Advanced Programming 2

DR. ELIAHU KHALASTCHI

2016

Architectural Patterns

DR. ELIAHU KHALASTCHI

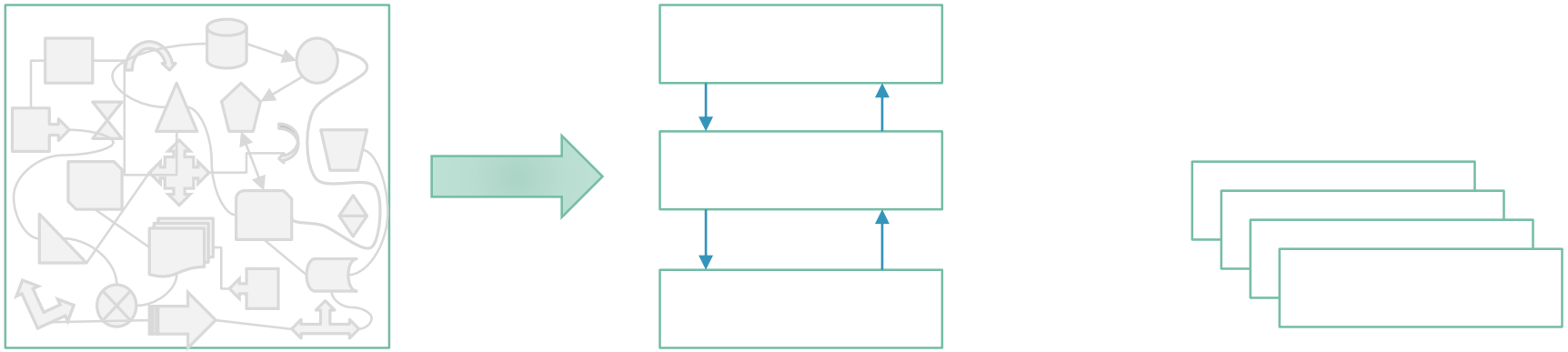
What are architectural patterns?

- Patterns that have a broader scope than design patterns
- Involve the architecture of our software

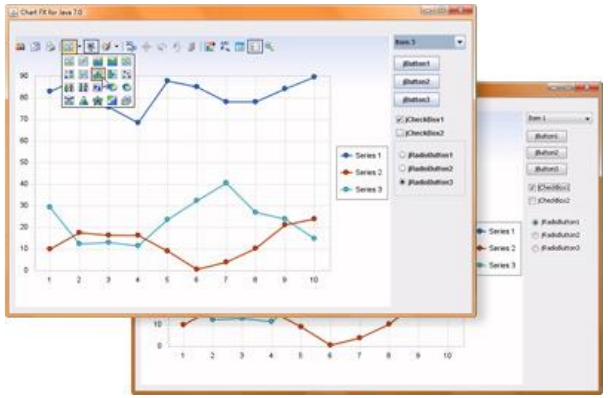
- There are several well-known architectural patterns
- We are going to learn about two:
  - MVC, MVC + Observer Design Pattern,
  - And later, the MVVM architecture, which is dedicated to the WPF technology of Microsoft

Dividing the code into layers

- We do not want to implement everything in 1 layer of code…
  - When something changes, everything has to be changed
- Instead, we want to divide the code into different layers
  - The code is modular
  - Different teams can work independently parallel to each other
  - Easier to trace and isolate bugs!

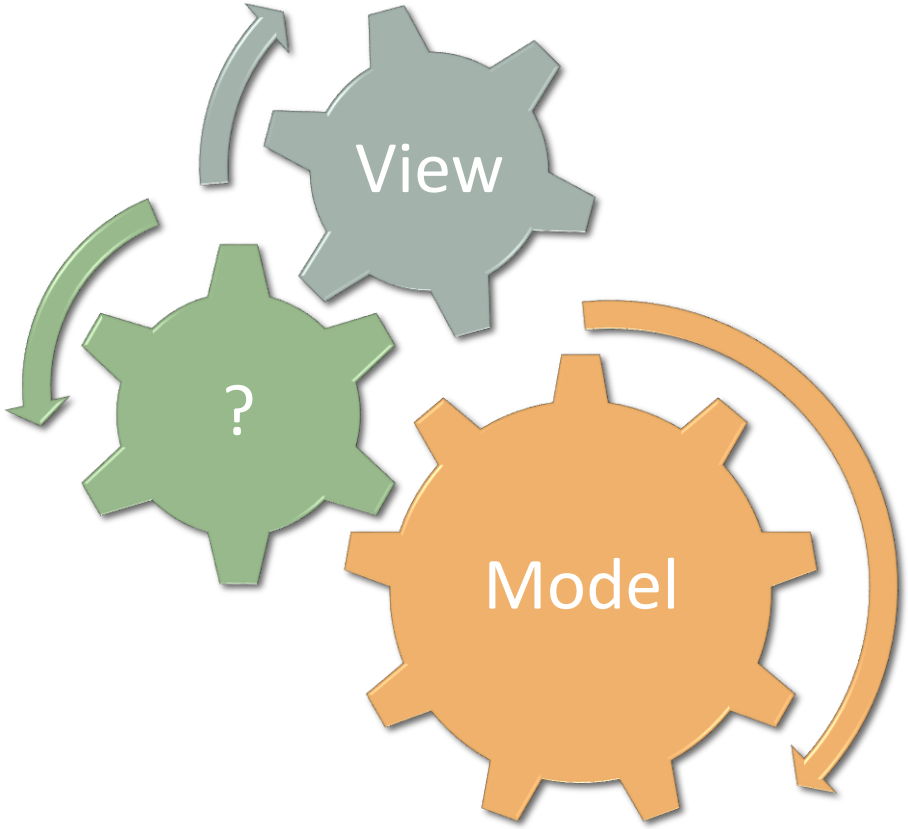


Separation of the **Model** and the **View**

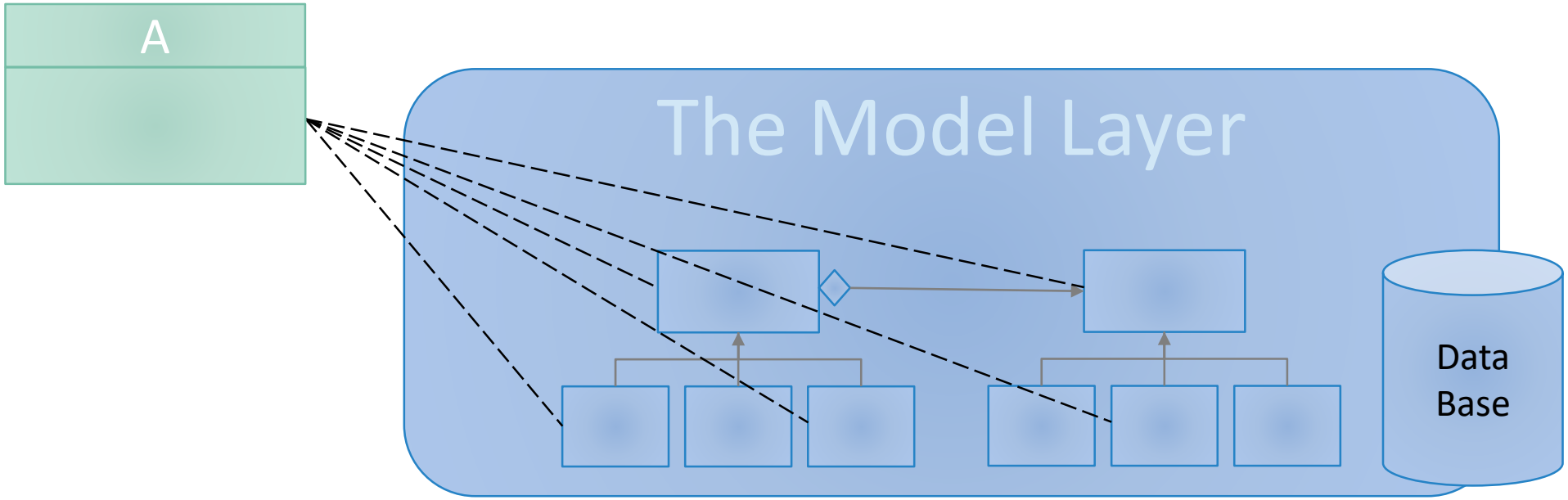


These layers should not “know” each other!

```
procedure bubbleSort( A : list of sortable items )
  n = length(A)
  repeat
    swapped = false
    for i = 1 to n-1 inclusive do
      /* if this pair is out of order */
      if A[i-1] > A[i] then
        /* swap them and remember something changed */
        swap( A[i-1], A[i] )
        swapped = true
      end if
    end for
  until not swapped
end procedure
```

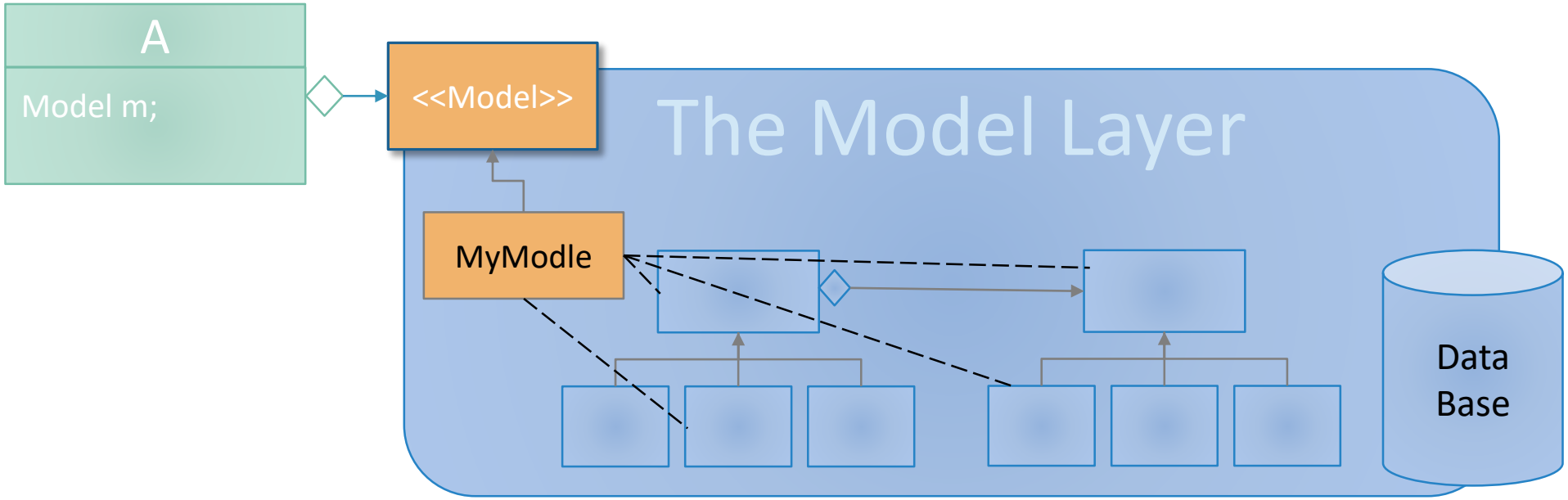


How to separate?



How to separate?

This is a “Façade” – an interface that defines the layer’s functionality

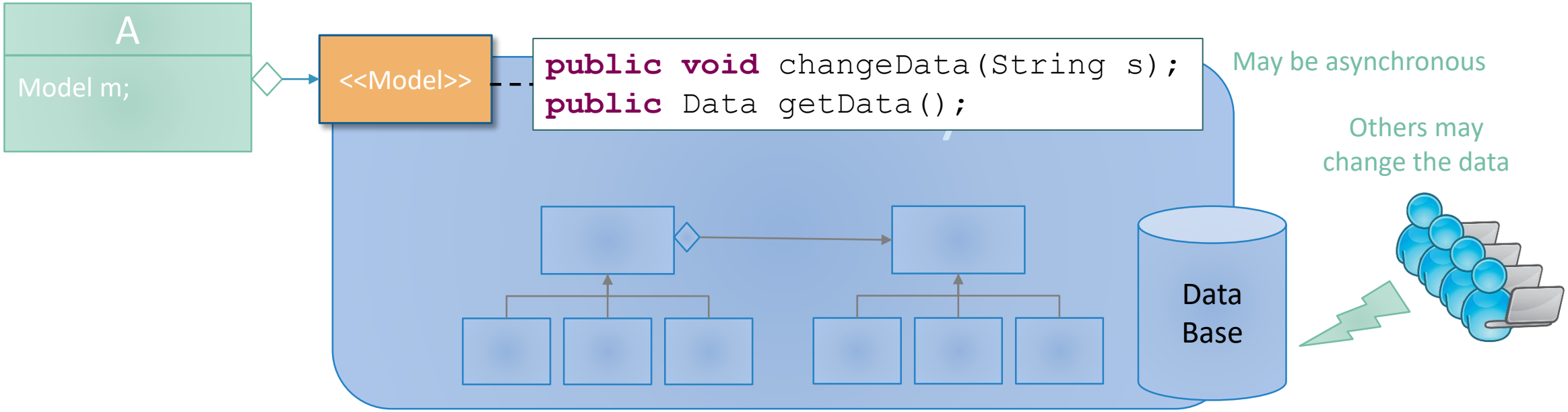


Basic façade instructions

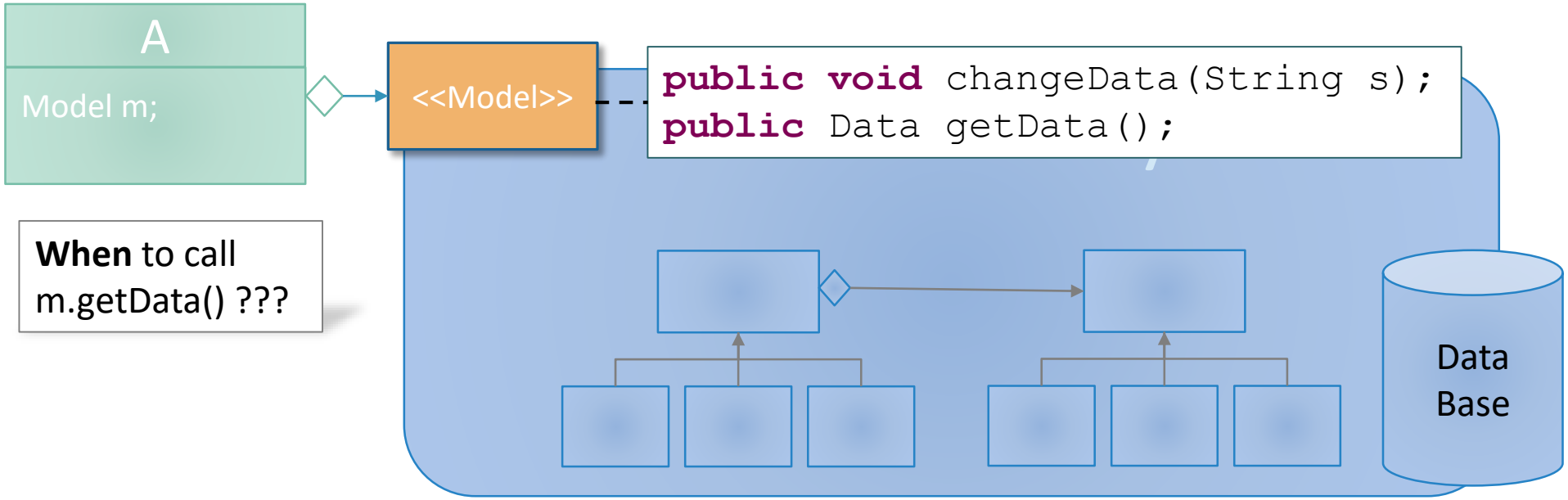
1) Do something
2) Give me something

Why not:

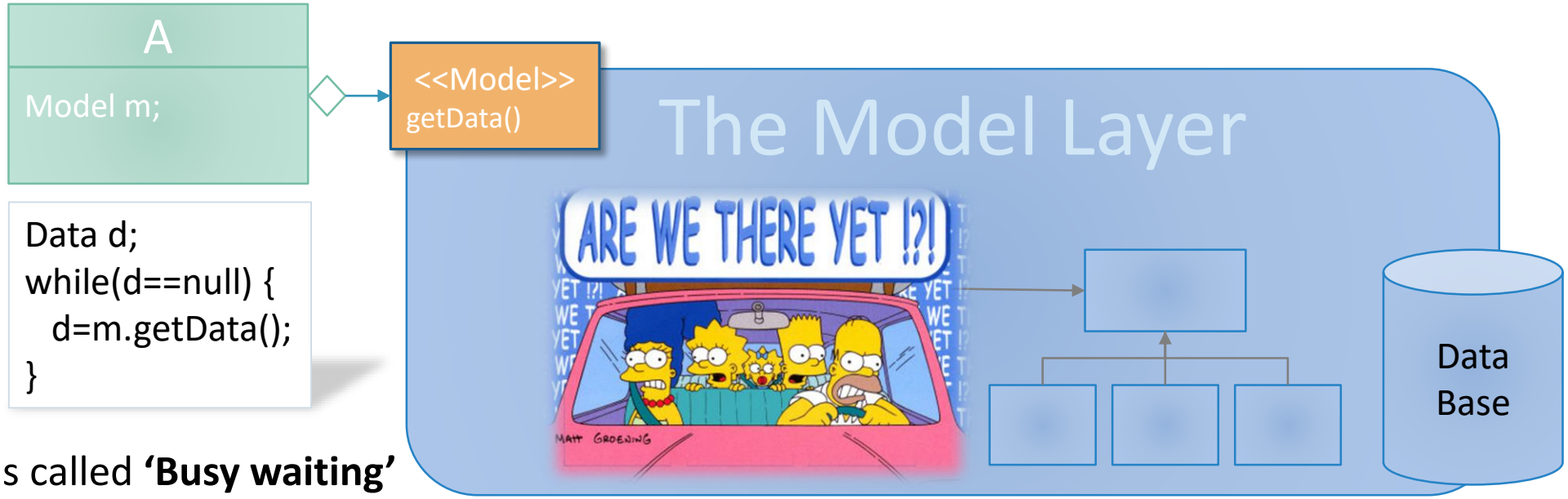
```
public Data changeData(String s);
```



A challenge…


A
Model m;
<<Model>>
public void changeData(String s);
public Data getData();
When to call m.getData() ???
Data Base

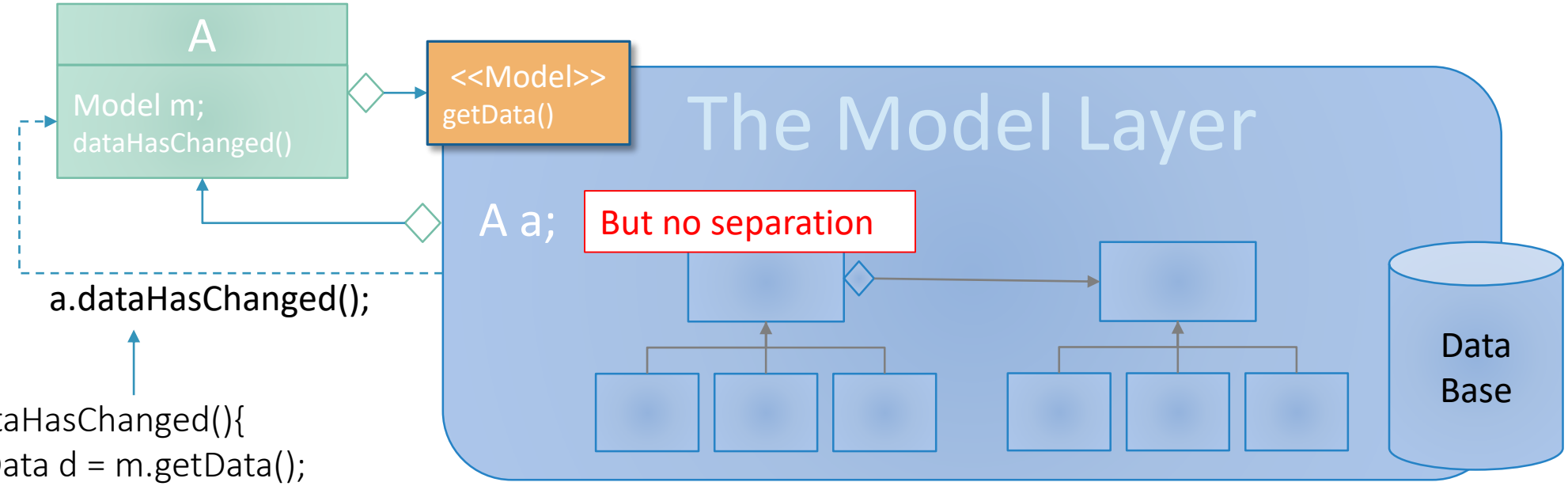
Bad idea #1 – busy waiting




```
Data d;
while(d==null) {
   d=m.getData();
}
```

This is called **'Busy waiting'**
It is a waste of CPU…

Bad idea #2 – no separation

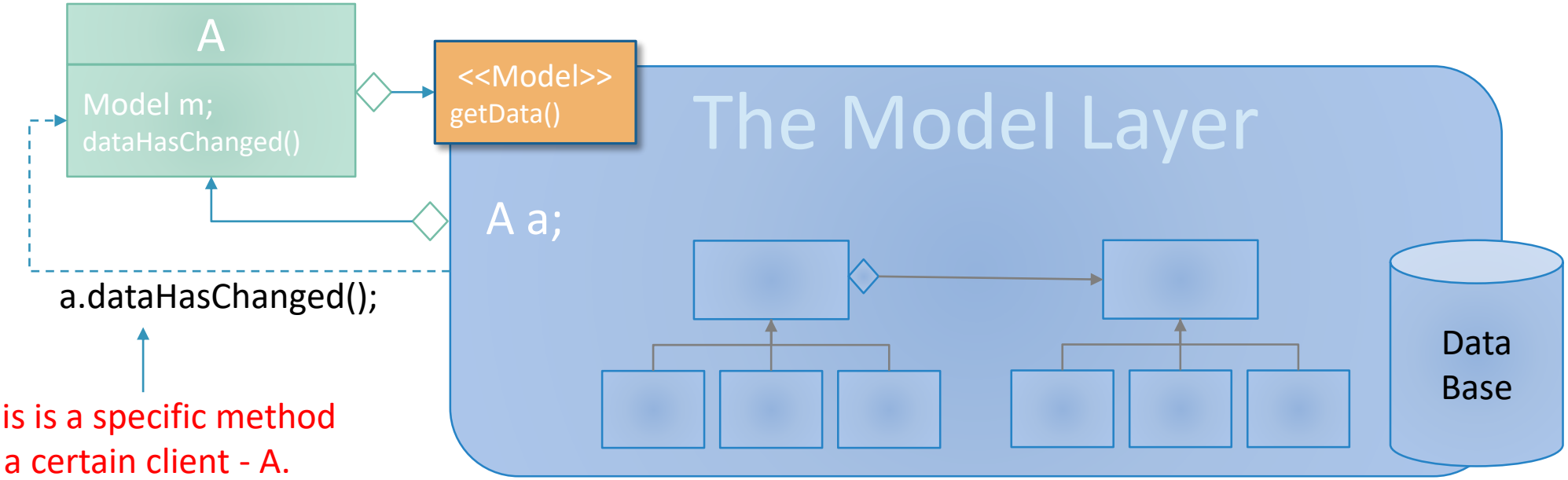


a.dataHasChanged();

```
dataHasChanged(){
   Data d = m.getData();
}
```

No busy waiting…

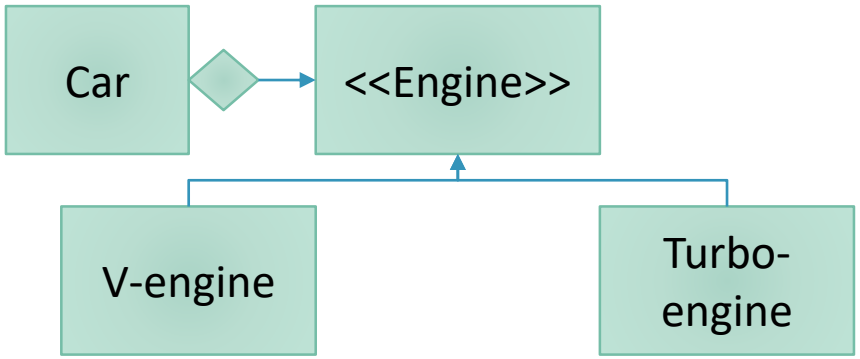
Bad idea #2 – no separation



This is a specific method
of a certain client - A.
what about other clients?
e.g., B, C etc.

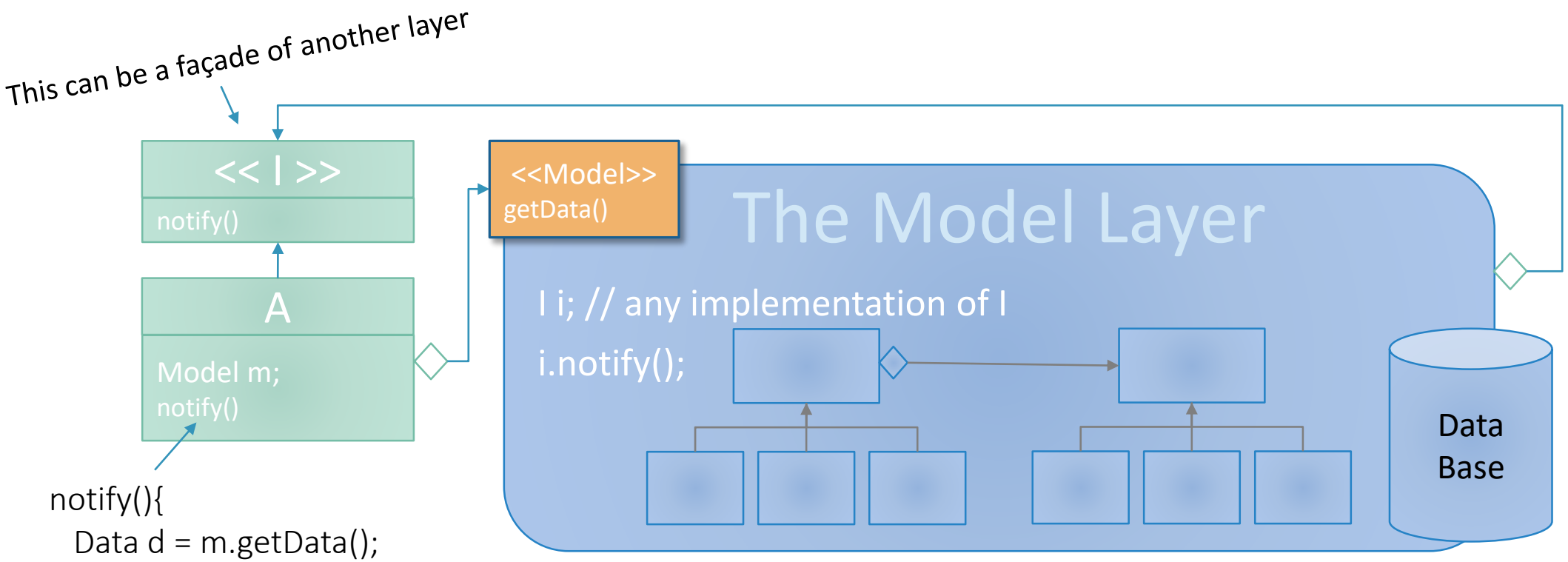
Solid principles of OO design

- S
  - Single Responsibility Principle
- O
  - Open / Closed Principle
- L
  - Liskov Substitution Principle
- I
  - Interface Segregation Principle
- D
  - **Dependency Inversion Principle**

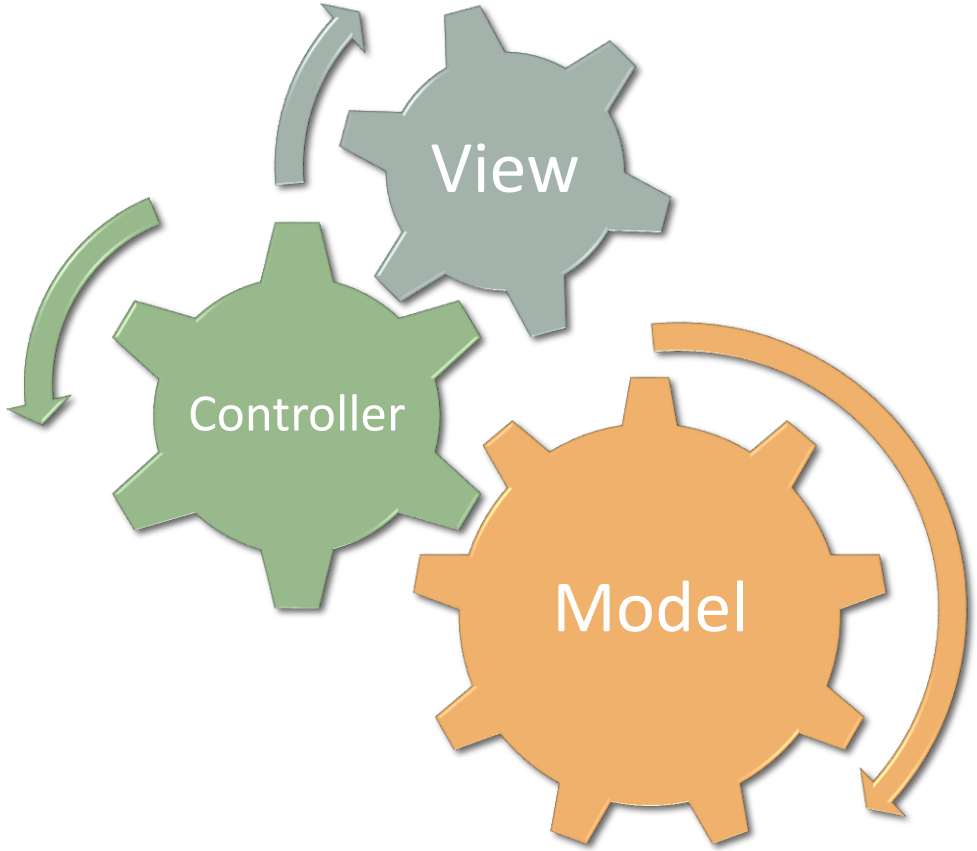


IOC – inversion of control
Control can be given to any type of engine.

Using IOC



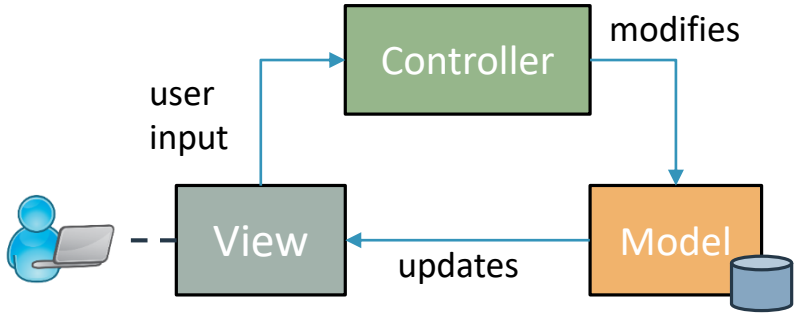

```
notify(){
  Data d = m.getData();
}
```

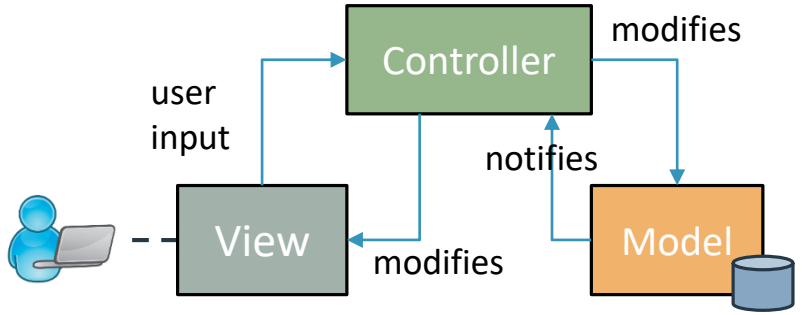


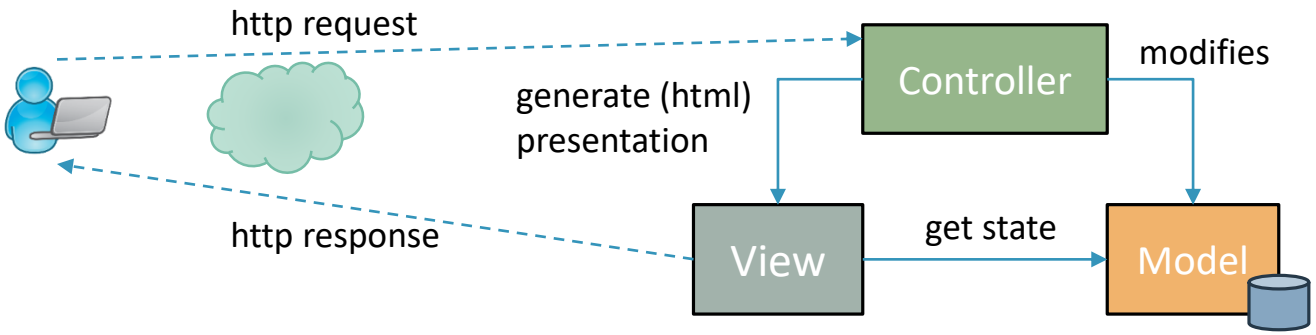
MVC

MODEL, VIEW, CONTROLLER

MVC variations



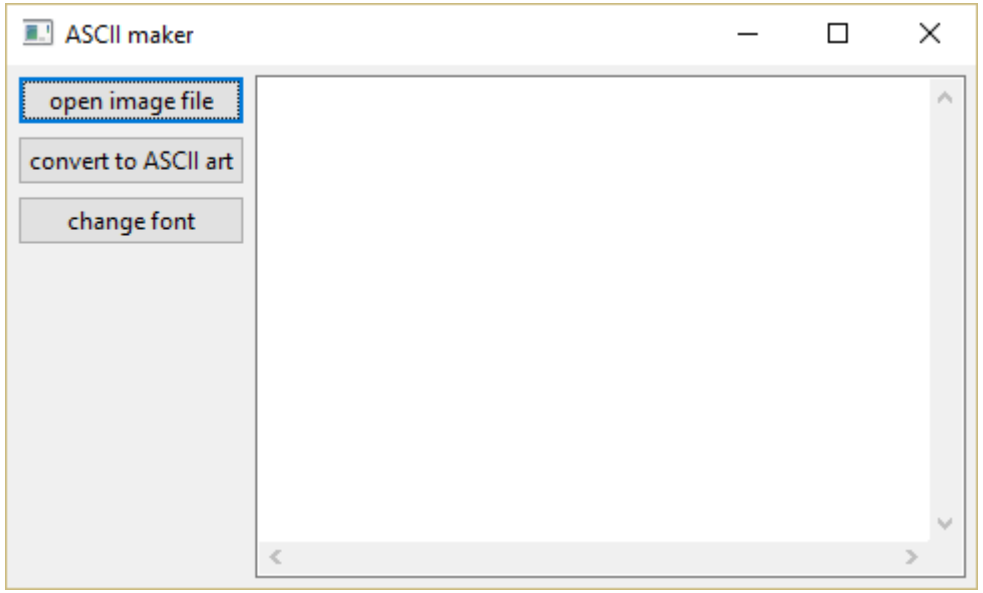


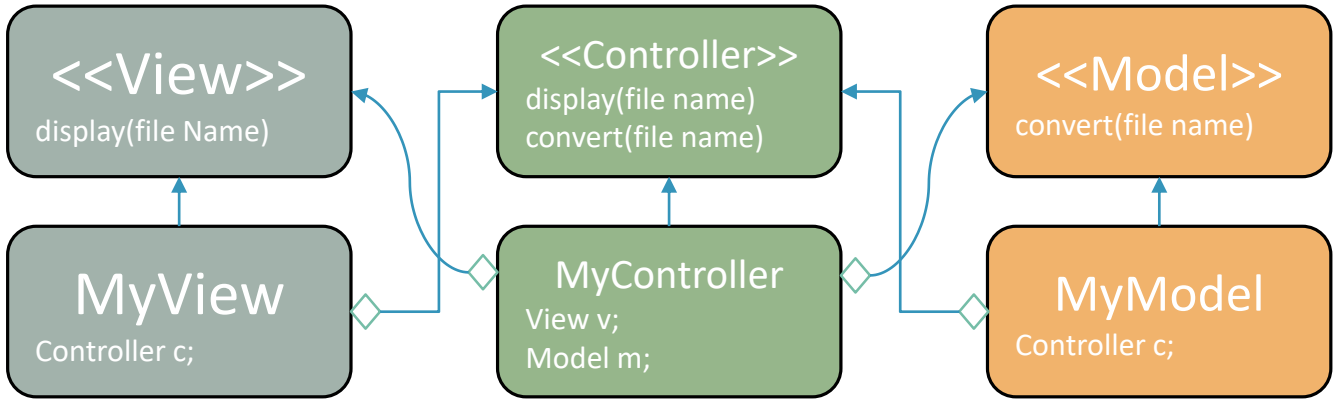


Web application

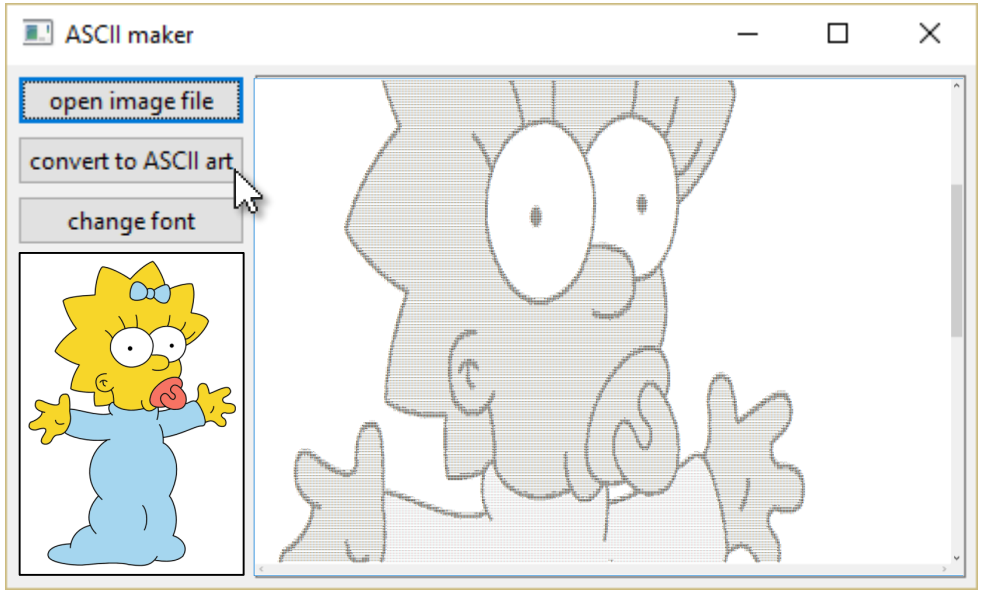
MVC Example:

```
static void Main(string[] args){
    Controller c = new MyController();
    View v = new MyView(c);
    Model m = new MyModel(c);
    c.setModel(m);
    c.setView(v);
    v.start();
}
```

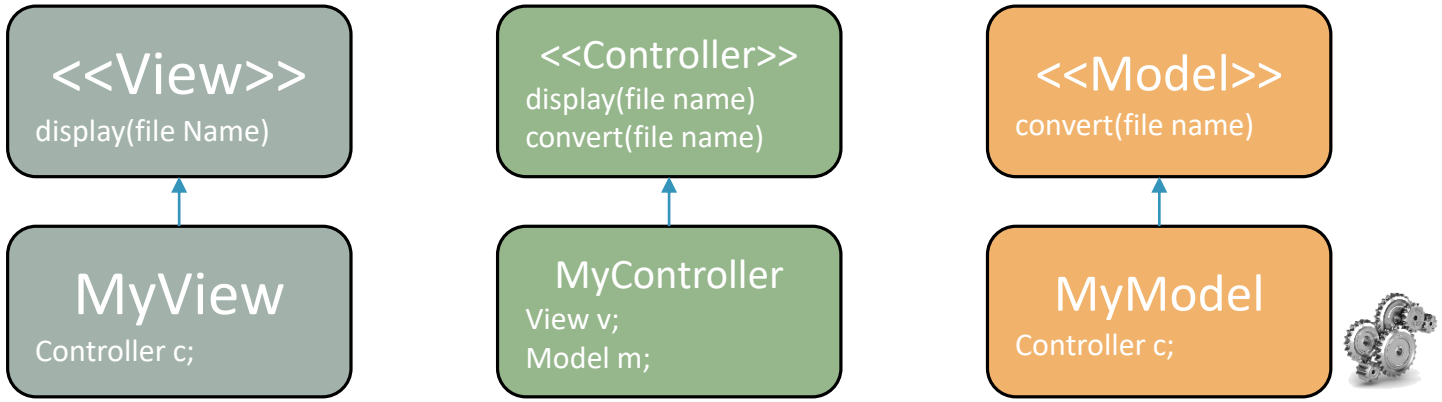




MVC Example:



Did the controller applied any control logic?

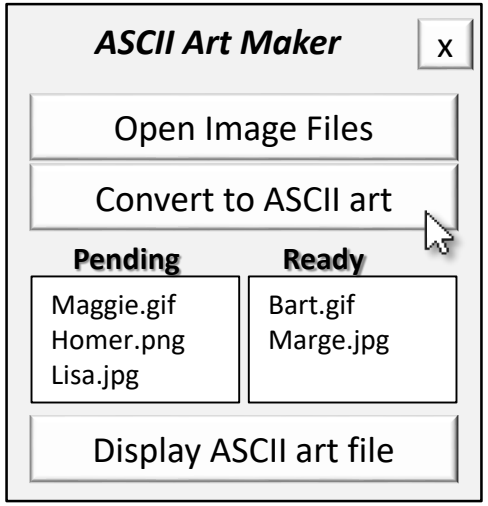


c.convert(image file)

m.convert(image file)
v.display(ASCII file)

c.display(ASCII file)

MVC – a better example



<<View>>

add(file Name)

c.convert(image files)

<<Controller>>

convert(File[] files)
start()

Has a priority queue of jobs

```
start() {
  in a new thread:
  while not stopped{
      if queue is not empty
         m.convert(first job in queue);
         v.add(first job in queue);
      else sleep();
   }
}
convert(File[] files){
  put files in the queue -
  shortest job first (SJF)
  wake up the thread
}
```

<<Model>>

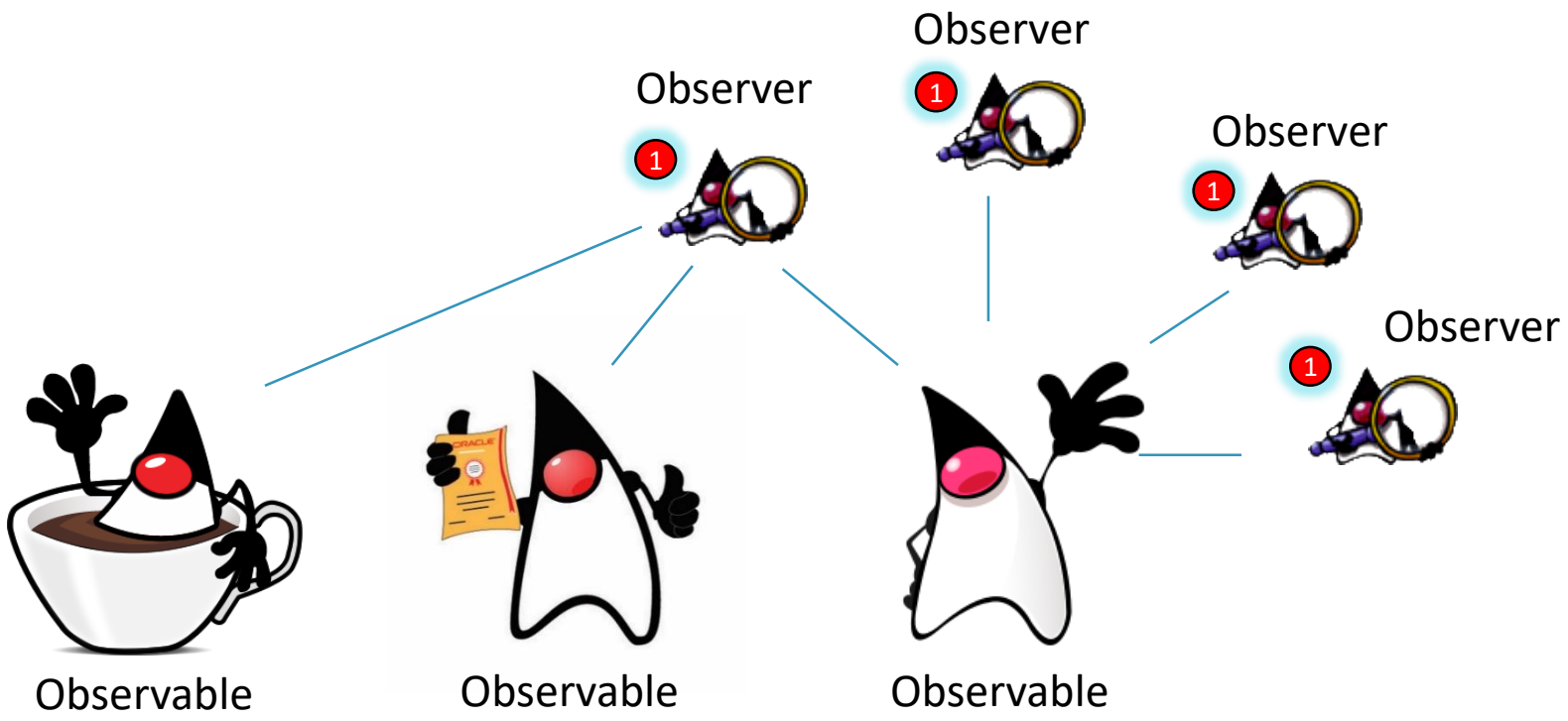
convert(file name)

Observer Design Pattern

A PART OF EVENT DRIVEN PROGRAMMING

A.K.A PUBLISHERS & SUBSCRIBERS

HOW TO **NOTIFY** THAT AN **EVENT** HAS OCCURRED AND ACTIVATE THE **EVENT HANDLER**


Observer
1
Observer
1
Observer
1
Observer
1
Observable
Observable
Observable

Observer Design Pattern

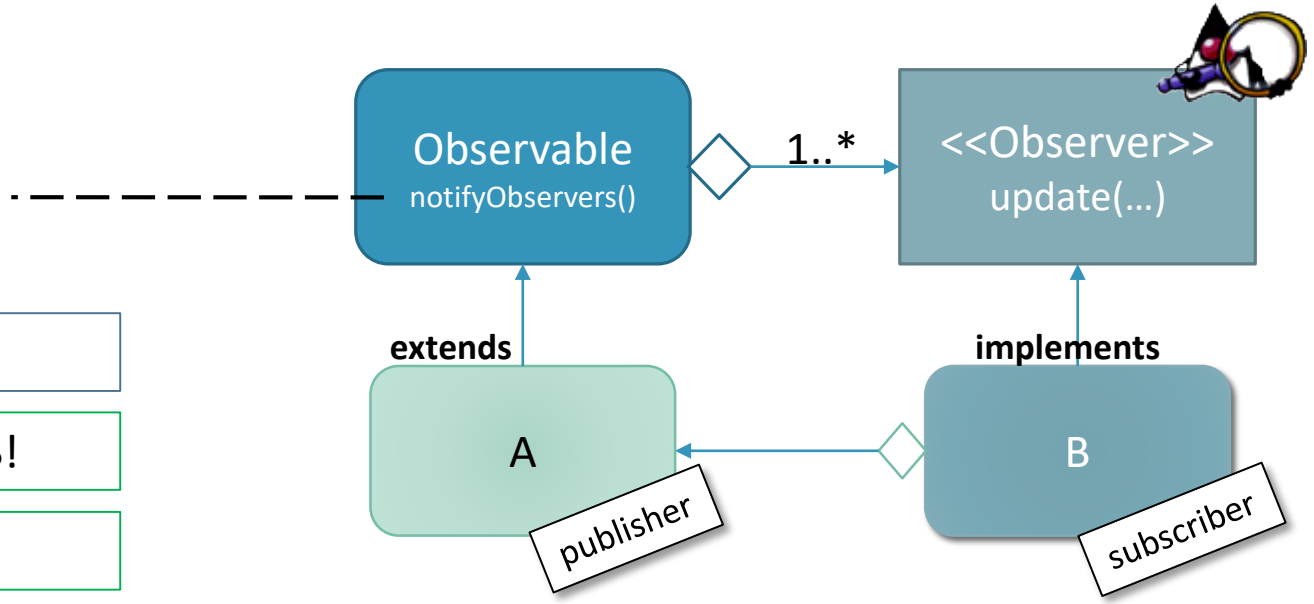
- First, let's learn this design pattern in ***JAVA***
- An ***Observable*** can **notify** many ***Observer***s
- An ***Observer*** can **subscribe** to many ***Observable***s

```
for(Observer o : observersList){
        o.update(…);
}
```

B should subscribe as A's observer

A can update B without knowing B!

If we want, B can tell A what to do



Observer Design Pattern

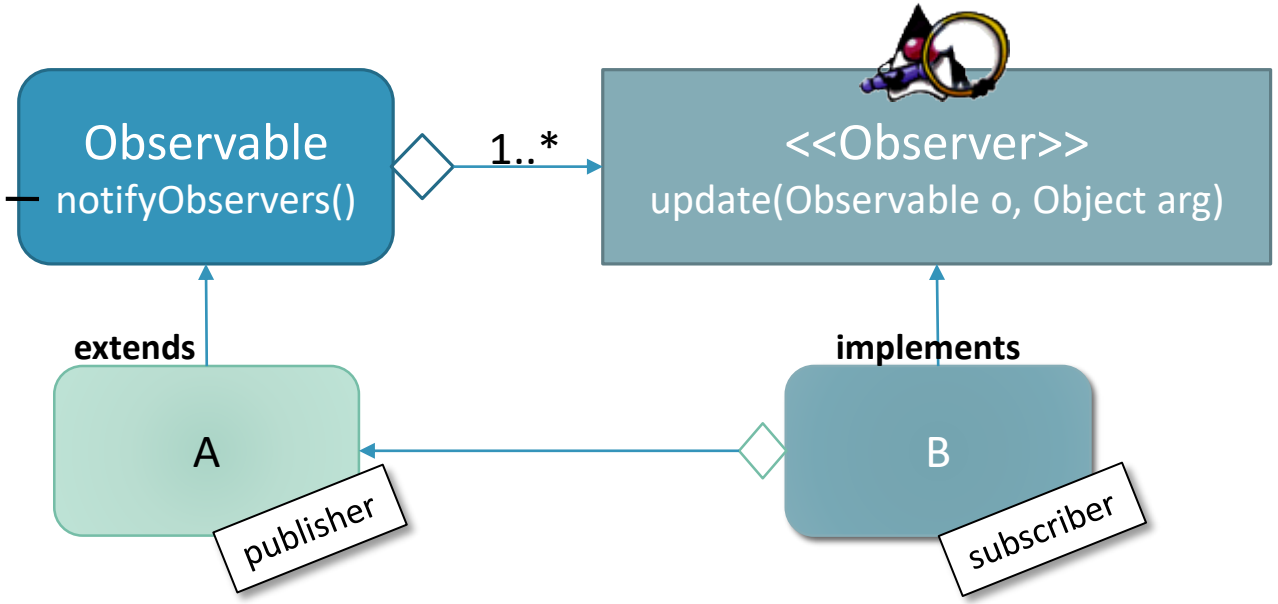
- First, let's learn this design pattern in ***JAVA***
- An ***Observable*** can **notify** many ***Observer***s
- An ***Observer*** can **subscribe** to many ***Observable***s

```
for(Observer o : observersList){
        o.update(this, args);
}
```

B should subscribe as A's observer

A can update B without knowing B!

If we want, B can tell A what to do



Observer Pattern Example

```java
public class A extends Observable{
  int x,y;
  public A() {
    x=0;
    y=0;
  }
  public void setXY(int x,int y){
    this.x=x;
    this.y=y;
     // actively notify all observers
     // and invoke their update method
     notifyObservers();
  }

  public int getX(){return x;}
  public int getY(){return y;}
}
```

Observable

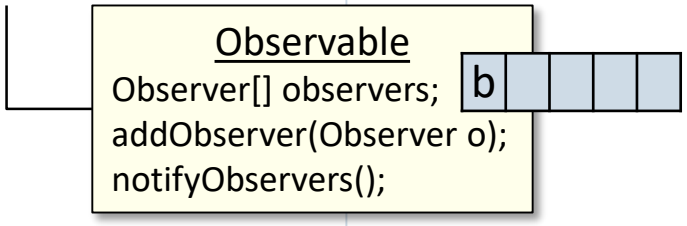
Observer[] observers;
addObserver(Observer o);
notifyObservers();

```java
public class B implements Observer{
  A a; // can observe others as well, e.g., A a1,a2,a3
  public B(A a) {
    this.a=a;
  }

  @Override
  public void update(Observable o, Object arg) {
    // this is invoked upon any change to object "a"
    // now we can actively get the state of object "a"
    if(o == a){
      System.out.println("a change has occurred");
      System.out.println("X="+a.getX()+" Y="+a.getY());
    }
```

```java
public static void main(String[] args) {
  A a=new A();
  B b=new B(a);
  // inherited from Observable
  // add b to the a's list of observers
  a.addObserver(b);

  a.setXY(5,5);
}
```

Observer Pattern Example

```java
public class A extends Observable{
  int x,y;
  public A() {
    x=0;
    y=0;
  }
  public void setXY(int x,int y){
    this.x=x;
    this.y=y;
     // actively notify all observers
     // and invoke their update method
     notifyObservers();
  }

  public int getX(){return x;}
  public int getY(){return y;}
}
```




```java
public class B implements Observer{
  A a; // can observe others as well, e.g., A a1,a2,a3
  public B(A a) {
    this.a=a;
  }

  @Override
  public void update(Observable o, Object arg) {
    // this is invoked upon any change to object "a"
    // now we can actively get the state of object "a"
    if(o == a){
      System.out.println("a change has occurred");
      System.out.println("X="+a.getX()+" Y="+a.getY());
    }
```

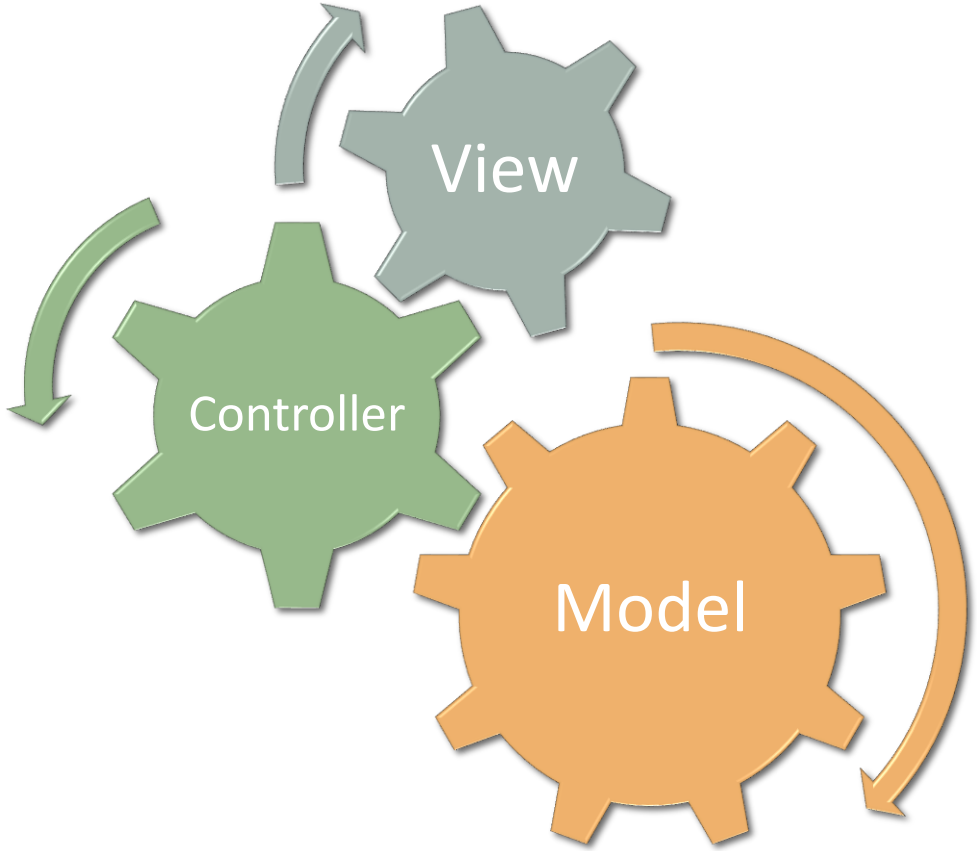
```java
public static void main(String[] args) {
  A a=new A();
  B b=new B(a);
  // inherited from Observable
  // add b to the a's list of observers
  a.addObserver(b);

  a.setXY(5,5);
}
```

Observer Pattern - delegates & events in C#

- The ***Observable*** defines an event variable of some known delegate type
- The ***Observer*** registers its own delegates to the observable
- The observable activates all the registered delegates whenever it is needed


```
// when it is needed to notify all observers
notify(this, theEventArgs);
```

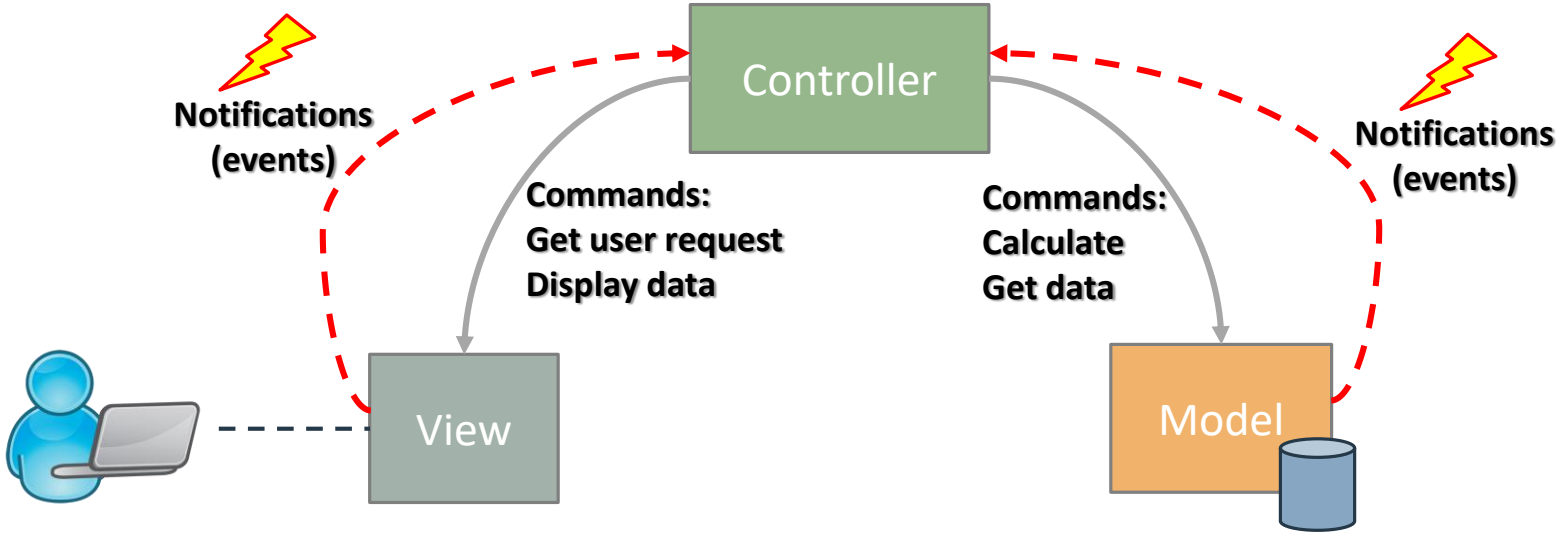
```
theObservable.notify +=
        delegate(Object sender, EventArgs e){
                // do something about the notification
        }
```



MVC

+ OBSERVER PATTERN

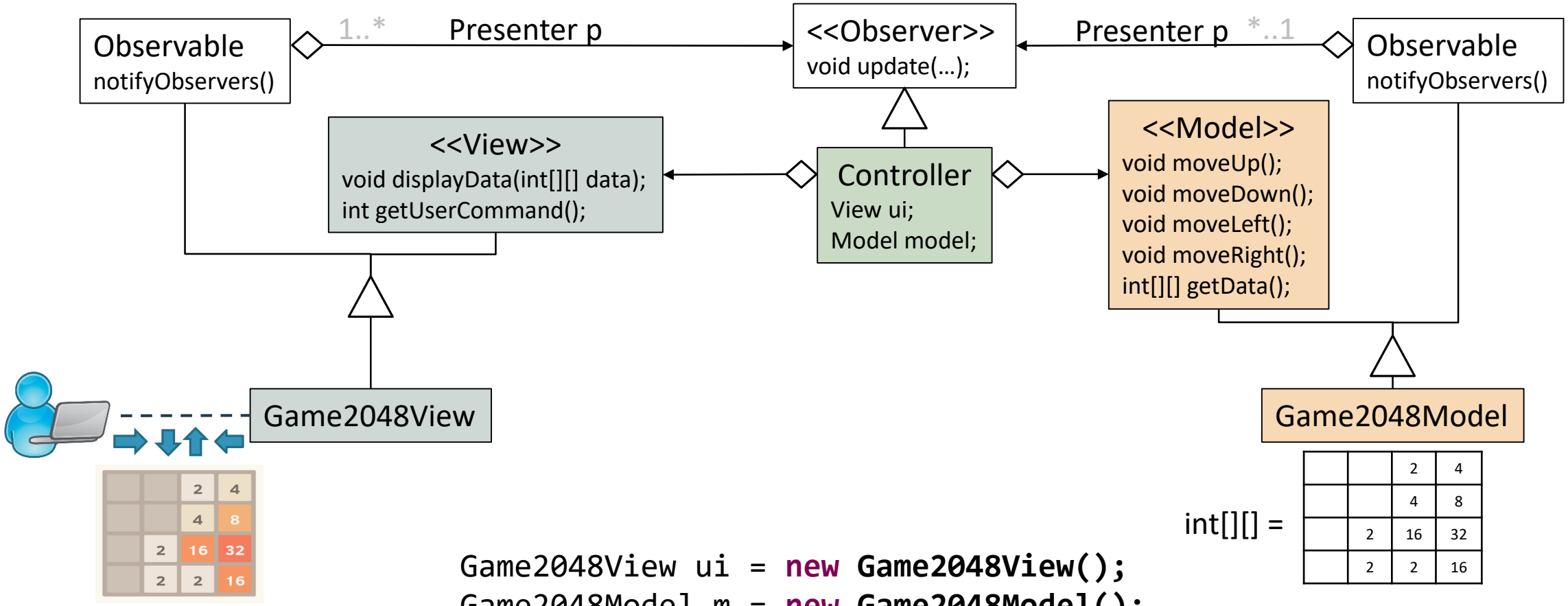
The MVP architectural design



MVP - 2048 puzzle examples

- a JAVA example – using the Observer Design Pattern
- a C# example – using delegates and events

JAVA:

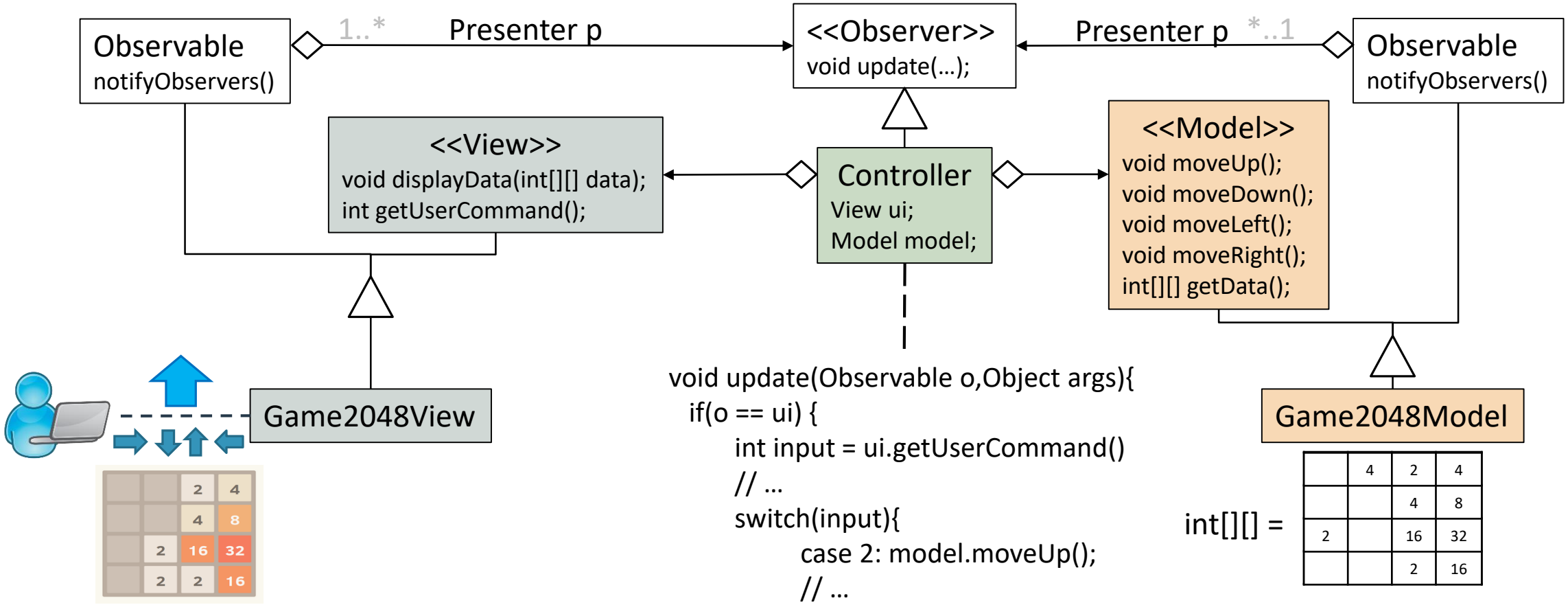


int[][] =

		2	4
		4	8
	2	16	32
	2	2	16

```
Game2048View ui = new Game2048View();
Game2048Model m = new Game2048Model();
Controller c=new Controller(ui, m);
ui.addObserver(c);
m.addObserver(c);
```

JAVA:

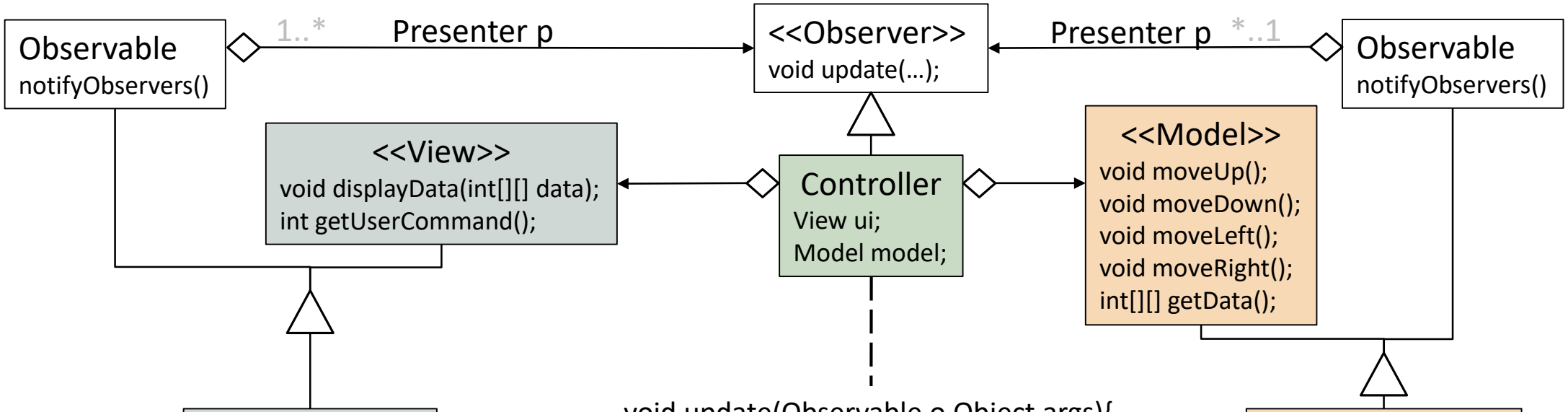



```
void update(Observable o,Object args){
  if(o == ui) {
      int input = ui.getUserCommand()
      // ...
      switch(input){
          case 2: model.moveUp();
          // ...
      }
  }
  // ...
}
```

int[][] =

	4	2	4
		4	8
2		16	32
		2	16

JAVA:



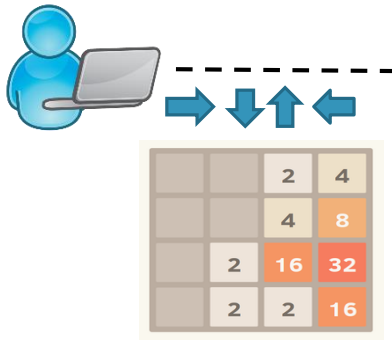

```
void update(Observable o,Object args){
  // ...
  if(o == model) {
      ui.displayData( model.getData() );
  }
  // ...
}
```

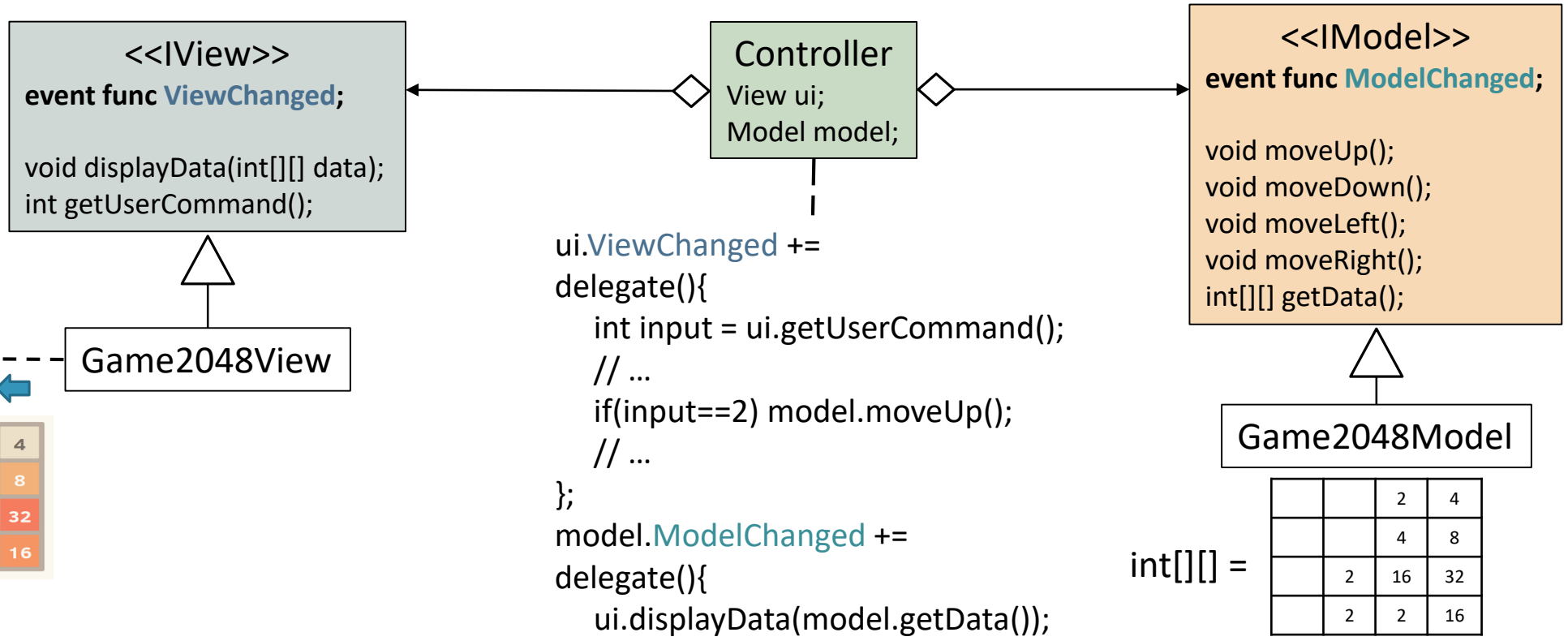
int[][] =

	4	2	4
		4	8
2		16	32
		2	16

MVP in C# - Simply use delegates & events

- We define: **delegate void func ();**

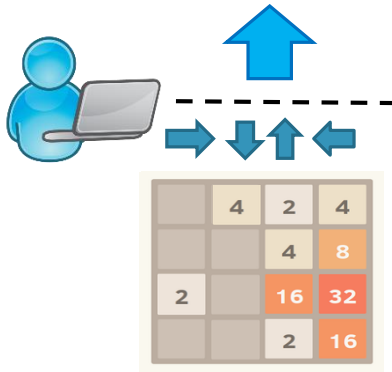





```
ui.ViewChanged +=
delegate(){
    int input = ui.getUserCommand();
    // …
    if(input==2) model.moveUp();
    // …
};
model.ModelChanged +=
delegate(){
    ui.displayData(model.getData());
};
```

MVP in C# - Simply use delegates & events

- We define: **delegate void func ();**



<<IView>>
event func ViewChanged;

void displayData(int[][] data);
int getUserCommand();

Game2048View

ViewChanged();

Controller
View ui;
Model model;

```
ui.ViewChanged +=
delegate(){
    int input = ui.getUserCommand();
    // …
    if(input==2) model.moveUp();
    // …
};
model.ModelChanged +=
delegate(){
    ui.displayData(model.getData());
};
```

<<IModel>>
event func ModelChanged;

void moveUp();
void moveDown();
void moveLeft();
void moveRight();
int[][] getData();

Game2048Model

ModelChanged();

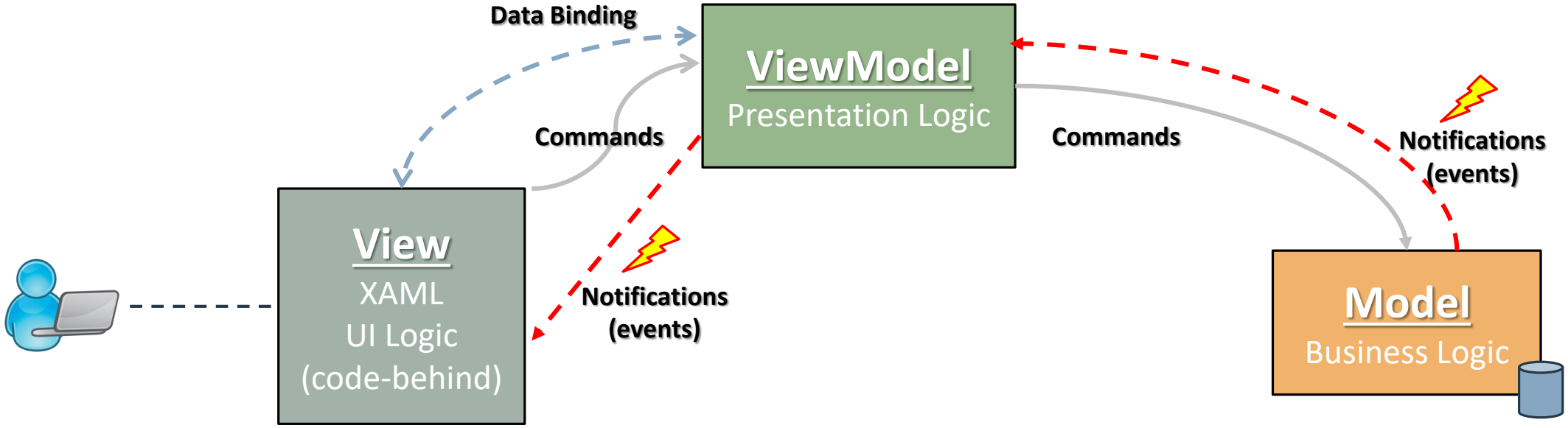
int[][] =

	4	2	4
		4	8
2		16	32
		2	16

In the next lessons…

THE MVVM ARCHITECTURE

MVVM


Data Binding
ViewModel
Presentation Logic
Commands
Commands
Notifications
(events)
View
XAML
UI Logic
(code-behind)
Notifications
(events)
Model
Business Logic