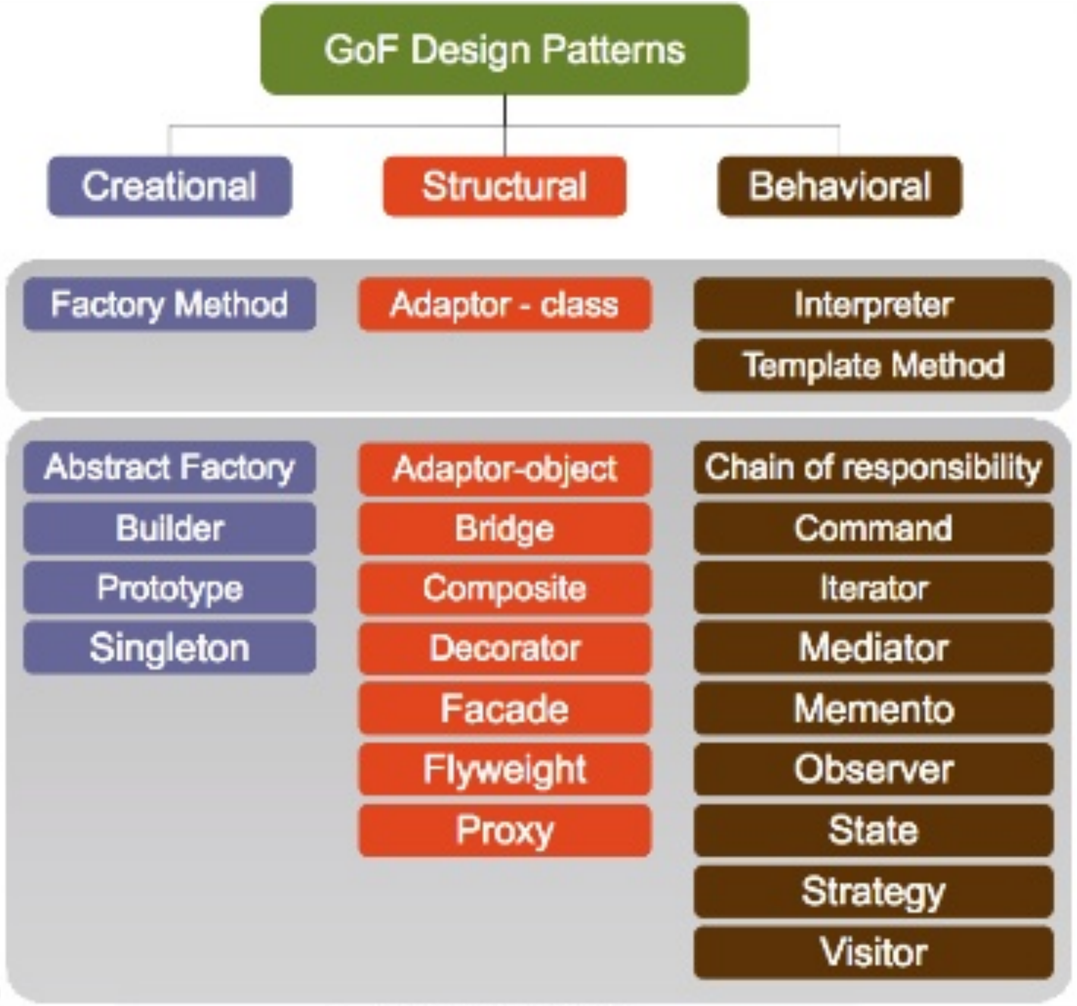
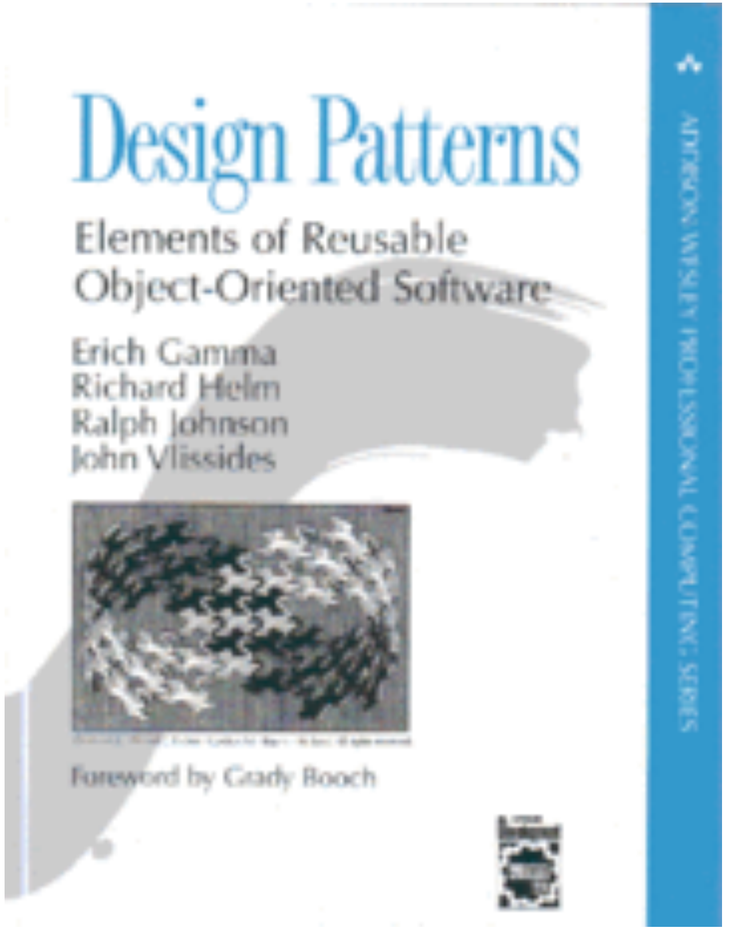
Information System Design

Lecture 6.5 : Design Patterns (Contd.)

Dr. Moustafa Alzantot

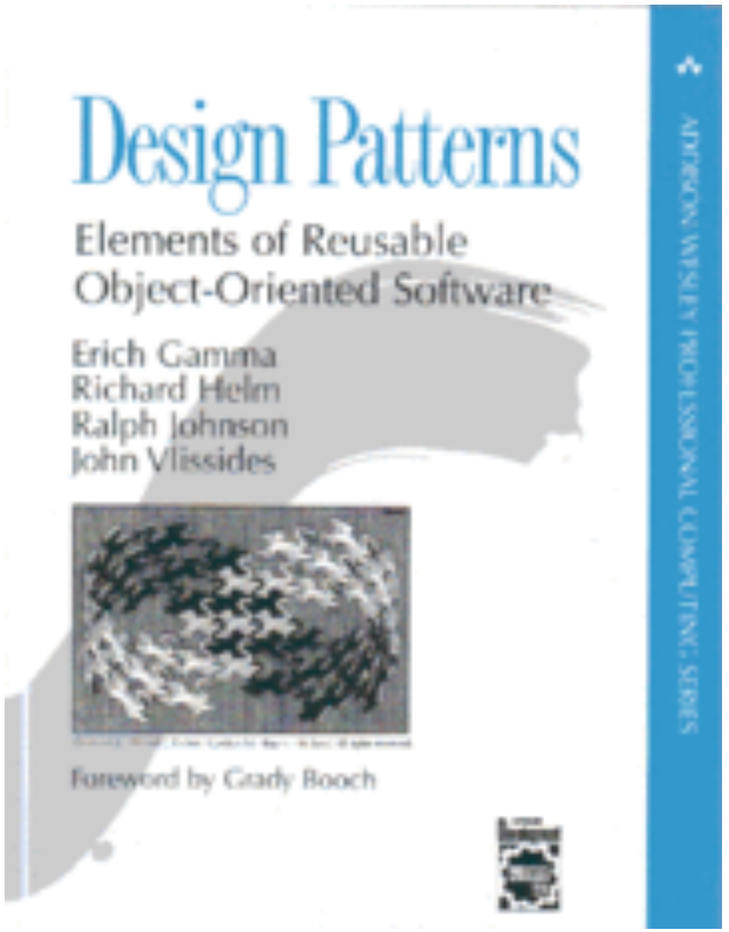
GoF Design Patterns





GoF Design Patterns

- GoF book describes 23 design patterns are categorized by their purpose into 3 categories:
  - **Creational:** related to how we create new objects.
  - **Structural:** concerned with patterns that use compositions of objects and classes to generate larger structures with new functionalities.
  - **Behavioral:** concerned with interactions between classes to divide responsibilities among themselves.



Structural Patterns Example

Decorator Pattern

Decorator Pattern

Decorator pattern is a widely used example of ***structural*** patterns in GoF book.

What problem does *decorator* solve ?

- Dynamically adding (or removing) functionalities to an individual objects without affecting other instances from the same class.
- Provides a more flexible alternative than subclassing to extend functionalities of classes/objects.

Decorator Pattern

Examples:

- **Example 1:** Imagine you have a GUI component (e.g. `TextView`), and you want to create a new extended version of it . For example:
  - `TextViewWithScrollbar`: adds scrollbar to scroll through the content.
  - `TextViewWith3DBorder`: adds a 3D border around that component.
- **Example 2:** Imagine you have a `FileStream` class that reads/writes data from/to a file. Ou want to enhance the performance: for example
  - Adding data compression
  - Adding data encryption

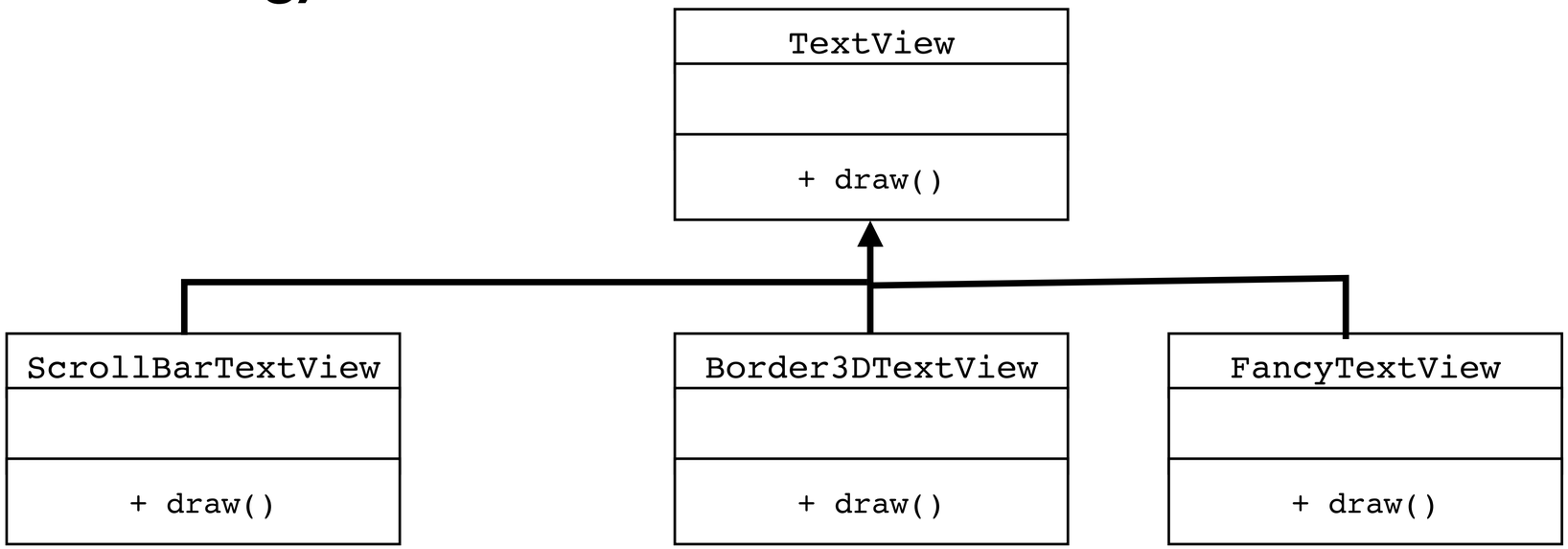
Decorator Pattern

Decorators offer a more flexible way to add functionality than inheritance (subclassing).

*Sometimes a large number of independent extensions are possible , and would produce a an **explosion of subclasses** to support every combination of extensions.*

Decorator Pattern

Decorators offer a more flexible way to add functionality than inheritance (subclassing).



How would you create a Fancy TextView that has both scrollbar and 3D border ?

Decorator Pattern

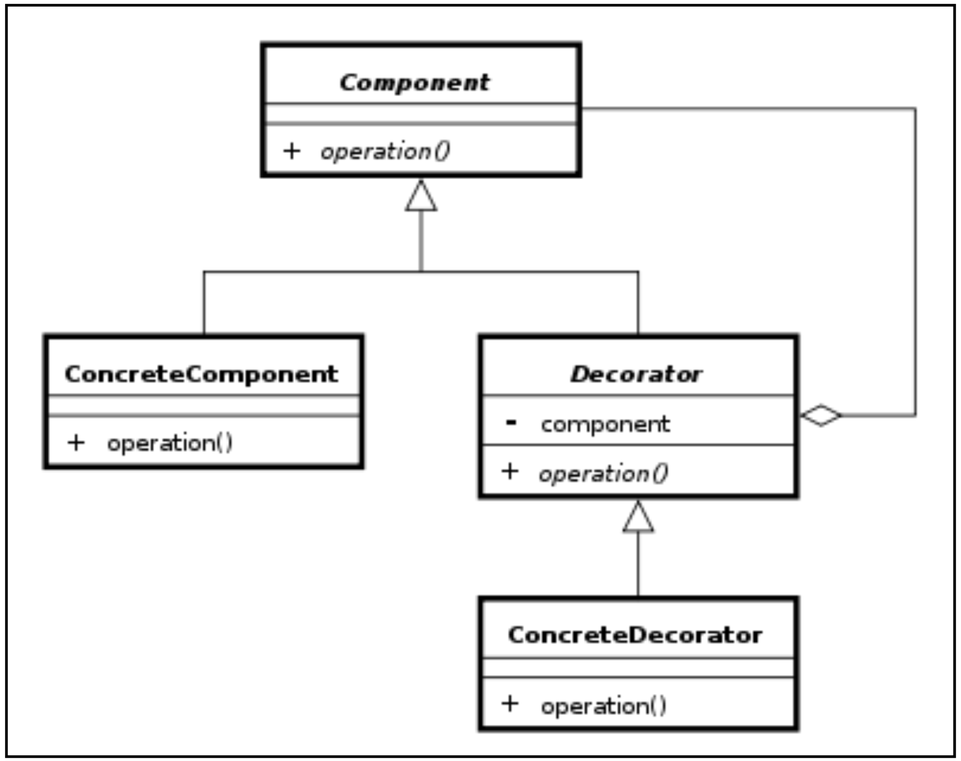
Decorator pattern extend the functionality by wrapping an object within another “decorator” object.

Decorators can be wrapped around each other, therefore allowing for flexible way to combine new added functionalities.

Decorator Pattern

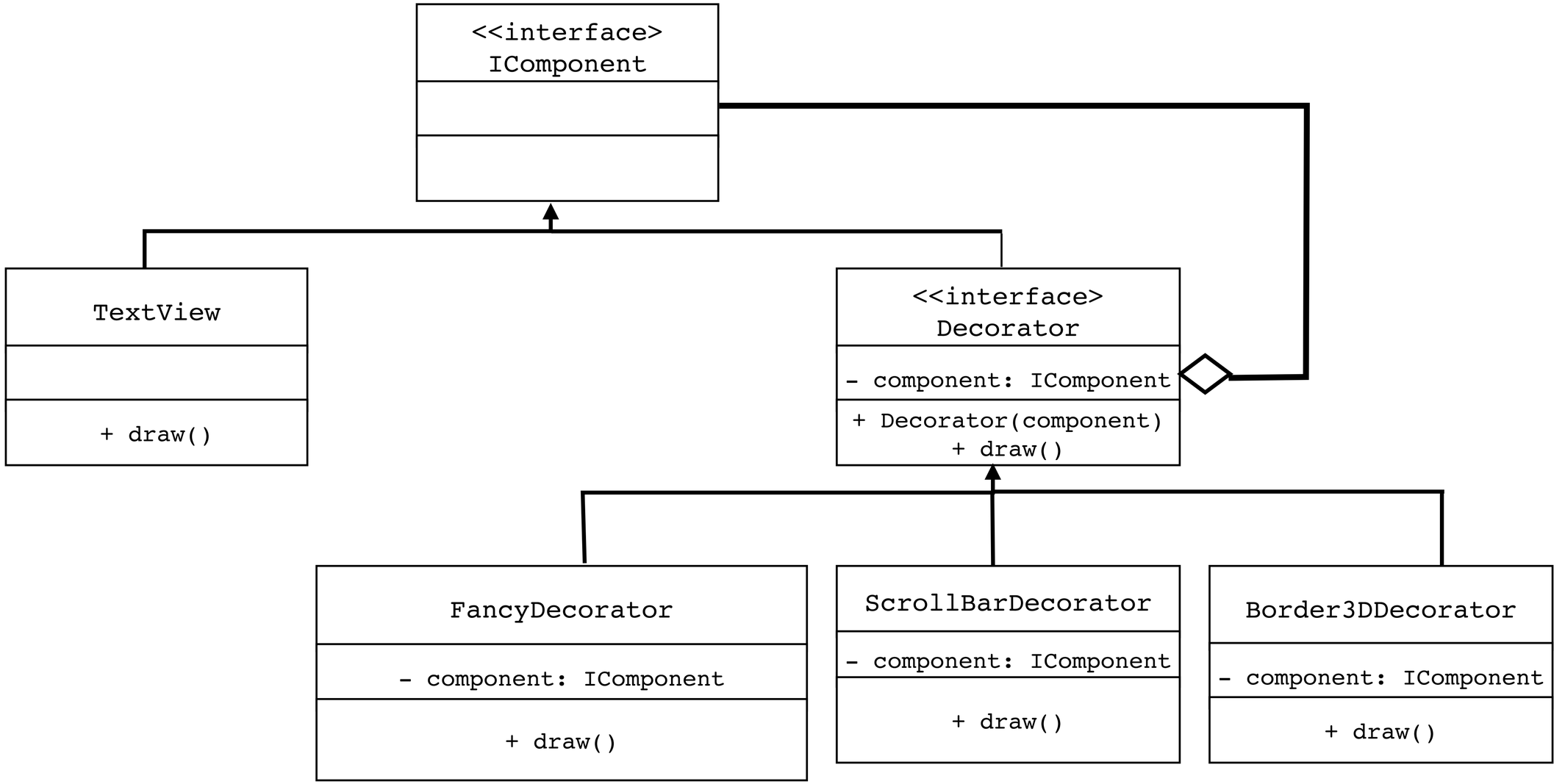
Decorator patter

- Define a new “**Decorator**” class that wraps the original class
- **Decorator** class has the same interfaces “Component”
- **Decorator** class encloses a component object inside it
- **Decorator** forwards requests to the component and may perform actions before or after any forwarding.



Decorator Pattern

Solution



Decorator Pattern

Decorator pattern extend the functionality by wrapping an object within another “decorator” object.

Decorators can be wrapped around each other, therefore allowing for flexible way to combine new added functionalities.

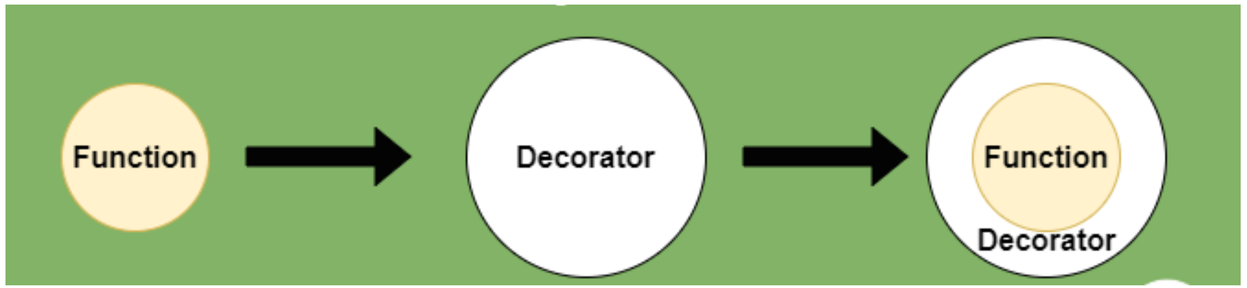
```
IComponent* text_view = new TextView();
IComponent* scrollbar_text_view = new ScrollBarDecorator(text_view);
IComponent* border_3d_scrollbar_text = new Border3DDecorator(scrollbar_text_view);
```

Decorator Pattern: function decorators

Another application of decorator patterns.

A common practice in programming languages such as C#, Python and Javascript.

A decorator wraps one piece of code with another.



Function Decorators: Python Example

```
def say_hello():
  return "hello world"

func = say_hello
func()
```

Output:

hello world

Function Decorators: Python Example

```
def upper_case_decorator(function):
  def wrapper():
    output = function()
    return output.upper()
  return wrapper

func = upper_case_decorator(say_hello)
func()
```

upper_case_decorator is wrapped around **say_hello** function

Output:

HELLO WORLD

Function Decorators: Python Example

```
def upper_case_decorator(function):
  def wrapper():
    output = function()
    return output.upper()
  return wrapper

@upper_case_decorator
def say_hello():
  return "hello world"

@upper_case_decorator
def say_bye():
  return "bye"

print(say_hello())
print(say_bye())
```

In Python, you can use @ function annotation to decorate a function with another.

Behavioral Patterns Example

Observer Pattern

Observer Pattern

Goal of *Observer* Pattern:

You have an object with some internal state. The state value for that object could change during the program. Whenever an event happens that causes the value of this object state to change, another objects should be updated.

- *This can be a 1:many relationship. I.e. many objects should notified of the event.*
- *We need to reduce the coupling between objects.*

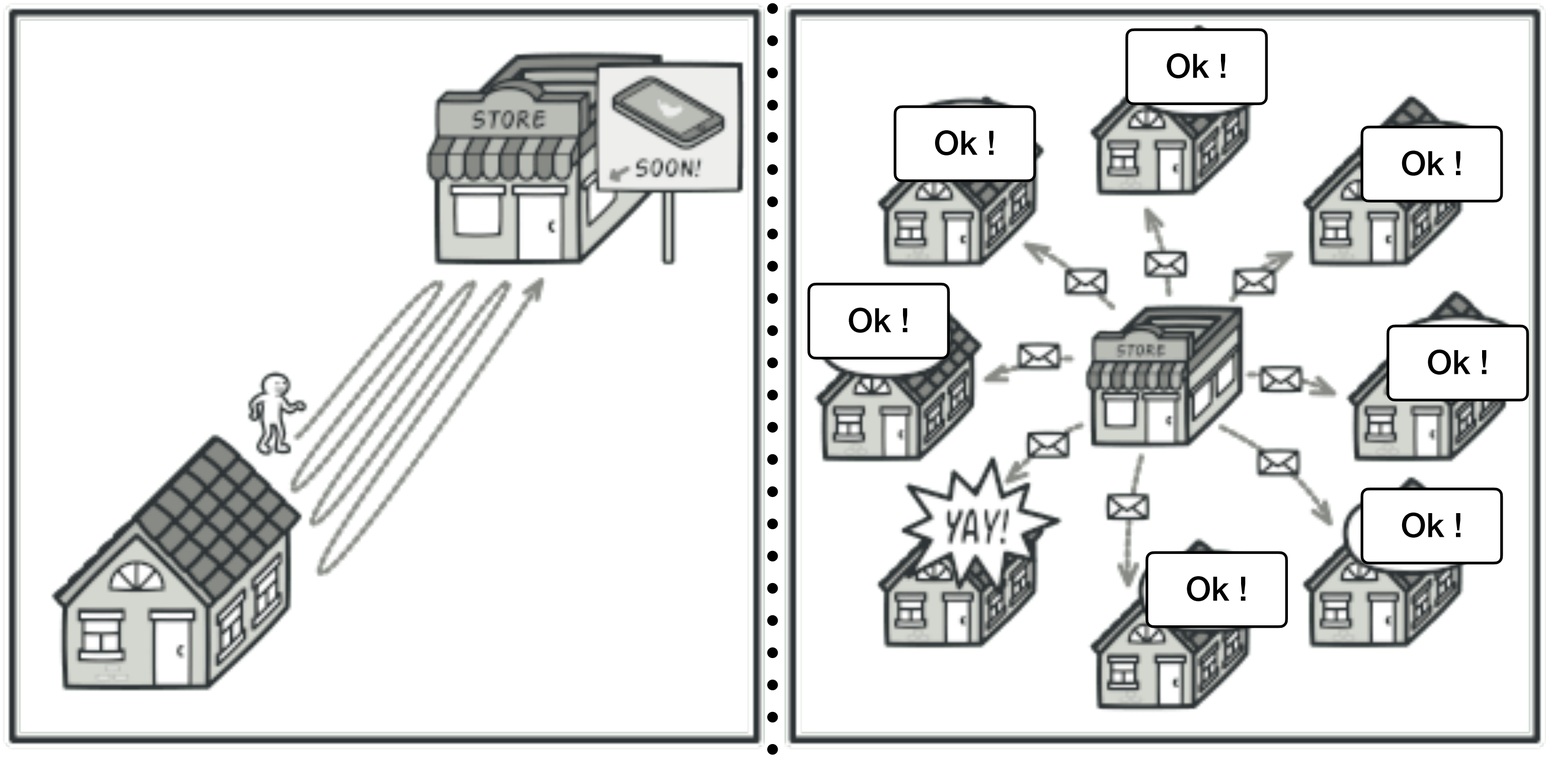
This is a very common situation in any complex system.

Observer Pattern

Common Use Cases of Observer Pattern:

- Handling GUI events (e.g. button clicked)
- Responding to file transfer operations (e.g. transfer has finished, error happened)
- Monitoring sensor values.

Observer Pattern



Polling

Observer Pattern
a.k.a. publisher-subscriber

Source: https://refactoring.guru/design-patterns/observer

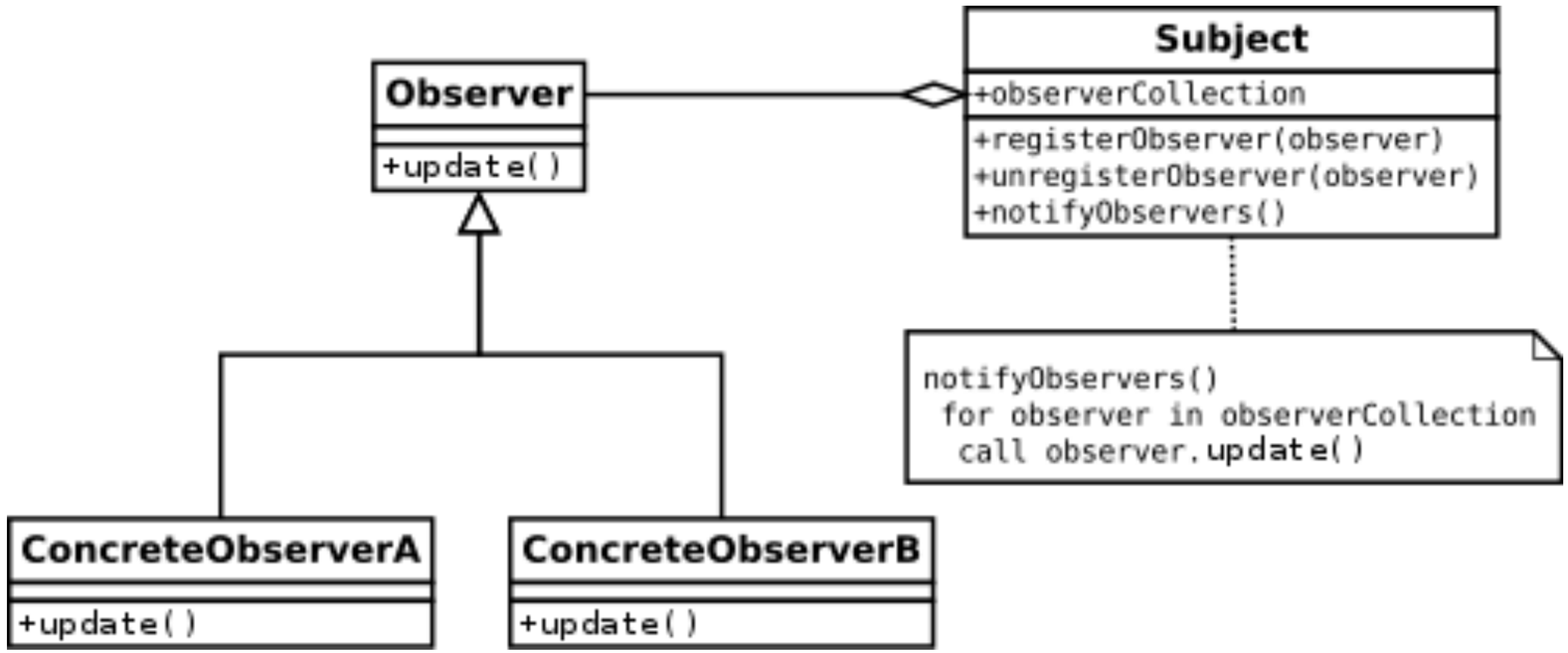
Observer Pattern

Key Players:

- **`Subject`:**
  - maintains a list of observers.
  - Provides an interface for registering (adding)/unregistering (removing) observers..
  - Maintains the object status, and when status changes it will send notification to all registered observers.
- **`Observer`:** An interface for Observers who are interesting in receiving notification for value changes.
- **`ConcreteObserver`**: Implements Observer interface. Will provide its own custom logic to handle the event notifications.

Observer Pattern

UML Class Diagram



Observer Pattern

UML Sequence Diagram

- Both observers `o1` and `o2` registers for notification (by calling subject `attach` method).

- A change in subject state, causes a call to **`notify()`** method which will call **`update()`** method of all attached observers.

- Observers interested in getting the new state value will call subject `getState()` function to read the new value.