YORK
UNIVERSITÉ
UNIVERSITY

EECS3311 – Software Development

Observer Design Pattern

Model View Controller

Event-Driven Software

Assessing Software Architectures

Weather-O-Rama Inc
Statement of Work

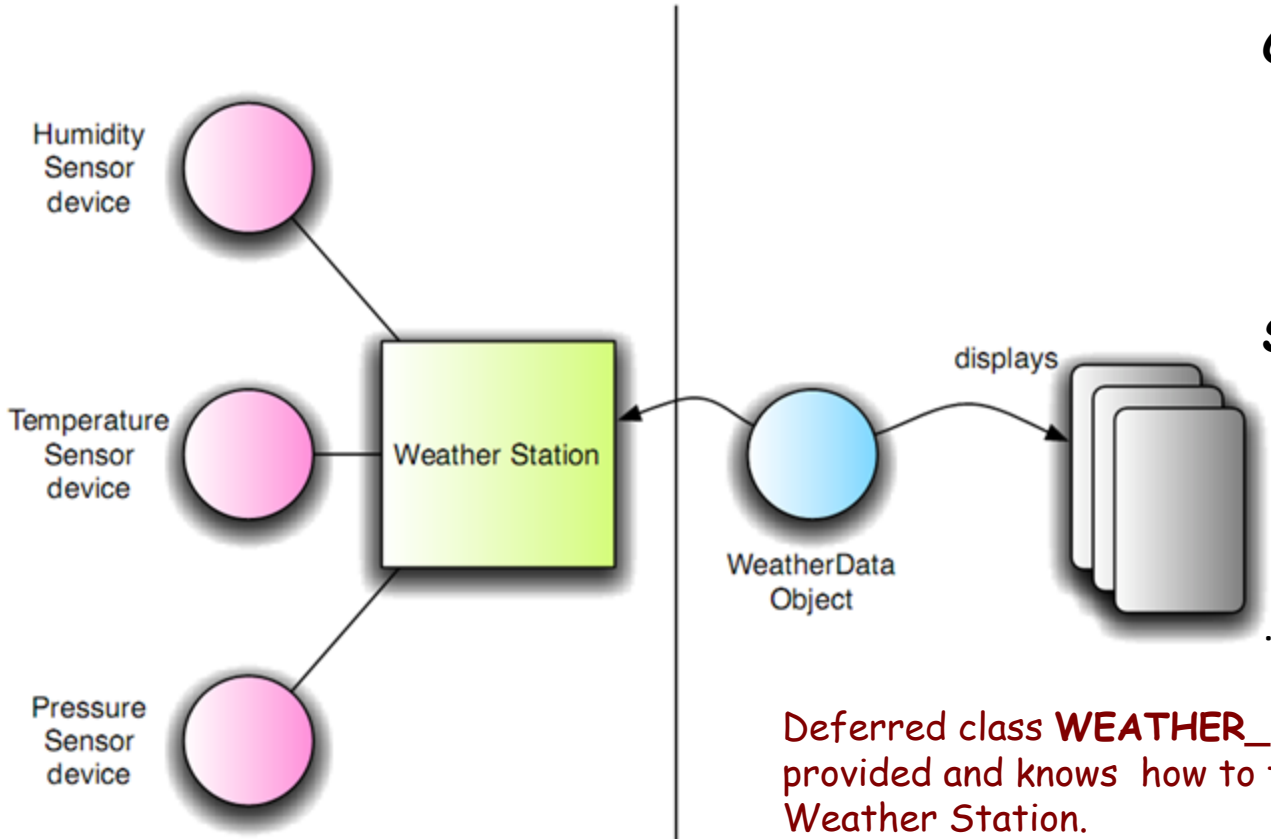
- Congratulations on being selected to build our next generation of internet based weather monitoring station.
- Details follow below.
- We look forward to seeing your design by tomorrow night!

Sincerely

John Hurricane CEO

P.S. We are overnighting the WEATHER_DATA source files to you

Weather-O-Rama



Current Conditions
Temp: 73F
Humidity: 60
Pressure: ↓

Statistics
Avg. Temp: 62F
Min. Temp: 50F
Max. Temp: 78F

… other views …..

Deferred class **WEATHER_DATA** is provided and knows how to talk to the Weather Station.

What must we implement?
Create an app that uses **WEATHER_DATA** to update the various displays real-time

Hardware provided by Weather-O-Rama

You have been hired by Weather-O-Rama Inc. to build its next generation Weather Monitoring Station. The system so far contains a class called `WEATHER_DATA` with the following features:

The `measurements_changed` method is called automatically when any of the three measurements (temperature, humidity, pressure) is changed (we don't know or care how it is called, we just know that it is).

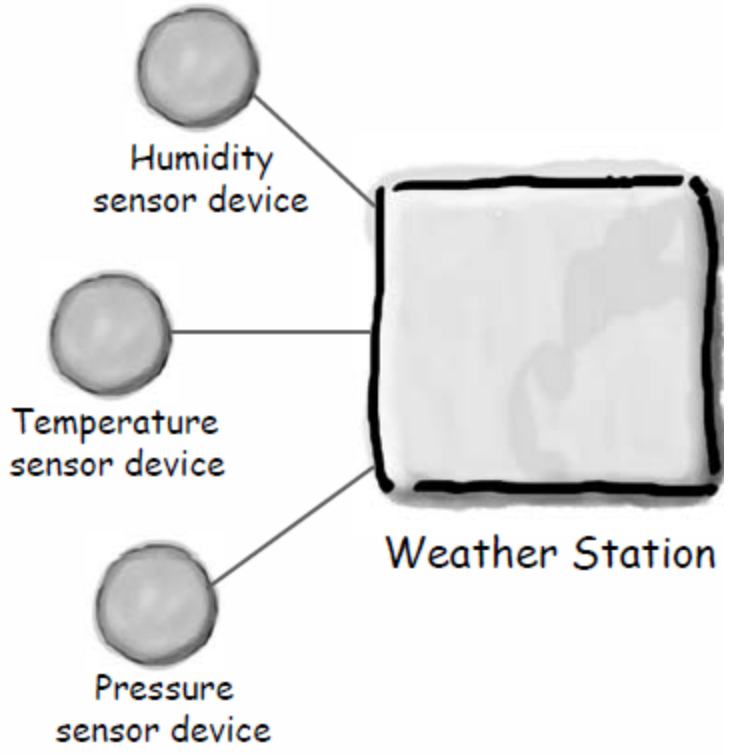
Your task is to

1. Fill in the code for `measurements_changed`.
2. Create an application that initially provides two display elements: current conditions and weather statistics. These displays are updated in real time as a `WEATHER_DATA` object acquires the most recent measurements.
3. Allow third party developers to create their own weather displays and plug them right in!

How would you design this system? Do not be concerned with GUI code for the displays. Textual displays are sufficient for this exercise.

We are provided with

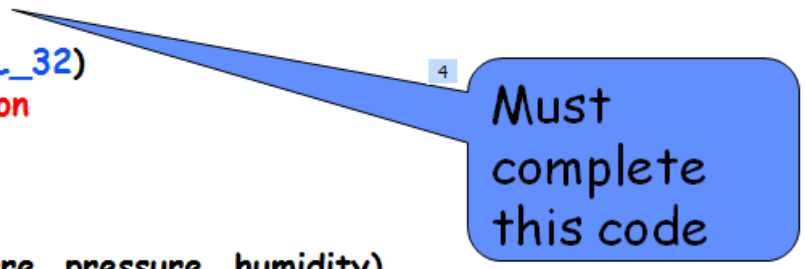
```
deferred class
  WEATHER_DATA

feature -- weather data available to observers
  temperature: REAL
  humidity: REAL
  pressure: REAL

  correct_limits (t, p, h: REAL_32): BOOLEAN
    ensure
      Result implies -36 <= t and t <= 60 -- oC
      Result implies 50 <= p and p <= 110 -- KPa
      Result implies 0.8 <= h and h <= 100 --%RH

  measurements_changed -- called from weather station after a set_measurement

  set_measurements (t, p, h: REAL_32)
      -- called from weather station
    require correct_limits (t, p, h)

invariant
        correct_limits (temperature, pressure, humidity)
end
```



Must complete this code

- We are provided with class WEATHER_DATA
- But this class is closed for business

```
deferred class
  WEATHER_DATA

feature -- weather data available to observers
  temperature: REAL
  humidity: REAL
  pressure: REAL

  correct_limits (t, p, h: REAL_32): BOOLEAN
    ensure
      Result implies -36 <= t and t <= 60 -- oC
      Result implies 50 <= p and p <= 110 -- KPa
      Result implies 0.8 <= h and h <= 100 --%RH

  measurements_changed -- called from weather station after a set_measurement

  set_measurements (t, p, h: REAL_32)
      -- called from weather station
    require correct_limits (t, p, h)

invariant
        correct_limits (temperature, pressure, humidity)
end
```



First Design

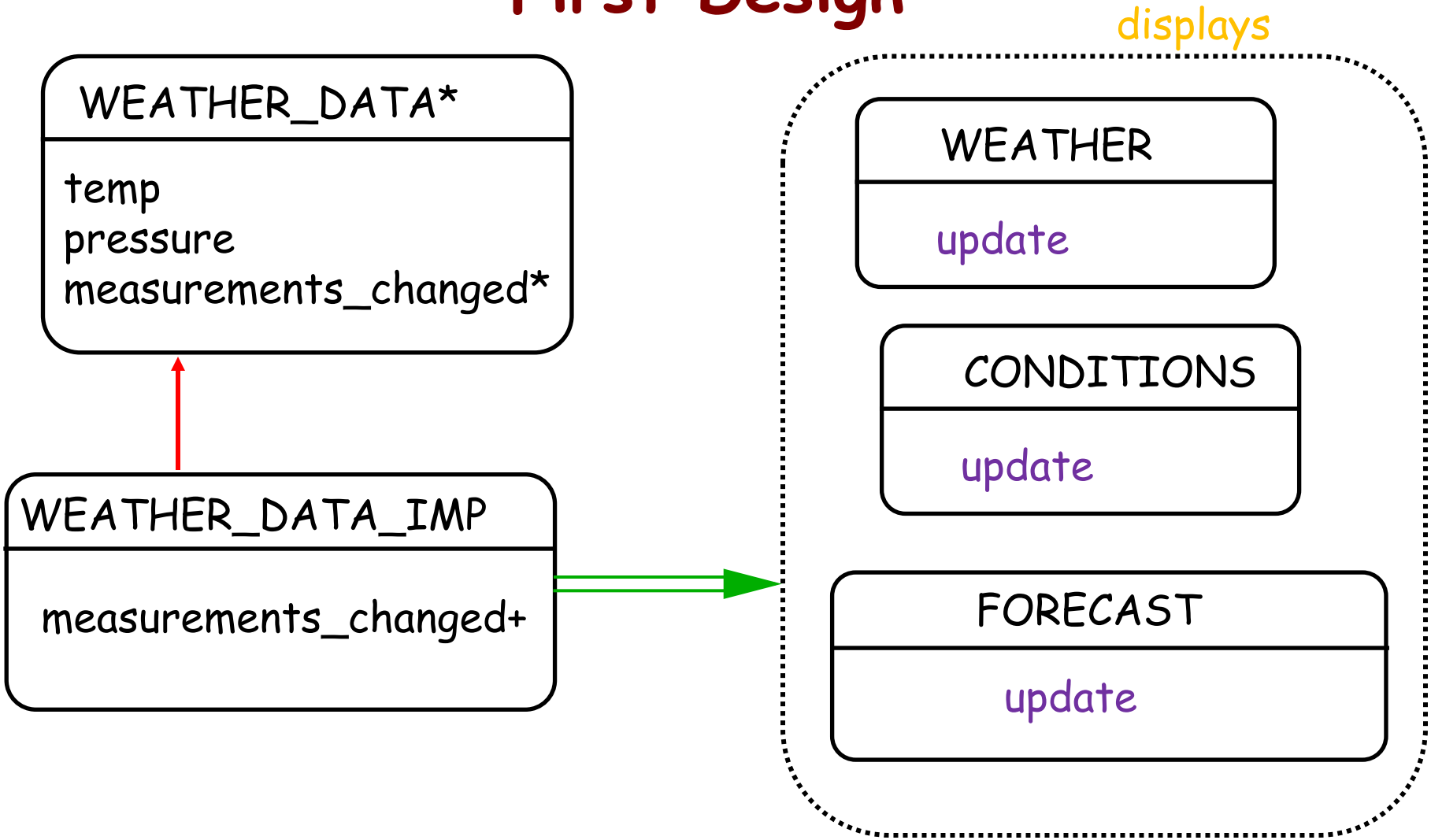
Use Inheritance
Open Closed Principle

```
class WEATHER_DATA_IMP inherit
  WEATHER_DATA

feature

  current_conditions: CURRENT_CONDITIONS
  statistics: STATISTICS
  forecast: FORECAST

  ...

  measurements_changed
    do
      current_conditions.update(temp, humidity, pressure)
      weather_stats.update(temp, humidity, pressure)
      forecast.update(temp, humidity, pressure)
    end
end
```

Problems?

on
update" interface

First Design



```
class WEATHER_DATA_IMP inherit
  WEATHER_DATA
feature  …
  measurements_changed
    do
      current_conditions.update(temp, humidity, pressure)
      weather_stats.update(temp, humidity, pressure)
      forecast.update(temp, humidity, pressure)
    end
end
```

Part that changes

Sharpen your pencil

Based on our first implementation, which of the following apply? (Choose all that apply.)

- ❑ A. We are coding to concrete implementations, not interfaces.
- ❑ B. For every new display element we need to alter code.
- ❑ C. We have no way to add (or remove) display elements at run time.
- ❑ D. The display elements don't implement a common interface.
- ❑ E. We haven't encapsulated the part that changes.
- ❑ F. We are violating encapsulation of the WeatherData class.

Problems?

First Design

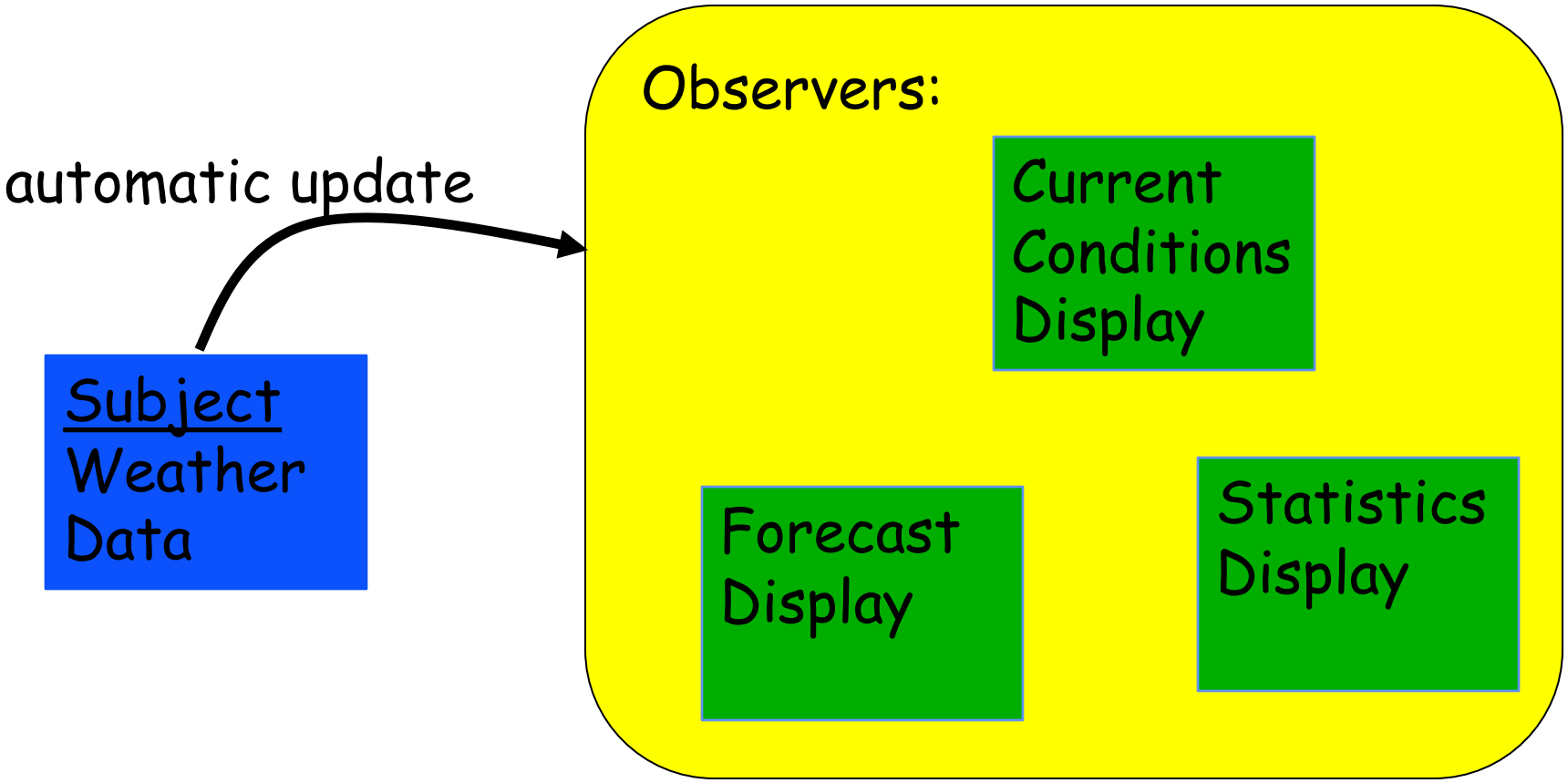
```
class WEATHER_DATA_IMP inherit
  WEATHER_DATA

feature

  current_conditions: CURRENT_CONDITIONS
  statistics: STATISTICS
  forecast: FORECAST

  ...


  measurements_changed
    do
      current_conditions.update(temp, humidity, pressure)
      weather_stats.update(temp, humidity, pressure)
      forecast.update(temp, humidity, pressure)
    end
end
```

Display One Display Two Display Three

Part that changes

- Is update coded to concrete implementation, or to interfaces?
- Do we need to add code to the class for each new display?
- Can we add/remove new displays from a client at runtime?
- Have we encapsulated that which changes?

Observer: ONE → MANY relationship



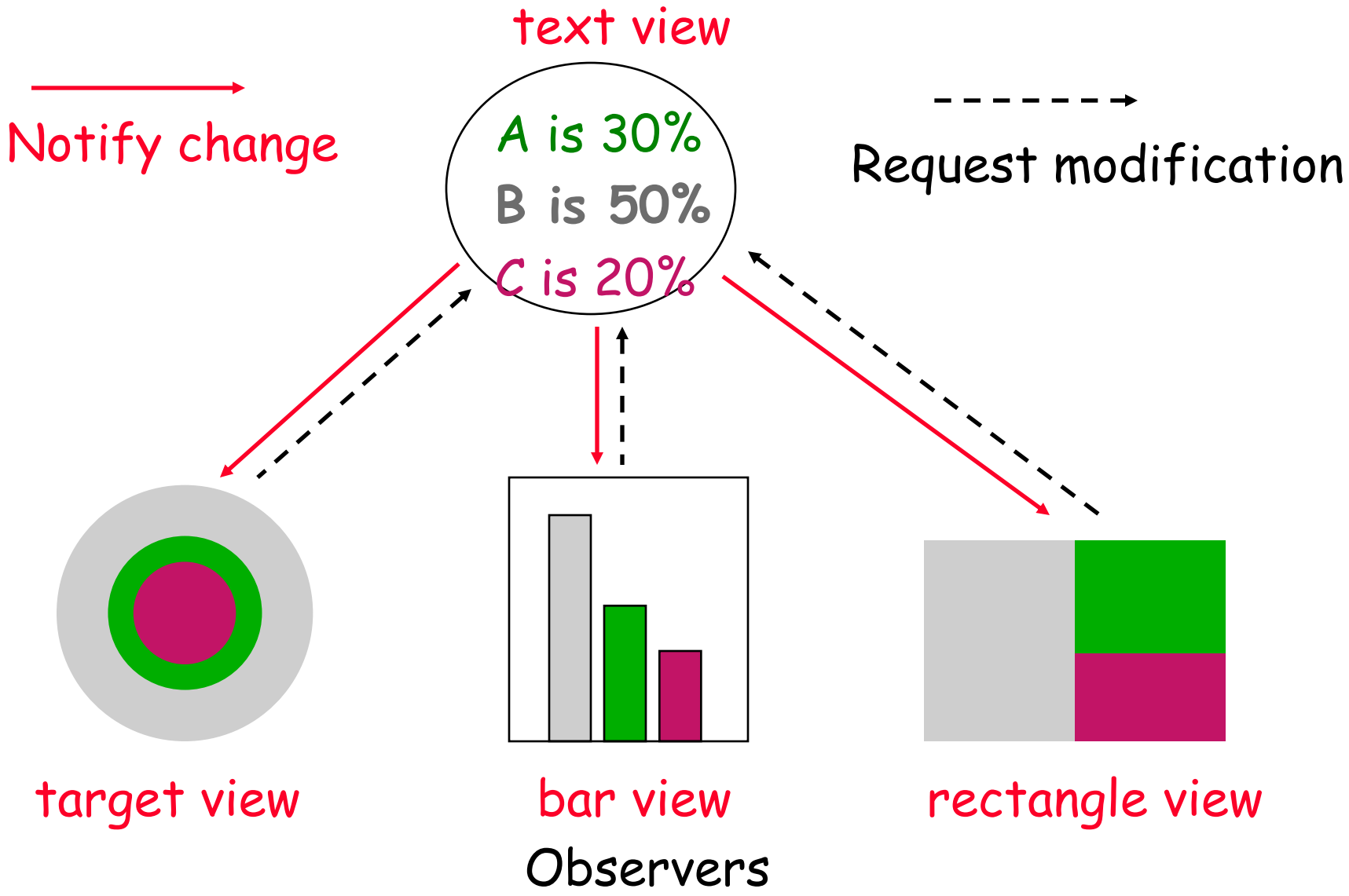
The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.

Observer Pattern

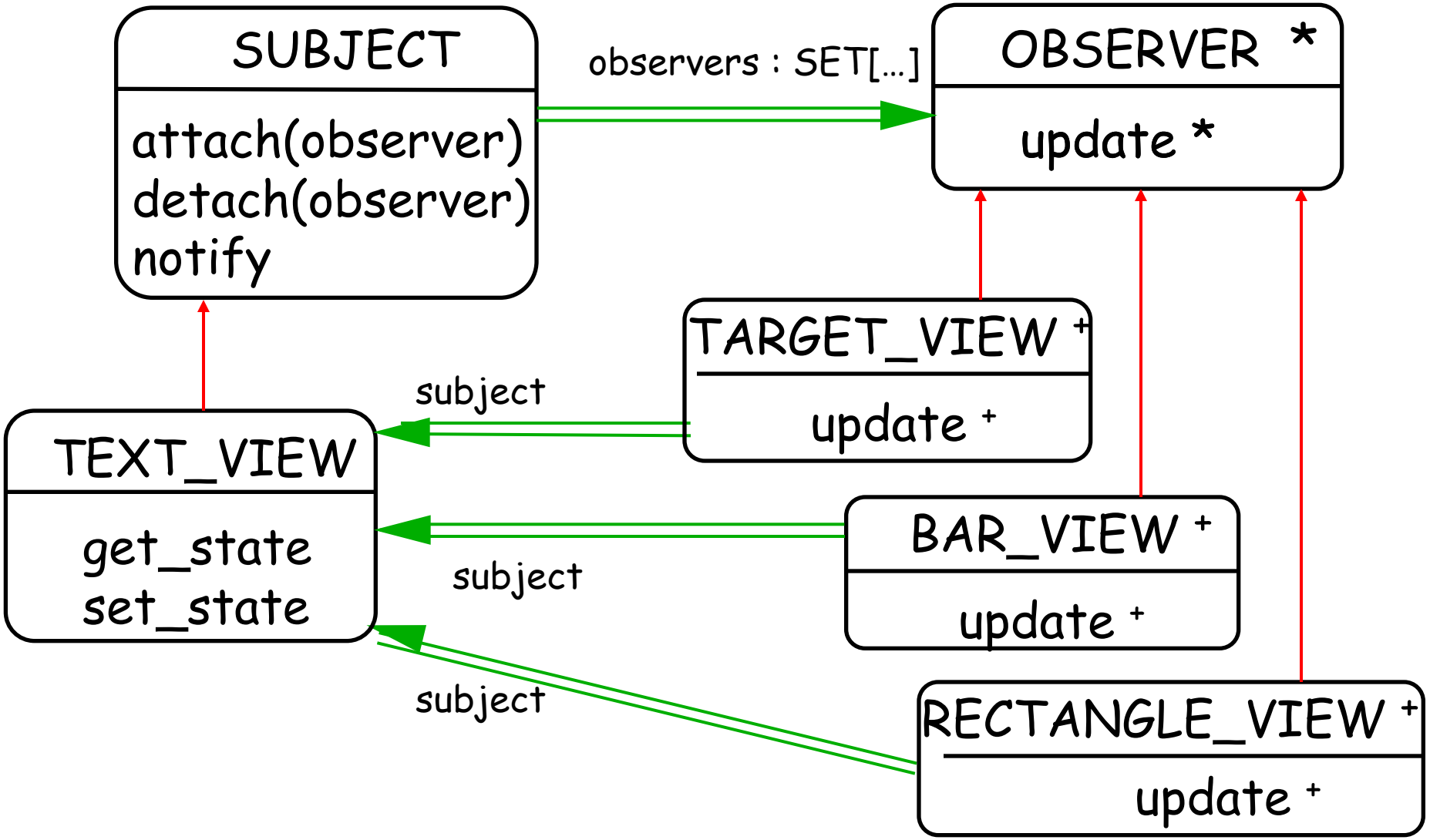
- **Intent**
  - Define one-to-many dependency
    - When one subject changes state, all observers are notified and correspondingly updated

- **Also known as**
  - Publish-Subscribe

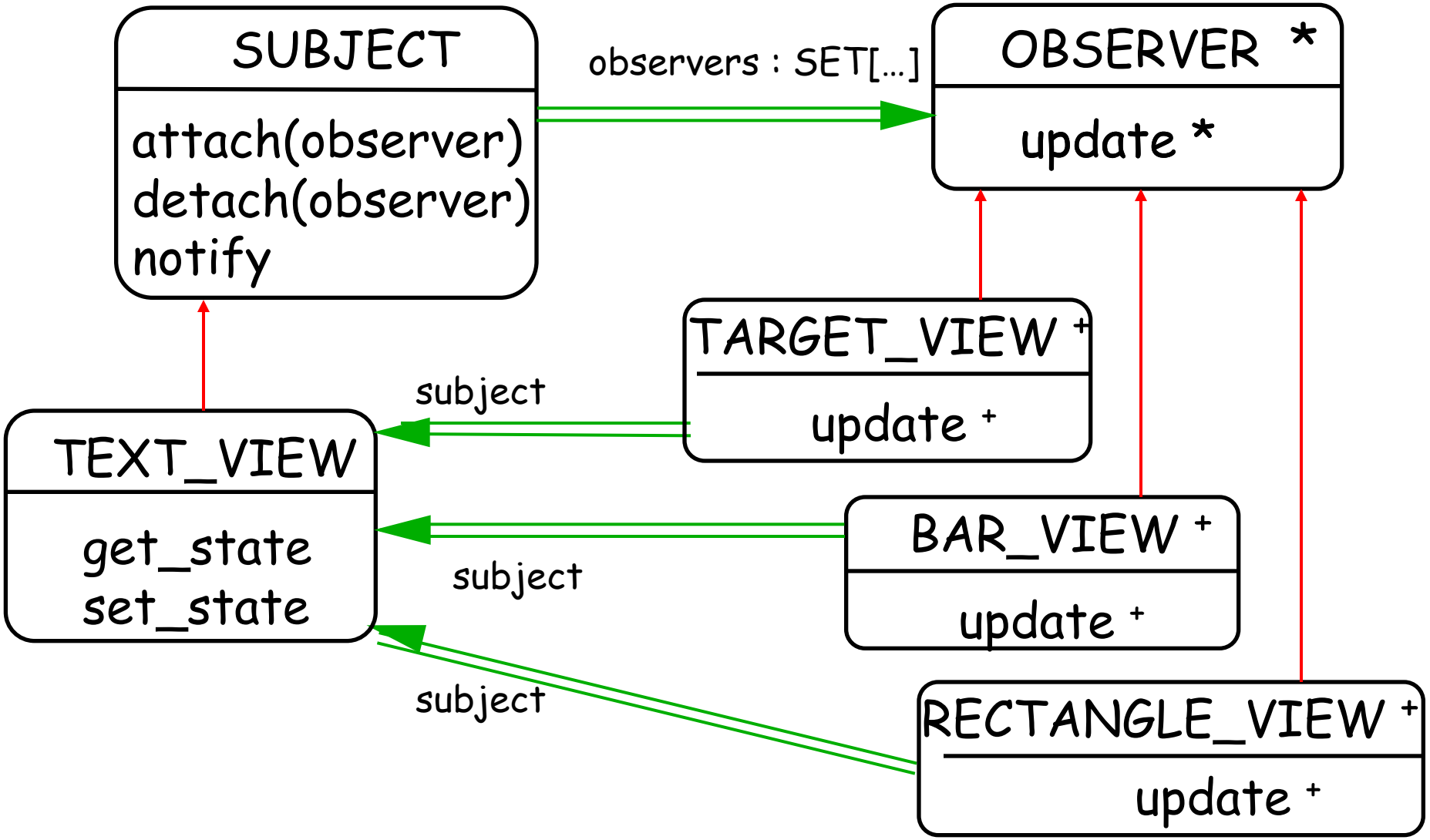
Observer – Motivation



Observer Architecture – Example



Observer Architecture – Example

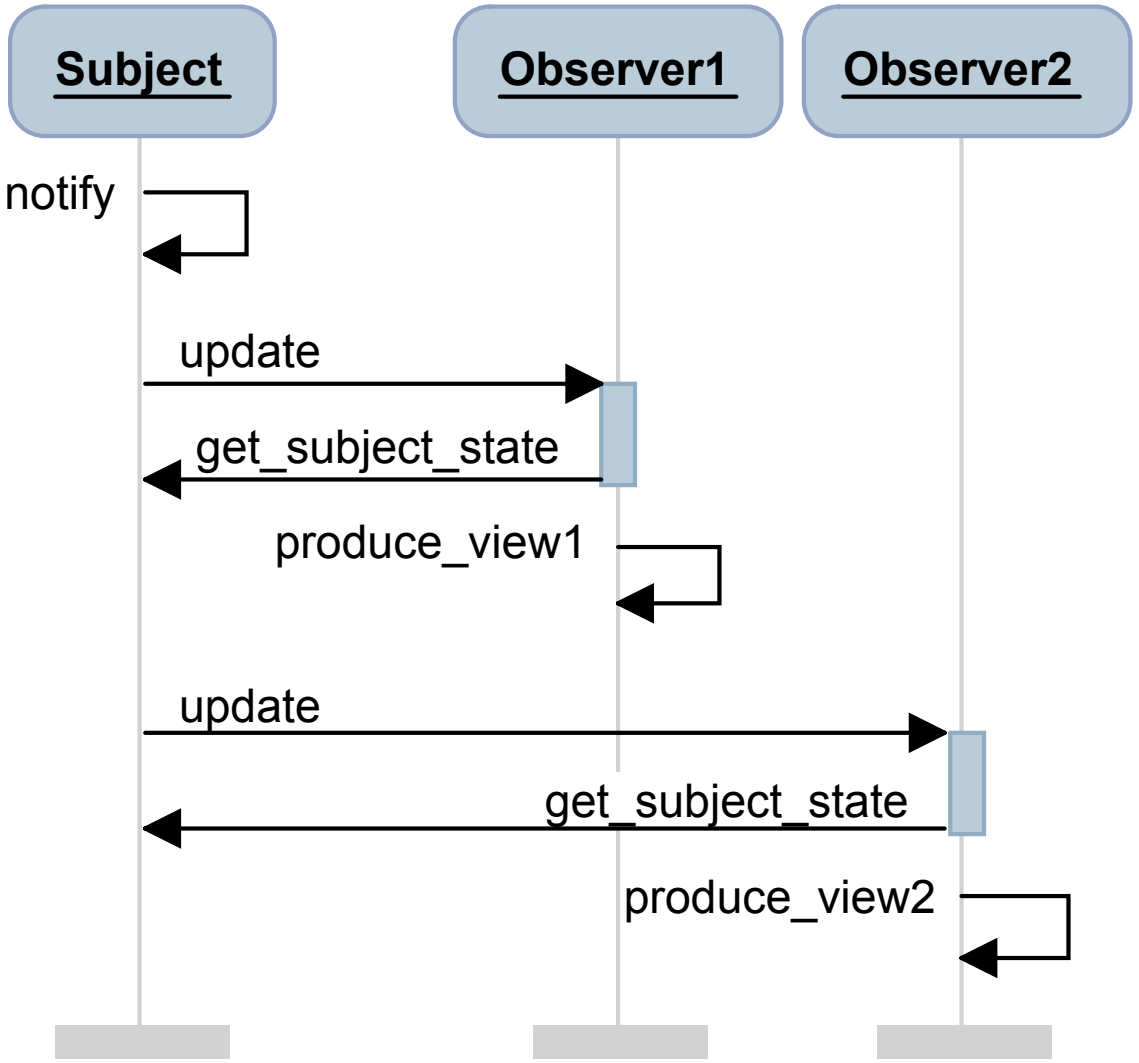


Observer

UML Sequence Diagram
Dynamic (runtime objects, not classes)

- as opposed to

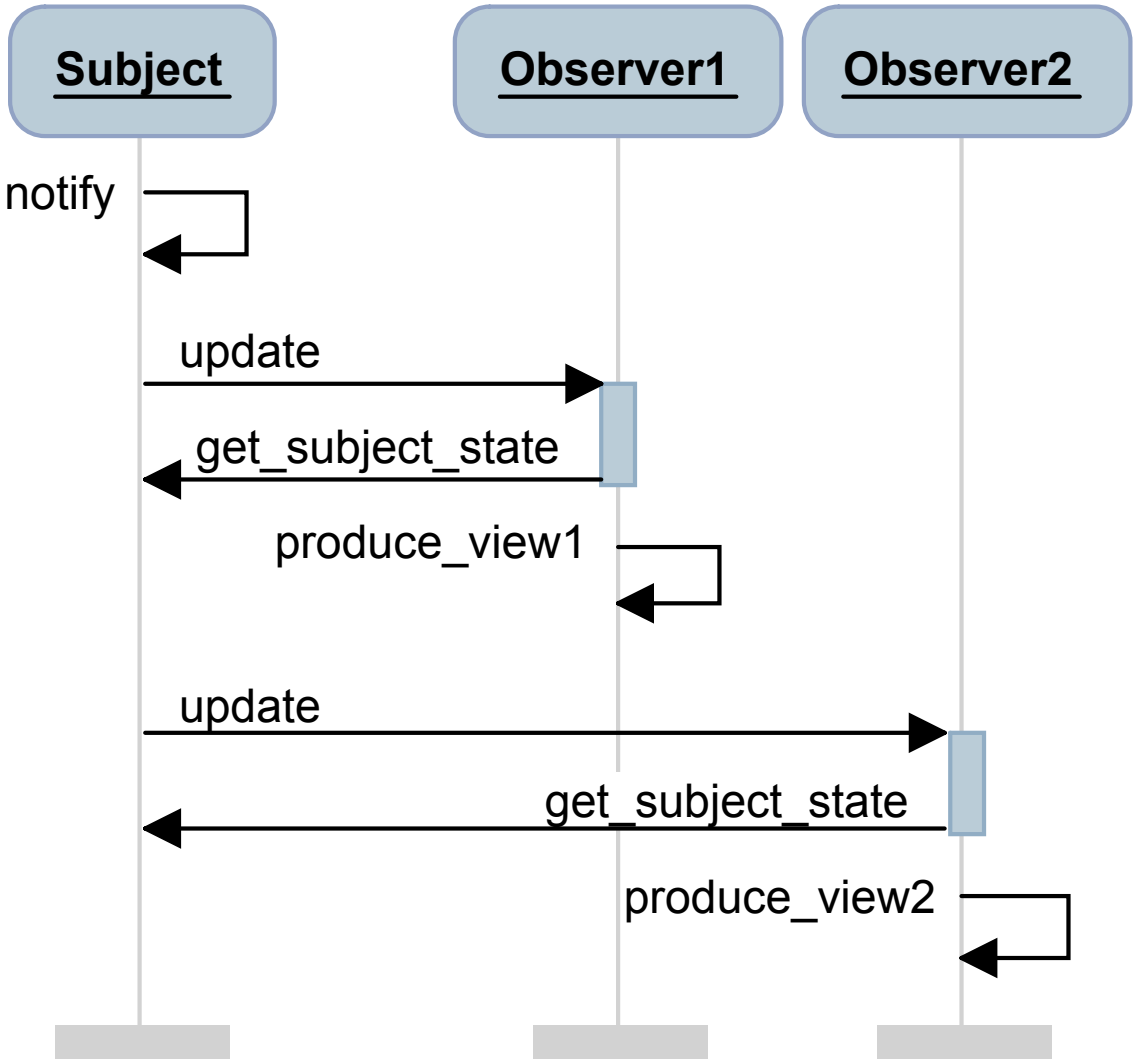
a Class Diagram which is static (compile time)



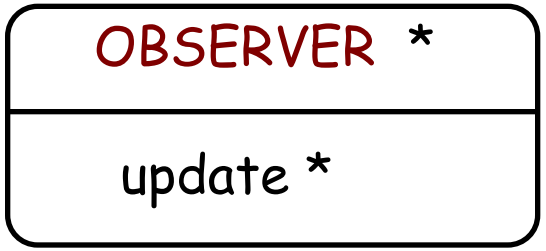
Observer

Sequence Diagram

- Use these dynamic diagrams in your design document



UML: https://www.websequencediagrams.com

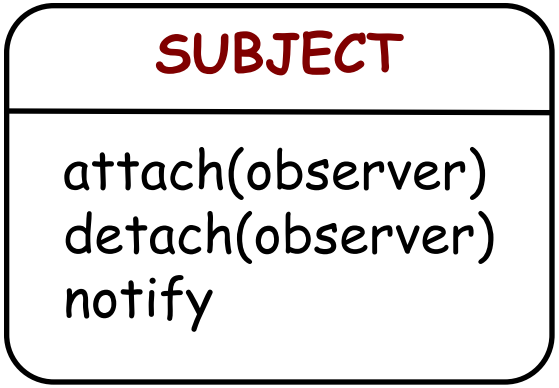



```
deferred class
 OBSERVER
feature -- to be effected by a descendant
   update
      -- update the observer's view of the subject
    deferred
    ensure
    up_to_date_with_subject
    end
```

How can we ensure that the update worked?

```
up_to_date_with_subject: BOOLEAN
     -- is this observer up to date with its subject
  deferred
  end
```

```
end
```




```
class SUBJECT create
  make
feature {OBSERVER}
  attach (o: OBSERVER)
    require
      o /= Void and not observers.has(o)
    ensure
      observers.has(o)

  detach (o: OBSERVER) ...

  notify ...

feature {NONE}
 observers: LIST [OBSERVER]
    -- list of observers attached to this subject
end
```

```
class SUBJECT feature …
  observers: LIST [OBSERVER] …
  attach (o: OBSERVER) …
  detach (o: OBSERVER) …

  notify
      -- Send an `update' message to each observer o
    do
      from observers.start
      until observers.after
      loop observers.item.update; observers.forth
      end
    ensure
      across observers as it all
            it.item.up_to_date_with_subject
invariant
  observers_not_void: observers /= Void
end
```

SUBJECT
attach(observer) detach(observer) notify

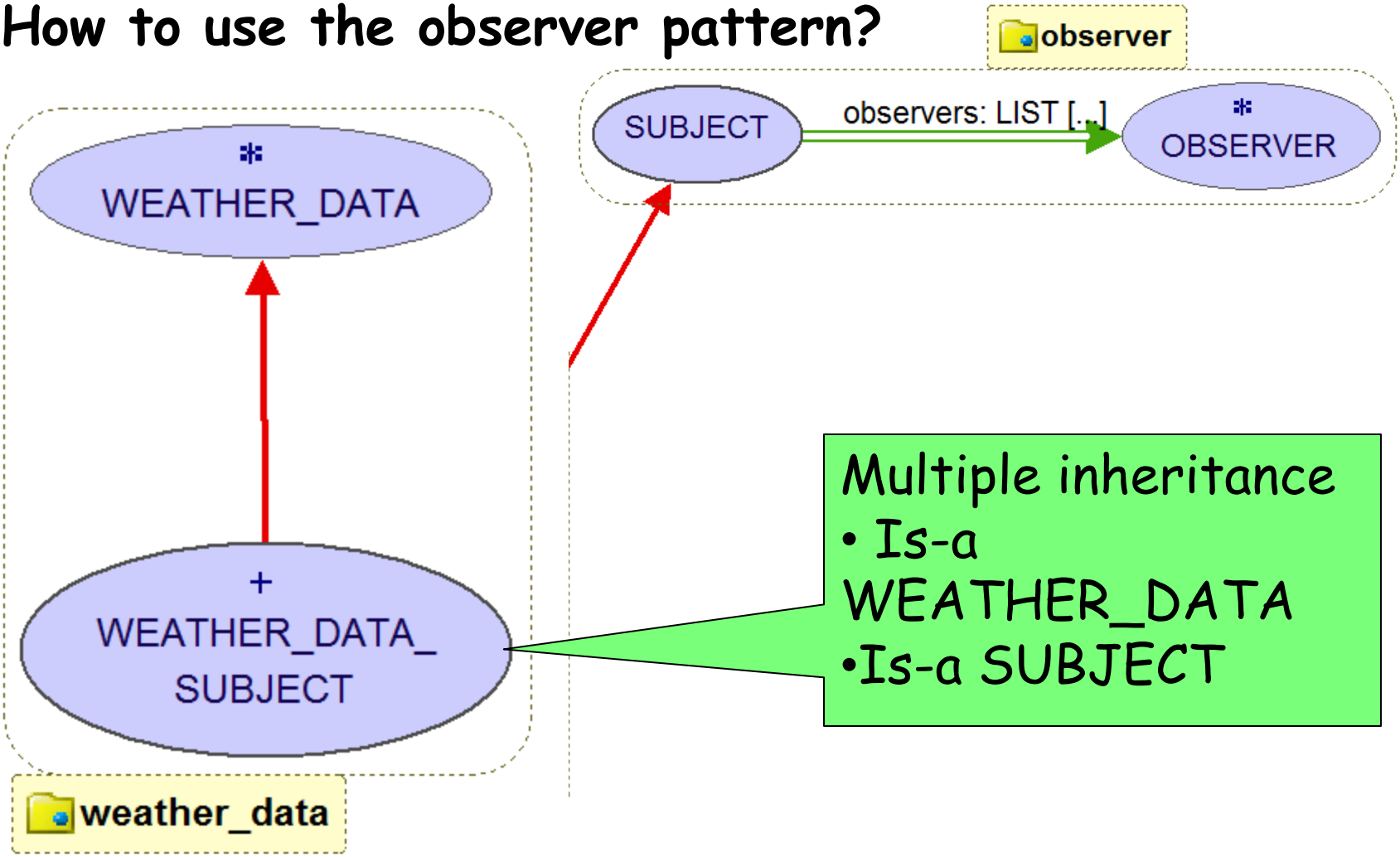
Dynamic binding

back to Weather-O-Rama

- Ok, so what classes will we need?
- Sketch a BON diagram

back to Weather-O-Rama

- WEATHER_DATA is closed; so inherit from it
- How to use the observer pattern?



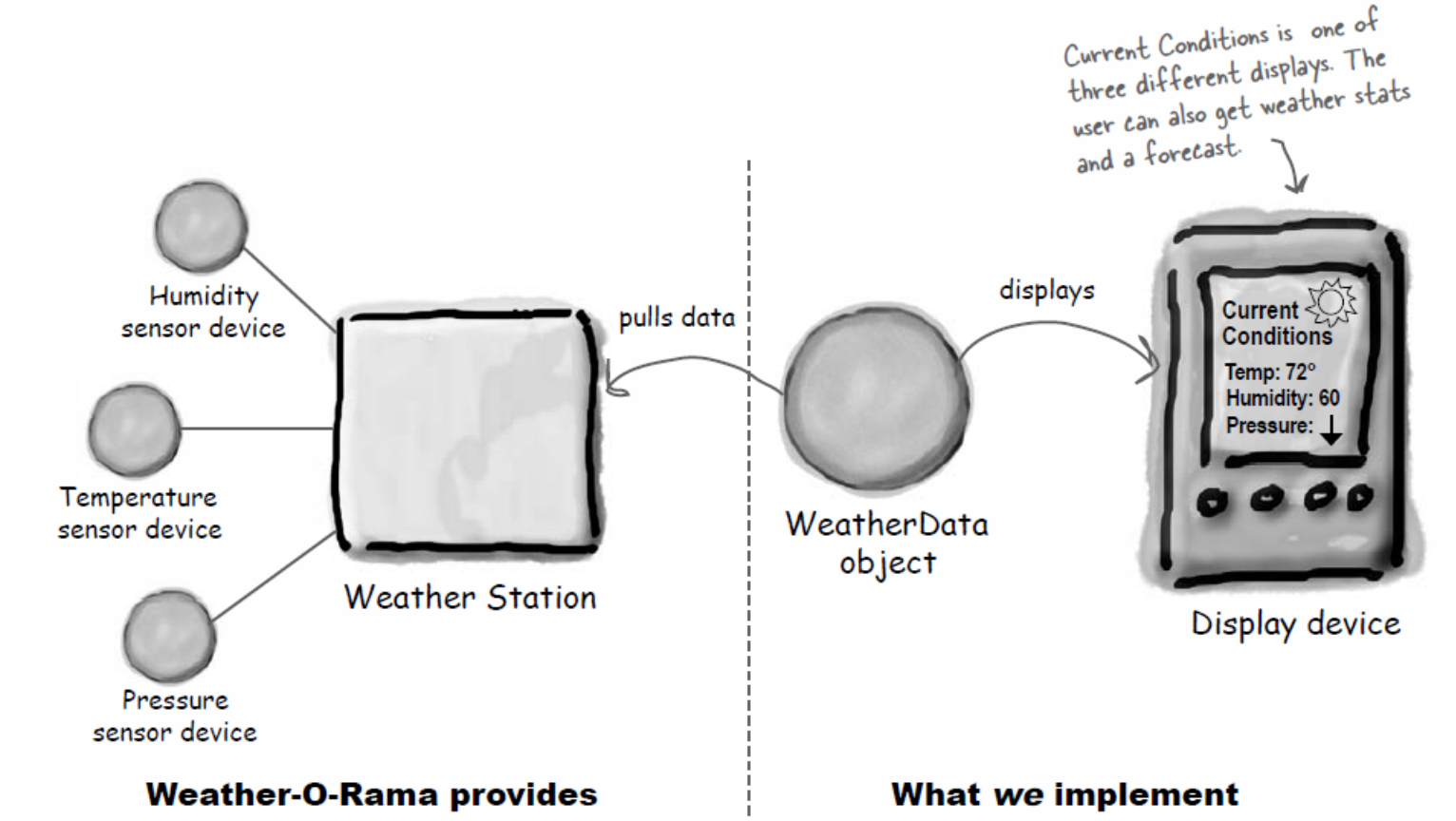
```
class
    WEATHER_DATA_SUBJECT

inherit
    WEATHER_DATA
        redefine
            make
        end

    SUBJECT
        rename
            make as subject_make
        end


create
    make

feature {NONE}

    make
        do
            subject_make
            Precursor
        end

feature

    measurements_changed
            -- called from weather station after set
        do
            notify
        ensure then
            observers_updated: observers.for_all (agent update_action_completed (?))
        end

end -- class WEATHER_DATA_SUBJECT
```

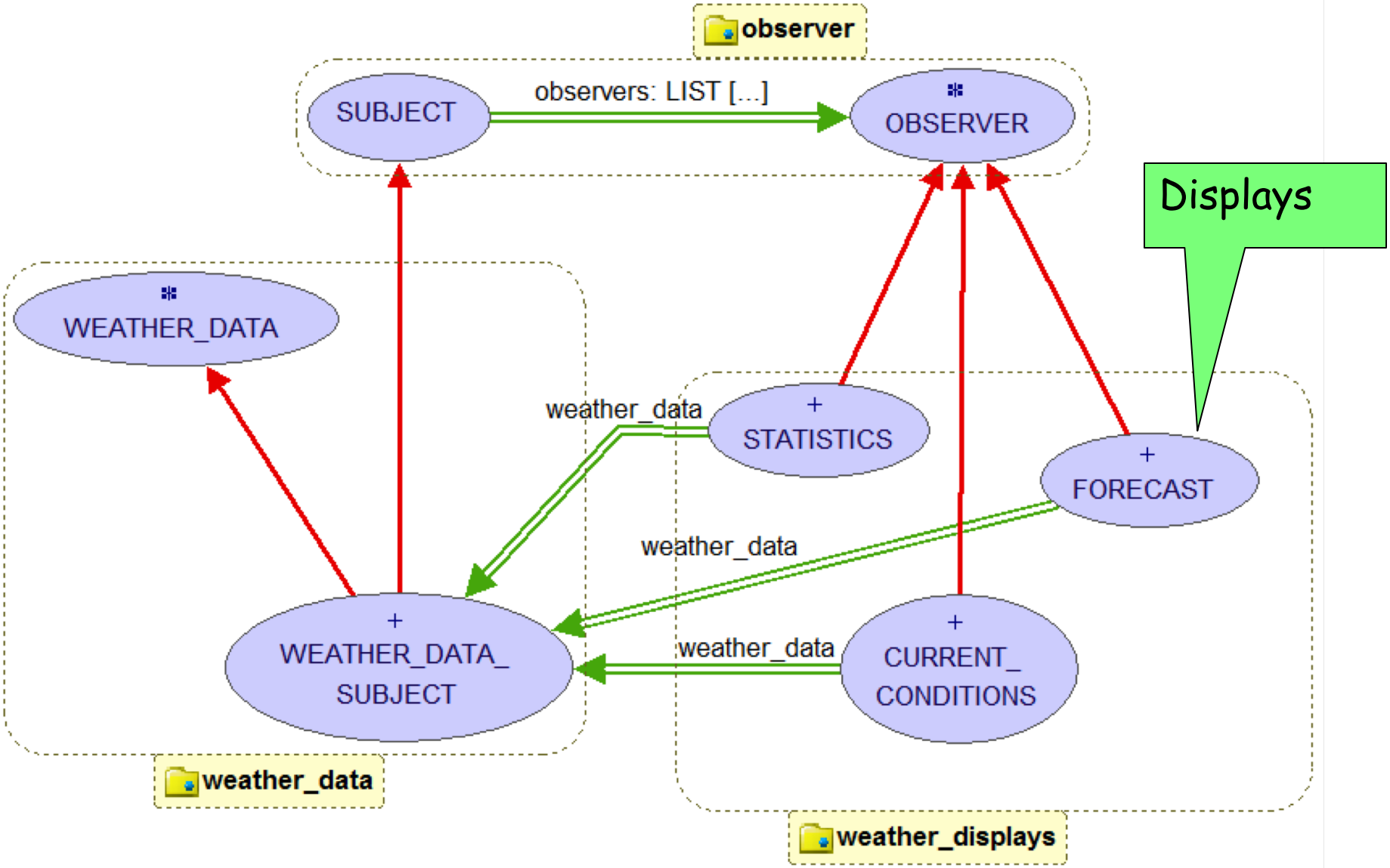
Inherit weather data behaviour such as temp, pressure, humidity

Inherit subject behaviour such as
• add an observer (display)
•notify all observers

Effect measurements changed via notify

What about the displays




observer
SUBJECT
observers: LIST [...]
OBSERVER
Displays
WEATHER_DATA
weather_data
STATISTICS
FORECAST
weather_data
WEATHER_DATA_
SUBJECT
weather_data
CURRENT_
CONDITIONS
weather_data
weather_displays

```
class  CURRENT_CONDITIONS inherit
  OBSERVER
create
  make
feature {NONE} -- initialize
  make (wds: WEATHER_DATA_SUBJECT)
    do
      weather_data := wds
      weather_data.attach (Current)
    end
feature
  weather_data: WEATHER_DATA_SUBJECT -- subject
  temperature: REAL
  humidity: REAL

  update
      -- update the observer's view of subject
    do
      temperature := weather_data.temperature
      humidity := weather_data.humidity
      display
    end  ...
end
```

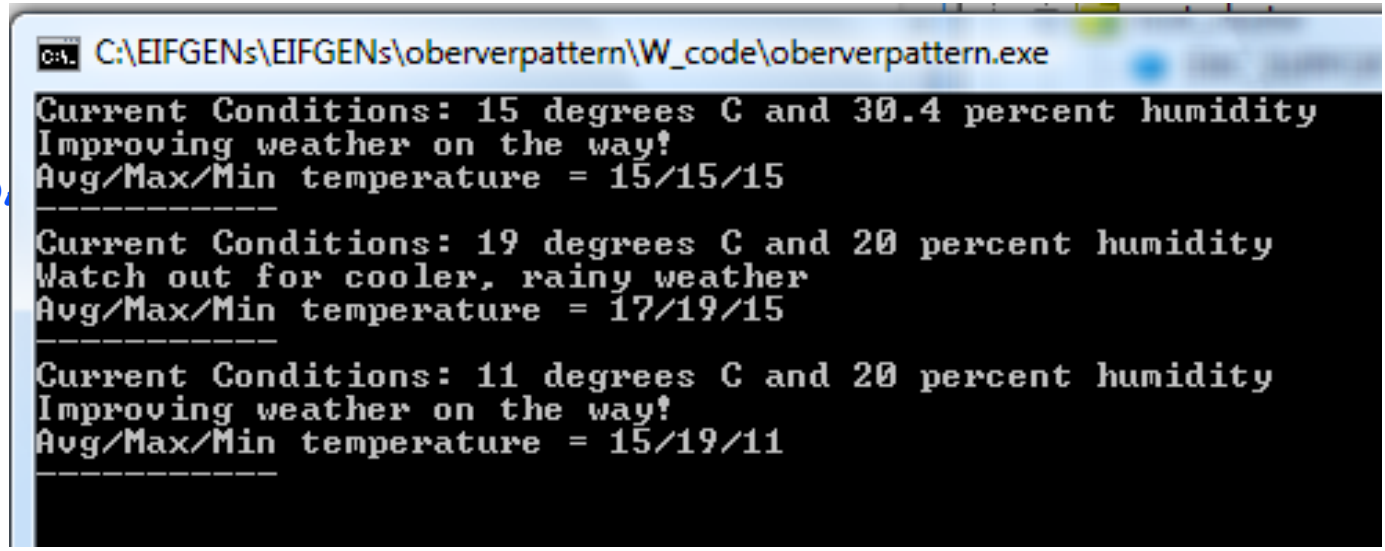
Attach the current display to the weather data subject

```
class  CURRENT_CONDITIONS inherit
  OBSERVER
create
  make
feature {NONE} -- initialize
  make (a_weather_data: WEATHER_DATA_SUBJECT) ...

feature
  weather_data: WEATHER_DATA_SUBJECT --
  temperature: REAL_32
  humidity: REAL_32

  update ...

  up_to_date_with_subject: BOOLEAN
      -- is this observer up to date with its subject
   do
     if temperature = weather_data.temperature
         and humidity = weather_data.humidity then
                 Result := true
   end

   display ...
end
```

- It is easy to forget to synch
- A contract failure will remind you, if you forget to do this

Power up weather station

```
class WEATHER_STATION create
  make
feature
  cc: CURRENT_CONDITIONS  -- displays
  fd: FORECAST
  sd: STATISTICS

  wd: WEATHER_D

make
  do
    create wd.make
    create cc.make

    wd.set_measurements (15, 60, 30.4); wd.measurements_changed

    wd.set_measurements (19, 56, 20); wd.measurements_changed

    wd.set_measurements (11, 90, 20); wd.measurements_changed

  end
end
```



Design Principle

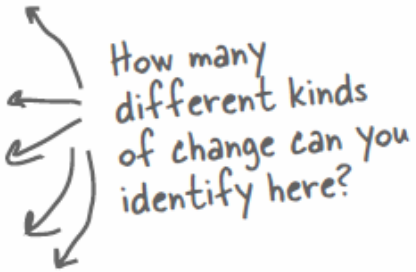
Strive for loosely coupled designs between objects that interact.

Why?

The only thing the subject knows about an observer is that it implements a certain interface (the Observer interface). It doesn't need to know the concrete class of the observer, what it does, or anything else about it.

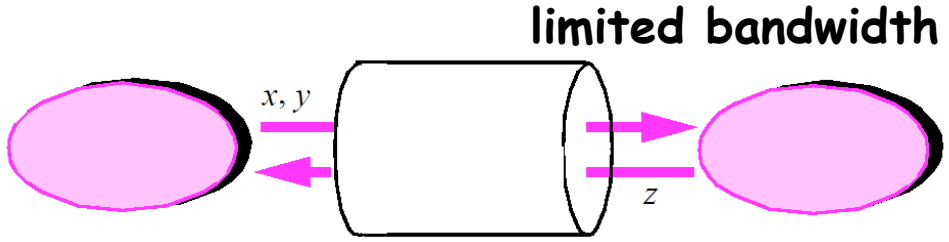
We can add new observers at any time. Because the only thing the subject depends on is a list of objects that implement the Observer interface, we can add new observers whenever we want. In fact, we can replace any observer at runtime with another observer and the subject will keep purring along. Likewise, we can remove observers at any time.

We never need to modify the subject to add new types of observers. Let's say we have a new concrete class come along that needs to be an observer. We don't need to make any changes to the subject to accommodate the new class type, all we have to do is implement the Observer interface in the new class and register as an observer. The subject doesn't care; it will deliver notifications to any object that implements the Observer interface.



Design Principle: Strive for loosely coupled designs between objects that interact.

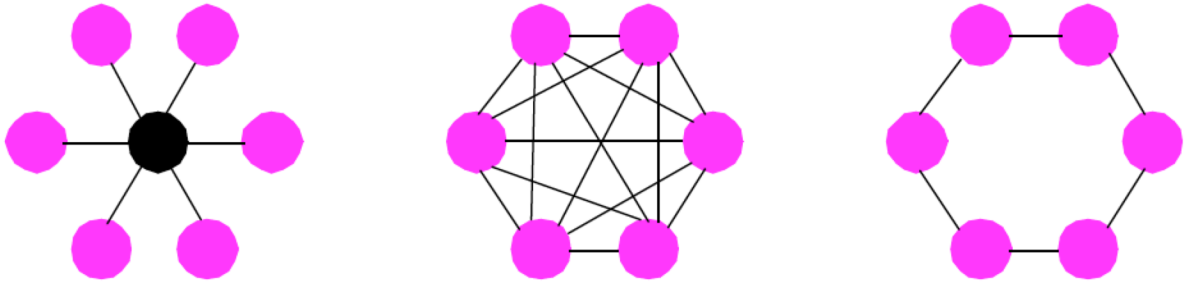
- *OOSC2*: Chapter 3 on modularity
  - Few interfaces principle (a module should communicate with as few others as possible)
  - Small interfaces principle (exchange as little information as possible)
  - Explicit interface principle (interface obvious from the text)
  - Information hiding
  - Self documentation principle



Few interfaces (OOSC2 p47)

Every module should communicate with as few others as possible.

Communication may occur between modules in a variety of ways. Modules may call each other (if they are procedures), share data structures etc. The Few Interfaces rule limits the number of such connections.



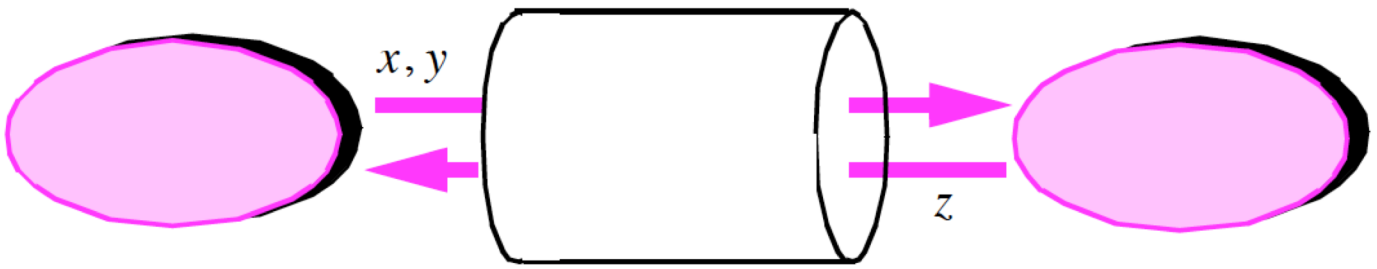
(A) **(B)** **(C)**

More precisely, if a system is composed of n modules, then the number of intermodule connections should remain much closer to the minimum, $n–1$, shown as **(A)** in the figure, than to the maximum, $n\ (n-1)\ /2$, shown as **(B)**.

Few interfaces (OOSC2 p48)

> If two modules communicate, they should exchange as little information as possible

An electrical engineer would say that the channels of communication between modules must be of limited bandwidth:



- The use of external variables in C like languages creates blocks of common data accessible from all modules
- But this also means that all modules can misuse this data and create tightly coupled modules

Design Principle: Strive for loosely coupled designs between objects that interact.

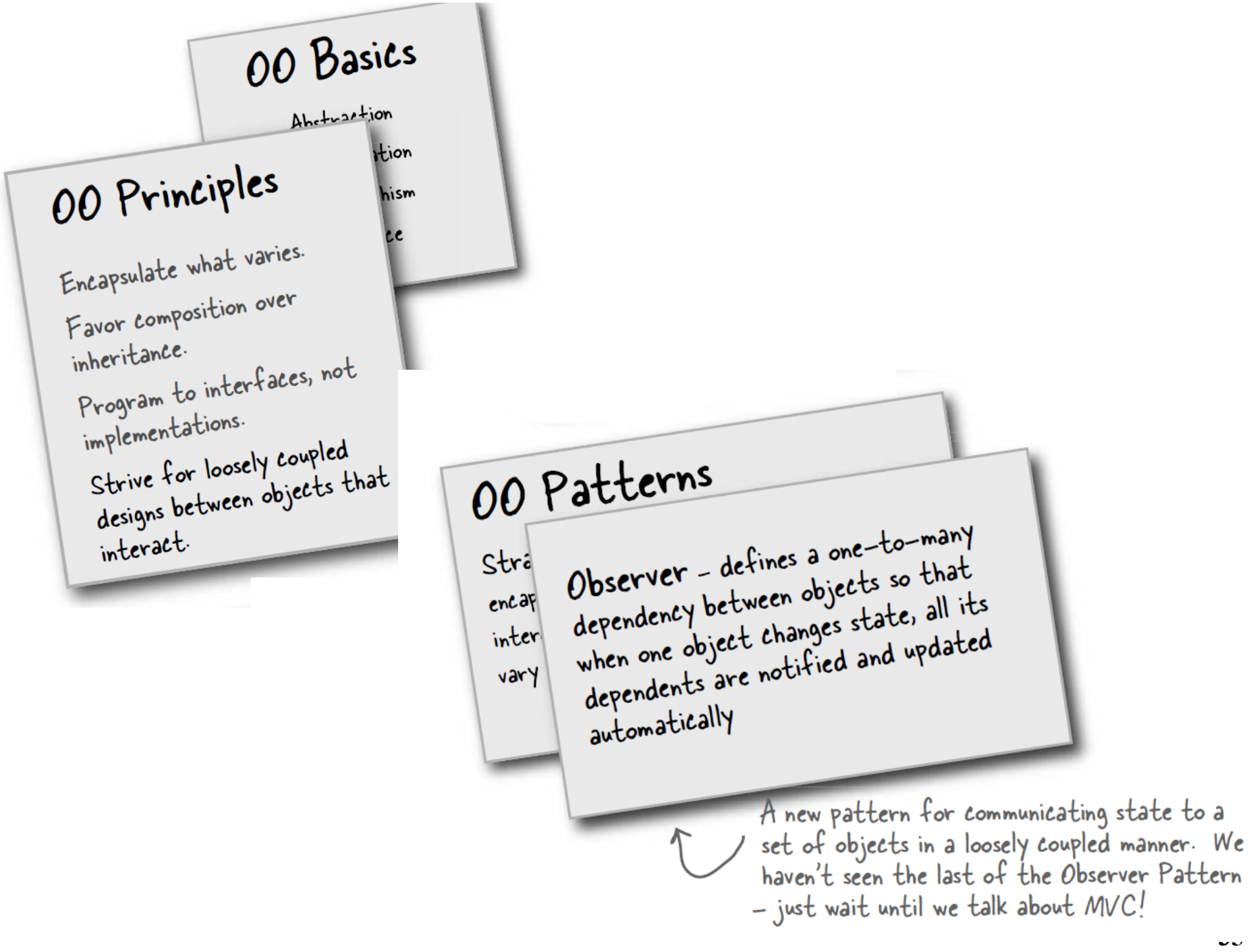
- Loosely coupled designs allow us to build flexible OO systems that can handle change because they minimize the dependency between objects

Add a new display for the heat index?

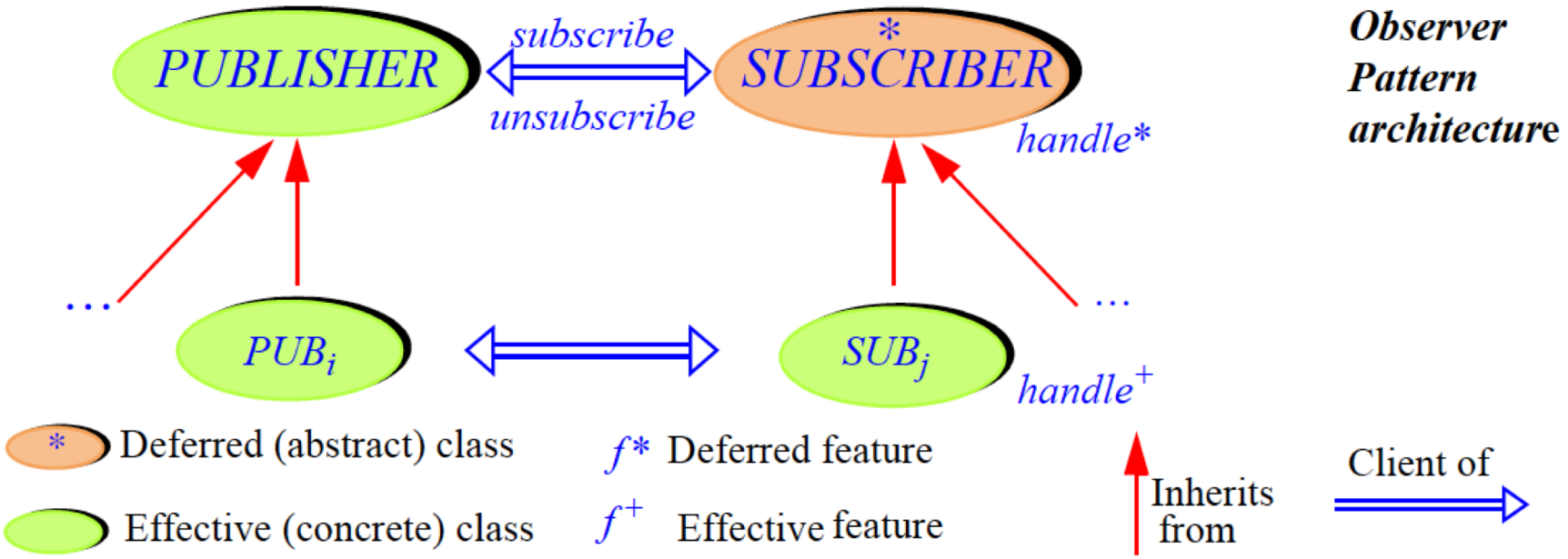
$$heatindex = 16.923 + 1.85212 * 10^{-1} * T + 5.37941 * RH - 1.00254 * 10^{-1} * T * RH + 9.41695 * 10^{-3} * T^2 + 7.28898 * 10^{-3} * RH^2 + 3.45372 * 10^{-4} * T^2 * RH - 8.14971 * 10^{-4} * T * RH^2 + 1.02102 * 10^{-5} * T^2 * RH^2 - 3.8646 * 10^{-5} * T^3 + 2.91583 * 10^{-5} * RH^3 + 1.42721 * 10^{-6} * T^3 * RH + 1.97483 * 10^{-7} * T * RH^3 - 2.18429 * 10^{-8} * T^3 * RH^2 + 8.43296 * 10^{-10} * T^2 * RH^3 - 4.81975 * 10^{-11} * T^3 * RH^3$$

Here's what changed in this output.

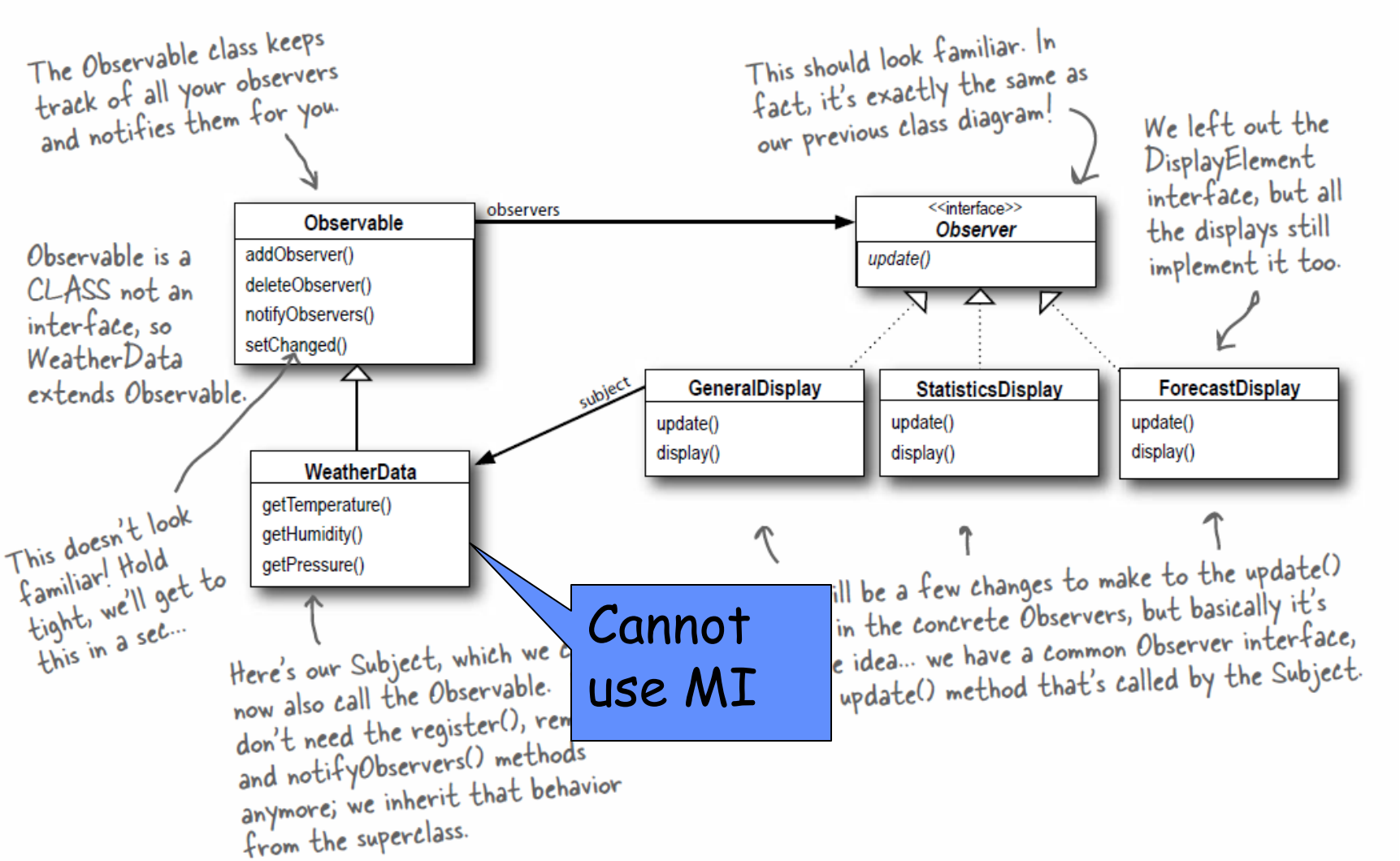
```
Current conditions: 80.0F degrees and 65.0% humidity
Avg/Max/Min temperature = 80.0/80.0/80.0
Forecast: Improving weather on the way!
Heat index is 82.95535
Current conditions: 82.0F degrees and 70.0% humidity
Avg/Max/Min temperature = 81.0/82.0/80.0
Forecast: Watch out for cooler, rainy weather
Heat index is 86.90124
Current conditions: 78.0F degrees and 90.0% humidity
Avg/Max/Min temperature = 80.0/82.0/78.0
Forecast: More of the same
Heat index is 83.64967
%
```


OO Basics
OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
OO Patterns
Observer – defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
A new pattern for communicating state to a set of objects in a loosely coupled manner. We haven't seen the last of the Observer Pattern – just wait until we talk about MVC!

Also called publisher-subscriber



Java's built-in pattern (e.g. in Swing)



Observer in Java

This display implements Observer so it can get changes from the WeatherData object.

It also implements DisplayElement, because our API is going to require all display elements to implement this interface.

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

    public CurrentConditionsDisplay(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }

    public void update(float temperature, float humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }

    public void display() {
        System.out.println("Current conditions: " + temperature
            + "F degrees and " + humidity + "% humidity");
    }
}
```

The constructor is passed the weatherData object (the Subject) and we use it to register the display as an observer.

When update() is called, we save the temp and humidity and call display().

The display() method just prints out the most recent temp and humidity.

Java Versionof Subject (restricted to interfaces)

```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
```

Both of these methods take an Observer as an argument; that is, the Observer to be registered or removed.

This method is called to notify all observers when the Subject's state has changed.

specific to weather station

```
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
```

These are the state values the Observers get from the Subject when a weather measurement changes

Critique of Observer pattern

- The issue of how to deal with update arguments in the observer is unpleasant. In the Java example, there would be many quasi-identical observers
- The subject is decoupled from the observers. But the observers are coupled to the the subject. For example, CURRENT_CONDITIONS has to subscribe to WEATHER_DATA.
- With a single general-purpose SUBJECT class, an observer can only subscribe to that subject. But observers/listeners/subscribers may want to to register with multiple events (press of a button, right-click etc).

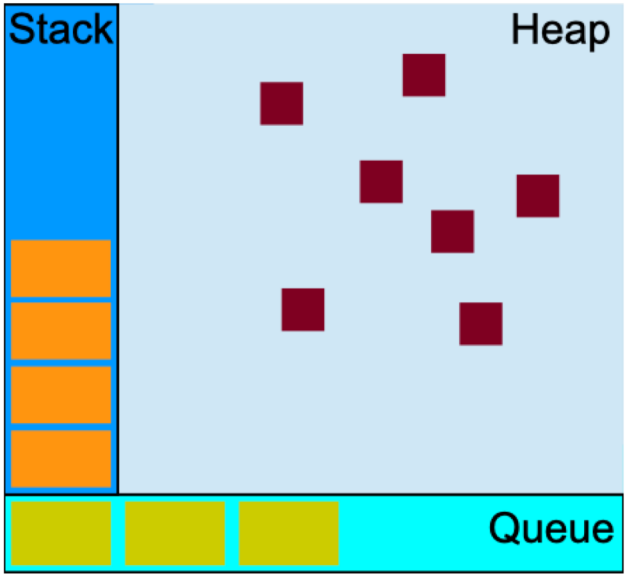
Design Decisions

- The preceding assessment of "Observer" is an example of how one may analyze a proposed software architecture.
- When you are presented with possible design alternatives, the evaluation criteria are always the same such as:
  - reliability (decreasing the likelihood of ERRORS),
  - reusability (minimizing the amount of work to integrate the solution into a new program),
  - extendibility (minimizing adaptation effort when the problem varies),
  - and simplicity

Event-Driven Programming

- Event-driven programming is a programming paradigm in which the flow of the program is determined by events such as user actions (mouse clicks, key presses), sensor outputs, or messages from other programs/threads.
- Event-driven programming is the dominant paradigm used in graphical user interfaces and other applications (e.g. JavaScript web applications) that are centered on performing certain actions in response to user input.
- In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function (lambda function) when one of those events is detected. In embedded systems the same may be achieved using hardware interrupts instead of a constantly running main loop. Event-driven programs can be written in any programming language, although the task is easier in languages that provide high-level abstractions, such as closures (lambda functions).

Javascript event driven loop



Memory Model

- **Stack** for function calls
- **Heap** for object allocation
- **Queue** for storing messages such as mouse clicks etc. (each message has an associated function to handle it)

```
1 while (queue.waitForMessage()) {
2   queue.processNextMessage();
3 }
```

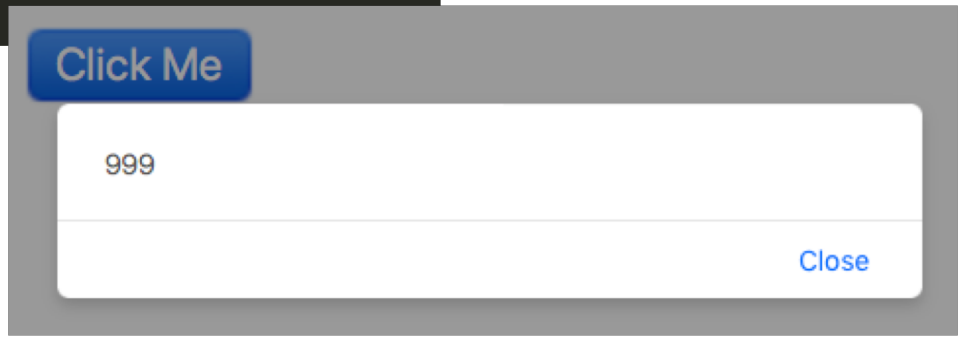
Event Driven Programming

```
<html>
<head>
<script type="text/javascript" src="https://ajax.microsoft.com/ajax/
jQuery/jquery-1.4.2.min.js"></script>

<script type="text/javascript">
    a = 998;
    function handleClick(){
        alert(a);
    }
</script>

<script type="text/javascript">
    a = a + 1;

    $(document).ready(
        function(){
                $("#ClickMe").click(handleClick);
                }
                    );
</script>

</head>
<body>
<p><input type="button" valu      ck Me" id="ClickMe" /></p>
</body>
</html>
```

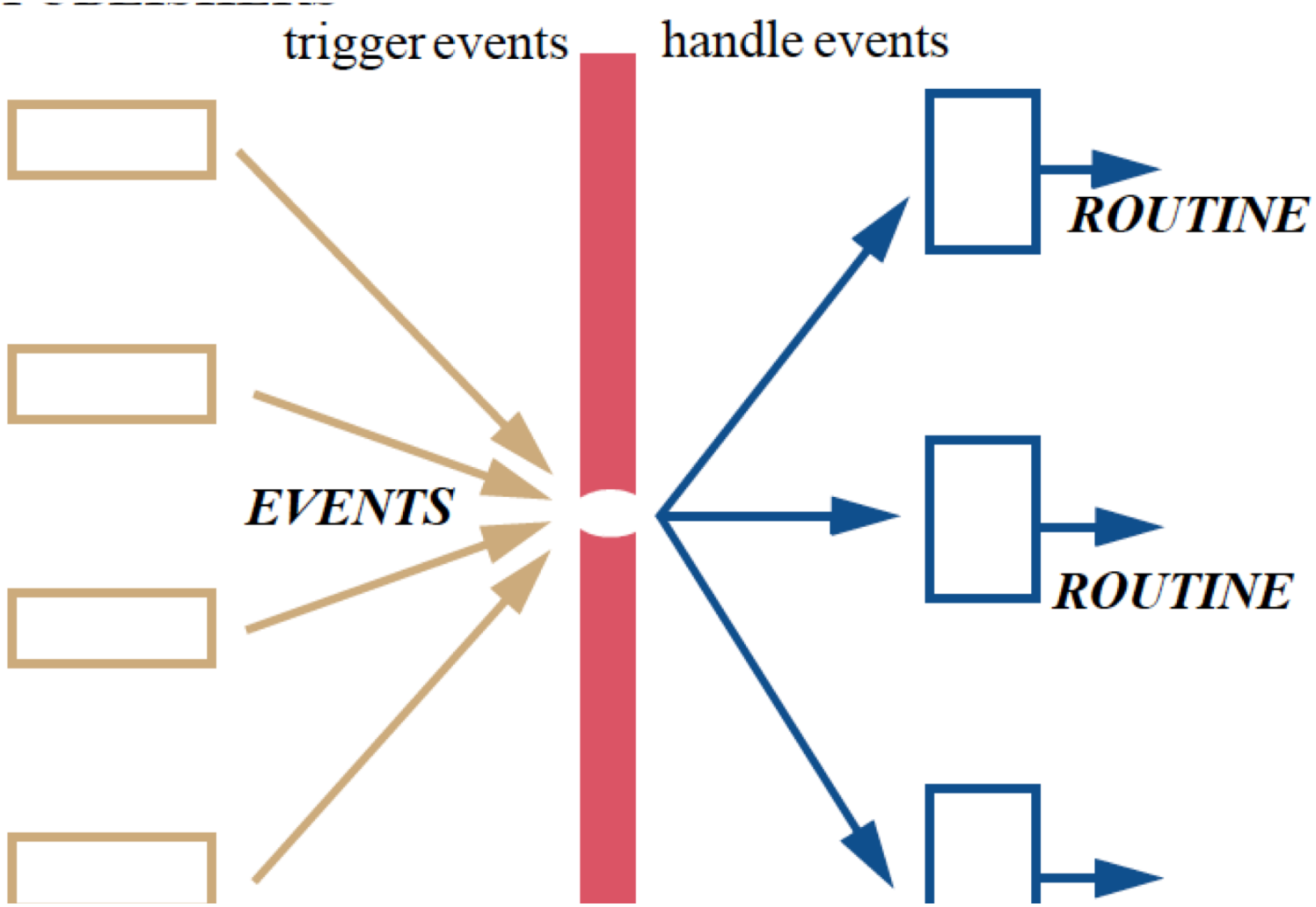
This code runs once and sits in global memory. An event driven interpreter in the browser is initiated.

Define/store a routine (an anonymous lambda function) to respond to a button click. A listener (observer) for button clicks is initiated.

When the event loop detects a click, function handleClick is executed. (repeatedly)



Event Based Programming (agents)



Event-Driven Programming

The event-driven scheme

E1 Some elements, *publishers*, make known to the rest of the system what *event types* they may trigger.

E2 Some elements, *subscribers*, are interested in *handling* events of certain event types. They *register* the corresponding actions.

E3 At any time, a publisher can *trigger* an event. This will cause execution of actions registered by subscribers for the event's type. These actions can use the event's arguments.

EVENT_TYPE API (agents)

```
note
    description: "Event Driv

class interface
    EVENT_TYPE [G -> TUPLE]

create
    default_create
```

```
feature

    actions: LIST [PROCEDURE [G]]

    count: INTEGER_32
        ensure
            Result = actions.count

    execute (p: PROCEDURE [G]; args: G)

    has (an_action: PROCEDURE [G]): BOOLEAN
        ensure
            Result = actions.has (an_action)
```

```
publish (args: G)
        -- update all subscribed listeners

subscribe (an_action: PROCEDURE [G])
        -- Register an action of this type
    require
        an_action_not_void: an_action /= Void
        an_action_not_already_attached: not has
    ensure
        added: count = old count + 1
        subscribed: actions.has (an_action)

unsubscribe (an_action: PROCEDURE [G])
        -- deregister and action of this type
    require
            has (an_action)
    ensure
        unsubsribed: not has (an_action)
```

Iterate over *actions* and execute each action procedure with argument *args*

Subscribe/Unsubscribe

```
actions: LIST [PROCEDURE [G]]

subscribe (an_action: PROCEDURE [G])
        -- Register an action of this type
    require
        an_action_not_void: an_action /= void
        an_action_not_already_attached: not has (an_action)
    do
        actions.extend (an_action)
    ensure
        added: count = old count + 1
        subscribed: actions.has (an_action)
    end

unsubscribe (an_action: PROCEDURE [G])
        -- deregister and action of this type
    require
        has (an_action)
    do
        actions.search (an_action)
        actions.remove
    ensure
        unsubsribed: not has (an_action)
    end
```

Publish

```
EVENT_TYPE [G -> TUPLE]
```

```
    actions: LIST [PROCEDURE [G]]

    publish (args: G)
            -- update all subscribed listeners
        do
--          actions.do all (agent execute (?, args))
            across actions as l_action loop
                execute(l_action.item, args)
            end
        end

    execute (p: PROCEDURE [G]; args: G)
        do
            p(args)
        end
```

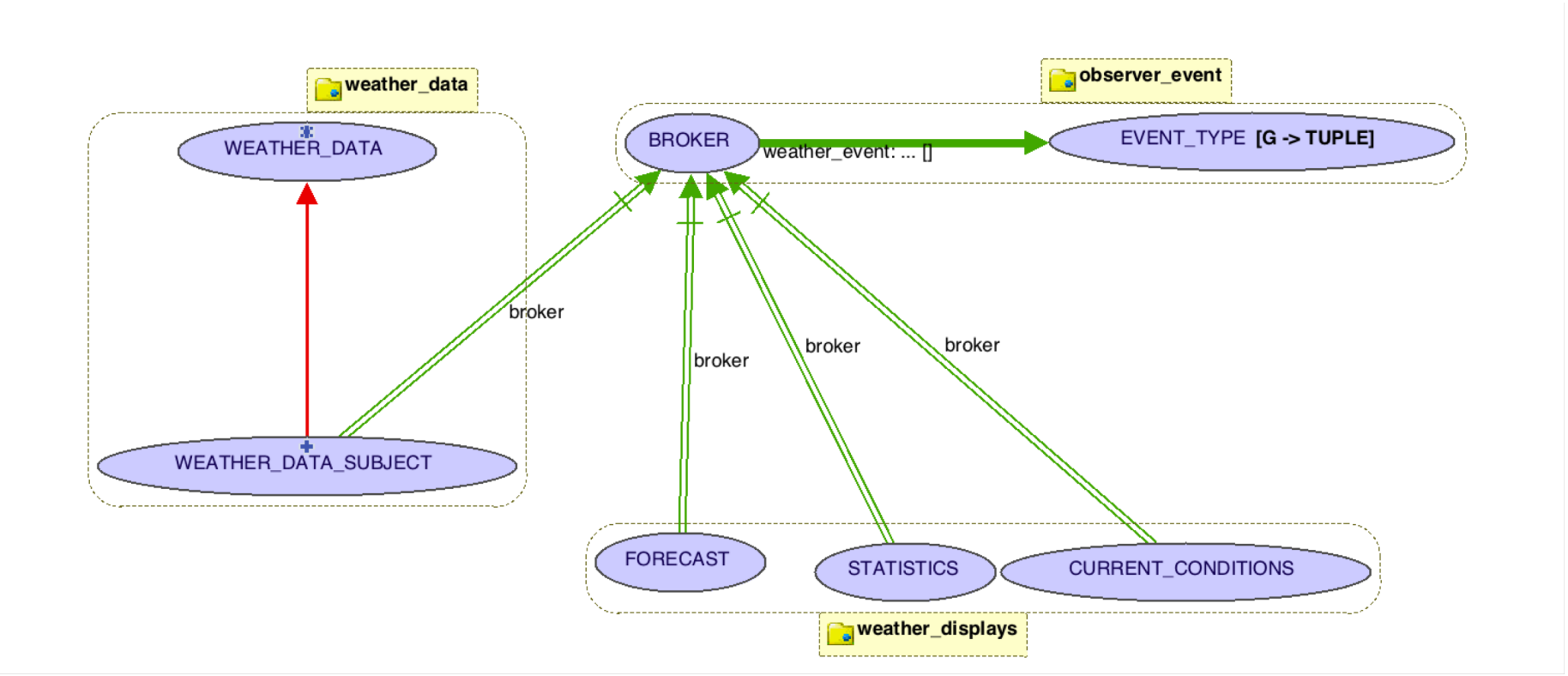
This is also ok

Execute procedure item with argument tuple ***args***

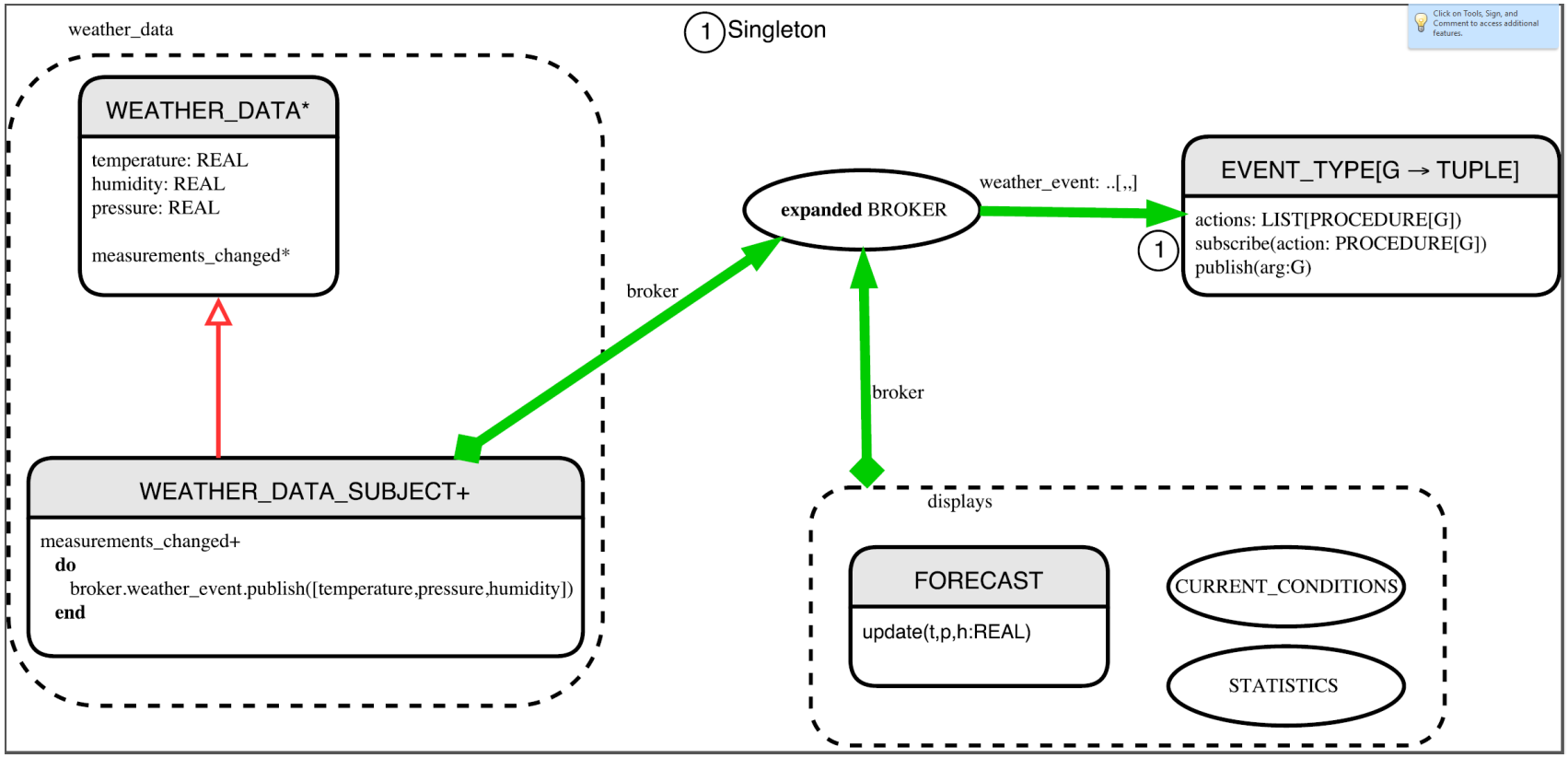
Broker

```
note
    description: "[
        Broker provides a Singleton Event Type
        for weather station updates.
        Other event types may be added as neecssary.
        ]"

expanded class
    BROKER

feature
    weather_event: EVENT_TYPE[
            TUPLE[temperature:REAL; pressure:REAL; humidity:REAL]
        ]
            -- event type for weather station
        once
            create Result
        end

end
```

IDE BON Diagram



For Design Documents use draw.ioTemplate



Simulator

```
class
    WEATHER_STATION_SIMULATOR
create
    make
feature
    cc: CURRENT_CONDITIONS
    fd: FORECAST
    sd: STATISTICS
    wd: WEATHER_DATA_SUBJECT

    make
        do
            create wd.make
            create cc.make
            create fd.make
            create sd.make

            io.new_line
            wd.set_measurements (15, 60, 30.4)
            wd.measurements_changed
            print("-----------%N")

            wd.set_measurements (19, 56, 20)
            wd.measurements_changed
            print("-----------%N")

            wd.set_measurements (11, 90, 20)
            wd.measurements_changed
            print("-----------%N")

--          io.read_character
        end

end
```

```
class
    WEATHER_DATA_SUBJECT
inherit
    WEATHER_DATA
create
    make

feature{NONE}
    broker: BROKER -- expanded singleton

feature --effect measurements_changed

    measurements_changed
        do
            broker.weather_event.publish (
                    [temperature,pressure,humidity]
                )
        end
end
```

```
measurements_changed
    do
        -- [temperature,pressure,humidity]
        broker.weather_event.publish (temperature,pressure,humidity)
    end
```

```
class CURRENT_CONDITIONS create
    make
feature{NONE} -- constructor

    broker: BROKER -- expanded singleton

    make
            -- subscribe to weather event type in broker
        do
            broker.weather_event.subscribe (agent update)
        end
feature
    temperature: REAL
    humidity: REAL


    update(t,p,h:REAL)
            -- update the observer's view of `s'
        do
            temperature := t
            humidity := h
            display
        end

    display
            -- display current conditions
```

```
class
    WEATHER_STATION_SIMULATOR
create
    make
feature
    cc: CURRENT_CONDITIONS
    fd: FORECAST
    sd: STATISTICS
    wd: WEATHER_DATA_SUBJECT

    make
        do
            create wd.make
            create cc.make
            create fd.make
            create sd.make

            io.new_line
            wd.set_measurements (15, 60, 30.4)
            wd.measurements_changed
            print("-----------%N")

            wd.set_measurements (19, 56, 20)
            wd.measurements_changed
            print("-----------%N")

            wd.set_measurements (11, 90, 20)
            wd.measurements_changed
            print("-----------%N")
```

```
Current Conditions: 15 degrees C and 30.4 percent humidity
Improving weather on the way!
Avg/Max/Min temperature = 15/15/15
-----------
Current Conditions: 19 degrees C and 20 percent humidity
Watch out for cooler, rainy weather
Avg/Max/Min temperature = 17/19/15
-----------
Current Conditions: 11 degrees C and 20 percent humidity
Improving weather on the way!
Avg/Max/Min temperature = 15/19/11
-----------
```

Adding cosmic ray detectors to weather station

- Each weather station will be equipped with a cosmic ray (large energetic particles) detector
  - basically a camera
- Detect change in temperature events as well as detect cosmic rays
- How would we do that?

```
class BROKER feature
    weather_event: EVENT_TYPE[TUPLE[ℝ, ℝ, ℝ]]
        once create Result end

    cosmic_ray: EVENT_TYPE[TUPLE[BOOLEAN, ℝ]]
            -- sensor ok and ray intensity
        once create Result end
end
```

Choose the right abstractions

- In the Observer pattern, the abstractions were SUBJECT and OBSERVER.
- Useful, but not good enough for a multi-subject publish-subscribe architecture
- In the weather system, what we needed was a BROKER. The only significant feature of a publisher is that it publishes events from a given event type in the broker, and the only significant feature of a subscriber is that it can subscribe to events from a given event type.
- The key abstraction is EVENT_TYPE

Information Hiding

- Each module (i.e. class) should have a 'secret', some operation that it hides from the other modules so that the other modules need not know the details
  - The secret of class WEATHER_DATA is its interaction and handshaking with the weather station to obtain the temperature, pressure etc. It is a hardware-hiding module.
  - The secret of class CURRENT_CONDITIONS is how it displays the temperature, humidity etc.
  - The secret of class BROKER is that (via EVENT_TYPE) it hides the publisher from the subscribers (views) and the subscribers from the publisher.

Assessing software architectures

The key to the quality of a software system is in its architecture, which covers such aspects as:

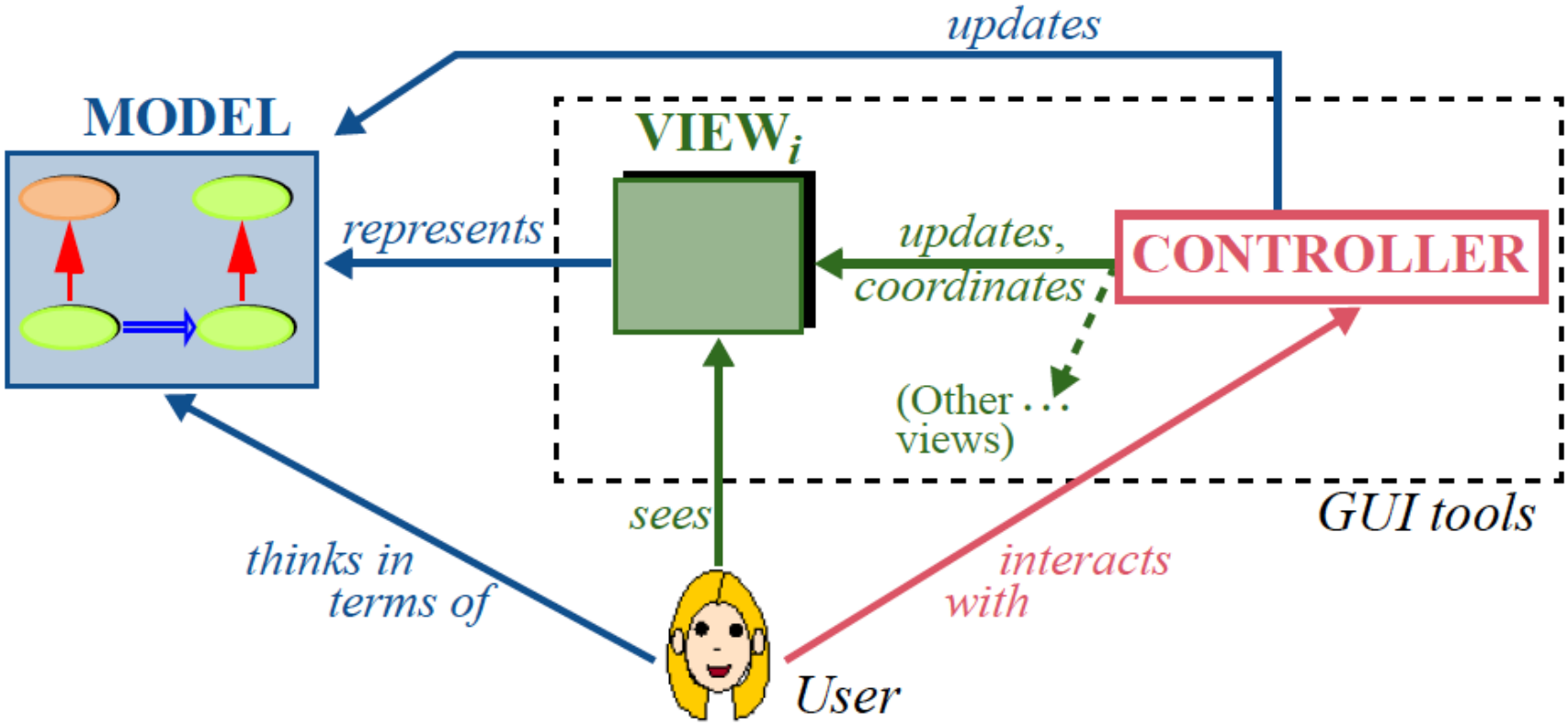
- The choice of classes, based on appropriate data abstractions.
- Deciding which classes will be related, with the overall goal of minimizing the number of such links (to preserve the ability to modify and reuse various parts of the software independently).
- For each such link, deciding between client and inheritance.
- Attaching features to the appropriate classes.
- Equipping classes and features with the proper contracts.
- For these contracts, deciding between a "demanding" style (strong preconditions, making the client responsible for providing appropriate values), a "tolerant" style (the reverse), or an intermediate solution.
- Removing unneeded elements.

Touch of Methodology:
Assessing software architectures

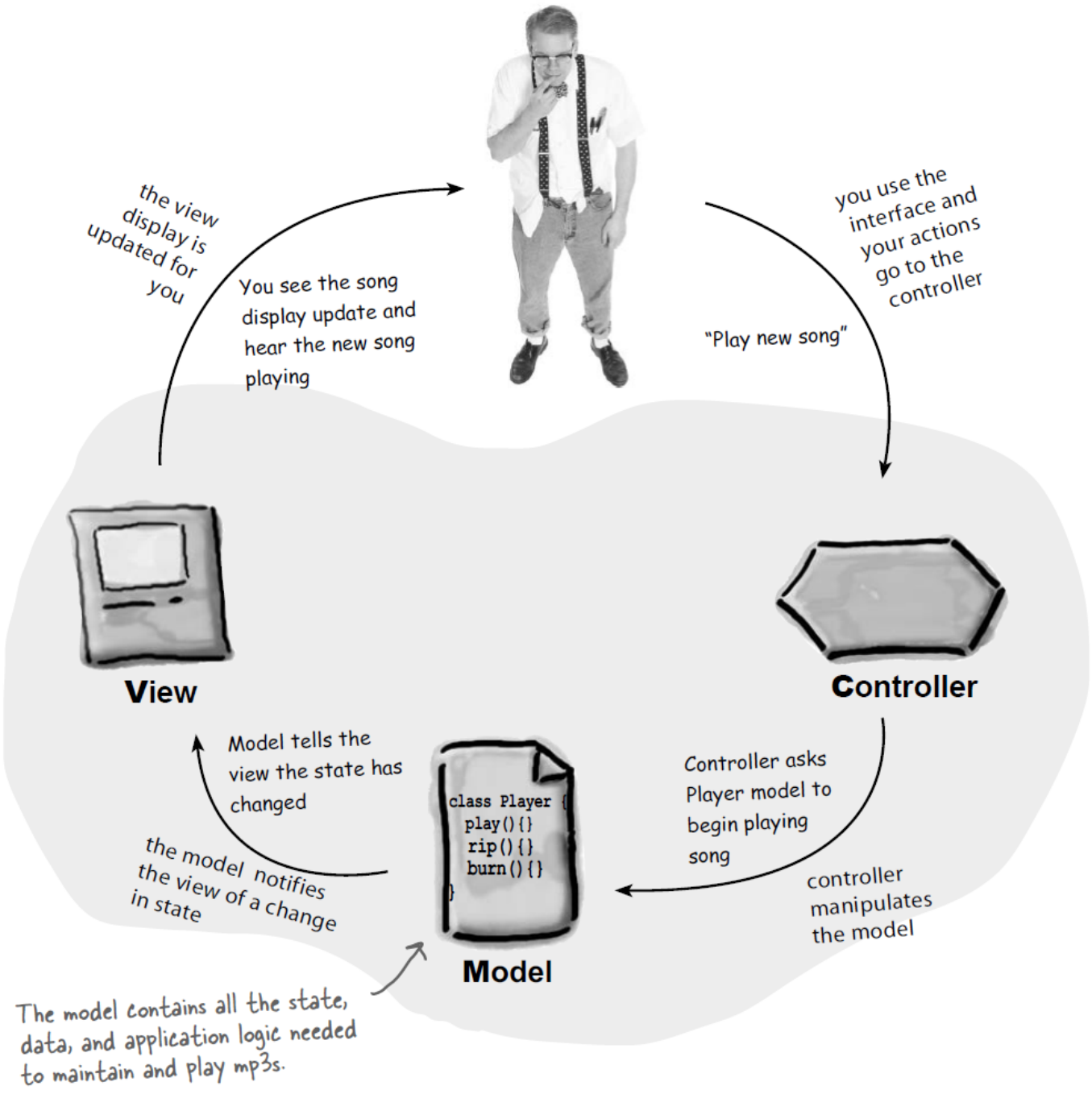
When examining possible design solutions for a given problem, discuss alternatives critically. The key criteria are: reliability, extendibility, reusability and simplicity.

- Avoiding code duplication and removing it if already present; techniques involve inheritance (to make two or more classes inherit from an ancestor that captures their commonality) as well as genericity, tuples and agents.
- Taking advantage of known design patterns.
- Devising good APIs: simple, easy to learn and remember, equipped with the proper contracts.
- Ensuring consistency: throughout the system, similar goals should be achieved through similar means. This governs all the aspects listed so far; for example, if you use inheritance for a certain class relationship, you should not use the client relation elsewhere if the conditions are the same. Consistency is also particularly important for an API, to ensure that once programmers have learned to use a certain group of classes they can expect to find similar conventions in others.

MVC (old)


updates
MODEL
VIEW$_i$
represents
updates,
coordinates
CONTROLLER
(Other ...
views)
GUI tools
sees
thinks in
terms of
interacts
with
User

MVC
iTunes
mp3 player



CONTROLLER

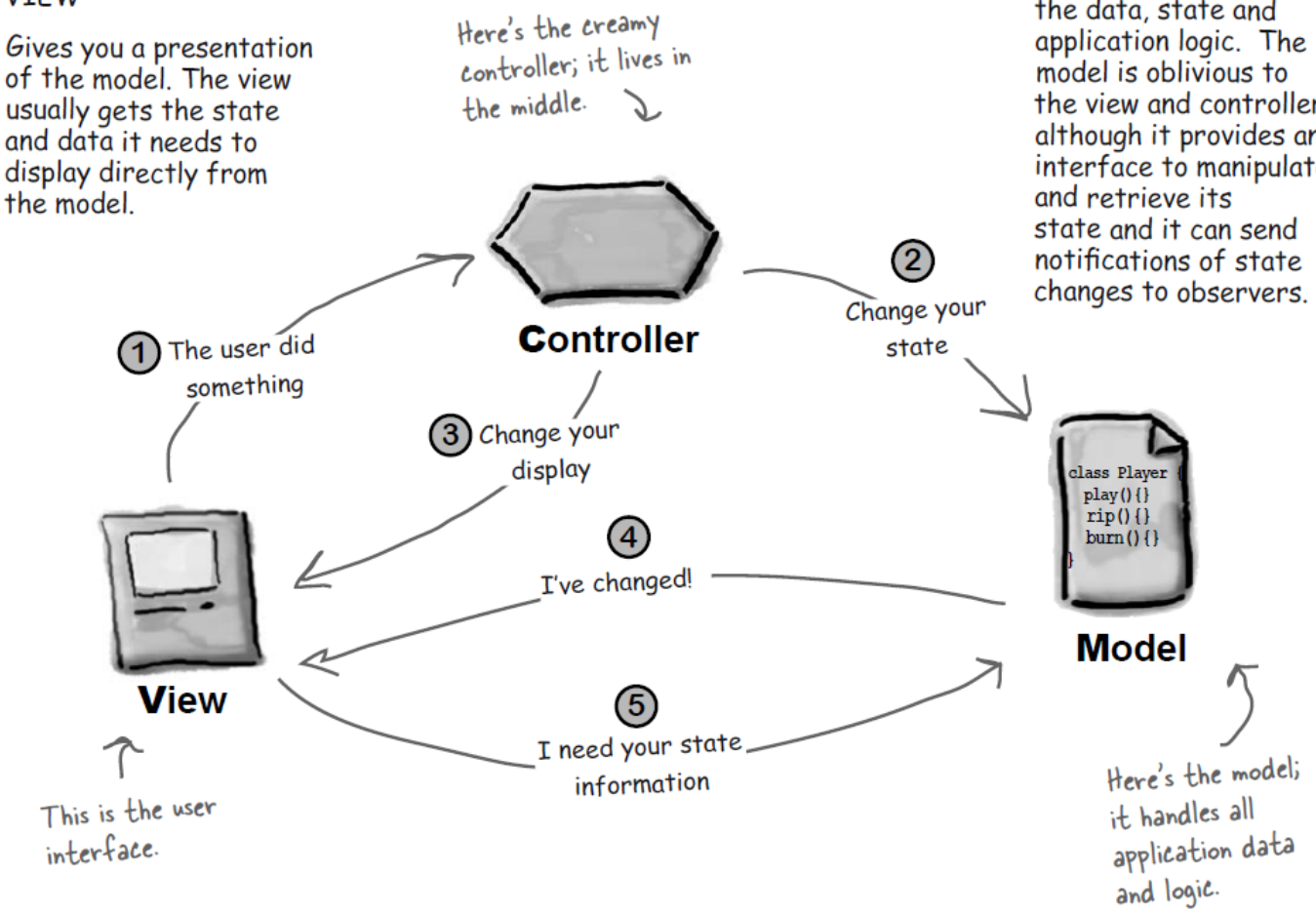
Takes user input and figures out what it means to the model.

MODEL

The model holds all the data, state and application logic. The model is oblivious to the view and controller, although it provides an interface to manipulate and retrieve its state and it can send notifications of state changes to observers.

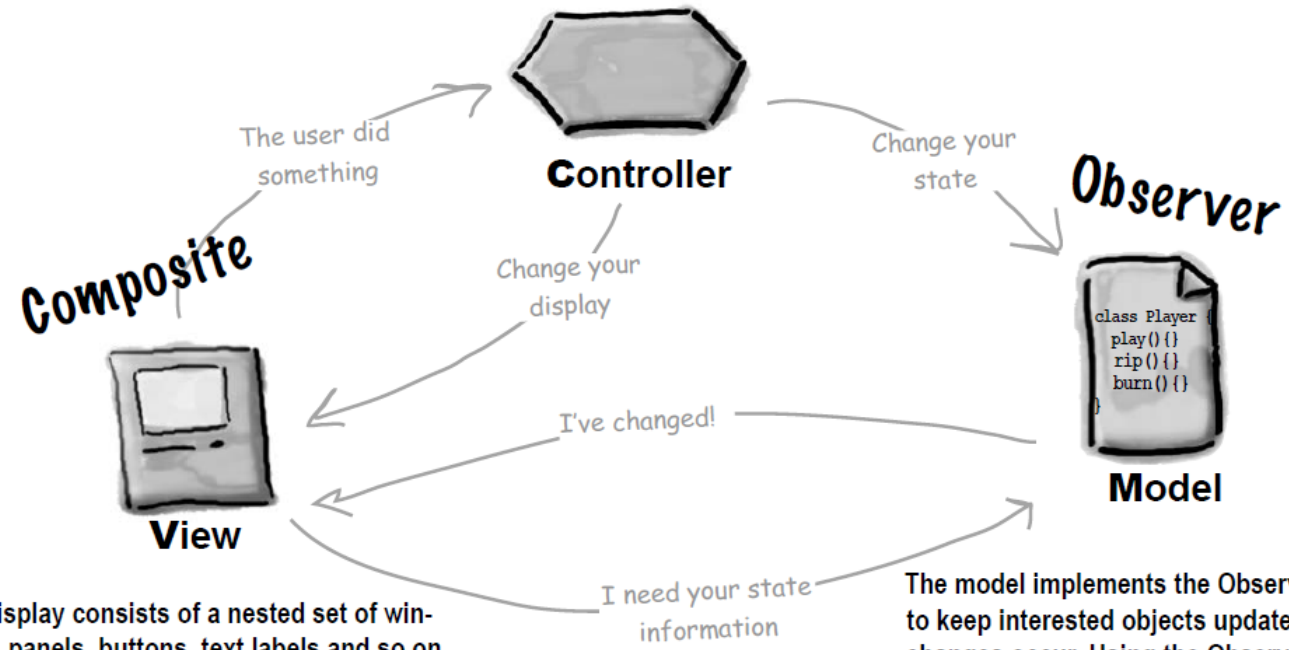
VIEW

Gives you a presentation of the model. The view usually gets the state and data it needs to display directly from the model.



Strategy

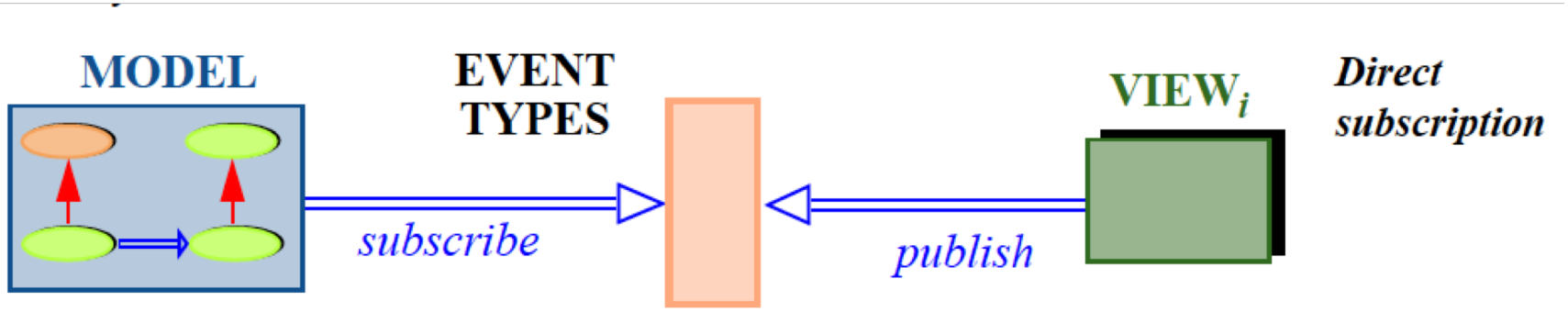
The view and controller implement the classic Strategy Pattern: the view is an object that is configured with a strategy. The controller provides the strategy. The view is concerned only with the visual aspects of the application, and delegates to the controller for any decisions about the interface behavior. Using the Strategy Pattern also keeps the view decoupled from the model because it is the controller that is responsible for interacting with the model to carry out user requests. The view knows nothing about how this gets done.



The display consists of a nested set of windows, panels, buttons, text labels and so on. Each display component is a composite (like a window) or a leaf (like a button). When the controller tells the view to update, it only has to tell the top view component, and Composite takes care of the rest.

The model implements the Observer Pattern to keep interested objects updated when state changes occur. Using the Observer Pattern keeps the model completely independent of the views and controllers. It allows us to use different views with the same model, or even use multiple views at once.

MVC revisited

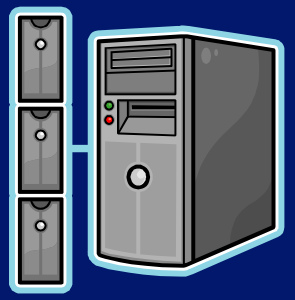


- There is no need for an explicit controller
- The model does not know about the view and the view does not know about the model

CGI: Dynamic Pages

HTTP

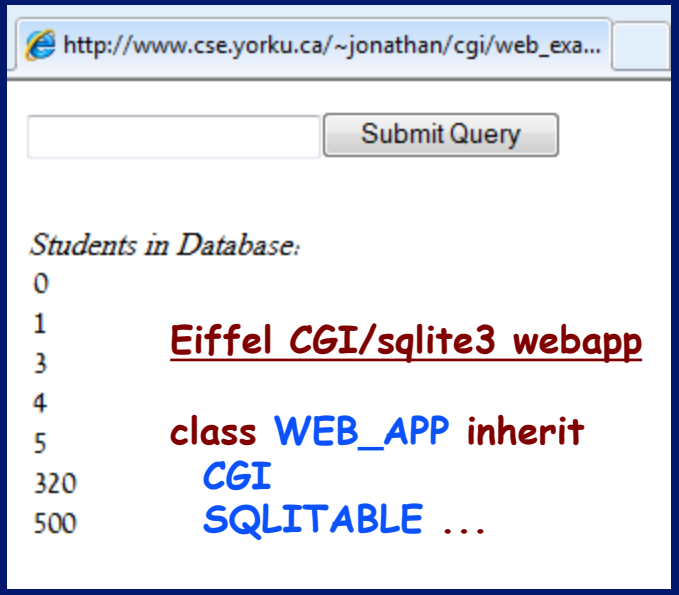
NFS



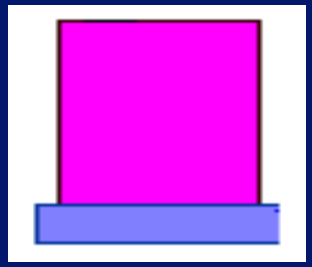
Client

Web Server

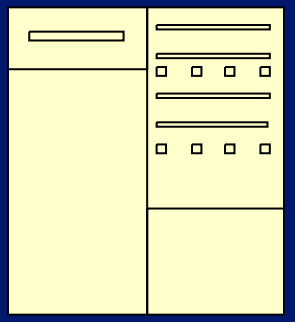
File Server



CGI



JDBC



App Server

DB Server

Key Characteristics of CGI

1. Browser sends an HTTP request to the Web Server
2. Web Server identifies the request as CGI
3. Web Server dispatches the request to the App Server
4. App Server impersonates the script's owner
5. App Server runs the script passing the request
6. Script outputs the response to Standard Output
7. App Server routes Standard Output to the Web Server
8. Web Server returns the response to the browser

UML Sequence Diagram

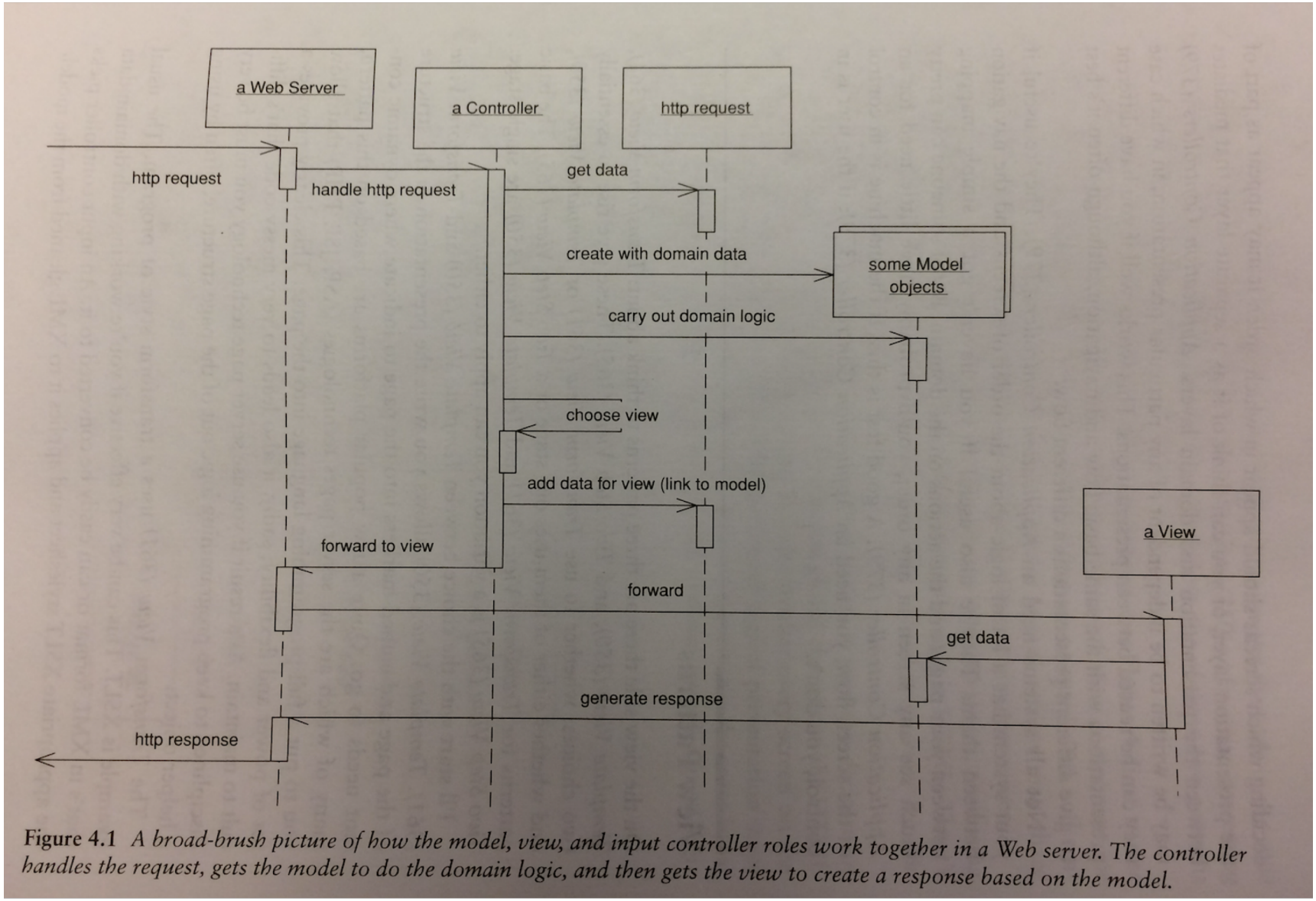
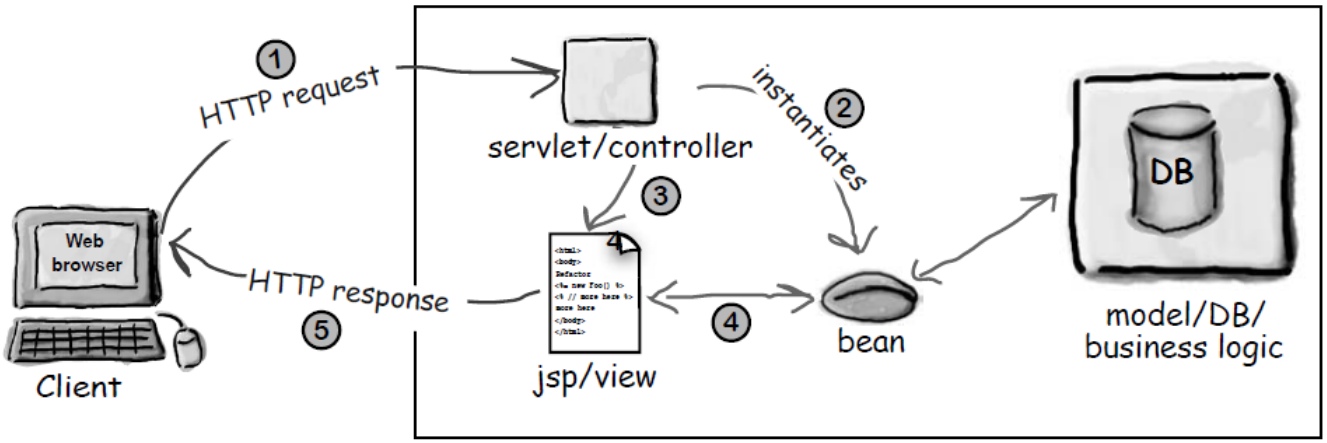


Figure 4.1 *A broad-brush picture of how the model, view, and input controller roles work together in a Web server. The controller handles the request, gets the model to do the domain logic, and then gets the view to create a response based on the model.*

Patterns of Enterprise Application Architecture, Marin Fowler, p57

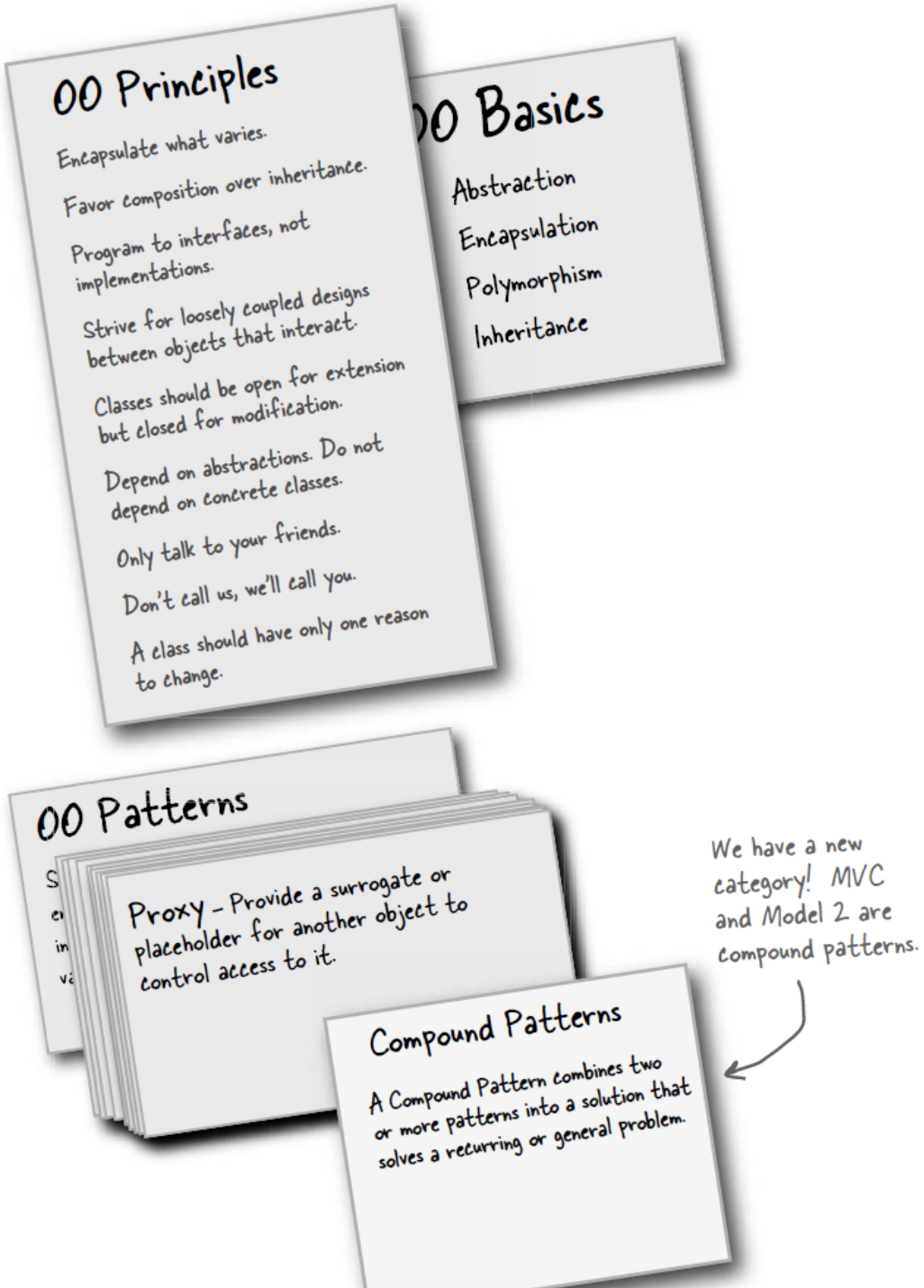


1. **You make an HTTP request, which is received by a servlet.**
   Using your web browser you make an HTTP request. This typically involves sending along some form data, like your username and password. A servlet receives this form data and parses it.

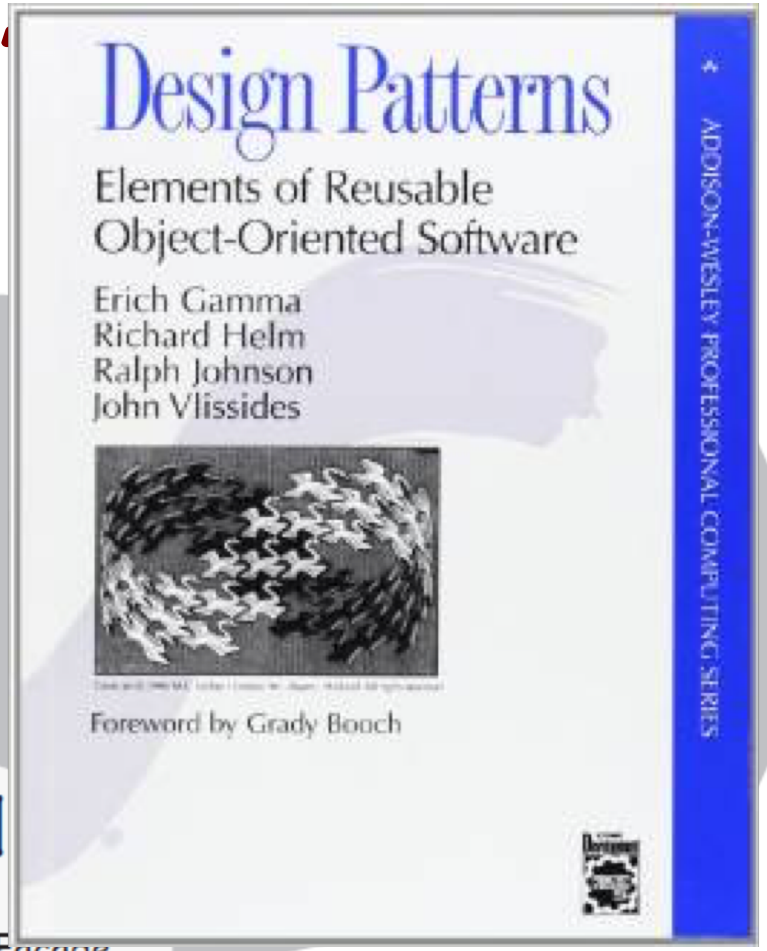
2. **The servlet acts as the controller.**
   The servlet plays the role of the controller and processes your request, most likely making requests on the model (usually a database). The result of processing the request is usually bundled up in the form of a JavaBean.

3. **The controller forwards control to the view.**
   The View is represented by a JSP. The JSP's only job is to generate the page representing the view of model (4 which it obtains via the JavaBean) along with any controls needed for further actions.

5. **The view returns a page to the browser via HTTP.**
   A page is returned to the browser, where it is displayed as the view. The user submits further requests, which are processed in the same fashion.


OO Principles
Encapsulate what varies.
Favor composition over inheritance.
Program to interfaces, not implementations.
Strive for loosely coupled designs between objects that interact.
Classes should be open for extension but closed for modification.
Depend on abstractions. Do not depend on concrete classes.
Only talk to your friends.
Don't call us, we'll call you.
A class should have only one reason to change.
OO Basics
Abstraction
Encapsulation
Polymorphism
Inheritance
OO Patterns
Proxy – Provide a surrogate or placeholder for another object to control access to it.
Compound Patterns
A Compound Pattern combines two or more patterns into a solution that solves a recurring or general problem.
We have a new category! MVC and Model 2 are compound patterns.

Google: "Gang of Four"



Creational

Singleton

Builder

Prototype

Abstract Factory

Factory Method

Structural

Decorator

Composite

Facade

Flyweight

Bridge

Adapter