Proposed MIDAD design.

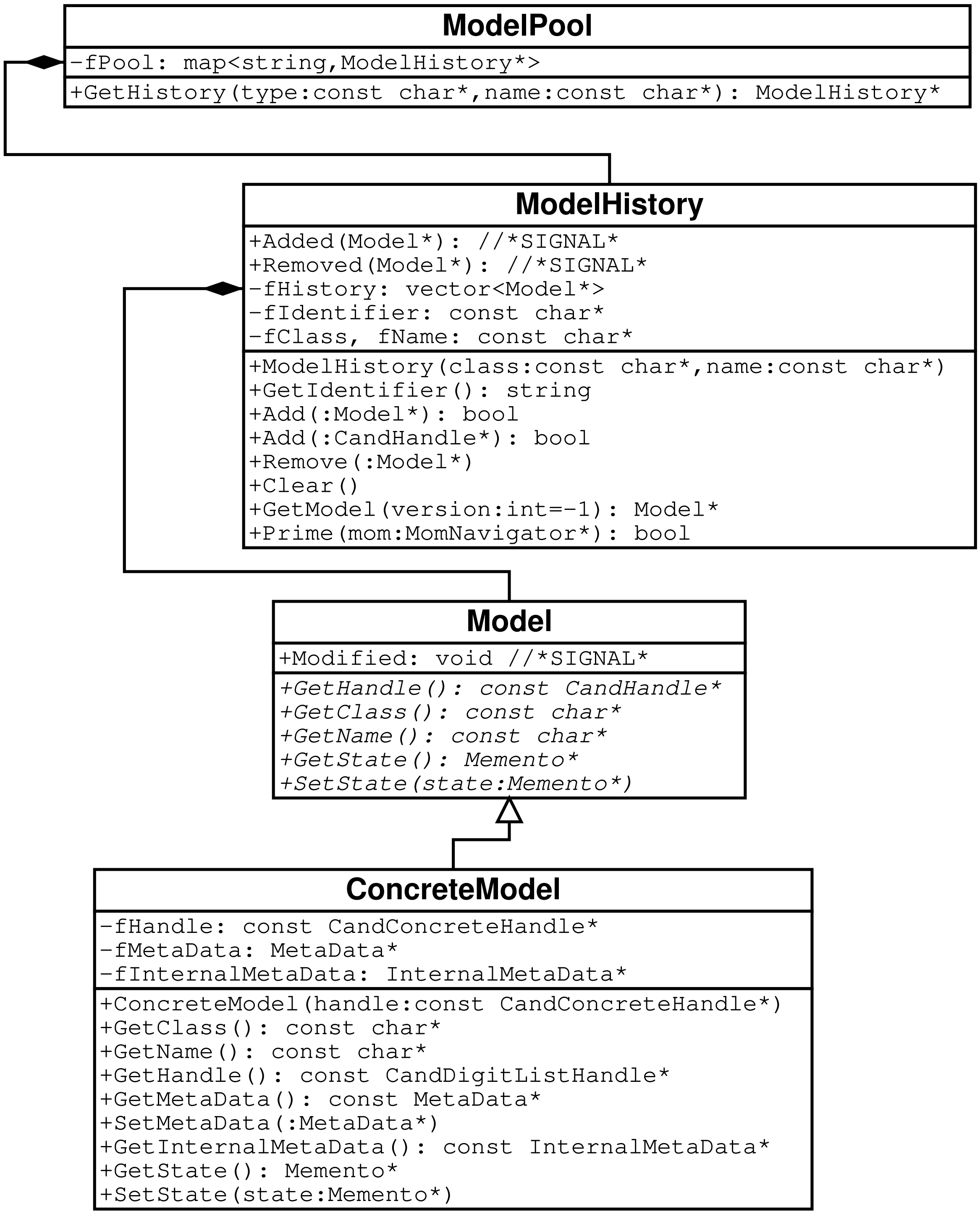
- Models.
- Views.
- Commands.
- Event control.
- Mark's GmsMenu.
- Conclusions.

http://minos.phy.bnl.gov/∼bviren/minos/software/

Overview of MIDAD Framework.

- Based on Model-View-Controller (MVC) pattern.
- A sub-framework with loose coupling to base framework, base framework should have few if any dependencies on MIDAD framework.
- Uses ROOT for graphics.
- Only proposing core classes (so far).
  - Models.
  - Views.
  - Control mechanism.
  - Event control and access to JobControl.

Models UML


ModelPool
-fPool: map<string,ModelHistory*>
+GetHistory(type:const char*,name:const char*): ModelHistory*
ModelHistory
+Added(Model*): //*SIGNAL*
+Removed(Model*): //*SIGNAL*
-fHistory: vector<Model*>
-fIdentifier: const char*
-fClass, fName: const char*
+ModelHistory(class:const char*,name:const char*)
+GetIdentifier(): string
+Add(:Model*): bool
+Add(:CandHandle*): bool
+Remove(:Model*)
+Clear()
+GetModel(version:int=-1): Model*
+Prime(mom:MomNavigator*): bool
Model
+Modified: void //*SIGNAL*
+GetHandle(): const CandHandle*
+GetClass(): const char*
+GetName(): const char*
+GetState(): Memento*
+SetState(state:Memento*)
ConcreteModel
-fHandle: const CandConcreteHandle*
-fMetaData: MetaData*
-fInternalMetaData: InternalMetaData*
+ConcreteModel(handle:const CandConcreteHandle*)
+GetClass(): const char*
+GetName(): const char*
+GetHandle(): const CandDigitListHandle*
+GetMetaData(): const MetaData*
+SetMetaData(:MetaData*)
+GetInternalMetaData(): const InternalMetaData*
+GetState(): Memento*
+SetState(state:Memento*)

Models Detailed.

Model:

- 1-to-1 corespondence between Concrete Models and CandHandles.
- A Concrete Model can have specific meta data.
- Does not modify its underlying Candidate.
- Identified by its CandHandle's type and name.
- If appropriate a Model can give and receive a Memento (state) (implement with streamers?).
- Emits "Modified" signal when its internal state changes.

ModelHistory:

- Stores history of one type and name of Model.
- Owns the Models it stores.
- Emits "Added" and "Removed" signals.

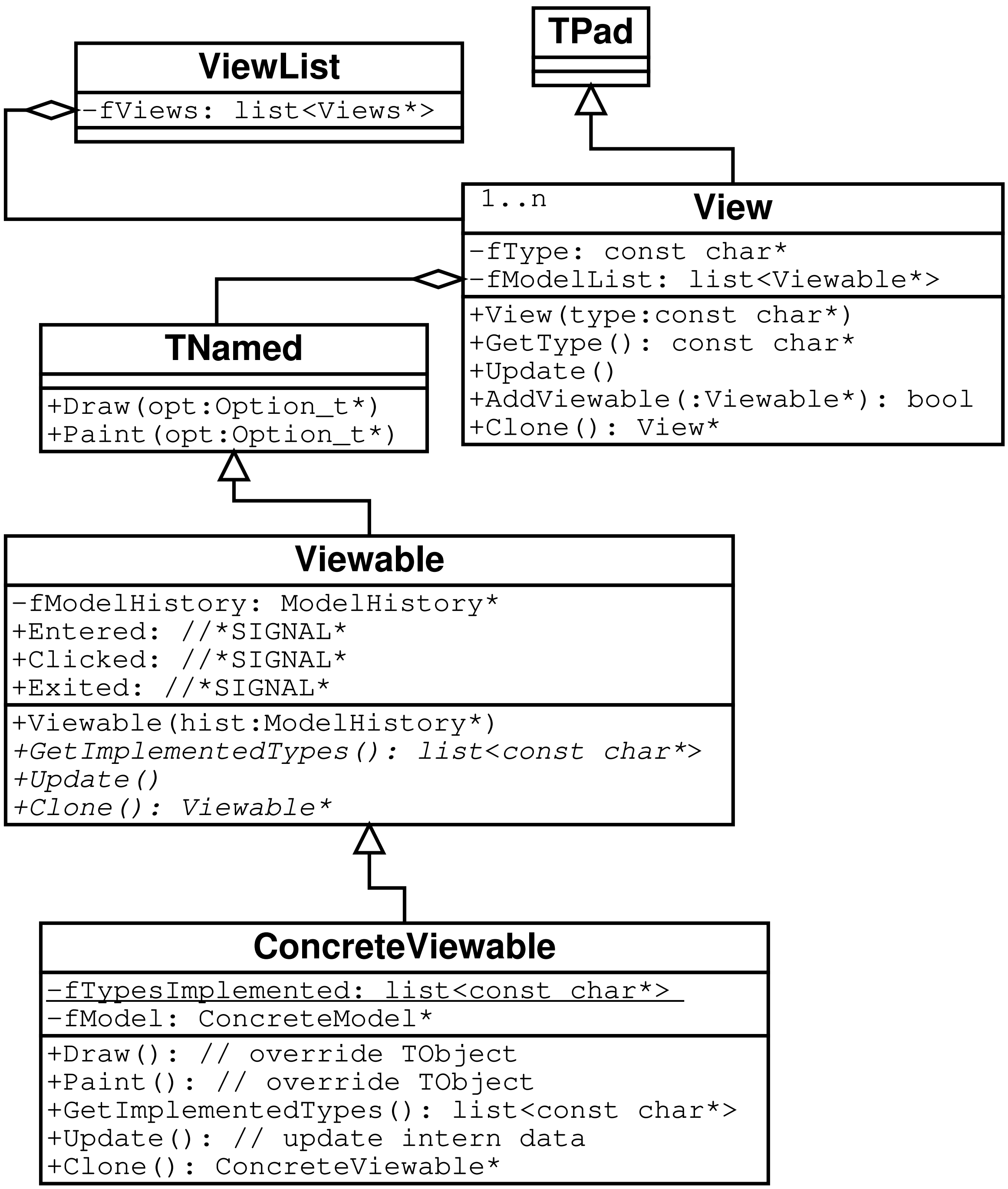
ModelPool:

- Holds (and owns) all existing ModelHistorys.

Uncertainties about Models

- I don't like the ModelHistory as is. I would like to change it to better fit with the Command mechanism. Possibly keeping the history in the CommandProcessor undo stack. (discussed later). Moving history into the base framework (into CandHandle?).
- As is, a Model treats a CandHandle as 'const'. Any change to a underlying Candidate is done by an external object (Algorithm) and a new Model is formed with a new CandHandle and added to the associated ModelHistory. It seems to me that event display should be, in some sense, treated like an interactive Algorithm, and thus be able to modify a (copy of a) Candidate. I am not sure how to fit this in.

Views UML


TPad
ViewList
-fViews: list<Views*>
1..n
View
-fType: const char*
-fModelList: list<Viewable*>
+View(type:const char*)
+GetType(): const char*
+Update()
+AddViewable(:Viewable*): bool
+Clone(): View*
TNamed
+Draw(opt:Option_t*)
+Paint(opt:Option_t*)
Viewable
-fModelHistory: ModelHistory*
+Entered: //*SIGNAL*
+Clicked: //*SIGNAL*
+Exited: //*SIGNAL*
+Viewable(hist:ModelHistory*)
+GetImplementedTypes(): list<const char*>
+Update()
+Clone(): Viewable*
ConcreteViewable
-fTypesImplemented: list<const char*>
-fModel: ConcreteModel*
+Draw(): // override TObject
+Paint(): // override TObject
+GetImplementedTypes(): list<const char*>
+Update(): // update intern data
+Clone(): ConcreteViewable*

Views Detailed.

View:

- "Is a" TPad.
- Has associated view type (U vs. Z, 3D, etc)
- Has a list of Viewables impl. view type.
- Can be deep cloned.

Viewable:

- "Is a" TNamed.
- Graphical representations of concrete Model.
- Implemented by overriding `TObject::Paint()` and/or `TObject::Draw()`
- Has a list of view types is can implement.
- Emits signals when pointer enters, exits or clicks.

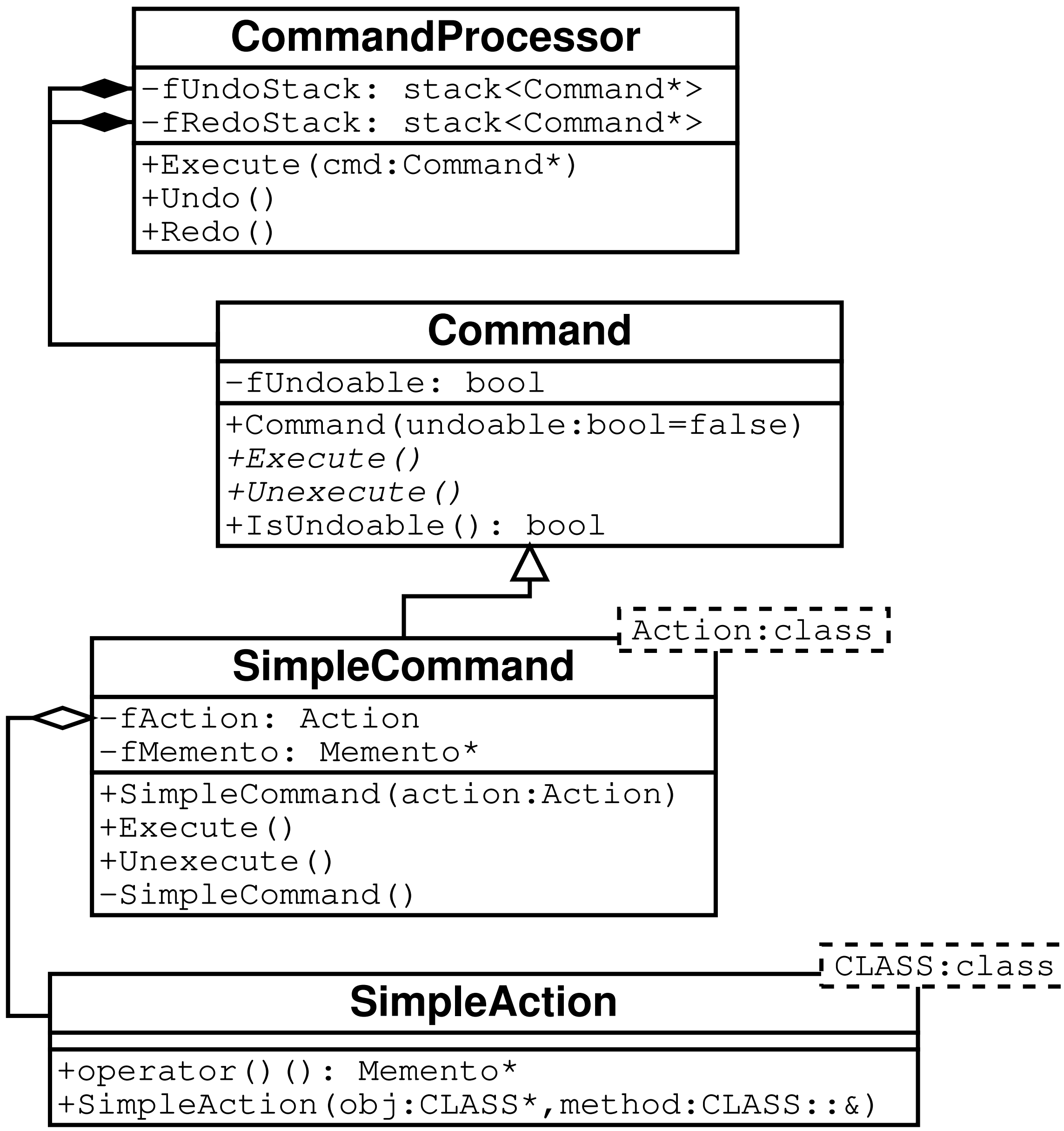
ViewList:

- Holds a list of all existing Views.

Additional View Functionality

- A mechanism like JobCModuleRegistry is needed to store the names, view types and a creator for each Viewable which is known to the system. Should work with explicit dynamic library loading.
- A uniform way for a View to export important actions to the outside world (ie, selections, drawing attributes of Viewables, etc).
- An external "Layers" mechanism for overlaying multiple like (and unlike?) Views.
- An external history mechanism (maybe using a Tab or Notebook widget) to allow easy switching between different versions of the data.

Commands (Controlling mechanism)



Commands Detailed.

Anything modifying a Model should be done via a Command executed by the CommandProcessor.

- A CommandProcessor executes a command and maintains a stack of undo and (possibly) redo commands.
- A Command encapsulate some action.
- A SimpleCommand is a concrete example of a command which takes some functional object with no args and returns a Memento.
- A SimpleAction is an action which calls `Model::GetMemento` and then calls the given method.

Uncertainties:

- How do Commands and ModelHistory interact. Redundant info?
- Not shown how to store and replay Commands, but not forseen to be a large challenge.

EventControl UML

EventController
-fMom: MomNavigator* -fJC: JobController* -fDefaultPath: string +EventChanged: void //*SIGNAL*
+Instance(): EventController& +NextEvent() +PrevEvent() +RunPath(name:string=0) +SetDefaultPath(name:string) +GetMom(): const MomNavigator*

EventControl Detailed.

- Holds a MomNavigator.
- Interfaces with JobControl.
- Provides interface to navigating events (next/prev).
- Provides interface to running JobPaths (maybe?)

Uncertainties:

- Should all interaction to JobControl be via JobMenu?
- If so, is this class even needed?

The JobMenu System

Description of the JobMenu system.

- Tree data structures, not a GUI.
- Decouples JobCModules and JobControl from outside classes.
- Lets Event Display and batch processing have same access to J.C.

Mark has proposed GmsMenu system:

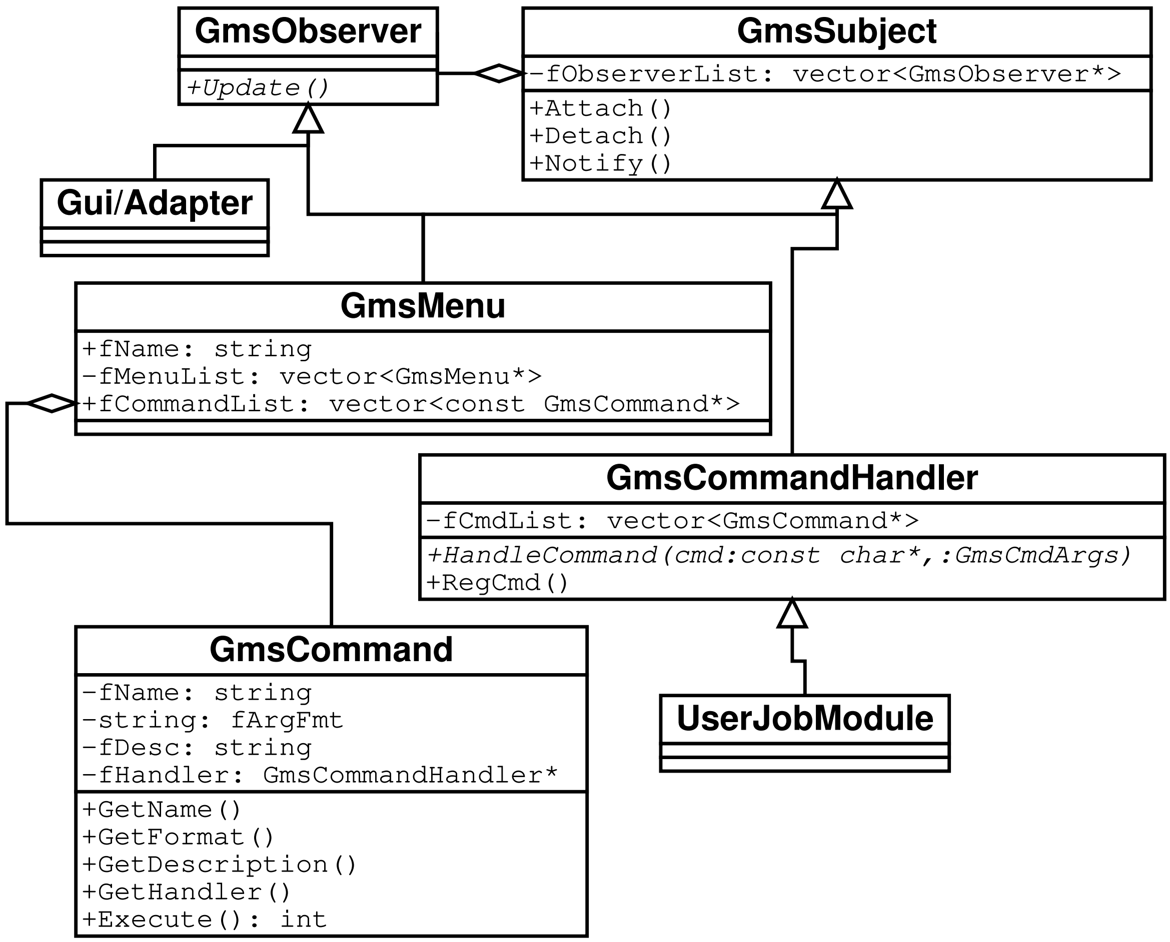
http://www.hepl.harvard.edu/∼messier/software/menu/

which uses Observer/Subscriber pattern. I have suggested instead to use signal/slot mechanism:

http://minos.phy.bnl.gov/∼bviren/minos/software/Menu.tar.gz

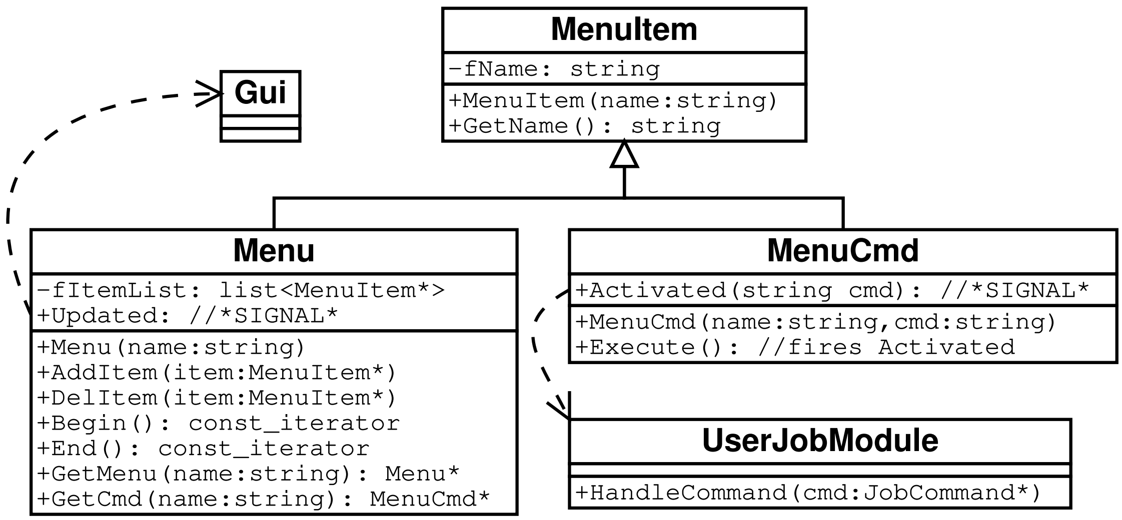
Otherwise, the two are similar.

Mark's proposed GmsMenu



- Gui depends on Menu or adapter needed.
- Subject/Observer notifies of changes.
- Menu has sub menus and commands.
- JobCModule is a cmd handler.
- `topMenu.GetSubMenu("submenu1")`
  `->GetSubMenu("submenu2")`
  `->GetCommand("command")`
  `->Execute();`

Menu system using signals



- Gui independent of Menu
- Menu holds MenuItems, (Menus and MenuCommands)
- Menu emits Updated when it or sub Menu is modified.
- MenuCommand emits Activated.
- `topMenu.GetMenu("submenu1")`
  `->GetMenu("submenu2")`
  `->GetCommand("command")`
  `->Execute();`

Conclusions.

- A rough design for the basic sub-framework exists.
- There is still some uncertainty on how best to design some aspects:
  - Model history? Wasn't there to be a history mechanism inside Candidate-land?
  - Should only Algorithms modify Candidates? Can Commands become interactive Algorithms somehow?
  - How to interface with JC? Completely via GmsMenu?
  - Can GmsMenu use Signals instead of explicit Publisher Subscriber pattern?
- I would like to get enough approval to start coding basic functionality ASAP, even if it means a rewrite later as I am worried that unknown implementation details will invalidate some design decisions.