses the Retrieval service of the Communications subsystem to pull the objects, and replaces previous copies of the same object in the warehouse. The only parameter is the type of a follow-up, communicated via an enumeration variable. This operation returns a list of all ids of the Patient objects which were pulled. These IDs can be used to access the pulled Patient objects in the warehouse using the *Get Model Object* operation.
Traceability	UC-210, UC-560

5 Class Interfaces

This section outlines the differences between the original system design document and the implementation of cuCare Prototype #2. Overall, most of the changes fall into the following three categories:

- New operations, such as the two new service operations described in the Subsystem Services section or the new class functions added to implement singleton functionality
- Changes concerning switching from passing a parameter by value to passing it by reference (i.e. as a pointer), and
- Addition of new parameters that provide the Storage subsystem with the information necessary for interacting with the database (i.e. unanticipated in the original design).

AccessControl (Access Control Subsystem)

OLD

AccessControl
+verifyUsername(string username) : LoginStatus +getLoginStatus() : LoginStatus

NEW

The AccessControl class interface now includes a new function:

+logout() : LoginStatus

For more information, see description of the corresponding operation in the Subsystem Services section.

PatientData (Patient Data Management Subsystem)

OLD

PatientData
+getPatientList(FilterType filterType, Physician physician, FollowupType followupType) : List<Patient>
+getFullPatient(Patient patient) : Patient
+createPatient(Patient patient) : void
+modifyPatient(Patient patient) : void
+createConsultation(Consultation consult) : void
+modifyConsultation(Consultation consult) : void
+createFollowup(Followup followup) : void
+modifyFollowup(Followup followup) : void
+getPhysicianList() : List<Physician>

NEW

The interface of the PatientData class has been changed as follows:

```
+getPatientList(FilterType filterType, int physicianId, FollowupStatus followupStatus) : List<Patient*>
+getFullPatient(int patientId): Patient*
+createPatient(Patient *patient) : int
+modifyPatient(Patient *patient) : void
+getConsultation(int consultationId) : Consultation*
+createConsultation(Consultation *consult, int parentId) : int
+modifyConsultation(Consultation *consult) : void
+getFollowup(int followupId) : Followup*
+createFollowup(Followup *followup, int parentId) : int
+modifyFollowup(Followup *followup) : void
+getPhysicianList() : List<Physician *>
```

Rationale:

- All passing of Model classes has been changed from passing by value to passing by reference to cut down on unnecessary copying.
- getConsultation() and getFollowup() functions have been created because the Model classes have been changed to contain lists of integer values with IDs of their child objects (e.g. list of consultations belonging to a given patient). As such, getFullPatient() would no longer provide the GUI subsystem with access to Consultation and Followup objects.
- createPatient(), createConsultation(), and createFollowup() functions have been changed to return the integer ID of the newly created object. The Client Data Management subsystem passes the request, via Communications, to the Storage subsystem which returns an integer ID value automatically generated by the SQLite database.
- createConsultation(), and createFollowup() now take a parented parameter, which provides the Storage subsystem with the information it needs to properly associate the new database entry with its respective patient or consultation record.

ClientData (Client Data Storage Subsystem)

OLD

ClientData
-Factory factory -Warehouse warehouse -NetworkSettings networkSettings
+T getModelObject<T>(int id) +list<int> pull(string className, map<string, string> filter) +list<int> getReportResults(ReportType type) +Time getAuditTime() +void create(ModelObject *object) +void modify(ModelObject *object) +void setAuditTime(Time time) +void setIPAddress(string ip) +void setPortNumber(int port)

NEW

The ClientData class interface now includes a new function:

+pullPatientsByFollowupStatus(FollowupStatus status) : list<int>

For more information, see description of the corresponding operation in the Subsystem Services section.

The rest of the interface of the ClientData class was changed as follows:

The single templated get<T> method was expanded into a get method for every type:

+getPhysician (int id) : Physician*
+getAdminAssistant (int id) : AdminAssistant*
+getSysAdmin (int id) : SysAdmin*
+getConsultation (int id) : Consultation*
+getPatient (int id) : Patient*
+getMedicalTest (int id) : MedicalTest*
+getReturnConsultation (int id) : ReturnConsultation*
+getReferral (int id) : getReferral*
+getMedicationRenewal (int id) : getMedicationRenewal*

During implementation it became apparent that the template function would need a specialization for every model type to know which map to look in for the requested object, which would defeat the purpose of using a template function in the first place. The approach taken also guarantees that the function will only be called with appropriate types, and avoids having to define the functions in the header file.

+pull(ModelObject* modelObject, int parentId = -1) : list<int>

Pull now takes a pointer to the model base class, ModelObject. The filtering process is now achieved by removing properties from that ModelObject, rather than simply passing a map of properties. The className string that was previously passed has changed into a tableName which is part of the ModelObject. parentId is an optional parameter (it is ignored if the value is negative), which describes the unique id of the parent object in the case of consultations and follow-ups.

+create(ModelObject *object, int parentId = -1) : int

Create was changed to return the newly created object's unique id, because this id is determined by the database on insertion. It also takes a parentId for the same reason that pull does.

+getInstance() : ClientData&

This static method allows access to the singleton instance of ClientData.

Object Interpreter (Object Interpreter Subsystem)

OLD

Object Interpreter
+Map<string, string> getPropertyMap (ModelObject *modelObject) +void getModelObject (map<string, string>, ModelObject *returnObject)

NEW

The interface of the Interpreter class was changed as follows:

+getPropertyMap(ModelObject *modelObject, out map<string, string>* properties) : void

The only change with this function was to move the properties pointer to be an out parameter instead of a return. This was done to keep the object interpreter functions more consistent with each other.

+getModelObject(map<string, string> *properties, out ModelObject *modelObject) : void

The map of properties was changed from a value to a pointer to reduce copying.

ClientNetworkInterface (Communications Subsystem)

OLD

ClientNetworkInterface
-serverIP : QHostAddress -serverPort : quint16
+create(in objectType : string, in pObjectMap : Map<string, string>*, out pReturnID : int*, out pErrorString : string*) : bool +push(in objectType : string, in pObjectMap : Map<string, string>*, out pErrorString : string*) : bool +pull(in objectType : string, in pFilterMap : Map<string, string>*, out pMapList : List<Map<string, string>*, out pErrorString : string*) : bool +setAuditTime(in auditTime : Time, out pErrorString : string*) : bool +retreiveAuditTime(out auditTime : Time*, out pErrorString : string*) : bool

NEW

The interface of the ClientNetworkInterface class was changed as follows:

+create(string tableName, string idKey, map<string, string> *pObjectMap, int *pOutID, string *pErrorString) : bool
+push(string tableName, string idKey, map<string, string> *pObjectMap, string *pErrorString) : bool
+pull(string tableName, string idKey, map<string, string> *pObjectMap, list< map<string, string> *> *pObjectList, string *pErrorString) : bool

Rationale:

- At the outset it was not apparent that the Storage subsystem required tableName and idKey data to be placed into a StorageObject (a new class, described in the System Decomposition section above). Putting this information in the map parameter was not consistent with its purpose. As such, tableName and idKey data was implemented to be passed as parameters.

- To reduce copying, the output parameter of the pull() function now contains a pointer to a map rather than passing it by value.

ServerController (Server Controller Subsystem)

OLD

ServerController
-storageRepo : Repository -auditor : AuditTimer
+create(in objectType : string, in pObjectMap : Map<string, string>*, out pReturnID : int*, out pErrorString : string*) : bool +push(in objectType : string, in pObjectMap : Map<string, string>*, out pErrorString : string*) : bool +pull(in objectType : string, in pFilterMap : Map<string, string>*, out pMapList : List<Map<string, string>*, out pErrorString : string*) : bool +setAuditTime(in auditTime : Time, out pErrorString : string*) : bool +getAuditTime(out auditTime : Time*, out pErrorString : string*) : bool

NEW

The ServerController class interface now includes two new functions:

+makeInstance(const string& timeString) : static bool

Singleton creator, called by the main() function to make the first instance of ServerController. Returns false if one already exists.

+getInstance() : static ServerController*

This static method allows access to the singleton instance of ServerController. Throws an exception if there isn't one.

The rest of the interface of the Server Controller class (ServerController) was changed as follows:

+create(string tableName, string idKey, map<string, string> *pObjectMap, int *pOutID, string *pErrorString) : bool
+push(string tableName, string idKey, map<string, string> *pObjectMap, string *pErrorString) : bool
+pull(string tableName, string idKey, map<string, string> *pObjectMap, list< map<string, string> *> *pObjectList, string *pErrorString) : bool

Rationale:

- At the outset it was not apparent that the Storage subsystem required tableName and idKey data to be placed into a StorageObject (new class, described in the Storage subsystem sub-section of the System Decomposition section). Putting this information in the map parameter was not consistent with its purpose. As such, tableName and idKey data was implemented to be passed as parameters.

- To reduce copying, the output parameter of the pull() function now contains a pointer to a map rather than passing it by value.

Repository (Storage Subsystem)

OLD

Repository
-Database* db
+bool pull (string table, StorageObject filters, list<StorageObject>* returnObjects)
+bool push (string table, StorageObject values)
+bool create (string table, StorageObject values, int &uid)
+bool runAudit ()

NEW

The interface of the Server Controller class (ServerController) was changed as follows:

+pull (StorageObject filters, out list<StorageObject>* return) : bool
+push(StorageObject values) : bool
+create(StorageObject values, int &uid) : bool

The 'table' string for these functions was folded into the StorageObject class.