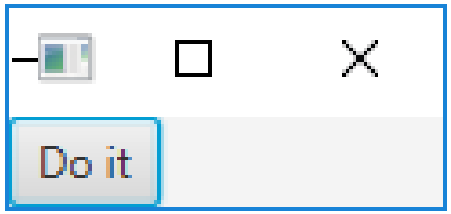
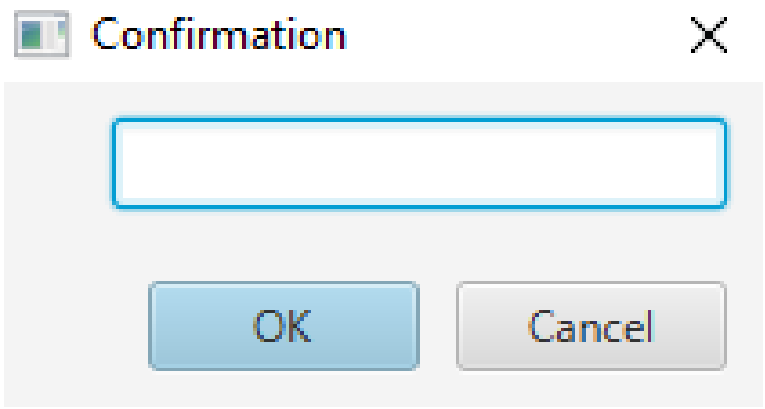
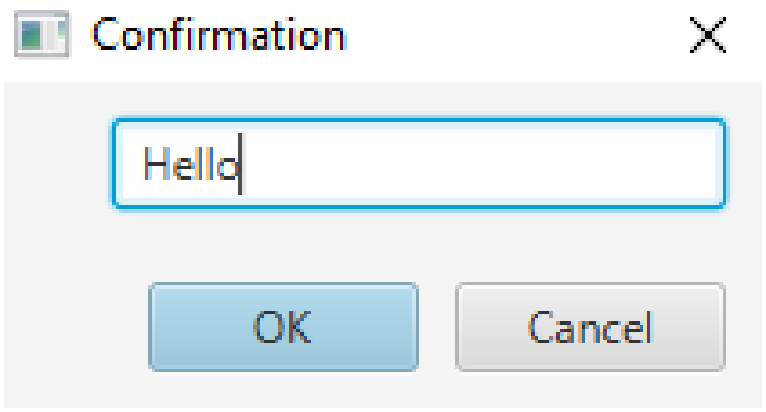
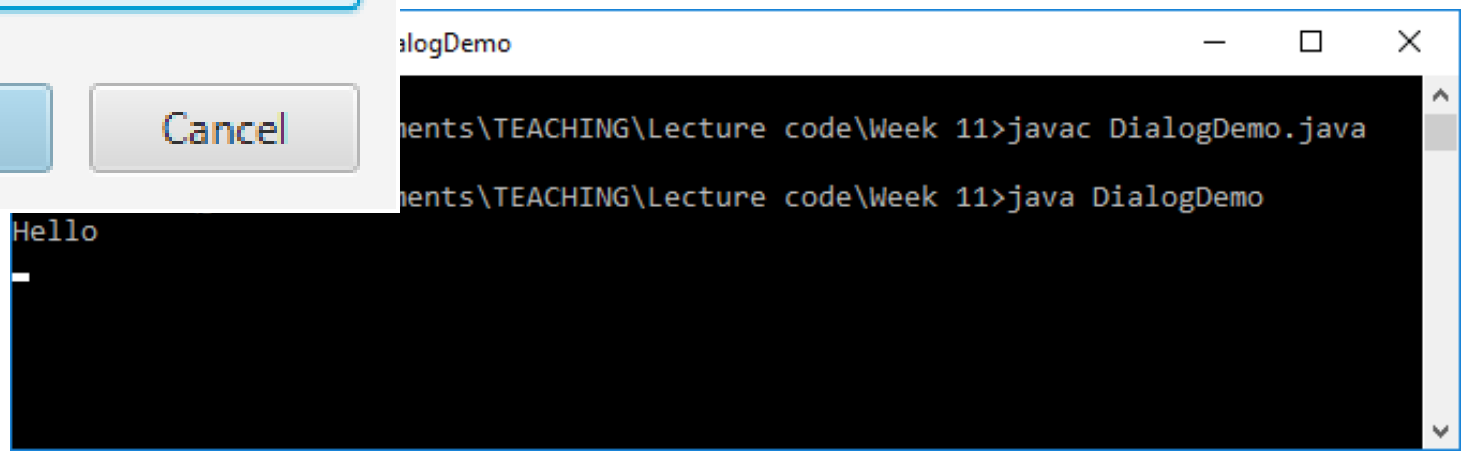
Week 11.1 — GUIs
Model and View

DialogDemo.java









Terminology — Cohesion

Cohesion:

- How well do the parts of the class (state and methods) group together.
- Do they all contribute to a clear purpose?

Eg: Car class contains:

- Fuel
- Steering
- Speed
- Route planner
- Public holiday calculator

High cohesion is preferable.
(Low cohesion may indicate that a class should be split).

Terminology — Coupling

Coupling:

- To what extent does this class depend on other classes?
- How many methods need to be called on how many other classes?

Low coupling is preferable.

- High coupling may indicate that a class has been split when it shouldn't have been.
- Classes coupled to lots of other classes are harder to write and test in isolation.

Note: both coupling and cohesion are relative terms.

Code Structure

There are lots of ways we could write GUI code so it "works" but not all of them are good.
Remember that we want to be able write larger programs and still have them be maintainable.

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.

Note: each category may have many classes in it.

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

Why?

- Decompose the task
- May want to be able change the interface
  - The model only cares that `.fight()` 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 duplicate code.

How do we achieve the separation?

This is a bit more subtle.

1. The 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 states change

How?

This is a bit more subtle.

1. The view and the model need to be able to communicate.
2. . . . which means calling methods (in OO anyway).

How?

This is a bit more subtle.

1. The view and the model need to be able to communicate.
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. The view and the model need to be able to communicate.
2. . . . which means calling methods (in OO anyway).
3. . . . which means they know each others' type?
4. . . . so no flexibility?
    - How do we get around this apparent limitation?

How?

The user of the interface[2] needs to get information and issue commands — which ultimately affect the model.
The interface needs to be aware if something has changed in the model[3]

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

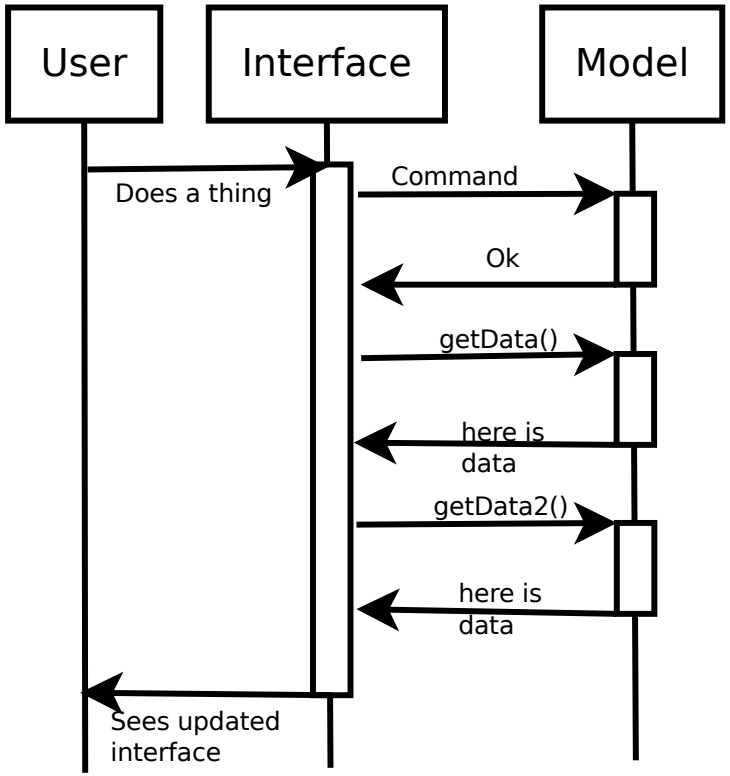
[3] Note the wording here, I haven't said that the model must **tell** the interface that something has changed.

One way access

Suppose:

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

One way: Interface → Model



This sort of structure does not require the model to know about the view.

Potentially multiple calls to model to get information about model's state.

Not a great example of the flexibility required by GUIs.
Be very careful about drawing conclusions about large systems from a single window.

Model changes

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

- Changes take time?
- There are multiple interfaces connected to the model.
- The model reads info from the environment
- ...

Possible strategy:

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
```

So, 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.
The interface **could** then ask for all the information from the model, or have details of what changed in a parameter.
So we could still be assuming the model is "small"; a large model would require detailed change instructions.

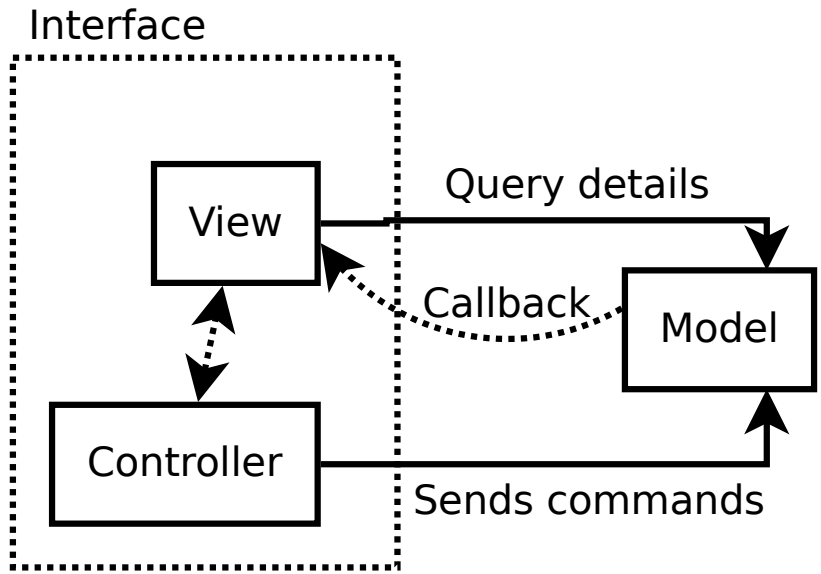
Input processing?

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

Possible arrangements

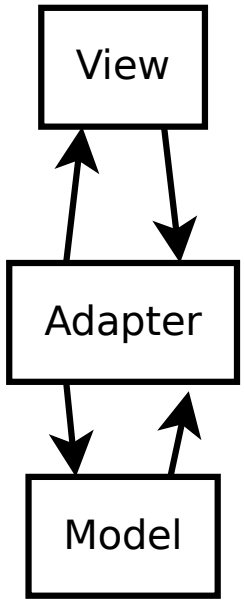
Model+View+Controller:



In this arrangement, the view gives you information and the controller processes input. Together, they make up the interface. The model doesn't need to know about the view, aside from the callback.
This can be good when your GUI design and programming are separate, or in a client-server model.

Possible arrangements

Model+View+Adapter:



In this arrangement, you isolate the view from the model by using the adapter. The view and model only need to know about methods in the adapter. This allows for richer interactions and multiple interfaces.

Summary

The following is not limited to GUIs:

- "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.

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