Proprosal of 4/11/2003

test-area-c has an elaboration of the current framework in the main client tree.

A Rough description of Framework for EditArea, patient/gmGP widget , ../business/gmClinicalPart interaction:

CONCEPT: Base classes provide utility and basic functionality :

e.g.

a. gmEditArea

_init_data(self):

override for custom data initialization; default is spaces for text fields, and zeros for numeric fields.

_save_data(self):

override for custom persistence.
The pattern here is
if self.getDataId():
# do create new entity
else:
# do update entity with id = self.getDataId()

delete_data(self):

ovveride for custom delete of an entity
The pattern here is
if self.getDataId() <> None:
# do delete entity where id = self.getDataId()

getInputFieldValues()

setInputFieldValues(map, id)

map is keyed by input field keys, to values to be put into fields. Id is the data id of this current entity, default is None , which means a new entity.

Map = getInputFields()

gives a copy of mapping of the input fields.

Map.keys() will give the keys needed for setInputFieldValues(map)

(fields, formatting, values) = get_fields_formatting_values():

gives default field list , a map of formatting keyed by elements of field list, and a map of current values keyed by elements of field list.

edit area responsibilities:

- will call the domain objects necessary for create/update/delete.

b. Patient/gmGPBlah_Blah, inheriting from gmPatientHolder

these are the container modules for gmEditArea.

Usually they will manage some sort of container (multiple entity) view of instances of entities edited by gmEditArea . Each one is custom, so the interface is fairly informal.

- inherits gmPatientHolder.

this wraps the logic for detecting when a different patient is selected.

- it holds a model reference to the top level business model gmCurrentPatient

- using this , it provides wrapper methods to access the useful child business objects. e.g.
  Self.get_past_history() will return a gmPastHistory business object which has been constructed for gmCurrentPatient.

  e.g.2. Likewise self.get_allergies()

  self.get_family_history(), self.get_prescriptions()

  note, these are business objects, not collections of entities.

- e.g. Self.get_allergies().get_allergy_items() returns a id indexed
- collection of allergies data maps belonging to this patient.
-
- e.g.2 self.get_past_history().get_significant_past_history()
- gets the non-active , significant past history items of this patient, as a collection of past history data maps keyed by id.
- The individual data maps are usually keyed with values that match an edit areas keyed input fields, (handled by mapping logic within the business object) , as well as perhaps other relevant data items.

e.g.
(gmPatientHolder instance) .get_past_history().get_significant_past_history()
will return a indexed map
{ id:map1, id2:map2, id3:map3 ... } where id is an integer
and map1 would contain items such as map1['condition'], map1['year'] ,
map1['notes1'] but also map1['notes'] .
By matching the keys between the collection of (id, map) returned by collection accessors of business objects, with the keys of the editarea, selection for update is simplified (just copy the map values of the selected map into the edit area, along with it's id).

Hence implementations overide
_updateUI(self):
and call the business/gmXXXXX entity managers, collection methods.
This provides a link from when the patient changes to when the displayed lists are changed in a gmGP widget.

The business gmClinicalPart derived objects use the external references in
self.id_patient()
self.id_health_issue()
self.id_episode() of gmClinicalPart .

for selecting the relational data relevant for the patient.

WALKTHROUGH OF ID_PATIENT DATA FLOW:

The flow of the patient id goes like this:

gmPersonSelect selects a person fires patient_activating and patient_selected signals.

patient_activating is picked up by all edit areas;
a comparison is made within the edit areas of the current input field values to an 'old' original input field value set (created by setInputFieldValues() : this is assumed non-dirty (not from the UI user input , but from the database)
If new is different from old, then the edit area data is unsaved dirty data, and the actual save_data is called

- invokes either create_xxxxx or update_xxxxxx on the relevant business object.

patient_selected is next picked up.
GmEditArea responds by init_data() , clearing input fields and setting original data to blanks.

gmGP_xxxxx responds, by calling
gmCurrentPatient(kwds['id']) . Kwds is the parameter sent by the patient_selected signal.

This constructs the top level gmCurrentPatient business object, and also the gmClinicalRecord object, and also the gmClinicalPart instances (which are derived from gmClinicalPart) .e.g. Business/gmPastHistory business/gmAllergies, business/gmPrescriptions

The constructor parameters get the id_patient from the gmCurrentPatient constuction, and , currently, gmCurrentPatient constructs new or retrieves old contexts of id_episode , id_encounter , (and possibly id_health_issue, please ask the implementor; all I know, is that it works), depending on the time And compared to last currentPatient entry in the database.

GmClinicalPart allows access of these needed identifiers through id_patient() , id_encounter() , id_episode(),
(through "friend" querying the gmCurrentPatient attributes (C++ exposing one's guts; forget it, just a joke))

The important thing, is that one can use methods on patient/gmPatientHolder to get business objects via methods such as get_past_history().

O-R (MANUAL) MAPPING ONTO CLIN_ROOT_ITEM
The business objects inherit from gmClinicalPart , which provides the relational id mapping for clin_root_item, (id_episode, id_encounter), and subclasses of gmClinicalPart can get the clin_root_items relavant to id_patient , for further transforming or whatever. If not accessing a relational data source, they have the id_patient identifier to start off with.
The business objects can then build collections (currently mapped by relational table primary key id) of data items relevant to that part of the business model, and provide access through collection methods e.g. For clients such as gmGP_xxxxx which displays the collections.

COORDINATION OF CONSTRUCTION:
on receiving the patient_selected signal,
GmPatientHolder first builds gmCurrentPatient and all its sub business objects, then calls gmGP_XXX.updateUI() , so that gmGP_xxxx widgets will have a constructed model from which to query for collections, and update its views.

I gather gmCurrentPatient is a Borg of some sort (for those trekky fans), so it has the same state shared amongst any instances of gmCurrentPatient.