Models and Views

Responsibilities for a typical GUI application

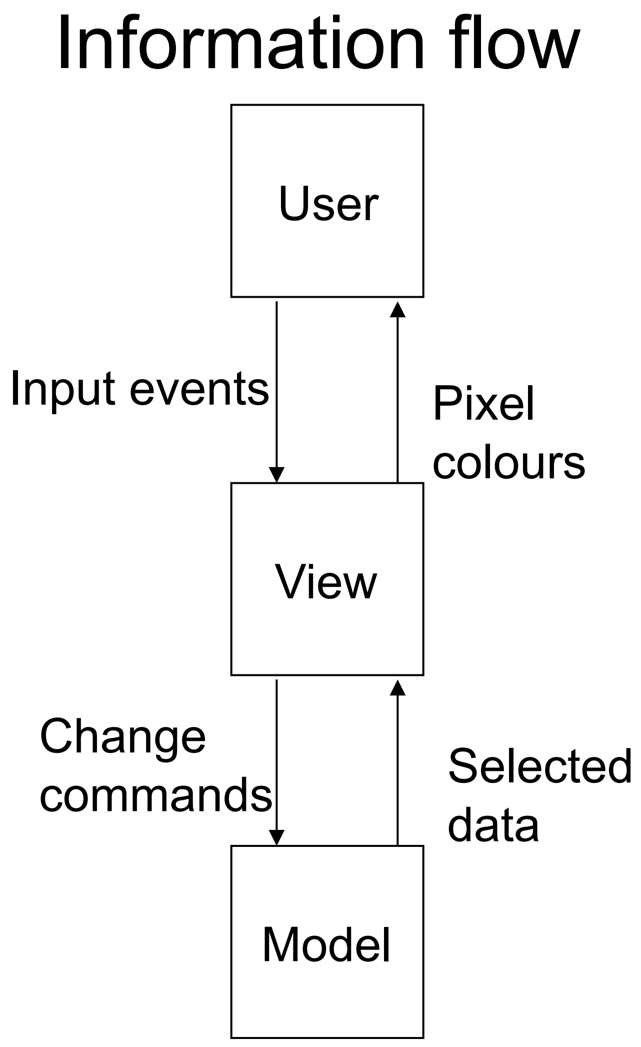
- Represent some data
- Receive input from the user
- Make changes to the data in response to input
- Present aspects of the data to the user

An example: An appointment calendar application

- Represent a set of appointments
- Receive commands to enter new appointments, delete appointments, change appointments
- Make changes to the set of appointments
- Present all appointments for a given day or week to the user.

Split the responsibilities

- Model
  - Store the data
  - Change the data
- View (a.k.a. Presenter)
  - Receive commands from the user (mouse clicks & key presses)
  - Present the data visually. (Translate to pixels)



Protection against change

- The model component is protected against changes in the GUI technology
- E.g. the same model component can be used for a desktop application and for a PDA version

Should we further split the View?

- Right now we are only looking at a top-level split.
- Furthermore in GUI applications, input and output are closely bound since the meaning of mouse clicks and key presses depends on where on the screen they are located and control of the screen is an output responsibility.
- The answer is: yes, but not yet.

Should we further split the model?

- Again the answer is: yes, but not yet.
- In many applications the storage function can be represented by classes that simply provide a conduit to a relational database
- The responsibility of sensibly changing the data is called the “business logic”
- Maintaining invariants on the data is the responsibility of the “business logic”.
- For example, the storage part may store dates as year, month, and day. The business logic must ensure that the day is not too big for the month.

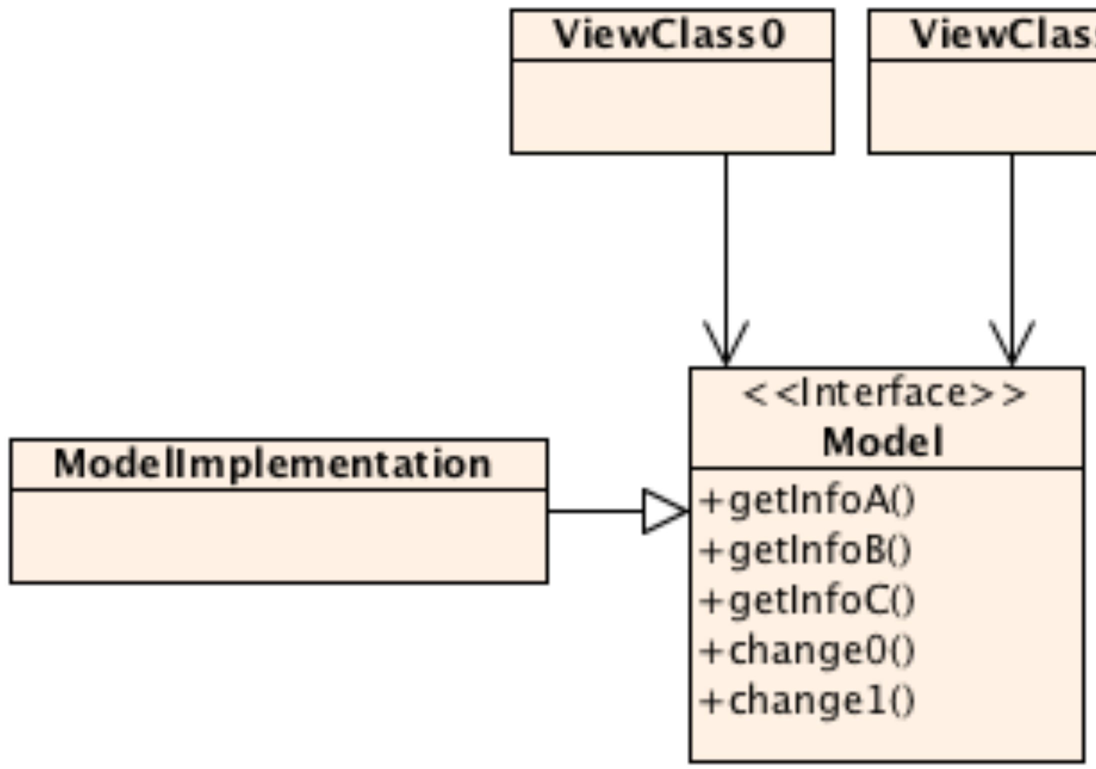
Dependence

- Since we want the model to be reusable it stands to reason that the model should not depend on the view.
- On the other hand, there seems to be no big problem with the view depending on the model.
- Never-the-less there is little harm in designing so that the view does not depend on the model class, but only an interface.

Push vs. Pull

- When information flows in the same direction as the method calls, we call it a “push”.
- When it flows opposite to the direction of the method calls, we call it “pull”.
- Since changes originate with the user and propagate to the view, it makes sense for the view to push changes to the model
- Since the model does not know what information the view needs, it makes sense for the view to pull the data it needs to display
- Luckily this means the method calls all go the same way.

This leads us to the following design

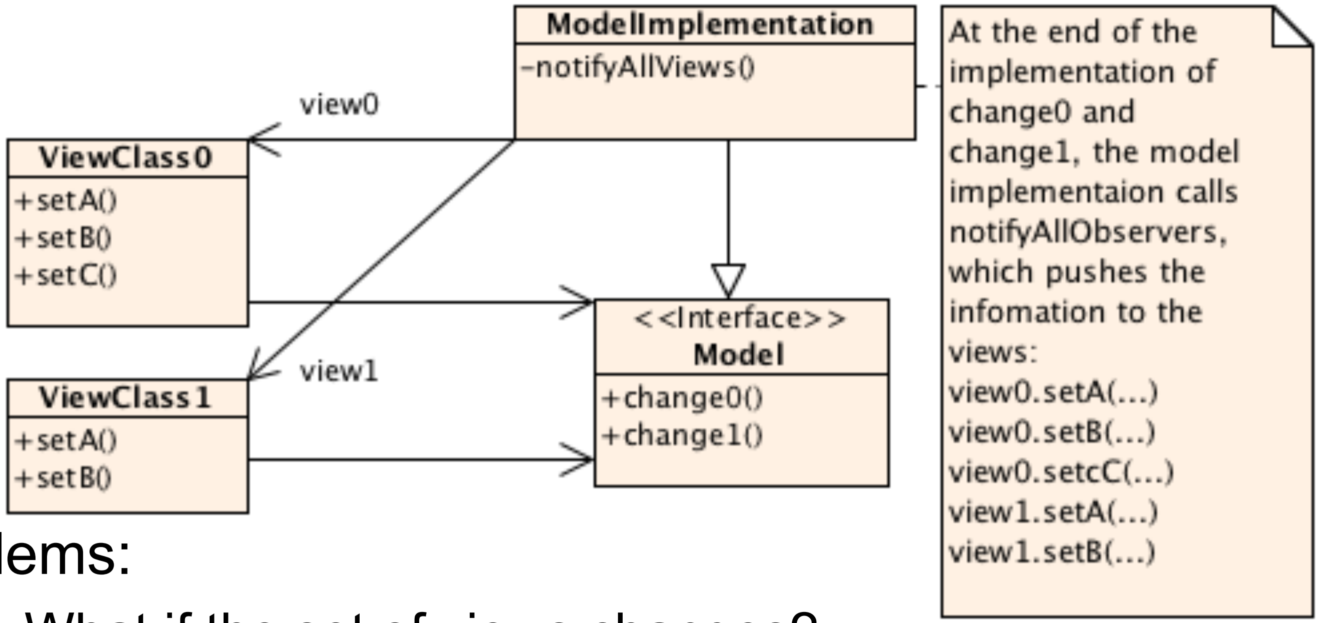


But Wait

- The view must be written to obtain all the data it needs after every change.
- This makes it hard to split the view into loosely connected (or unconnected parts)
- For example we might have a one-week view and a one-day view.
- We might have a "view" that ensures the desktop application is synchronized to the PDA application.
- There is little reason for these different views to know each other, but currently they must so that they all "refresh" themselves when any one makes a change.
- Furthermore the model might change in response to a message that does not come from any view, as when an appointment becomes "due"

Solution 0

- The model knows all the views.
- The model pushes changed information to the views



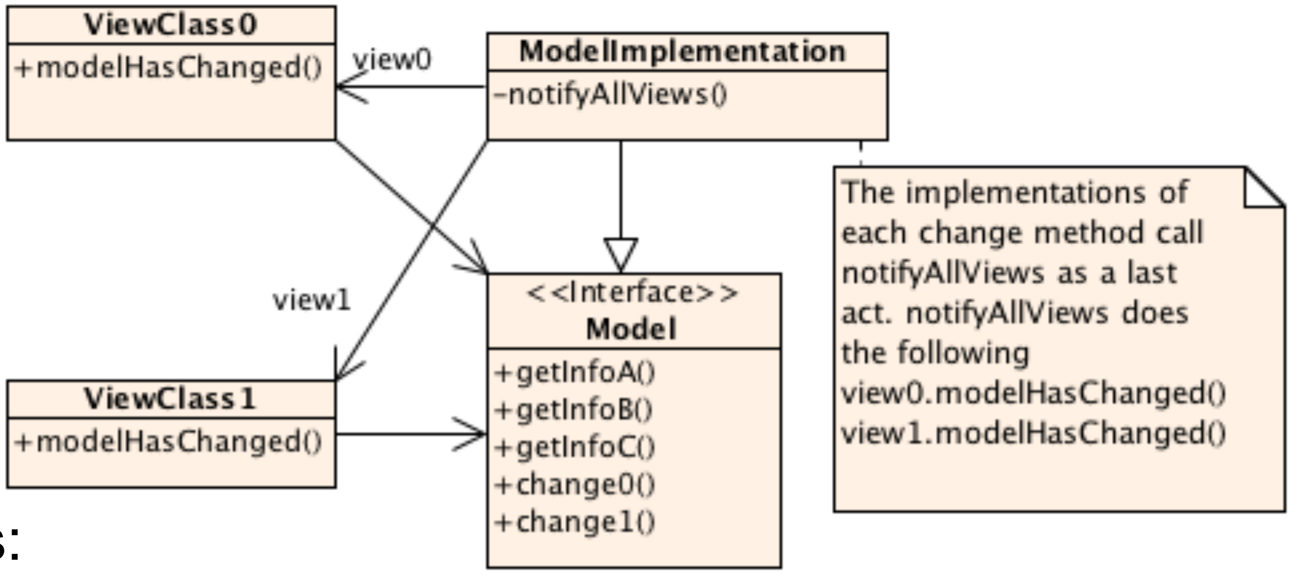
At the end of the implementation of change0 and change1, the model implementaion calls notifyAllObservers, which pushes the infomation to the views:
view0.setA(...)
view0.setB(...)
view0.setcC(...)
view1.setA(...)
view1.setB(...)

Problems:

- What if the set of views changes?
- What if the information needed by a view changes
- Either way, the model must be recoded.
- Poor design!

Solution 1

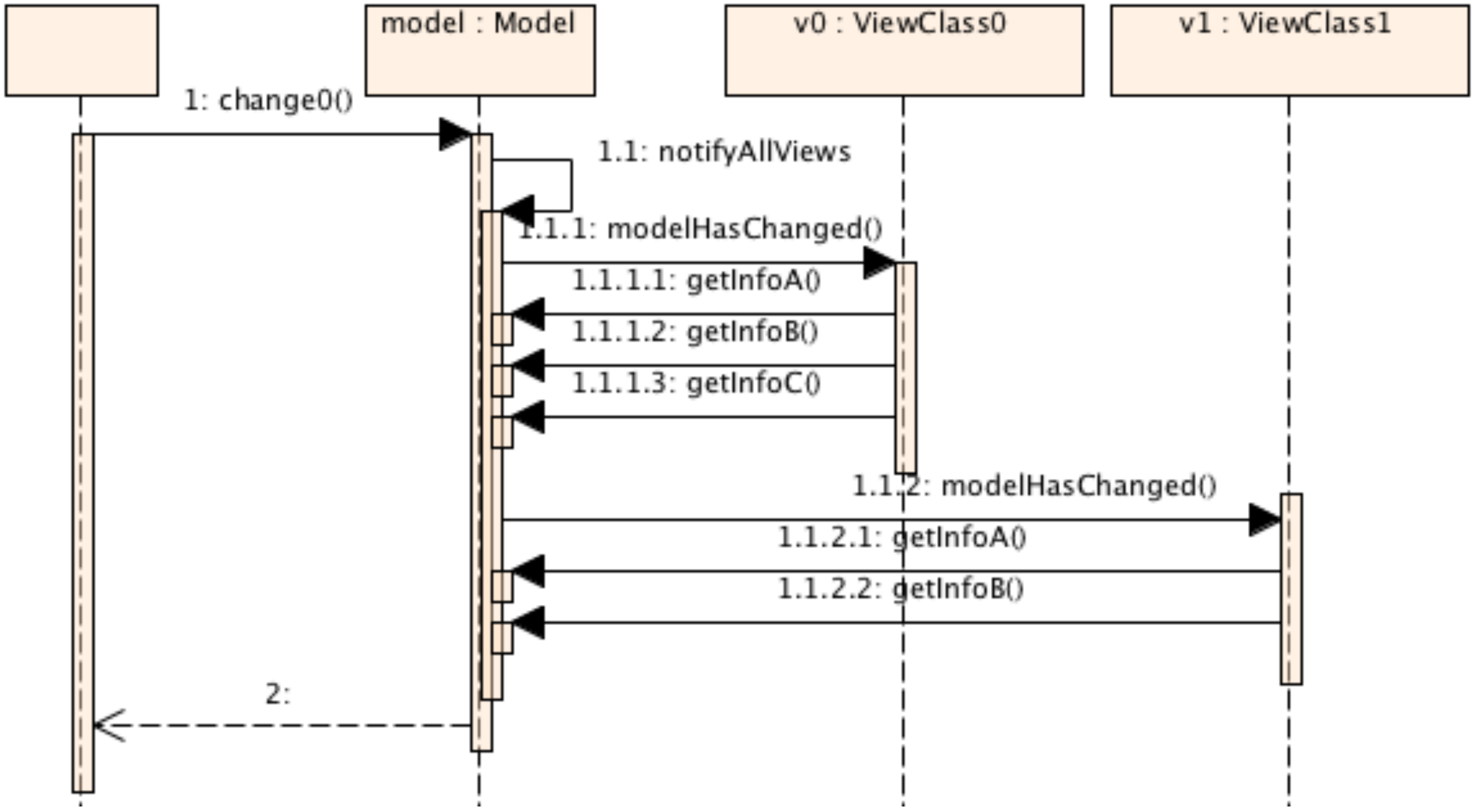
- The model knows the views
- But the views pull the information they need
- The model alerts the views that there was a change to the state.



Problems:

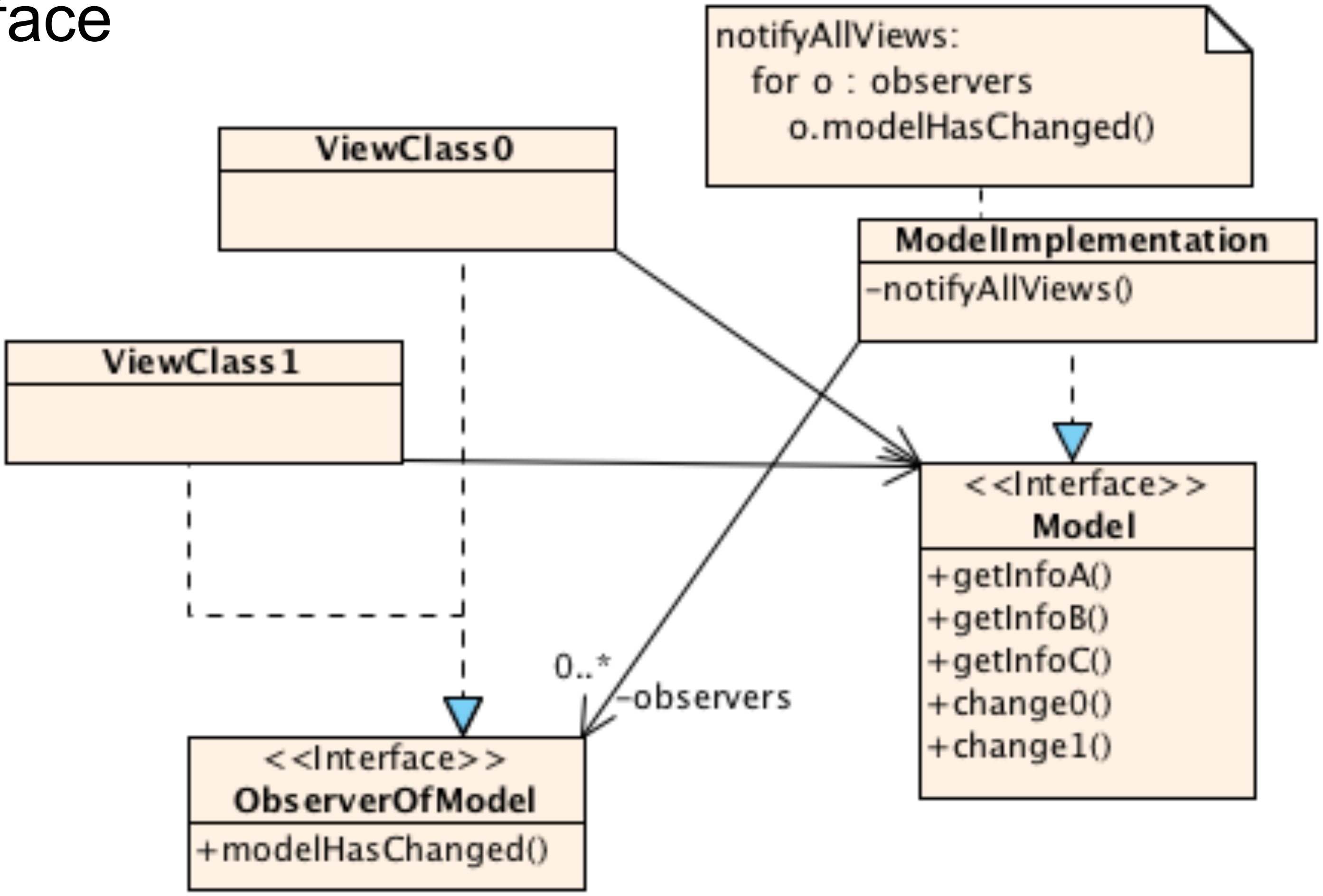
- What if the set of views changes?
- Either way, the model must be recoded.

Solution 1 & 2 call flow



Solution 2

Keep a list of observers, each implementing an interface



Observer Pattern

- As we will see later, this pattern of object relationships and interactions is called the “Observer Pattern”