327: Object-oriented programming

Lecture 16

11/1/2021

Professor Barron

Iterators

- Iterators are a common object-oriented pattern
- Classic…
  - for (int i=0, i< 10; i++) {
    printf("%d\n", arr[i]);}
- Iteration is controlled externally
- General iterator pattern…
  - while not iterator.done():
    print(iterator.next())
- Iteration controlled internally

for x in Iterator:

class Iterator:

def prev():

def current():

def set_current(e):

Iterators

- This pattern is baked into many languages
  - Python, all for loops use iterators
  - Java, for each loops when an object implements Iterable interface
  - C++, incrementing/decrementing pointers steps through arrays (size aware) and classes can overload pointer operations

ptr++
*ptr

for (Element x: elementList)

Iterator protocol

- An object that is “iterable” must implement __iter__
- This returns an Iterator object
- The iterator must implement __next__ and when it is done it must raise a StopIteration exception

lst = [1, 2, 3]

lst_iter = lst.__iter__() iter(lst)

lst_iter2 = lst.__iter__()

next(lst_iter) next(lst_iter2)

next(lst_iter)

Iterators in Python

- for loops only work on iterators
- Also used for comprehensions and generators

```
for (i, x) in enumerate(lst):
```

i: index, x: elem

```
gen = (int(x) for x in strings)
```

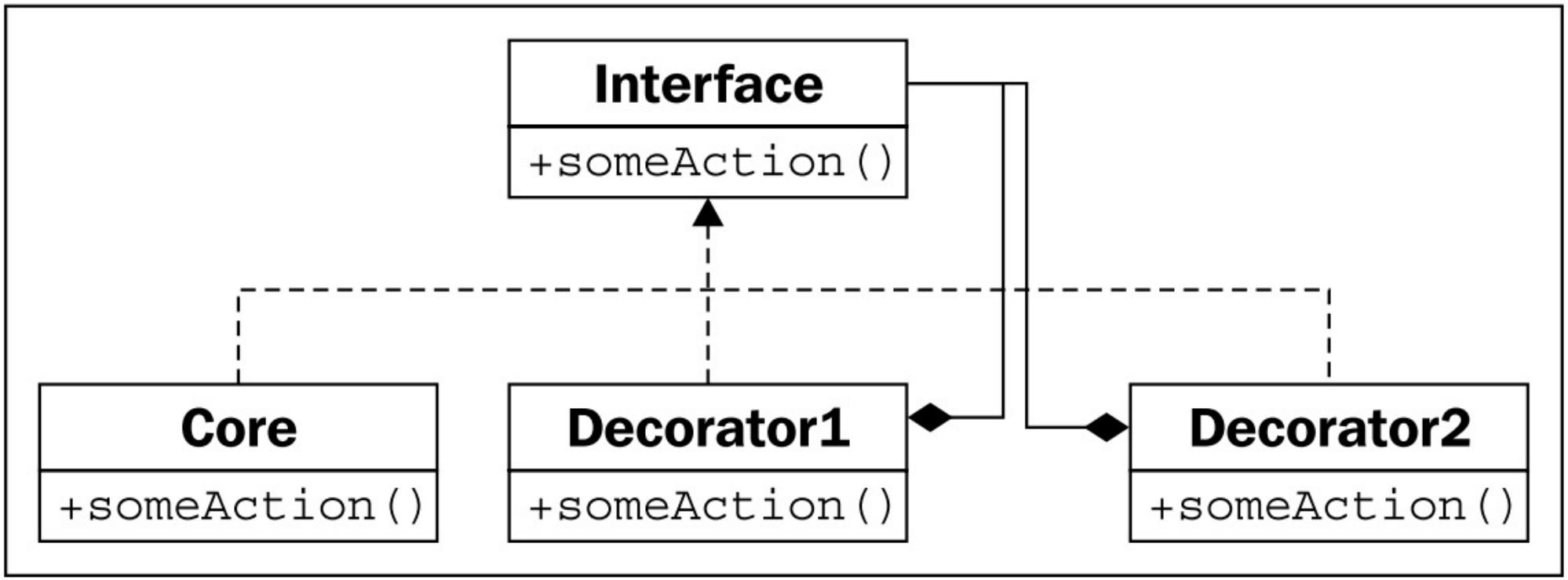
```
[int(x) for x in strings]
```

```
ints = []
for x in strings:
    ints.append(x)
```

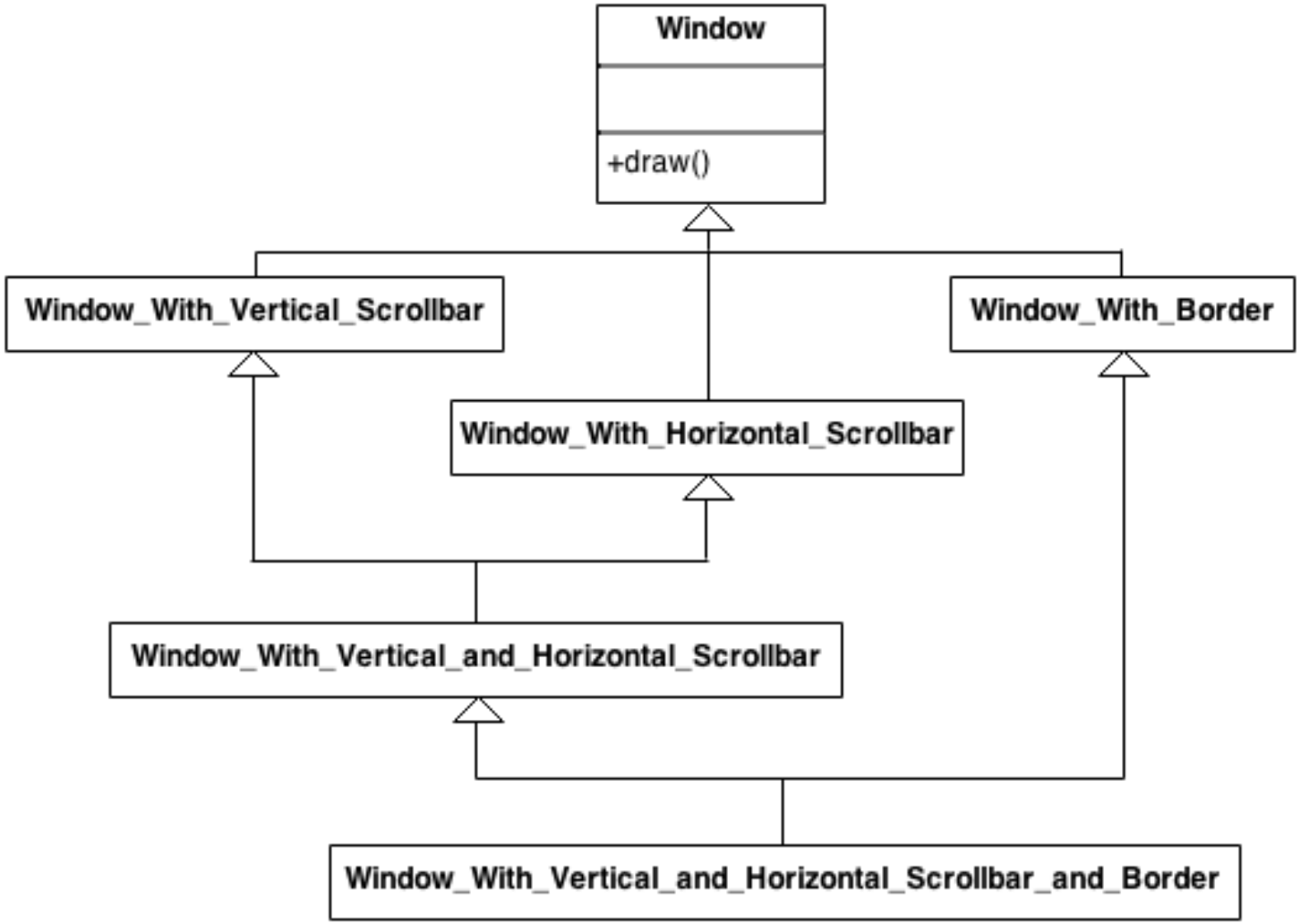
Decorator Pattern

- Structural pattern
- Add capabilities to an object dynamically (without monkey patching)
- Wrap an object with another object
- Can recursively wrap with more decorators
- Core interface remains untouched
- Another pattern that made its way into Python syntax

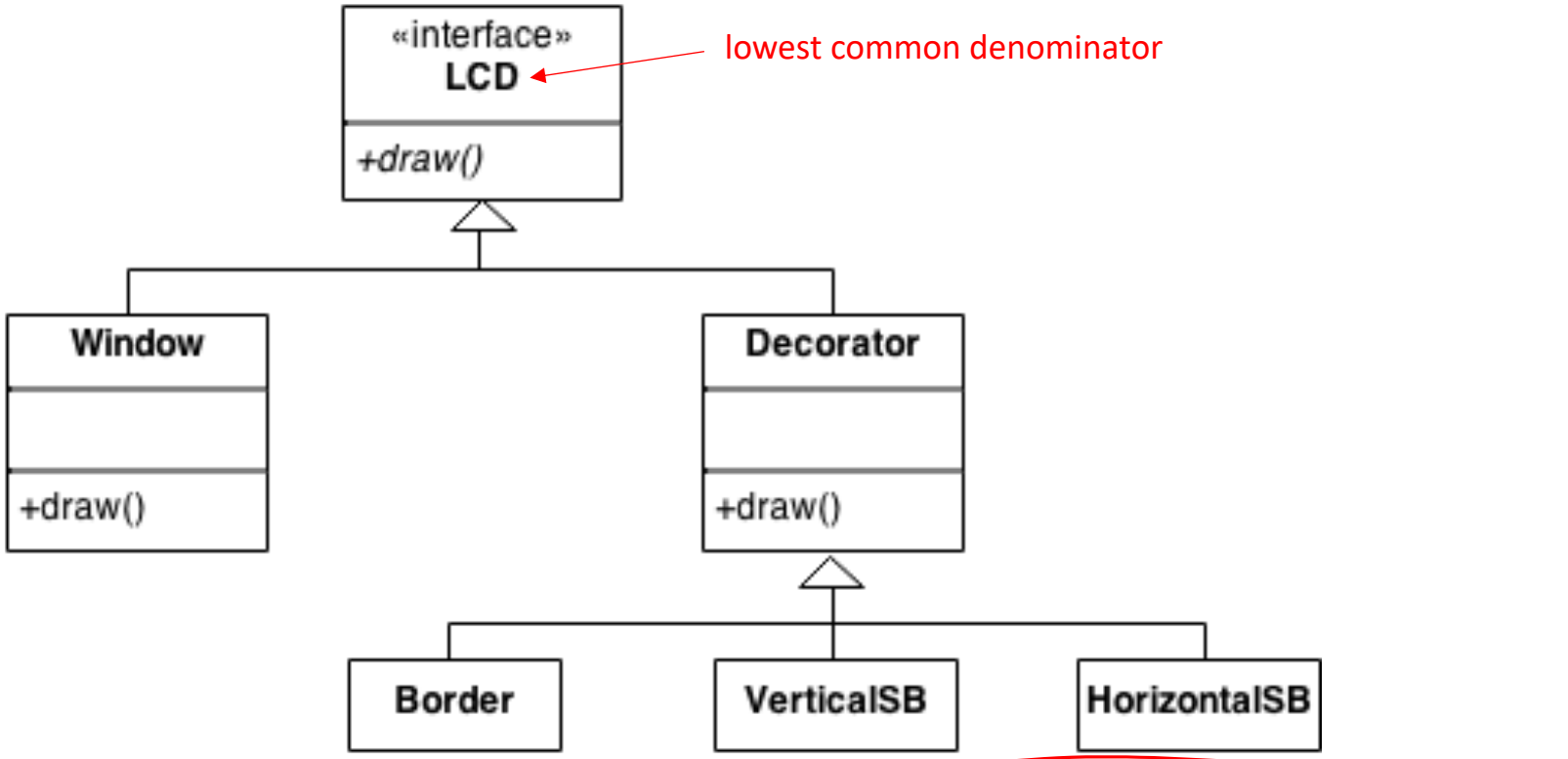
Decorator class diagram

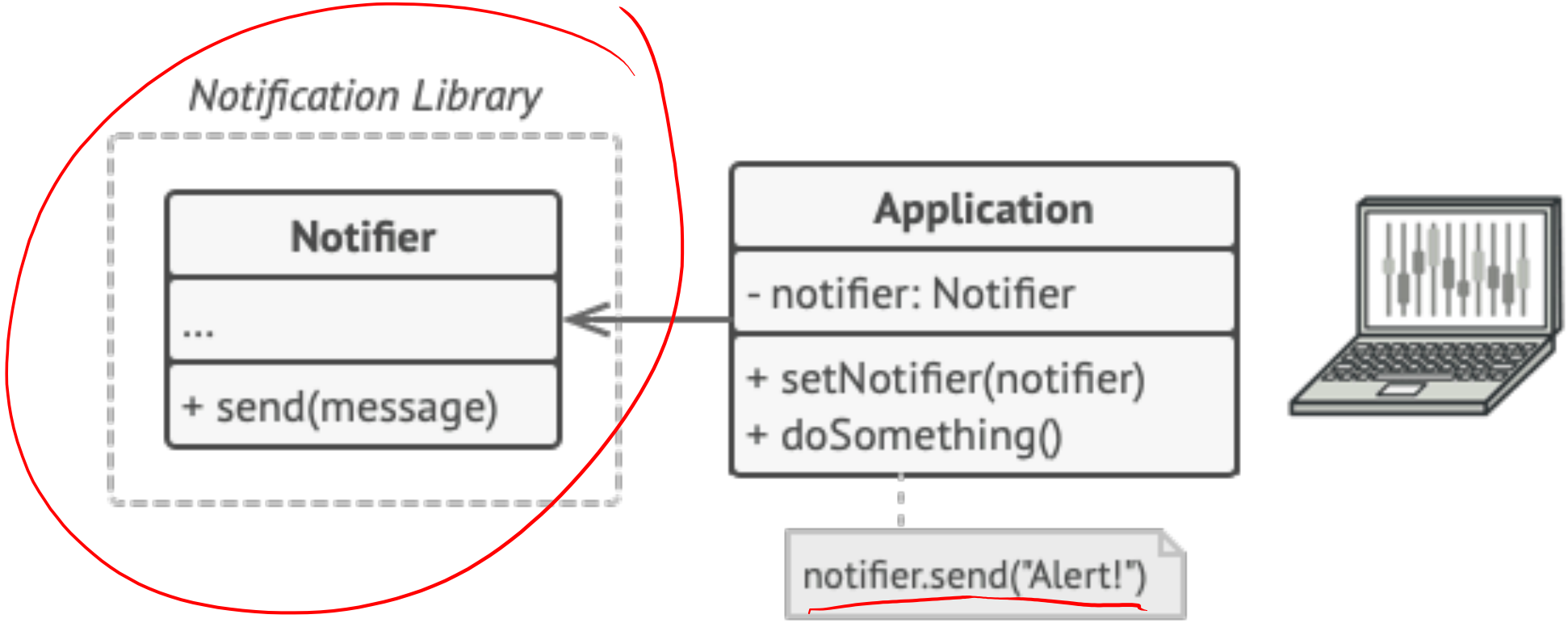


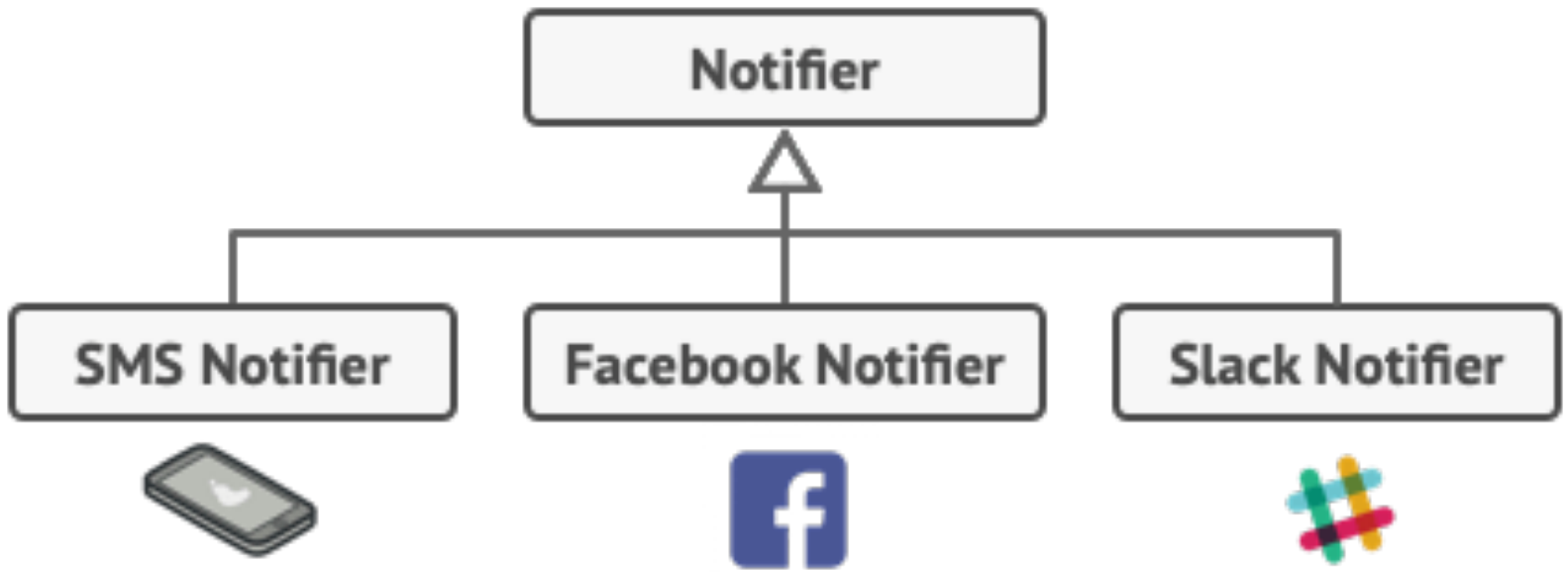
User interface example

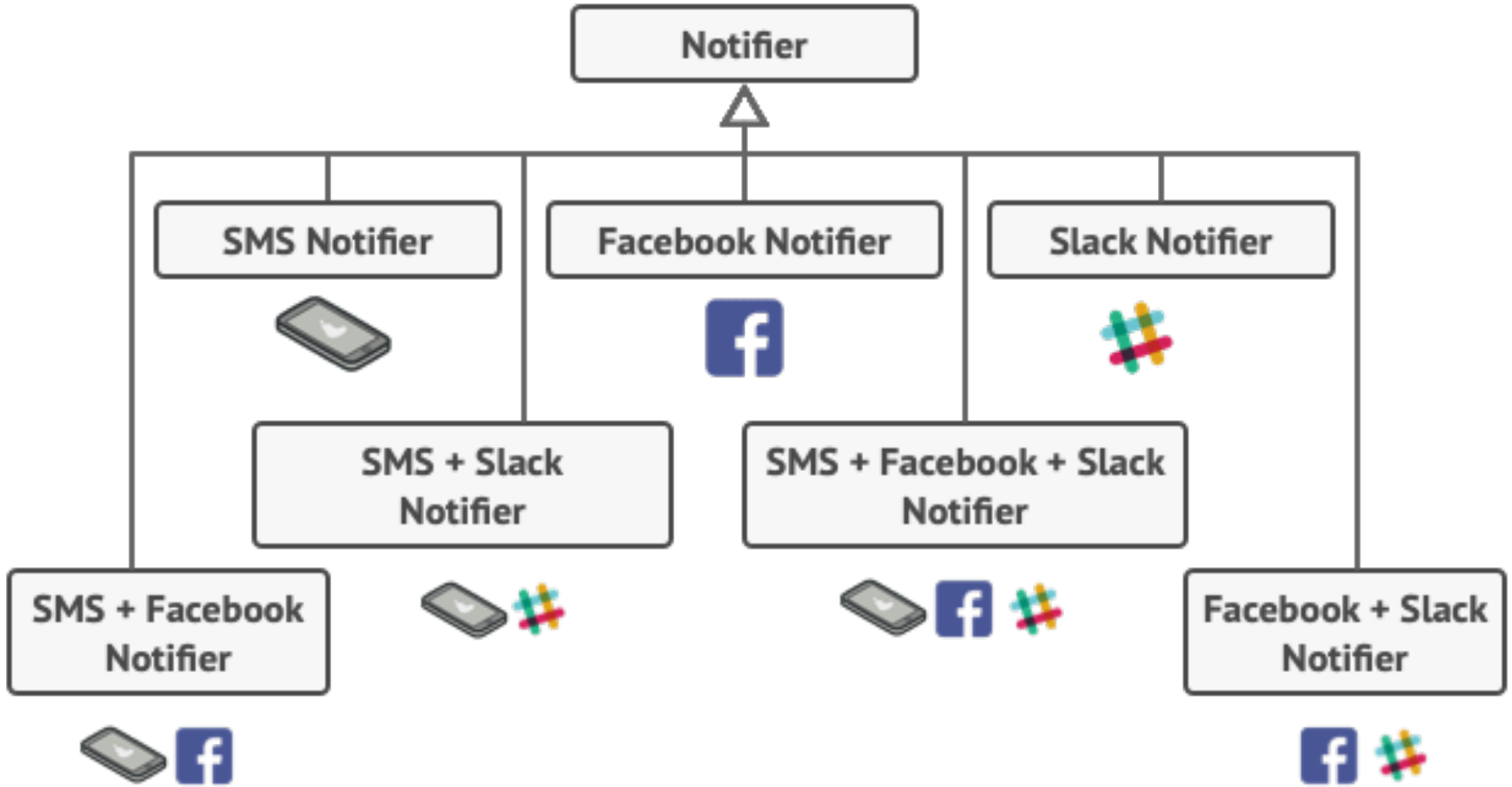


User interface example



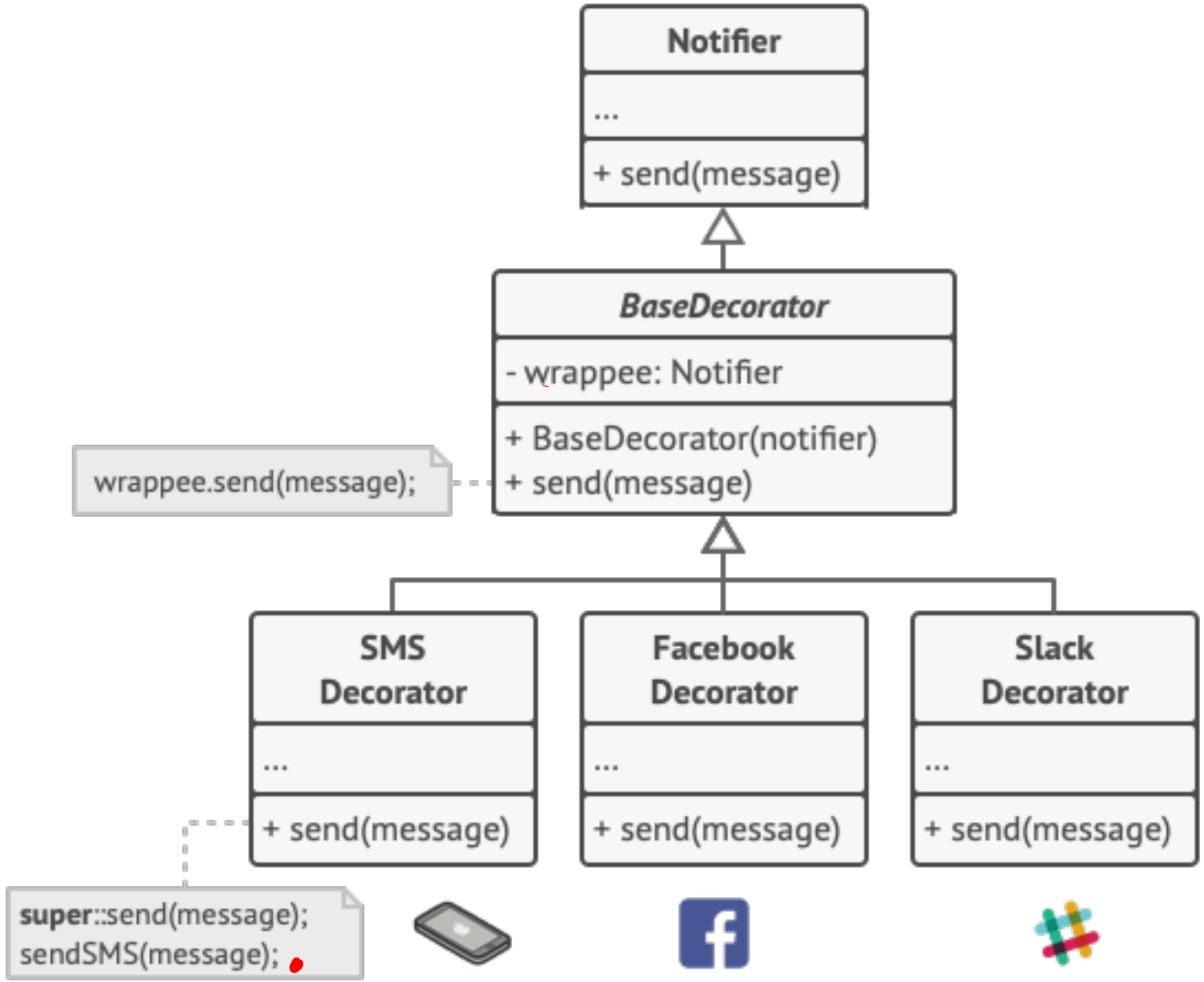

Notification Library
Notifier
...
+ send(message)
Application
- notifier: Notifier
+ setNotifier(notifier)
+ doSomething()
notifier.send("Alert!")


Notifier
SMS Notifier
Facebook Notifier
Slack Notifier


Notifier
SMS Notifier
Facebook Notifier
Slack Notifier
SMS + Slack Notifier
SMS + Facebook + Slack Notifier
SMS + Facebook Notifier
Facebook + Slack Notifier

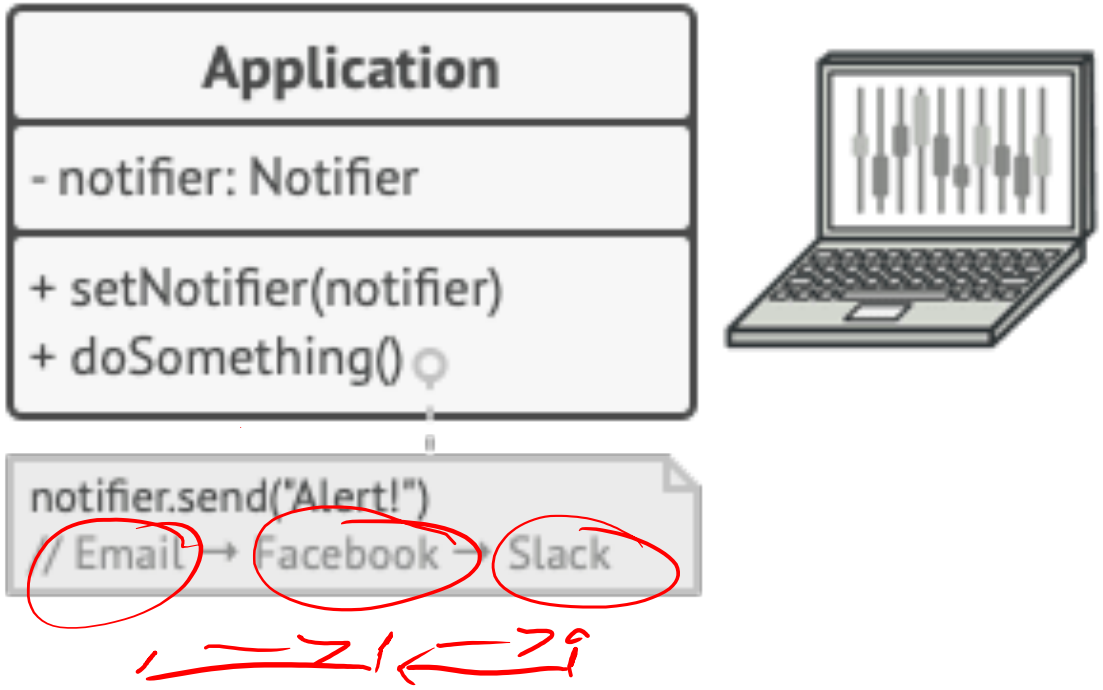
Inheritance vs “Composition”

- Note that, is sometimes “composition” used as a catch all term for “has-a” relationships
- Inheritance is static
- “has-a” relationships are more flexible by swapping out a reference
  - Inheritance is kind of like composition if you think about the subclass as holding a reference to the parent
  - that reference can’t be changed, and you only get one (unless willing to deal with multiple inheritance)


Notifier
...
+ send(message)
BaseDecorator
- wrappee: Notifier
+ BaseDecorator(notifier)
+ send(message)
wrappee.send(message);
SMS Decorator
...
+ send(message)
Facebook Decorator
...
+ send(message)
Slack Decorator
...
+ send(message)
super::send(message);
sendSMS(message);

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```



Decorators in Python

- In addition to decorating objects…
- Can also decorate functions
  - Remember functions are objects
- Special syntax for statically defined decorators
  - @decorator_name before definition
  - we've seen this before
  - less powerful
    - baked into the source code
    - can't do it at runtime
  - often easier to read