Programmazione

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

Design pattern

Observer

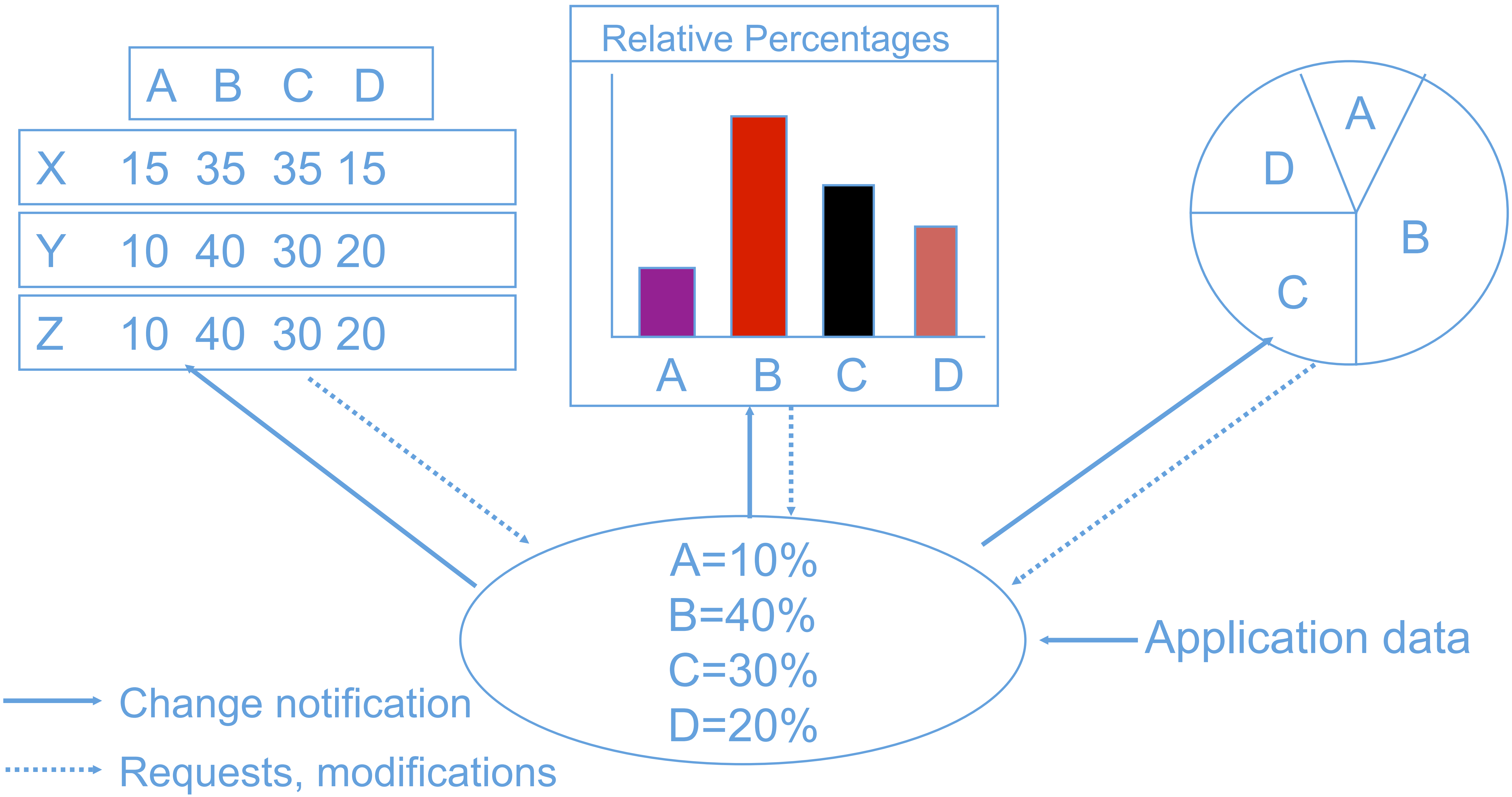
Some motivations

- 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

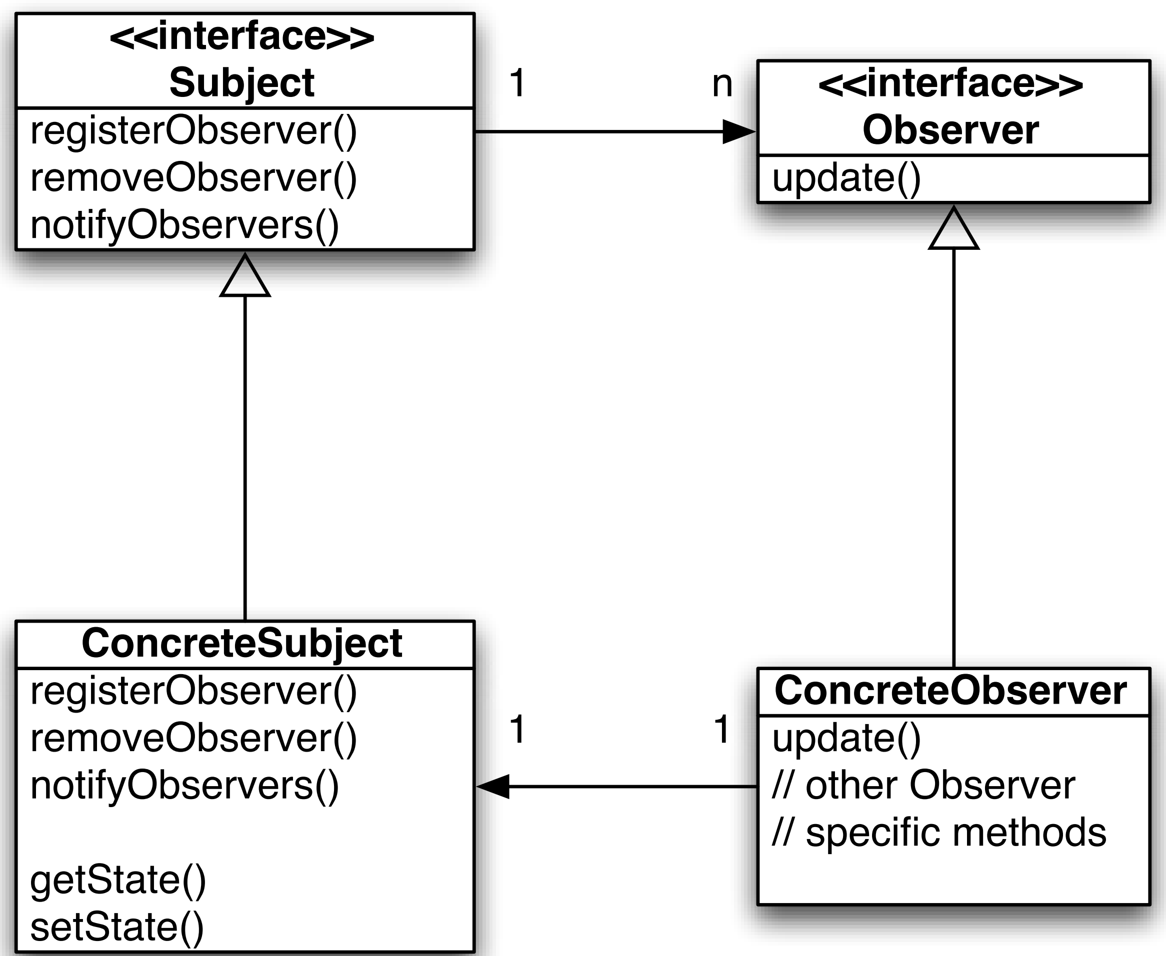
Participants

- 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() = 0 {};


  public: virtual void
registerObserver( Observer* o ) = 0;

  public: virtual void
removeObserver( Observer* o ) = 0;

  public: virtual void
notifyObservers() const = 0;

};
```

```
class Observer {

  protected: virtual ~Observer() = 0 {
  };

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

};
```

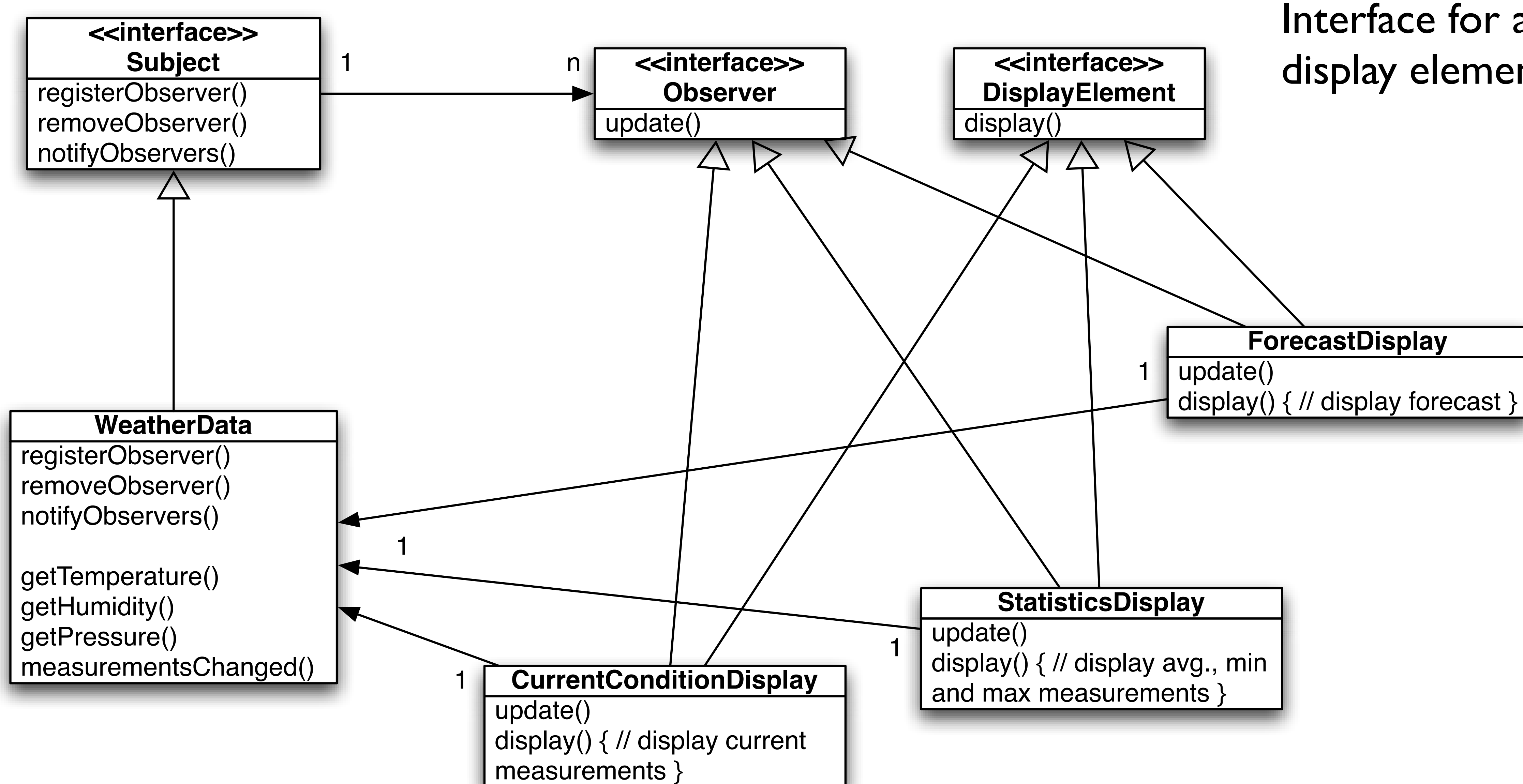
Observer example

```
class Subject {
  protected: virtual ~Subject() = 0 {};


  public: virtual void
registerObserver( Observer* o ) = 0;

  public: virtual void
removeObserver( Observer* o ) = 0;

  public: virtual void
notifyObservers() const = 0;

};
```

```
class Observer {

  protected: virtual ~Observer() = 0 {
  };

  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

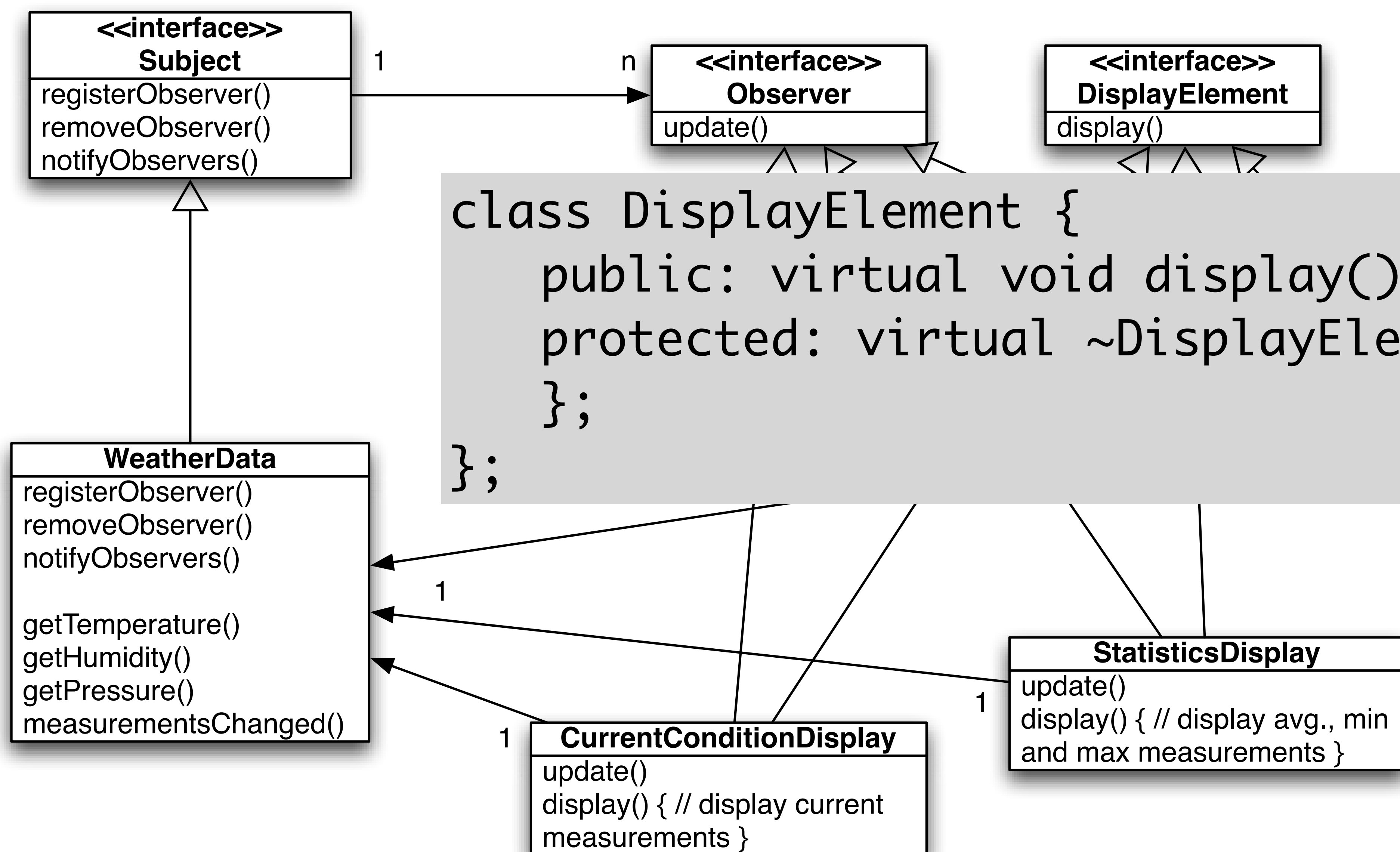


Interface for all the display elements

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: float pressure;

hu
      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: float pressure;

hu

      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 temperature, float humidity,
float pressure ) {
    temperature = temperature;
    humidity = humidity;
    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 temperature, float humidity,
float pressure ) {
    temperature = temperature;
    humidity = humidity;
    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 temperature, float humidity,
float pressure ) {
    temperature = temperature;
    humidity = humidity;
    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

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

The constructor gets the Subject and use it to register to it as an 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 temperature, float humidity,
float pressure ) {
    temperature = temperature;
    humidity = humidity;
    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.

Remind to unregister the observer when it is destroyed.

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

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

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

```
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

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

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

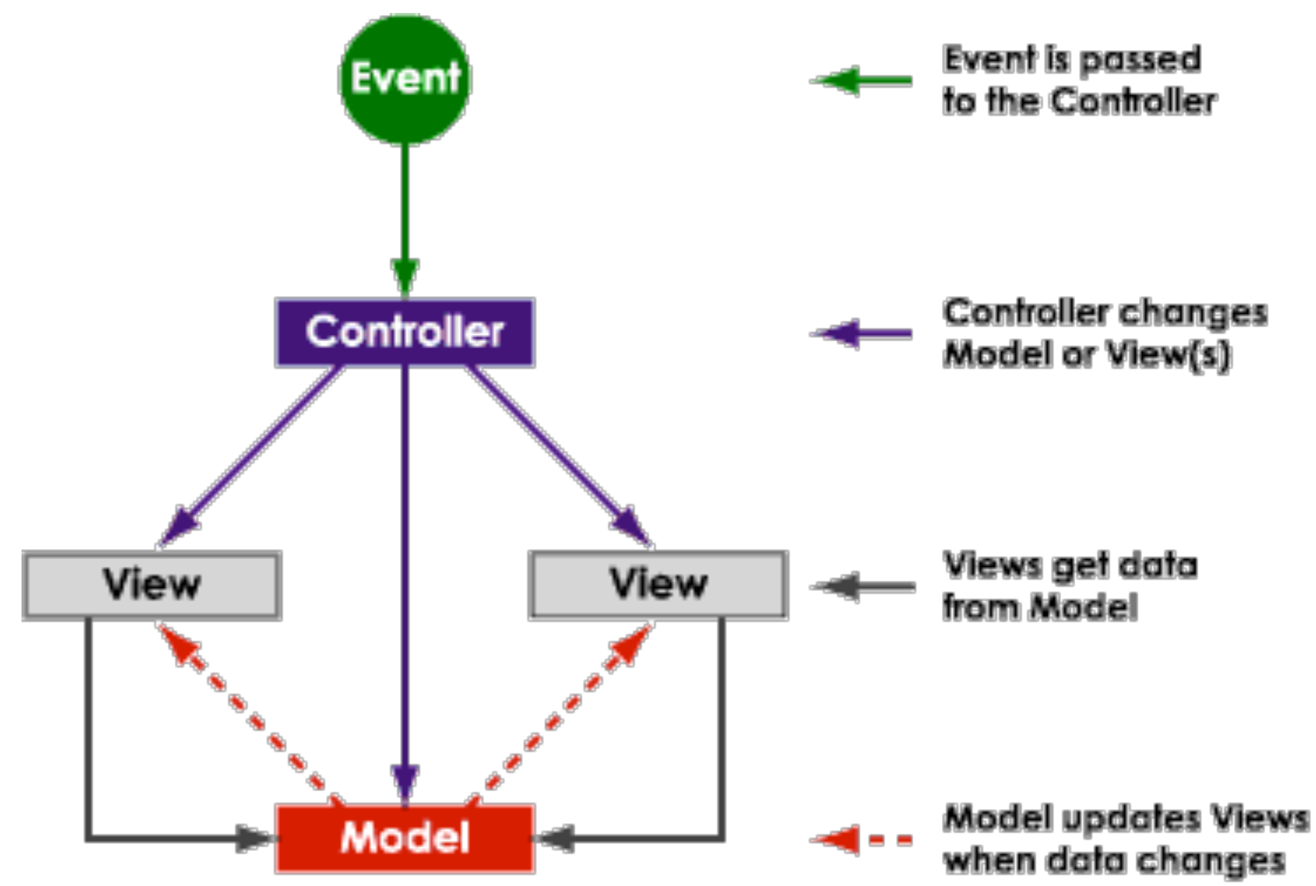
check the flag to start the notifications

The observer pattern and GUIs

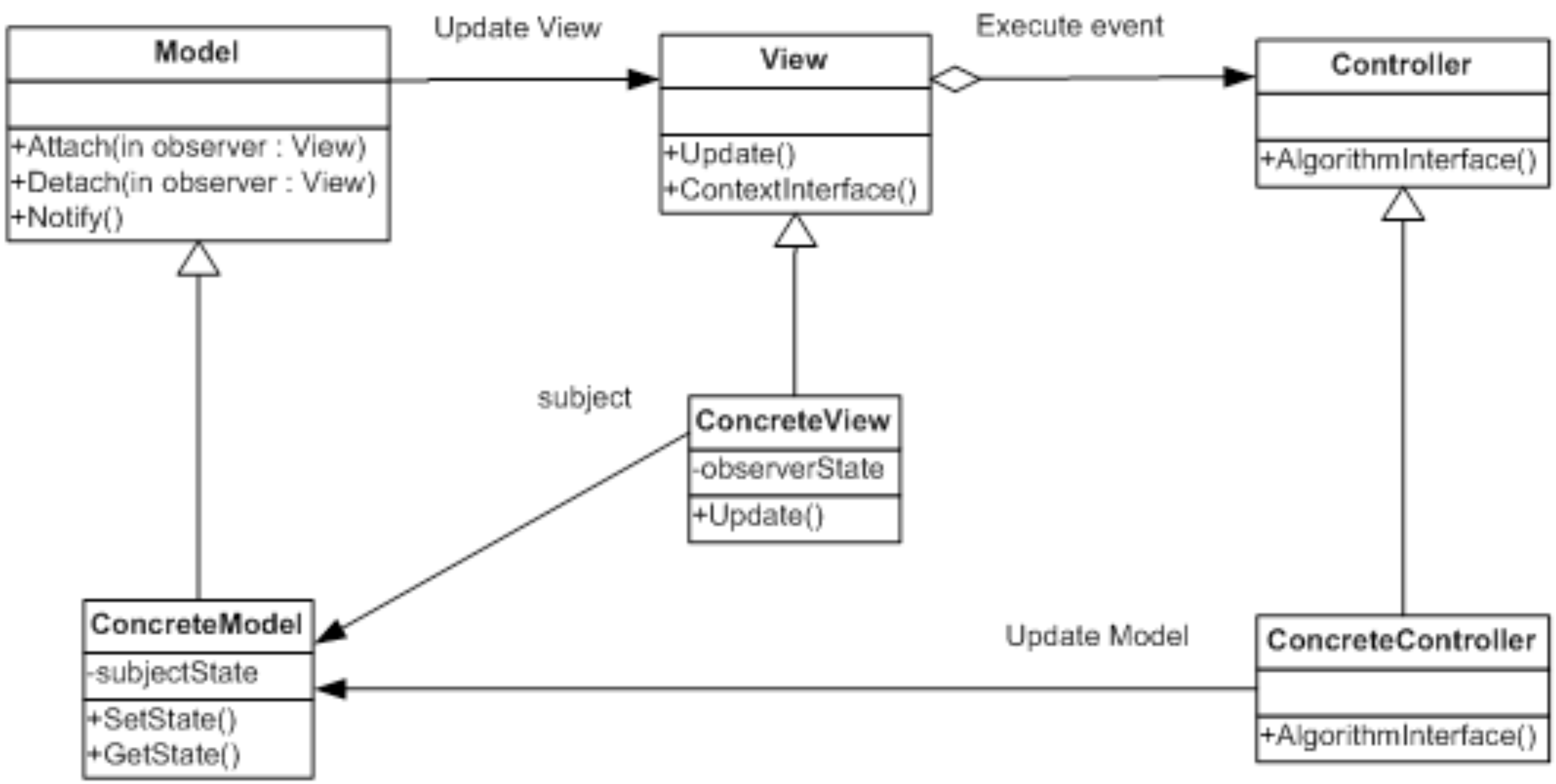
- The observer pattern is also very often associated with the model-view-controller (MVC) paradigm.
- In MVC, the observer pattern is used to create a loose coupling between the model and the view.
  Typically, a modification in the model triggers the notification of model observers which are actually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

MVC 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 UML schema



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

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
imple
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 been rendered.
          drawHud(event);
          // etc.
    }
};
```

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

Credits

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