CSSE2002/7023

Semester 2, 2021

Programming in the Large

Week 9.2: GUI Design — Model and View

In this Session

- GUI Design Principles
- Model and View Separation
- Call Back / Observer
- MVC / MVA / MVP / MVVM

Code Structure

There are lots of ways we could write GUI code so it “works.”

- but not all of them are good

We want to be able write large programs that are maintainable.

GUI Design Principles

- Each GUI class is a cohesive concept
- Restrict coupling between the GUI and the program logic

Model / View Separation[1]

Model

- Represents an entity in the system
- Stores state
- Has invariants
- Has methods to enforce the invariants

View

- A presentation of the state and (usually) a way to interact with it.

[1] Terminology from "Applying UML and Patterns" — Larman

Why?

- Decompose the task
- May want to be able change the interface
    - model only cares that a method is called, not which objects are involved
- You might want to support multiple interfaces.
    - text, vocal, remote network, multiple access panels, . . .
- Responsibility for enforcing invariants should be in one place, not many.
    - minimise duplicated code

How?

This is a bit more subtle.

1. View and the model need to be able to communicate.
   - interface needs to be able to find current state
   - model *may* need to tell interface when state changes

How?

This is a bit more subtle.

1. View and the model need to be able to communicate.
    - interface needs to be able to find current state
    - model *may* need to tell interface when state changes
2. . . . which means calling methods (in OO anyway).

How?

This is a bit more subtle.

1. View and the model need to be able to communicate.
   - interface needs to be able to find current state
   - model *may* need to tell interface when state changes
2. ...which means calling methods (in OO anyway).
3. ...which means they know each others' type?

How?

This is a bit more subtle.

1. View and the model need to be able to communicate.
    - interface needs to be able to find current state
    - model *may* need to tell interface when state changes
2. . . . which means calling methods (in OO anyway).
3. . . . which means they know each others' type?
    - Java won't compile if it doesn't know the method you're calling exists

How?

This is a bit more subtle.

1. View and the model need to be able to communicate.
   - interface needs to be able to find current state
   - model *may* need to tell interface when state changes
2. . . . which means calling methods (in OO anyway).
3. . . . which means they know each others' type?
   - Java won't compile if it doesn't know the method you're calling exists
4. . . . so no flexibility?
   How do we get around this apparent limitation?

How?

User of the interface[2] needs to get information and issue commands — which ultimately affect the model.

Interface needs to be aware if the model[3] has changed .

[2]Why not "view"? We'll get to that.

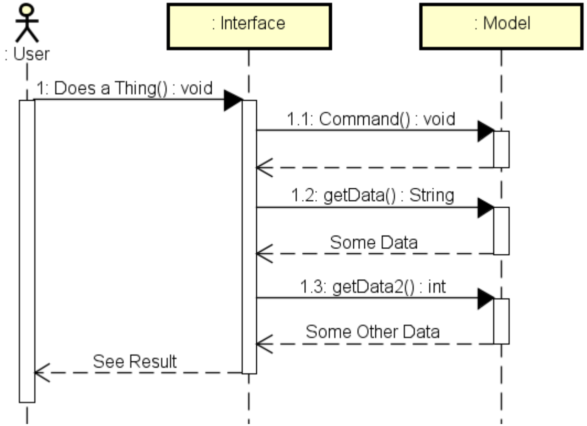
[3]Note the wording here, I haven't said that the model must tell the interface that something has changed.

One Way Access

Suppose:

1. All changes in the model are synchronous with commands from the interface.
2. Information required from the model is relatively small.

One Way: Interface $\rightarrow$ Model



Pro: Does not require the model to know about the view.

Con: Be very careful about drawing conclusions about large systems from a trivial example.

Model Changes

Suppose the model can change out of step with the interface.

- Changes take time?
- Multiple interfaces are connected to the model.
- Model reads info from the environment.
- ...

Possible strategies:

- Polling
  - Has it changed yet?
    Has it changed yet?
    ...
  - Generally frowned upon
- Call back / Observer
  - Model calls a method to notify something

Call Back / Observer

```
interface ThingThatCares {
    public void itHappened(Details details);
}
...
public class UserInterface implements ThingThatCares {...}
```

Similar to the way GUI events are handled.

Reduces coupling because although the model needs a reference to another object, it doesn't need to know anything about it other than the fact it has a particular method.

Interface *could* then ask for all the information from the model, or could have details of what changed in a parameter.

- Details will couple the interface to the model.

We are still assuming the model is "small."

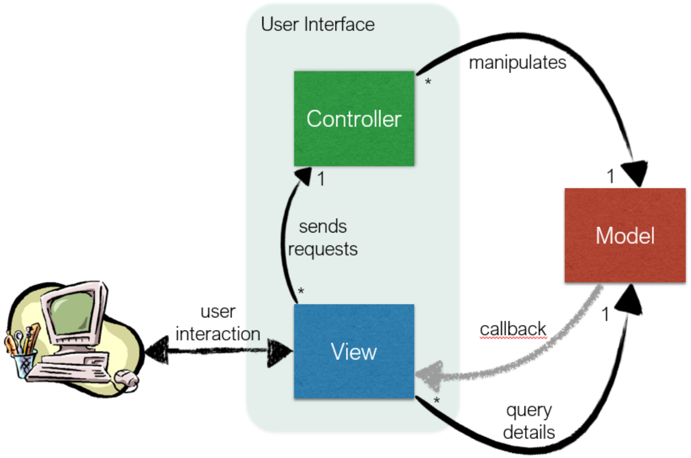
- Large model would require detailed change instructions.

Input Processing?

In the following, input processing could mean two things:

1. Getting values from the interface components to assemble a method call.
    - Probably belongs in the interface somewhere (since the interface knows what components are there).
    - If the processing to work out what to do is very complex, then maybe it needs to be in a separate class.
2. Making changes to the model based on that call.
    - Belongs in the model

Model – View – Controller

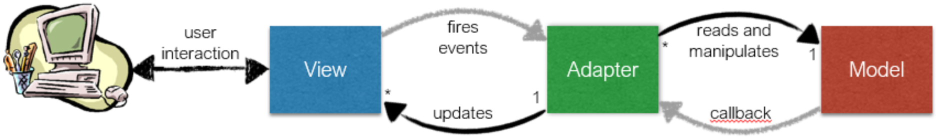


View displays information and the controller processes input. Together, they make up the interface.

Model doesn't need to know about view, aside from the callback.

Works well when your GUI design and programming are separate.

Model – View – Adapter



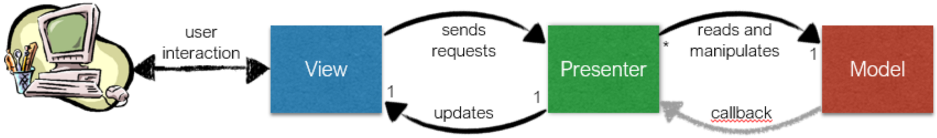
Isolate the view from the model by using an adapter.

View and model only need to know about the interfaces implemented by the adapter.

- In theory adapter can work with multiple views
    - less common in practice

Allows for richer user interactions.

Model – View – Presenter



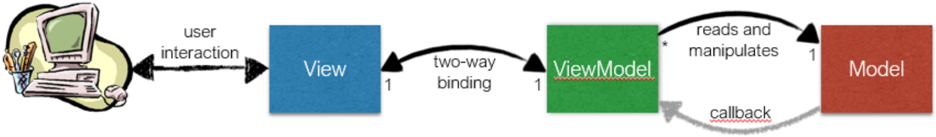
Isolate the view from the model by using a presenter.

View knows about the presenter.

Model only needs to know about the interface implemented by the presenter.

Clear separation of interface oriented logic (*presenter*) from system logic (*model*).

Model – View – ViewModel



ViewModel encapsulates view state and logic.

Controller behaviour provided by ViewModel logic and a data binding library.

- See use of properties and bind method in View and ViewModel classes in assignment.

Simplifies testing of UI logic.

Less flexible than MVC.

- Can't easily have multiple different views for a ViewModel.
- See inventors comments of pros & cons.

Summary

“Observers” reduce coupling[4].

“Observers,” or other call-back mechanisms, allow for model updates at “unexpected” times.

Adapters permit more ignorance about what is on the other end.

These points are not limited to GUIs.

MVC, MVA, MVP and MVVM diagrams adapted, by permission, from Stephan Rauh’s Model-View-Whatever.

[4] In that less needs to be known about the other object.