Programmazione

Prof. Marco Bertini
marco.bertini@unifi.it
http://www.micc.unifi.it/bertini/

Design pattern

Observer

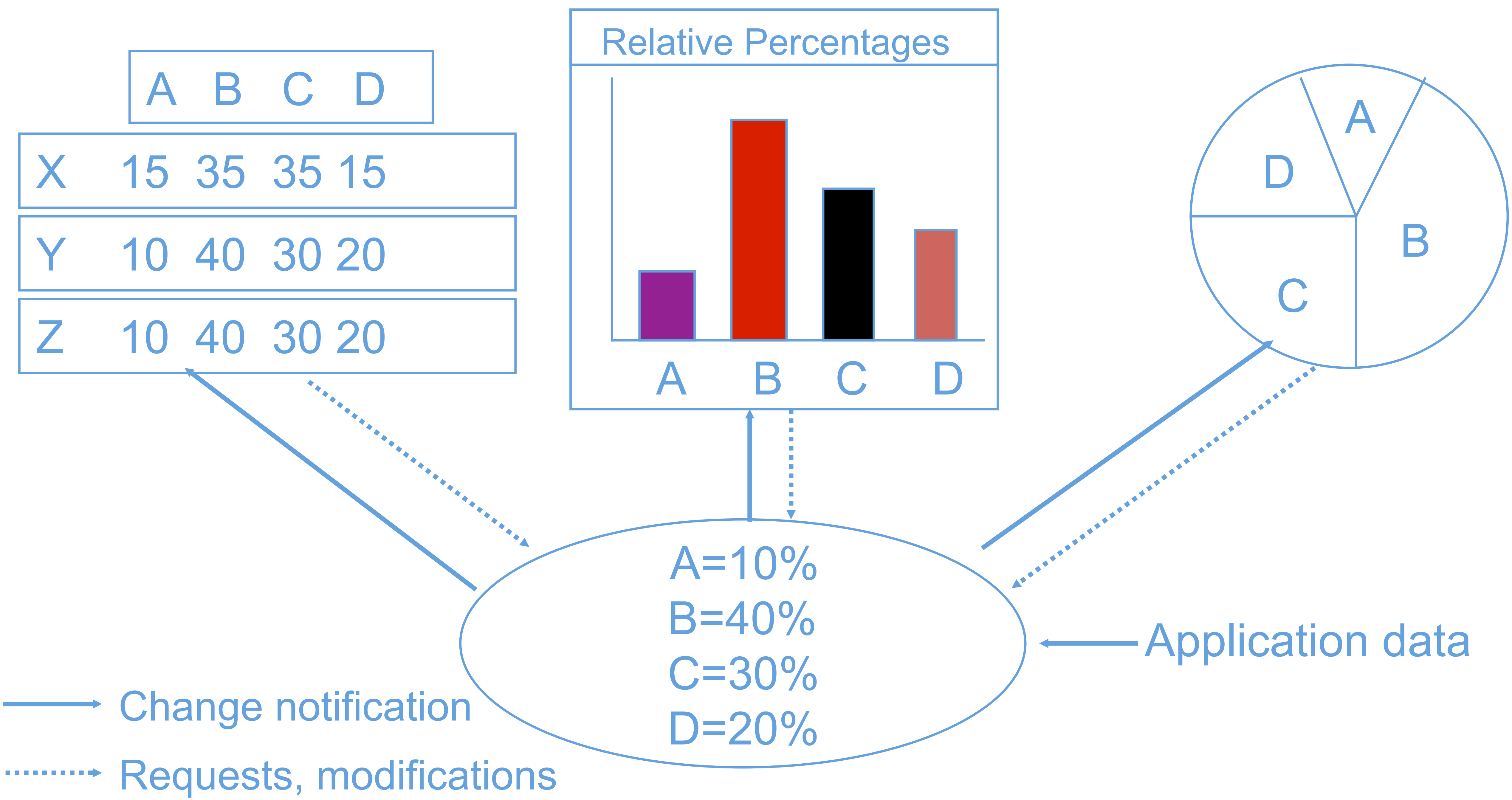
Some motivations

- In many programs, when a object changes state, other objects may have to be notified
- This pattern answers the question: How best to notify those objects when the subject changes?
- And what if the list of those objects changes during run-time?

Some examples

- Example: when an car in a game is moved
  - The graphics engine needs to know so it can re-render the item
  - The traffic computation routines need to re-compute the traffic pattern
  - The objects the car contains need to know they are moving as well
- Another example: data in a spreadsheet changes
  - The display must be updated
  - Possibly multiple graphs that use that data need to re-draw themselves

Another example



Observer Pattern

- Problem
  - Need to update multiple objects when the state of one object changes (one-to-many dependency)
- Context
  - Multiple objects depend on the state of one object
  - Set of dependent objects may change at run-time
- Solution
  - Allow dependent objects to register with object of interest, notify them of updates when state changes
- Consequences
  - When observed object changes others are notified
  - Useful for user interface programming, other applications

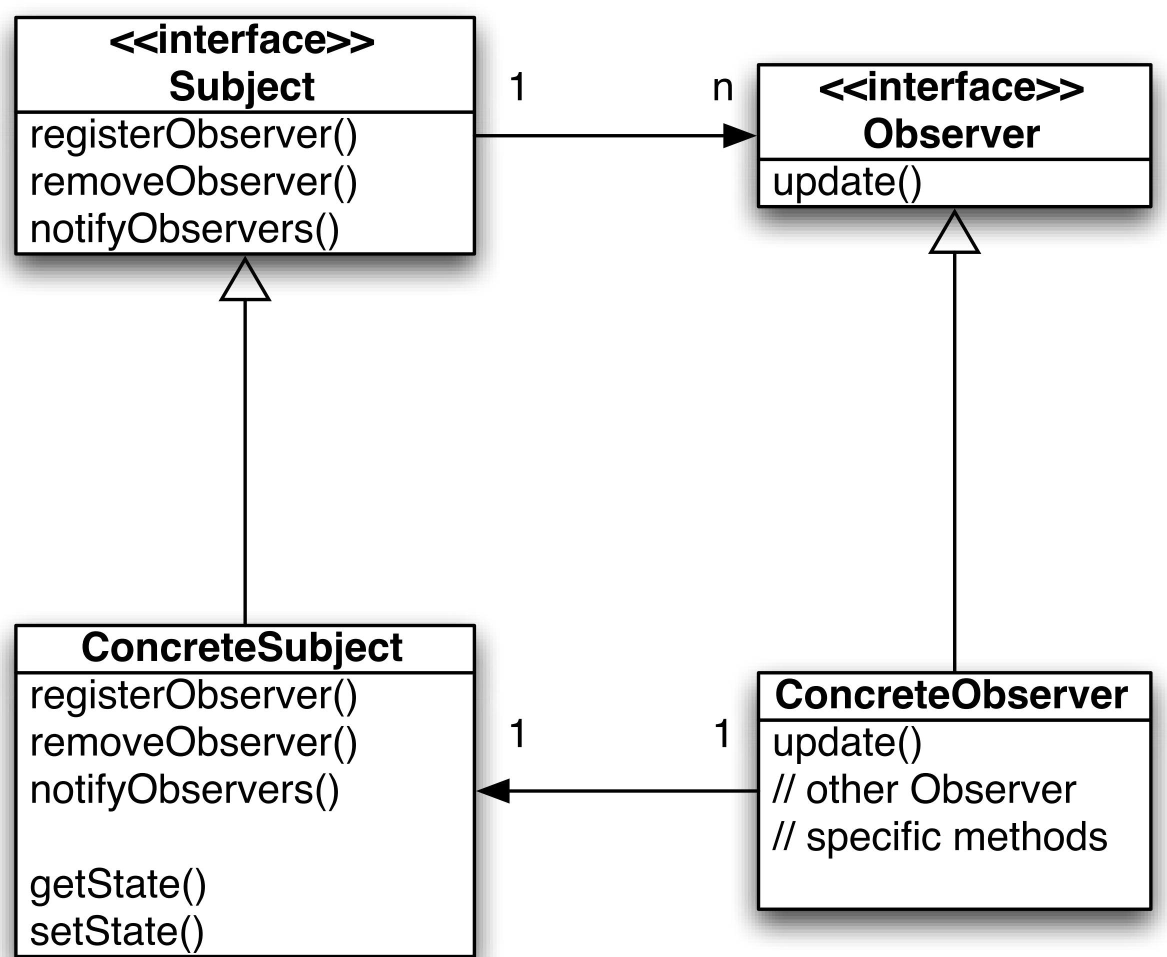
Participants

- The key participants in this pattern are:
- The Subject, which provides an (virtual) interface for attaching and detaching observers
- The Observer, which defines the (virtual) updating interface
- The ConcreteSubject, which is the class that inherits/extends/implements the Subject
- The ConcreteObserver, which is the class that inherits/extends/implements the Observer
- This pattern is also known as dependents or publish-subscribe

Observer UML class diagram

The Subject interface is used by objects to (un)register as Observers.
Each Subject my have several Observers.

Each potential Observer has to implement this interface.
The update() method gets called when the Subject changes its state.



A concrete subject has to implement the Subject interface.
The notifyObservers() method is used to update all the current observers whenever state changes.

Concrete observers have to implement the Observer interface.
Each concrete observer registers with a concrete subject to receive updates.

The concrete subject may have methods for setting and getting its state.

Some interesting points

- In the Observer pattern when the state of one object changes, all of its dependents are notified:
  - the subject is the sole owner of that data, the observers are dependent on the subject to update them when the data changes
  - it's a cleaner design than allowing many objects to control the same data

Loose coupling

- The Observer pattern provides a pattern where subjects and observers are loosely coupled (minimizing the interdependency between objects):
  - the only thing the subject knows about an observer is that it implements an interface
  - observers can be added/removed at any time (also runtime)
  - there is no need to modify the subject to add new types of observers (they just need to implement the interface)
  - changes to subject or observers will not affect the other (as long as they implement the required interface)

Observer example

```
class Subject {
protected:
  virtual ~Subject() {};

public:

  virtual void
registerObserver( Observer* o ) = 0;

  virtual void
removeObserver( Observer* o ) = 0;

  virtual void notifyObservers() const
= 0;

};
```

```
class Observer {

protected:
virtual ~Observer() {};

public:
virtual void update(float temp,
   float humidity, float pressure) = 0;

};
```

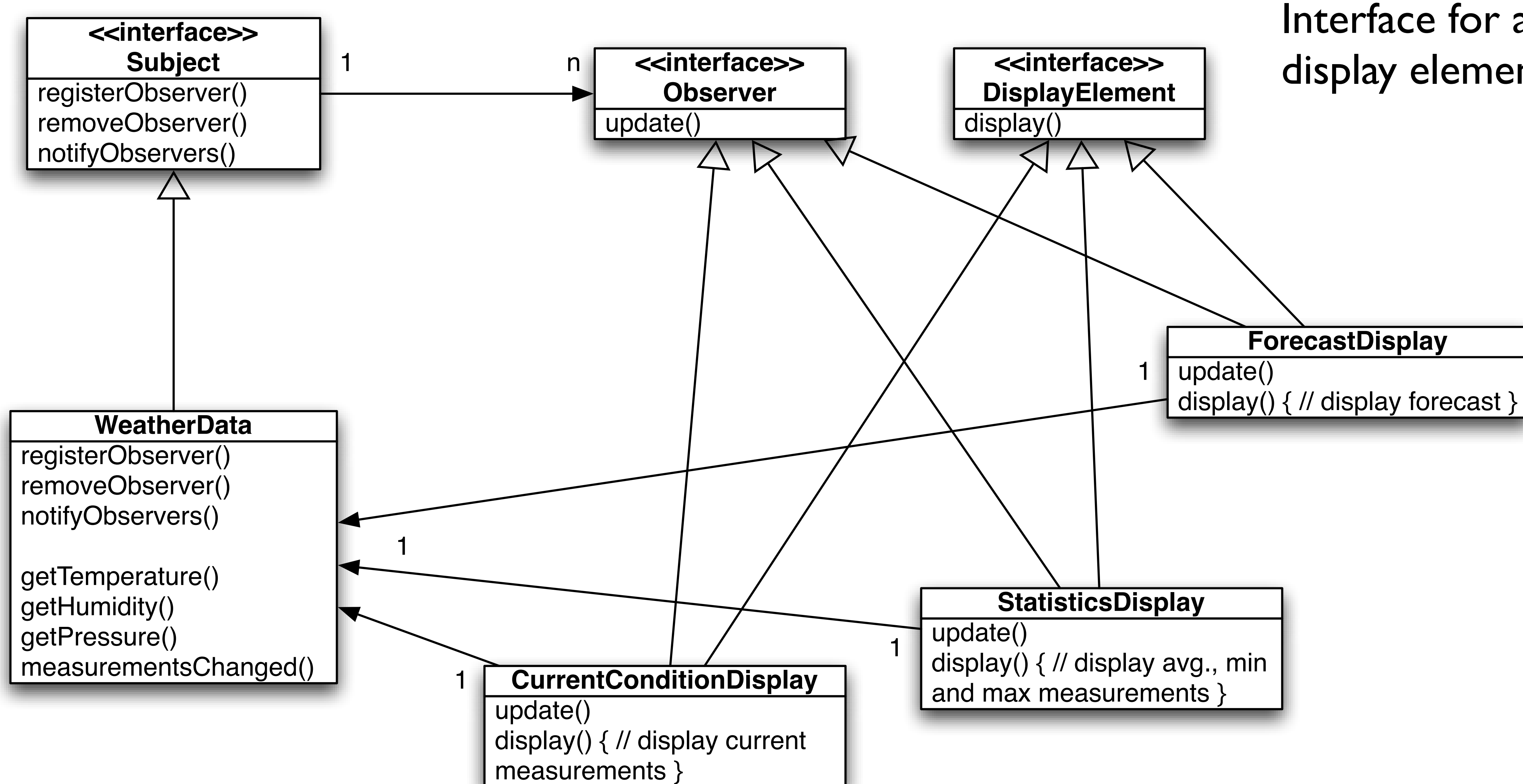
Observer example

```
class Subject {
protected:
  virtual ~Subject() {};

public:

  virtual void
registerObserver( Observer* o ) = 0;

  virtual void
removeObserver( Observer* o ) = 0;

  virtual void notifyObservers() const
= 0;

};
```

```
class Observer {

protected:
virtual ~Observer() {};

public:
virtual void update(float temp,
   float humidity, float pressure) = 0;

};
```

The update method gets the state values from the subject: they'll change depending on the subject, in this example is a weather station

Observer example

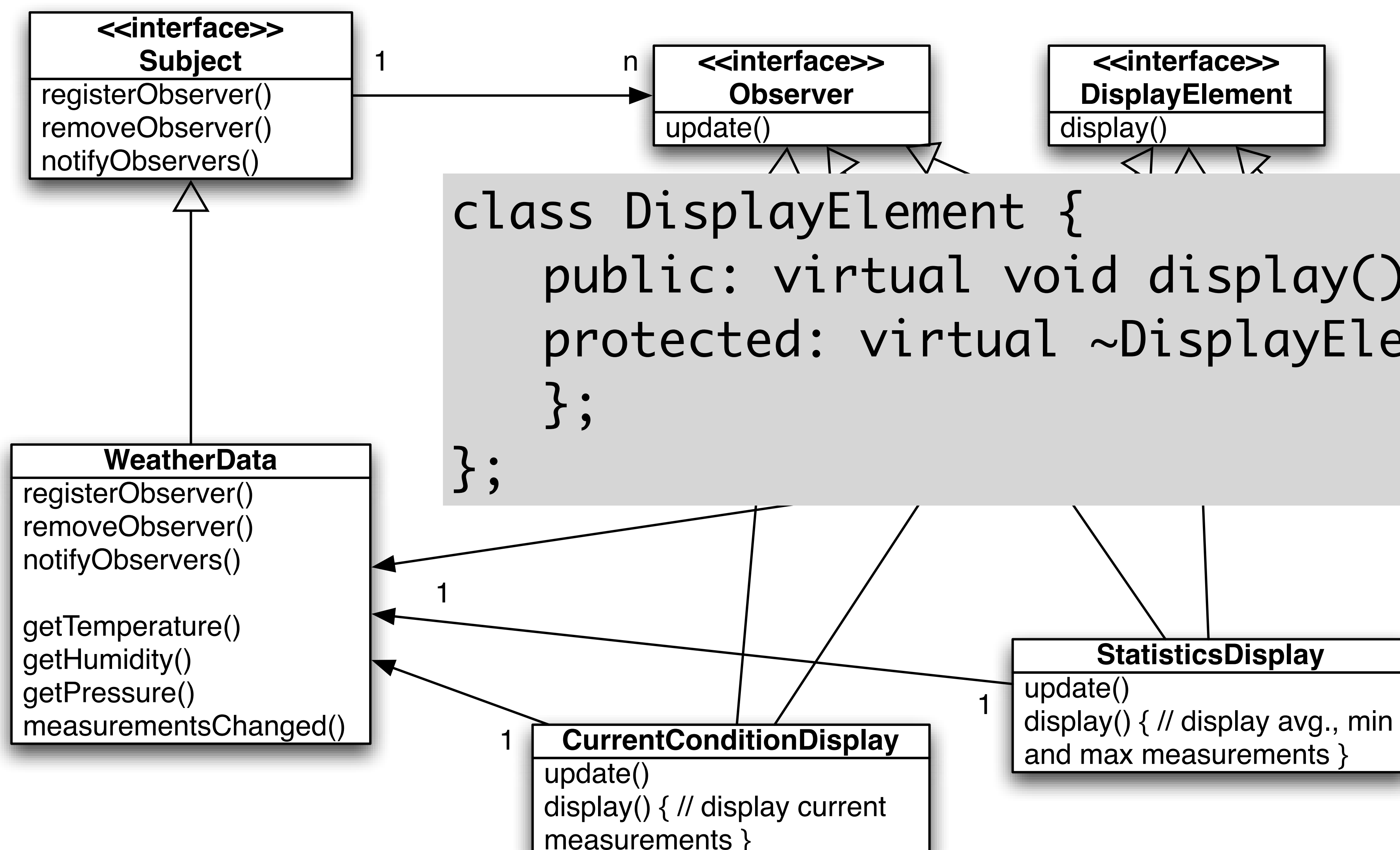


Interface for all the display elements

Different types of observers

Observer example

Interface for all the display elements




```
class DisplayElement {
    public: virtual void display() const = 0;
    protected: virtual ~DisplayElement() = 0 {
    };
};
```

Different types of observers

Implementing the Subject interface

```
class WeatherData : public Subject {
  private: list< Observer* > observers;
  private: float temperature;
  private: float humidity;
  private: float pressure;
  public: WeatherData() :   temperature( 0.0 ),
humidity( 0.0 ), pressure( 0.0 ) {    }
  public: void
  registerObserver( Observer* o ) {
      observers.push_back(o);
  }
  public: void
  removeObserver( Observer* o ) {
    observers.remove(o);
  }
  public: void notifyObservers() const {
    for( list< Observer* >::iterator itr =
observers.begin(); observers.end() != itr; ++itr ) {
      (*itr)->update( temperature, humidity, pressure );
    }
  }
```

```
  public: void measurementsChanged() {
    notifyObservers();
  }
  public: void setMeasurements( float temperature,
              float humidity, float pressure ) {
    temperature = temperature;
    humidity = humidity;
    pressure = pressure;
    measurementsChanged();
  }
  // other WeatherData methods here
  public: float getTemperature() const {
    return temperature;
  }
  public: float getHumidity() const {
    return humidity;
  }
  public: float getPressure() const {
    return pressure;
  }
};
```

Implementing the Subject interface

```
class WeatherData : public Subject {
  private: list< Observer* > observers;
  private: float temperature;
  private: float humidity;
  private: f
  public: We
humidity( 0.
  public: vo
  registerObserver( Observer* o ) {
      observers.push_back(o);
  }
  public: void
  removeObserver( Observer* o ) {
    observers.remove(o);
  }
  public: void notifyObservers() const {
    for( list< Observer* >::iterator itr =
observers.begin(); observers.end() != itr; ++itr ) {
      (*itr)->update( temperature, humidity, pressure );
    }
  }
```

The weather station device would call this method, providing the measurements

```
  public: void measurementsChanged() {
    notifyObservers();
  }
  public: void setMeasurements( float temperature,
            float humidity, float pressure ) {
    temperature = temperature;
    humidity = humidity;
    pressure = pressure;
    measurementsChanged();
  }
  // other WeatherData methods here
  public: float getTemperature() const {
    return temperature;
  }
  public: float getHumidity() const {
    return humidity;
  }
  public: float getPressure() const {
    return pressure;
  }
};
```

Implementing the Subject interface

When measurements are updated then the Observers are notified

The weather station device would call this method, providing the measurements

```
class WeatherData
  private: list< Observer* > observers;
  private: float temperature;
  private: float humidity;
  private: f
  public: We
humidity( 0.
  public: vo
  registerObserver( Observer* o ) {
      observers.push_back(o);
  }
  public: void
  removeObserver( Observer* o ) {
    observers.remove(o);
  }
  public: void notifyObservers() const {
    for( list< Observer* >::iterator itr =
observers.begin(); observers.end() != itr; ++itr ) {
      (*itr)->update( temperature, humidity, pressure );
    }
  }
```

```
  public: void measurementsChanged() {
    notifyObservers();
  }
  public: void setMeasurements( float temperature,
             float humidity, float pressure ) {
    temperature = temperature;
    humidity = humidity;
    pressure = pressure;
    measurementsChanged();
  }
  // other WeatherData methods here
  public: float getTemperature() const {
    return temperature;
  }
  public: float getHumidity() const {
    return humidity;
  }
  public: float getPressure() const {
    return pressure;
  }
};
```

Implementing a concrete observer

```
class CurrentConditionsDisplay : public Observer,
private DisplayElement {
  private: Subject* weatherData;
  private: float temperature;
  private: float humidity;


  public: CurrentConditionsDisplay( Subject*
weatherData ) : weatherData( weatherData ),
temperature( 0.0 ), humidity( 0.0 ) {
    weatherData->registerObserver( this );
  }
  public: ~CurrentConditionsDisplay() {
    weatherData->removeObserver( this );
  }
```

```
  public: void update( float temp, float hum, float
pres ) {
    temperature = temp;
    humidity = hum;
    display();
  }
  public: void display() const {
    cout.setf( std::ios::showpoint );
    cout.precision(3);
    cout << "Current conditions: "     << temperature;
    cout << " C° degrees and " << humidity;
    cout << "% humidity" << std::endl;
  }
};
```

Implementing a concrete observer

```
class CurrentConditionsDisplay : public Observer,
private DisplayElement {
  private: Subject* weatherData;
  private: float temperature;
  private: float humidity;

  public: CurrentConditionsDisplay( Subject*
weatherData ) : weatherData( weatherData ),
temperature( 0.0 ), humidity( 0.0 ) {
    weatherData->registerObserver( this );
  }
  public: ~CurrentConditionsDisplay() {
    weatherData->removeObserver( this );
  }
```

The constructor gets the Subject and use it to register to it as an observer.

```
  public: void update( float temp, float hum, float
pres ) {
    temperature = temp;
    humidity = hum;
    display();
  }
  public: void display() const {
                                        .nt );
    cout << "Current Conditions: "    << temperature;
    cout << " C° degrees and " << humidity;
    cout << "% humidity" << std::endl;
  }
};
```

Implementing a concrete observer

The reference to the subject is stored so that it is possible to use it to un-register

```
class CurrentConditionsDisplay : public Observer,
private DisplayElement {
  private: Subject* weatherData;
  private: float temperature;
  private: float humidity;


  public: CurrentConditionsDisplay( Subject*
weatherData ) : weatherData( weatherData ),
temperature( 0.0 ), humidity( 0.0 ) {
    weatherData->registerObserver( this );
  }
  public: ~CurrentConditionsDisplay() {
    weatherData->removeObserver( this );
  }
```

The constructor gets the Subject and use it to register to it as an observer.

```
  public: void update( float temp, float hum, float
pres ) {
    temperature = temp;
    humidity = hum;
    display();
  }
  public: void display() const {
                                      .nt );

    cout << "Current conditions: "    << temperature;
    cout << " C° degrees and " << humidity;
    cout << "% humidity" << std::endl;
  }
};
```

Implementing a concrete observer

The reference to the subject is stored so that it is possible to use it to un-register

```
class CurrentConditionsDisplay : public Observer,
private DisplayElement {
  private: Subject* weatherData;
  private: float temperature;
  private: float humidity;


  public: CurrentConditionsDisplay( Subject*
weatherData ) : weatherData( weatherData ),
temperature( 0.0 ), humidity( 0.0 ) {
    weatherData->registerObserver( this );
  }
  public: ~CurrentConditionsDisplay() {
    weatherData->removeObserver( this );
  }
```

The constructor gets the Subject and use it to register to it as an observer.

When update is called it stores the data, then display() is called to show them

```
  public: void update( float temp, float hum, float
pres ) {
    temperature = temp;
    humidity = hum;
    display();
  }
  public: void display() const {
    .nt );
    cout << "Current Conditions: "    << temperature;
    cout << " C° degrees and " << humidity;
    cout << "% humidity" << std::endl;
  }
};
```

Implementing a concrete observer

The reference to the subject is stored so that it is possible to use it to un-register

```
class CurrentConditionsDisplay : public Observer,
private DisplayElement {
  private: Subject* weatherData;
  private: float temperature;
  private: float humidity;


  public: CurrentConditionsDisplay( Subject*
weatherData ) : weatherData( weatherData ),
temperature( 0.0 ), humidity( 0.0 ) {
    weatherData->registerObserver( this );
  }
  public: ~CurrentConditionsDisplay() {
    weatherData->removeObserver( this );
  }
```

When update is called it stores the data, then display() is called to show them

```
  public: void update( float temp, float hum, float
pres ) {
    temperature = temp;
    humidity = hum;
    display();
  }
  public: void display() const {
                                      nt );

    cout << " Current Conditions: "    << temperature;
    cout << " C° degrees and " << humidity;
    cout << "% humidity" << std::endl;
```

The constructor gets the Subject and use it to register to it as an observer.

Remind to unregister the observer when it is destroyed.

Test the pattern

```
int main( ) {

  WeatherData weatherData;

  CurrentConditionsDisplay currentDisplay( &weatherData );
  StatisticsDisplay statisticsDisplay( &weatherData );
  ForecastDisplay forecastDisplay( &weatherData );

  weatherData.setMeasurements( 80, 65, 30.4f );
  weatherData.setMeasurements( 82, 70, 29.2f );
  weatherData.setMeasurements( 78, 90, 29.2f );

  return 0;
}
```

Test the pattern

```
int main( ) {

  WeatherData weatherData;

  CurrentConditionsDisplay currentDisplay( &weatherData );
  StatisticsDisplay statisticsDisplay( &weatherData );
  ForecastDisplay forecastDisplay( &weatherData );

  weatherData.setMeasurements( 80, 65, 30.4f );
  weatherData.setMeasurements( 82, 70, 29.2f );
  weatherData.setMeasurements( 78, 90, 29.2f );

  return 0;
}
```

Create the concrete subject

Test the pattern

Create the concrete subject

Create the displays and pass the concrete subject

```
int main( ) {

  WeatherData weatherData;

  CurrentConditionsDisplay currentDisplay( &weatherData );
  StatisticsDisplay statisticsDisplay( &weatherData );
  ForecastDisplay forecastDisplay( &weatherData );

  weatherData.setMeasurements( 80, 65, 30.4f );
  weatherData.setMeasurements( 82, 70, 29.2f );
  weatherData.setMeasurements( 78, 90, 29.2f );

  return 0;
}
```

Test the pattern

```
int main( ) {

  WeatherData weatherData;

  CurrentConditionsDisplay currentDisplay( &weatherData );
  StatisticsDisplay statisticsDisplay( &weatherData );
  ForecastDisplay forecastDisplay( &weatherData );

  weatherData.setMeasurements( 80, 65, 30.4f );
  weatherData.setMeasurements( 82, 70, 29.2f );
  weatherData.setMeasurements( 78, 90, 29.2f );

  return 0;
}
```

Create the concrete subject

Create the displays and pass the concrete subject

Simulate measurements

Push or pull ?

- In the previous implementation the state is pushed from the Subject to the Observer
- If the Subject had some public getter methods the Observer may pull the state when it is notified of a change
  - If the state is modified there's no need to modify the update(), change the getter methods

Pull example

```
public: void update( ) {
  temperature = weatherData->getTemperature();
  humidity = weatherData->getHumidity();
  display();
}
```

Pull example

The update() method in the Observer interface now is decoupled from the state of the concrete subject

```
public: void update( ) {
  temperature = weatherData->getTemperature();
  humidity = weatherData->getHumidity();
  display();
}
```

Pull example

The update() method in the Observer interface now is decoupled from the state of the concrete subject

```
public: void update( ) {
  temperature = weatherData->getTemperature();
  humidity = weatherData->getHumidity();
  display();
}
```

We just have to change the implementation of the update() in the concrete observers

... example

This must be a pointer to the concrete subject instead of Subject interface, or you can not use its getter methods

The update() method in the Observer interface now is decoupled from the state of the concrete subject

```
public: void update( ) {
  temperature = weatherData->getTemperature();
  humidity = weatherData->getHumidity();
  display();
}
```

We just have to change the implementation of the update() in the concrete observers

Flexible updating

- To have more flexibility in updating the observers the Subject may have a setChanged() method that allows the notifyObservers() to trigger the update()

```
setChanged() {
  changed = true;
}
public: void notifyObservers() const {
  if( changed ) {
      for( list< Observer* >::iterator itr = observers.begin();
           observers.end() != itr; ++itr ) {
        Observer* observer = *itr;
        observer->update( temperature, humidity, pressure );
      }
      changed = false;
  }
}
```

Flexible updating

- To have more flexibility in updating the observers the Subject may have a setChanged() method that allows the notifyObservers() to trigger the update()

```
setChanged() {
  changed = true;
}
public: void notifyObservers() const {
  if( changed ) {
      for( list< Observer* >::iterator itr = observers.begin();
           observers.end() != itr; ++itr ) {
        Observer* observer = *itr;
        observer->update( temperature, humidity, pressure );
      }
      changed = false;
  }
}
```

call the setChanged() method when the state has changed enough to tell the observers

Flexible updating

- To have more flexibility in updating the observers the Subject may have a setChanged() method that allows the notifyObservers() to trigger the update()

```
setChanged() {
  changed = true;
}
public: void notifyObservers() const {
  if( changed ) {
      for( list< Observer* >::iterator itr = observers.begin();
           observers.end() != itr; ++itr ) {
        Observer* observer = *itr;
        observer->update( temperature, humidity, pressure );
      }
      changed = false;
  }
}
```

call the setChanged() method when the state has changed enough to tell the observers

check the flag to start the notifications

Observer and pointers

- In the pull version of Observer the Subject contains a pointer to an Observer, and the Observer can hold a pointer to the Subject.
- Neither object owns the other, and either object can be deleted at any time. They are separate objects that just happen to communicate with each other via pointers.
- Remind: before an Observer is deleted, it must be unregistered from every Subject it is observing. Otherwise, the Subjects will keep trying to Update it via the invalid pointer, causing undefined behavior.

Observer and pointers

- In the pull version of Observer the Subject contains a pointer to an Observer, and the Observer can hold a pointer to the Subject.
- Neither object owns the other, and either object can be deleted at any time. They are separate objects that just happen to communicate with each other via

Do this in the destructor

- Remind: before an Observer is deleted, it must be unregistered from every Subject it is observing. Otherwise, the Subjects will keep trying to Update it via the invalid pointer, causing undefined behavior.

Observer and pointers

Also remind to destroy the Subject after its Observers have been destroyed, or some Observers will try to unregister from an already destroyed Subject (again using an invalid pointer…

Do this in the destructor

- Remind: before an Observer is deleted, it must be unregistered from every Subject it is observing. Otherwise, the Subjects will keep trying to Update it via the invalid pointer, causing undefined behavior.

Observer and video games

- Some game engines (e.g. OGRE3D) let programmers extend Ogre::FrameListener and implement:
```
virtual void frameStarted(const FrameEvent& event)
virtual void frameEnded(const FrameEvent& event)
```
- These are methods called by the main game loop before and after the 3D scene has been drawn. Add code in those methods to create the game.

Observer and video games

- Some progr... impler... virt... Fram... virt... Fram...

```
class GameFrameListener : public Ogre::FrameListener {
public:
    virtual void frameStarted(const FrameEvent& event) {
        // Do things that must happen before the 3D scene
        // is rendered (i.e., service all game engine
        // subsystems).
        pollJoypad(event);
        updatePlayerControls(event);
        updateDynamicsSimulation(event);
        resolveCollisions(event);
        updateCamera(event);
        // etc.
    }
    virtual void frameEnded(const FrameEvent& event) {
        // Do things that must happen after the 3D scene
        // has rendered.
        drawHud(event);
        // etc.
    }
};
```

- These ... before ...
  Add code in those methods to create the game.

Model-View-Controller

The observer pattern and GUIs

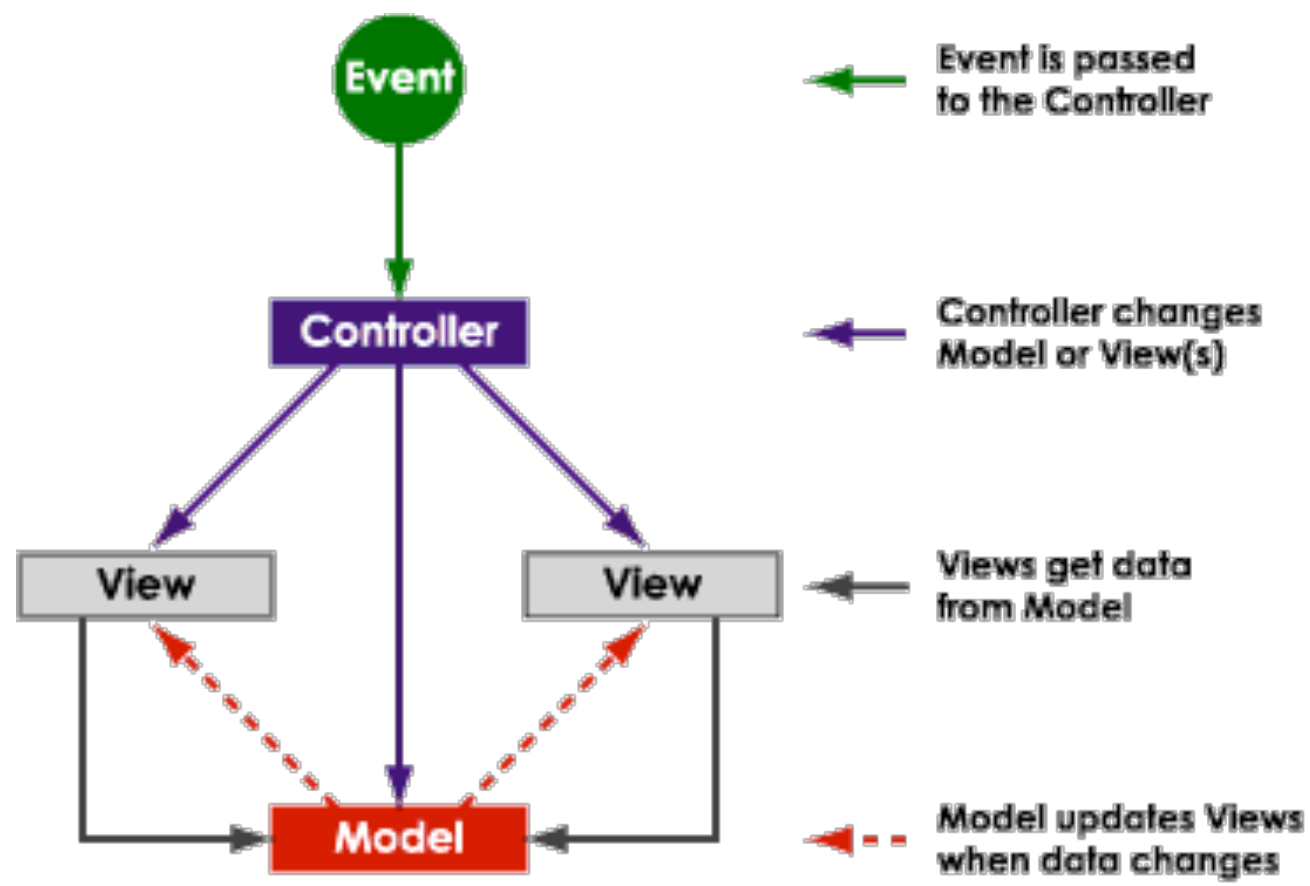
- The observer pattern is also very often associated with the model-view-controller (MVC) paradigm.
- In MVC, the observer pattern is used to create a loose coupling between the model and the view.
  Typically, a modification in the model triggers the notification of model observers which are actually the views.

MVC: GoF brief description

- “MVC consists of three kinds of objects. The **model** is the application object, the **view** is its screen presentation, and the **controller** defines the way the user interface reacts to user input. Before MVC, user interface designs tended to lump these objects together. MVC decouples them to increase flexibility and reuse.”

MVC schema

- The model maintains data, views display all or a portion of the data, and controller handles events that affect the model or view(s).
- Whenever a controller changes a model's data or properties, all dependent views are automatically updated.



MVC schema

Controller

- knows how this particular application works
- controls the view and the model

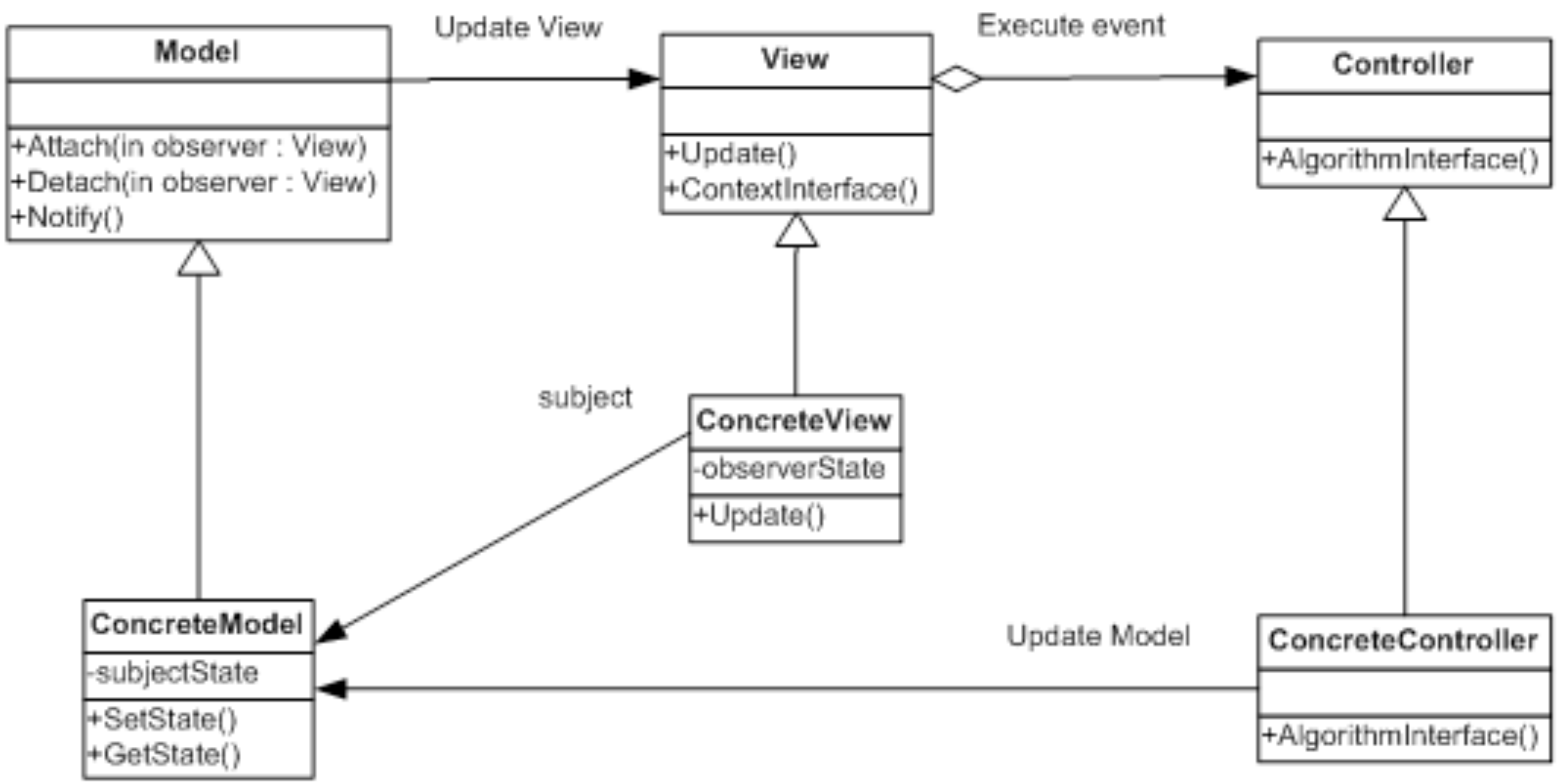
Model

- "real world"
- works when the controller asks it to work
- updates the view

View

- user interface
- knows how to communicate with the end user

MVC UML schema



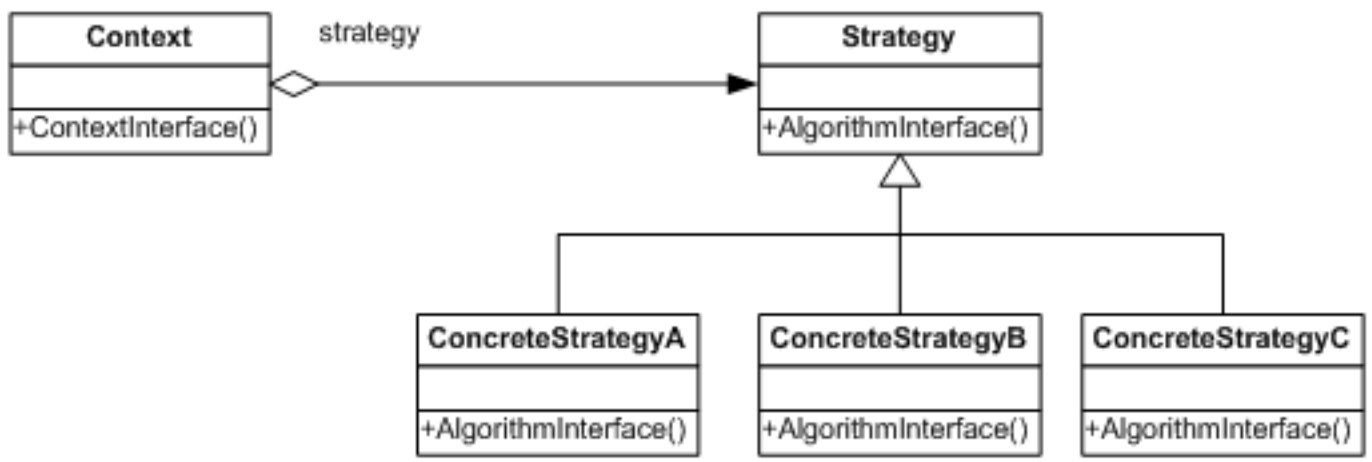
- The Model acts as a Subject from the Observer pattern and the View takes on the role of the Observer object.

MVC: a composite pattern

- MVC is not defined as a design pattern per-se in GoF, but is rather referred to as a "set of classes to build a user interface" that uses design patterns such as Observer, Strategy, and Composite."
- The relation between Model and View is implemented as Observer…

MVC: a composite pattern

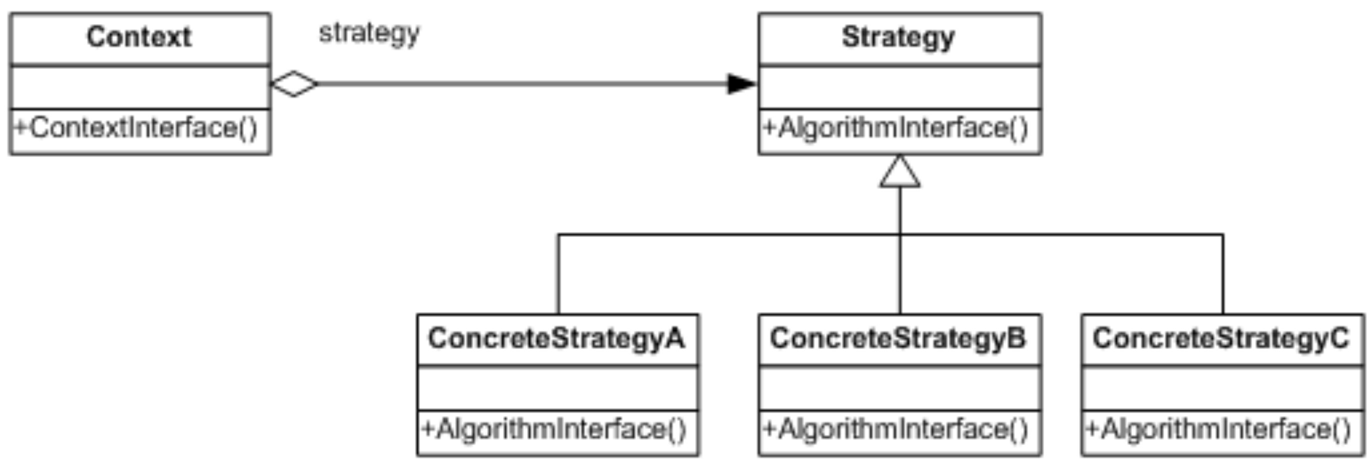
- The relation between View and Controller is implemented using the Strategy pattern



- The View uses the Controller to implement a specific type of response. The controller can be changed to let the View respond differently to user input.

MVC: a composite pattern

- The relation between View and Controller is implemented using the Strategy pattern



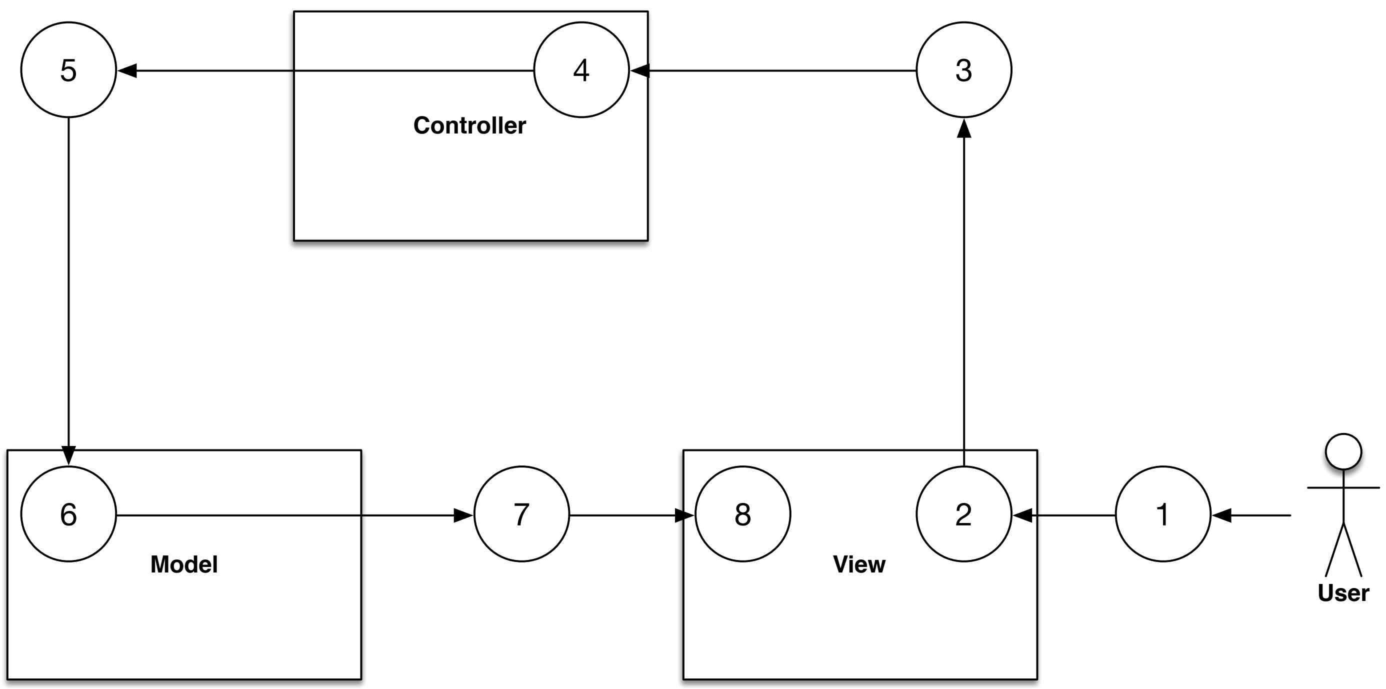
Strategy pattern: we have objects that hold alternate algorithms to solve a problem.

In this DP an algorithm is separated from the object that uses it, and encapsulated as its own object.

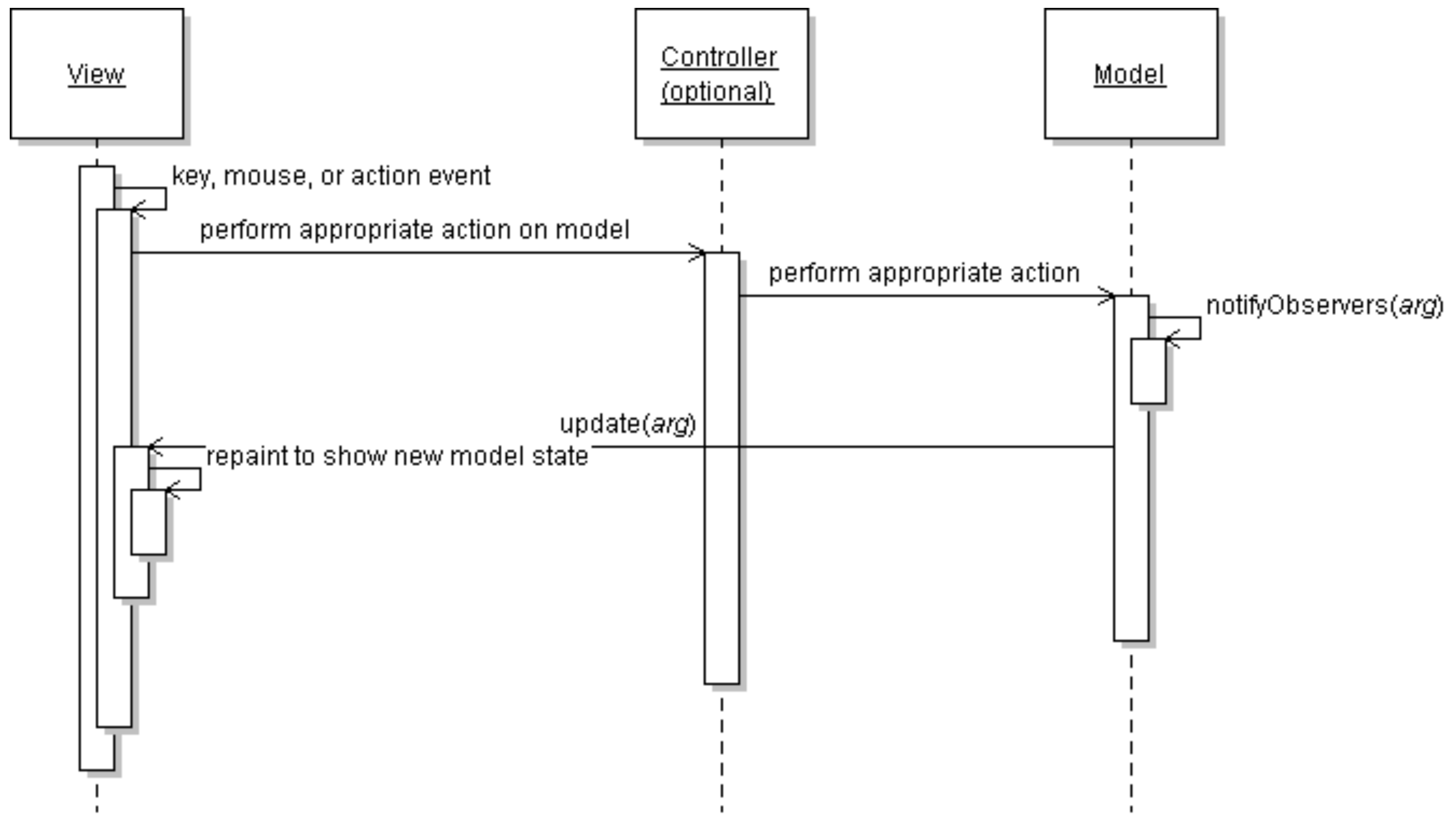
- A concrete strategy implements one behavior, one implementation of how to solve the same problem
- It separates algorithm for behavior from object that wants to act
- Allows changing an object's behavior dynamically without extending / changing the object itself

MVC in action

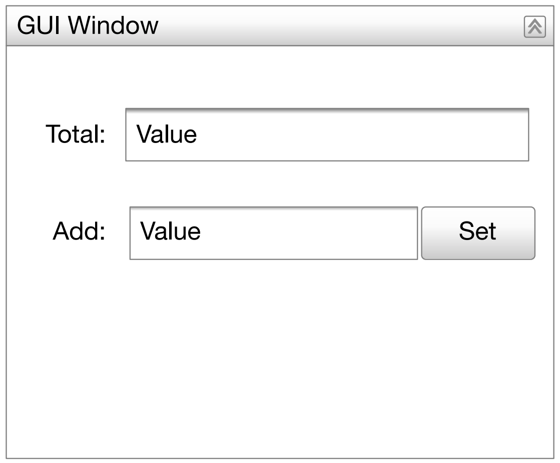
1. The end user manipulates the view (e.g. presses a button).
2. The user's actions are interpreted by the view.
3. The view passes the interpreted commands to the controller.
4. The controller decides what to do…
5. …and the controller makes the model act.
6. The model acts independently, and according to the requests of the controller
7. The model notifies the views interested in its updates
8. Each notified view displays the information in its own way.



MVC: sequence diagram



MVC in action



Controller

```
Controller::setBalance(int val) {
  model->setTotal(v);
}
```

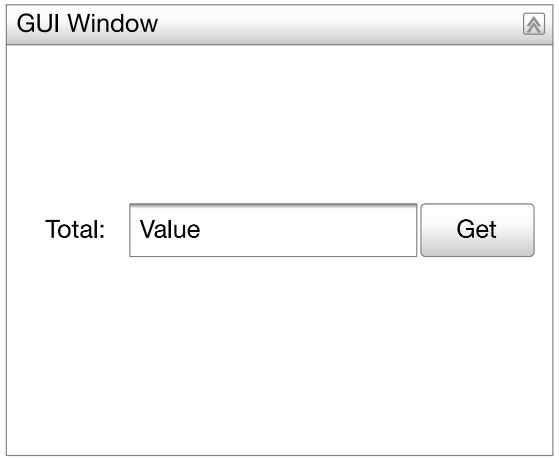
Model

```
Model::setTotal(int val) {
  total = v;
  notify();
}

Model::notify() {
  for( view : views )
    view->showTotal(total);
}
```

View

```
View::setButtonPressed() {
  int value = addTextField.getInt();
  controller->setBalance(value);
}

View::showTotal(int total) {
  totalTextField.setInt(total);
}
```

MVC in action - 2



View

```
View::getButtonPressed() {
  controller->getBalance();
}

View::showTotal(int total) {
  totalTextField.setInt(total);
}
```

Controller

```
Controller::getBalance() {
  int total = model->getTotal();
  view->showTotal(total);
}
```

Model

```
Model::getTotal() {
  return total;
}

Model::notify() {
  for( view : views )
    view->showTotal(total);
}
```

Model

- represents the“real world”
- is capable of completing “real world”tasks independently
- is controlled by the controller
  - the controller makes the model act
  - the model updates the view
- has methods to set/get its state

View

- it is the user interface
  - all the callback functions of the windowing system
  - all the widgets of the windowing system
- knows how to communicate with the enduser
  - knows how to present things to the end user
  - knows how to receive the end user's actions
- does not decide what to do with the user's actions: lets the controller decide
- has feedback, manipulation and query methods:
  - has methods like showValueInTextField(), setRadioButtonX() to present things to the end user,
  - has methods like onButtonXXXPressed(), onSliderYYYMoved() to capture the end user's actions,
  - has methods like getDropDownSelection(), getCheckBoxXX() to capture the end user's selections made some time ago

Controller

- Controls the application
  - makes application specific decisions
  - knows how this application should behave
- Controls the application by making the model and the view act
  - knows WHAT the model and the view are capable of doing
  - knows WHAT the model and the view should do
  - doesn't know HOW things are done inside the model and the view

Example

```
class Observer {
public:
    virtual ~Observer() {}
    virtual  void update() = 0;
};
```

```
class Subject {
public:
    virtual ~Subject() {}
    virtual void notify() = 0;
    virtual void addObserver(Observer* o) = 0;
    virtual void removeObserver(Observer* o) = 0;
};
```

Example: Model

```
class Model : public Subject {

public:

int getData(){
    return data;
}

void setData(const int i) {
    data = i;
    notify();
}

virtual void
addObserver(Observer* o) {
    observers.push_back(o);
}

virtual void
removeObserver(Observer* o) {
    observers.remove(o);
}

virtual void notify() {
    for (Observer* observer :
                        observers)
        observer->update();
}

private:

  int data = 0;

  list<Observer*> observers;

};
```

Example: Controller

```
class Controller {
public:

  Controller(Model* m) : model(m) {}

  void increment() {
       int value = model->getData();
       value++;
       model->setData(value);
  }

  void decrement() {
       int value = model->getData();
       value--;
       model->setData(value);
  }

private:

  Model* model;

};
```

Example: View

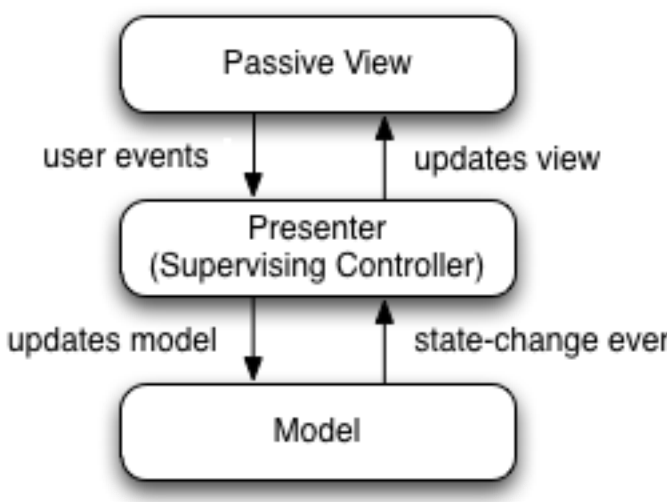
```
class View : public Observer {
public:

View(Model* m, Controller* c) {
     model = m;
     model->addObserver(this);
     controller = c;
}

virtual ~View() {
    model->removeObserver(this);
}

void displayTextField(int i) {
     cout << "Text field: " << i << endl;
}

virtual void update() {
     int value = model->getData();
     displayTextField(value);
}

void incrementButton() {
     controller->increment();
}

void decrementButton() {
     controller->decrement();
}

private:

  Model* model;
  Controller* controller;

};
```

Example: main

```
Model* model = new Model;
Controller* controller = new Controller(model);
View* view = new View(model, controller);
// simulate user interaction:
view->incrementButton();
view->incrementButton();
view->incrementButton();
view->decrementButton();
```

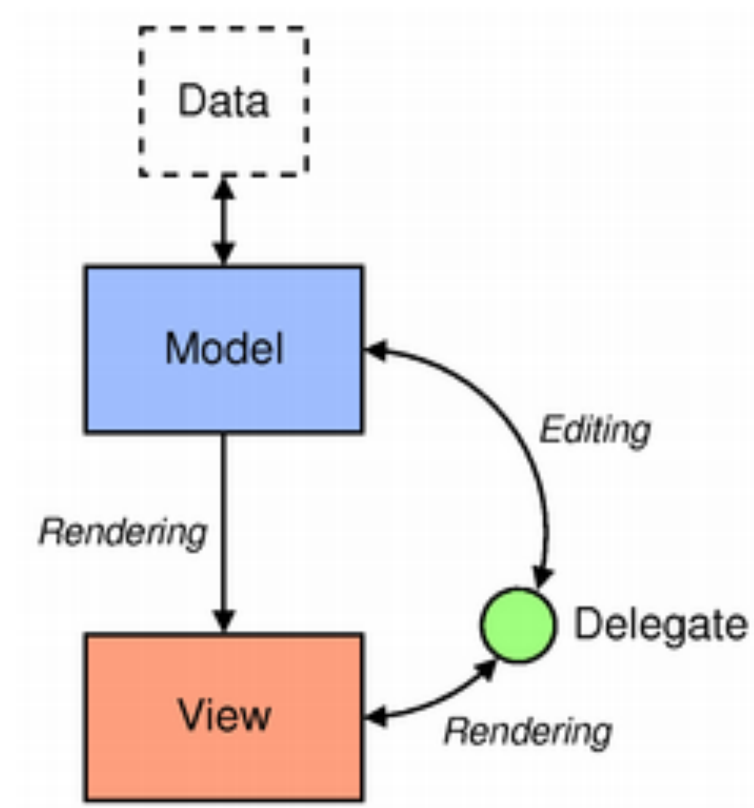
View and Controller and MVP

- Often View and Controller classes are merged
- There are variations of MVC, like MVP (Model-View-Presenter) where View and Controller are merged. A new actor, called Presenter is introduced; it can access the View and the Model directly, and the Model-View relationship can still exist where relevant. The Presenter can update the model and the view directly.



MVC and QT

- The QT Framework has a different architecture: Model/View/Delegates
- Delegates: classes that provide complete control over presentation and editing of data items, instead of using controllers. A Delegate has access to both view and model.
- If you do not use the delegates, then in QT you use a Model/View architecture, in which the Controller is merged with the View



MVC with wxWidgets and QT

wxWidgets

- It is an open source cross-platform GUI toolkit.
  - Programmers use the same API to write GUI applications on multiple platforms
  - The same application is recompiled on a different system and it will look as native on each system
  - Provides helper libraries for networks, multimedia, multithreading, HTML viewing, etc.

wxWidgets: model

It is the same of the previous example:

```
class Model : public Subject {
public:
    int getData() const {
        return data;
    }
    void setData(const int i);
    virtual void addObserver(Observer* o)
override;
    virtual void removeObserver(Observer* o) override;
    virtual void notify() override;
private:
    int data = 0;
    std::list<Observer*> observers;
};
```

wxWidgets: Controller

- Also the controller is the same of the previous example:

```
#include "Model.h"
class Controller {
public:
    Controller(Model* m) : model(m) {}
    void increment();
    void decrement();
private:
    Model* model;
};
```

wxWidgets: View

- The view is necessarily different: it has to deal with the way in which wxWidgets shows the GUI elements.
- We can extend a base class `wxFrame`, that represents a window whose size and position can (usually) be changed by the user.
  - It usually has thick borders and a title bar, and can optionally contain a menu bar, toolbar and status bar. A frame can contain any window that is not a frame or dialog.

wxWidgets: View

- Since View extends `wxFrame` it is possible to show it on the screen. Since it has to be an Observer for the Model we can use multiple inheritance.
- As a `wxFrame` it contains all the GUI widgets, and places them within the panel of the frame.
- It contains pointers to model and controller and uses them to get the data to be shown and to invoke the methods to change the model.

wxWidgets:View

```
class View : public wxFrame, public Observer {
private:

protected:
    Model* model;
    Controller* controller;

    wxStaticText* staticText;
    wxTextCtrl* textCtrl;
    wxButton* incrementButton;
    wxButton* decrementButton;

    virtual void onIncrementButtonClick( wxCommandEvent& event );
    virtual void onDecrementButtonClick( wxCommandEvent& event );

public:
    View(Model* model, Controller* controller, wxWindow* parent=NULL, wxWindowID id =
wxID_ANY, const wxString& title = wxEmptyString, const wxPoint& pos =
wxDefaultPosition, const wxSize& size = wxSize( 500,300 ), long style =
wxDEFAULT_FRAME_STYLE|wxTAB_TRAVERSAL );
    virtual ~View();
    virtual void update() override;
};
```

Some GUI widgets receive an input from the user, e.g. buttons are clicked. Upon these events there is code that has to be executed (callbacks)

wxWidgets: View

```
class View : public wxFrame, public Observer {
private:

protected:
    Model* model;
    Controller* controller;

    wxStaticText* staticText;
    wxTextCtrl* textCtrl;
    wxButton* incrementButton;
    wxButton* decrementButton;

    virtual void onIncrementButtonClick( wxCommandEvent& event );
    virtual void onDecrementButtonClick( wxCommandEvent& event );
```

Some GUI widgets receive an input from the user, e.g. buttons are clicked. Upon these events there is code that has to be executed (callbacks)

The callbacks invoke the controller methods:

```
void View::onIncrementButtonClick(wxCommandEvent &event) {
    controller->increment();
}

void View::onDecrementButtonClick(wxCommandEvent &event) {
    controller->decrement();
}
```

wxWidgets:View

- The constructor sets up the GUI elements and registers to the model:

```
View::View(Model* model, Controller* controller, wxWindow* parent,
wxWindowID id, const wxString& title, const wxPoint& pos, const wxSize&
size, long style ) : wxFrame( parent, id, title, pos, size, style ) {
    this->model = model;
    this->model->addObserver(this);
    this->controller = controller;

    this->SetSizeHints( wxDefaultSize, wxDefaultSize );
    wxBoxSizer* frameSizer;
    frameSizer = new wxBoxSizer( wxHORIZONTAL );
    staticText = new wxStaticText( this, wxID_ANY, wxT("Value"),
                                   wxDefaultPosition, wxDefaultSize, 0 );
    staticText->Wrap( -1 );
    frameSizer->Add( staticText, 0, wxALL, 5 );
    //...
```

wxWidgets: View

- The `update()` method gets data from the model and updates the widgets of the GUI:

```
void View::update() {
    int value = model->getData();
    wxString wxIntString =
                    wxString::Format(wxT("%i"),value);
    textCtrl->ChangeValue(wxIntString);
}
```

- When building the GUI the callbacks are associated to the events generated by the widgets:

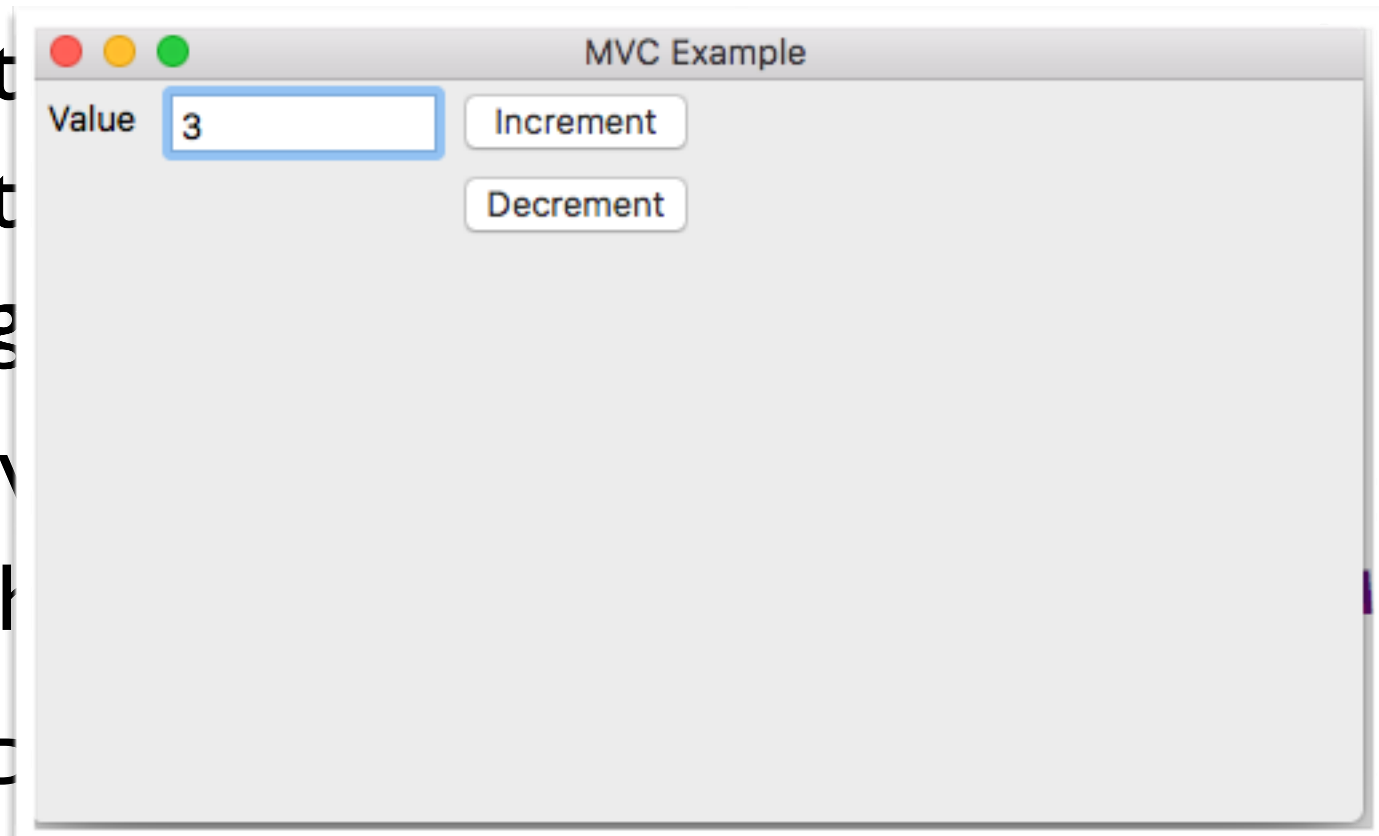
```
incrementButton->Connect( wxEVT_COMMAND_BUTTON_CLICKED,
wxCommandEventHandler( View::onIncrementButtonClick ), NULL, this );
decrementButton->Connect( wxEVT_COMMAND_BUTTON_CLICKED,
wxCommandEventHandler( View::onDecrementButtonClick ), NULL, this );
```

wxWidgets: main

- In wxWidgets there's no explicit `main()`
- We create a `wxApp` object, with an `OnInit()` method that instantiate at least a `wxFrame` or wxDialog and shows it.
- In our MVC we also instantiate a model, a controller and set their connections.
- A `wxIMPLEMENT_APP` macro instantiates the `wxApp` objects and executes it, basically providing the `main()` procedure access point to the program.

wxWidgets: main

- In wxWidgets there's no explicit `main()`
- We creat … `:()`
  method t … or
  wxDialog
- In our M … ontroller
  and set t
- A `wxIMP` … the
  `wxApp` objects and executes it, basically providing the `main()` procedure access point to the program.



QT

- It is an extremely popular open source cross-platform GUI toolkit.
  - Programmers use the same API to write GUI applications on multiple platforms
  - The same application is recompiled on a different system and it will adapt its look on each system
    - Differently from wWidgets the QT toolkit does not use the native widgets but re-implements them with low level graphic APIs
  - It provides many helper libraries, from databases to networking

QT: model and controller

- They remain the same of the wxWidgets example, and of the text-based example
- QT offers several classes that can be used as models, but for the sake of consistency this example shows how to implement the same general architecture of the previous example

QT: view

- The view class extends some QT window class or specific QT view class
  - In this example we are going to extend QMainWindow and Observer
  - Widgets and description of how to place and connect them in the panel is within a different class, that is included as an attribute

QT: view helper

- This simple class (usually automatically generated with a RAD tool) contains widgets and a setup method that places them within a window.
- Instead of registering callbacks QT connects so called SIGNALs (generated by some widget or object) to SLOTs (associated to methods/ function):

```
QObject::connect(incrementButton, SIGNAL(clicked()),
                 View, SLOT(onIncrementButton()));

QObject::connect(decrementButton, SIGNAL(clicked()),
                 View, SLOT(onDecrementButton())
```

QT: view

```
class View : public QMainWindow, public
Observer {
    Q_OBJECT
public:
    View(Model* m, Controller* c,
         QWidget *parent = 0);
    ~View();
    virtual void update() override;

private slots:
    void onIncrementButton();
    void onDecrementButton();

private:
    ViewWindow* ui;
    Model* model;
    Controller* controller;
};
```

```
View::View(Model* m, Controller* c,
           QWidget* parent) :
       QMainWindow(parent),
       ui(new ViewWindow()),
       model(m), controller(c) {

    model->addObserver(this);
    ui->setupUi(this);
    update();
}

void View::update() {
    int value = model->getData();
    ui->textCtrl
     ->setText(QString::number(value));
}
```

QT: view

The methods associated with the slots invoke the controller methods:

```
void View::onIncrementButton() {
    controller->increment();
}

void View::onDecrementButton() {
    controller->decrement();
}
```

```
class View : p
Observer {
    Q_OBJECT
public:
    View(Model
        QWidg
    ~View();
    virtual void update() override;

private slots:
    void onIncrementButton();
    void onDecrementButton();

private:
    ViewWindow* ui;
    Model* model;
    Controller* controller;
};
```

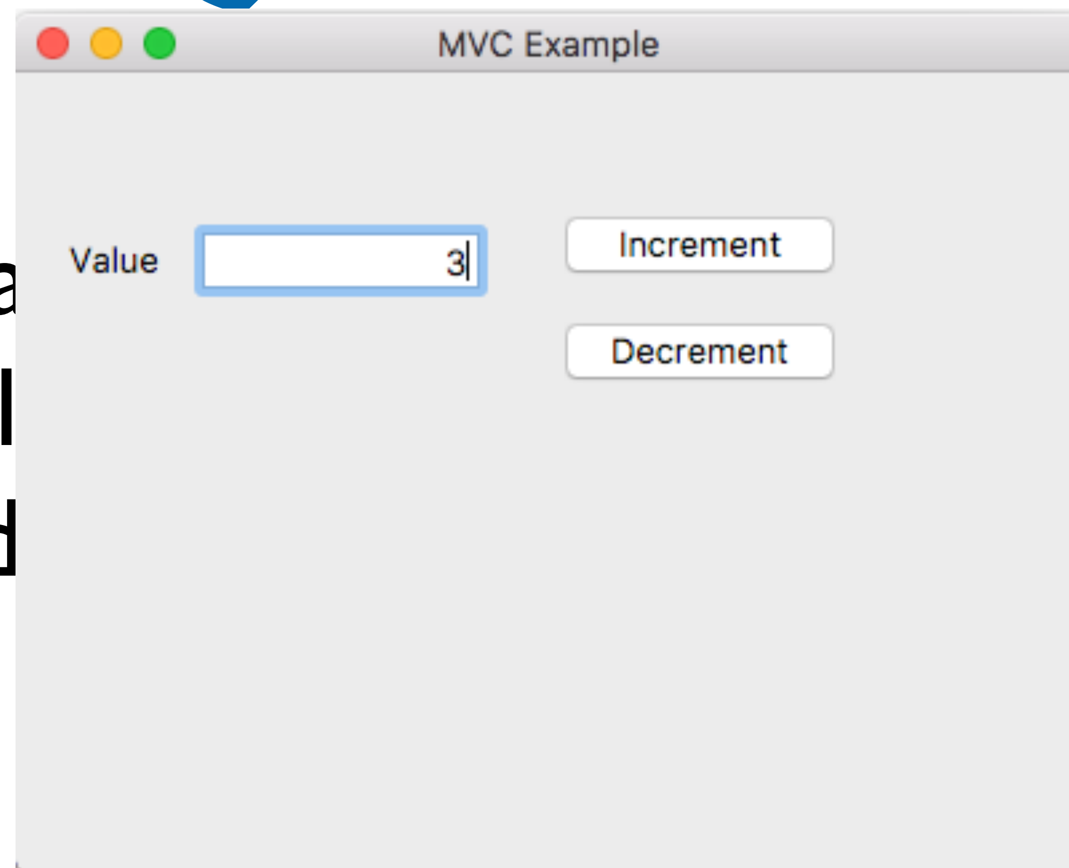
```
Controller* c,
arent) :
rent),
dow()),
roller(c) {

r(this);
    ui->setupUi(this);
    update();
}

void View::update() {
    int value = model->getData();
    ui->textCtrl
     ->setText(QString::number(value));
}
```

QT: main

- A QApplication object is instantiated in main(), then model, controller and view are instantiated and the view object is shown with the show() method:

```
QApplication app(argc, argv);

Model* model = new Model;
Controller* controller = new Controller(model);
View view(model, controller);
view.show();

return app.exec();
```

QT: main



- A QApplica… ed in main(), then model … e instantiated … shown with the show()

```
QApplication app(argc, argv);

Model* model = new Model;
Controller* controller = new Controller(model);
View view(model, controller);
view.show();

return app.exec();
```

Reading material

- M. Bertini, "Programmazione Object-Oriented in C++", parte II, cap. 3

Credits

- These slides are based on the material of:
  - Glenn Puchtel
  - Fred Kuhns, Washington University
  - Aditya P. Matur, Purdue University