Design Patterns (2)

November 13, 2018

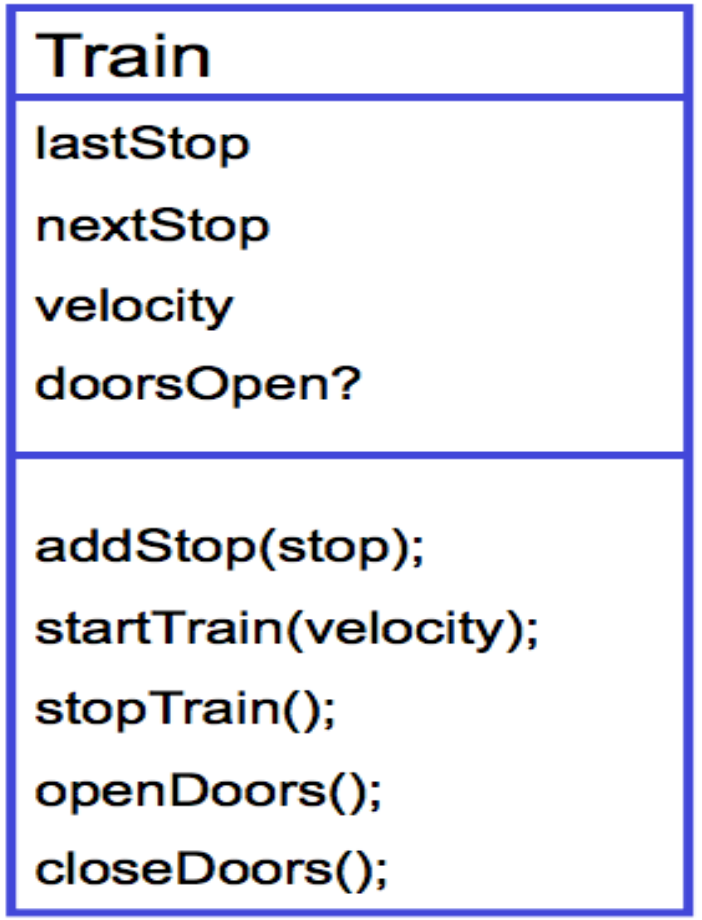
Byung-Gon Chun

(Slide credits: George Candea, EPFL and Armando Fox, UCB)

Just Enough UML

Class Diagrams

- Describe classes
  - In the OO sense
  - Statically: what interacts with what, but not what happens
- Each box is a class
  - Name
  - (public) fields
  - (public) methods
- The more detail, the more it becomes a design



Class Diagrams: Relationships

- Many different kinds of edges to show different relationships between classes

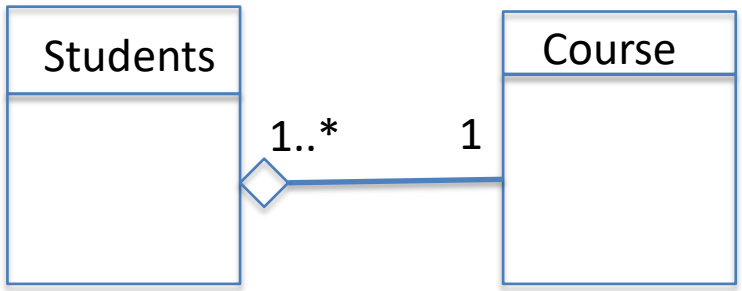
- Associations
  - multiplicity
- Aggregation
- Composition
- Inheritance

Associations

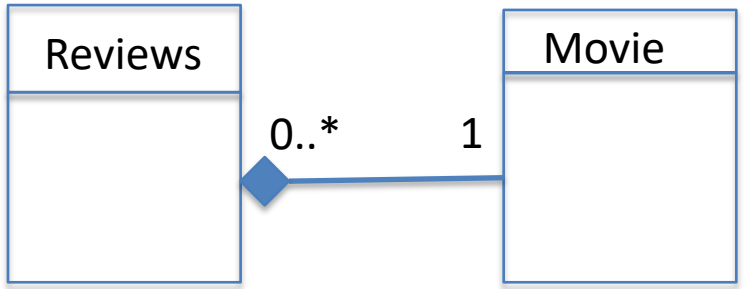
- Capture n-m relationships
  - Like entity-relationship diagrams (from databases)
  - “Connected to” relationship
- Label endpoints of edge with cardinalities
  - Use * for arbitrary
- Typically realized with embedded references

Two Kinds of Owning Associations: Aggregation and Composition

- In an **aggregation**, the owned objects survive destruction of the owning object

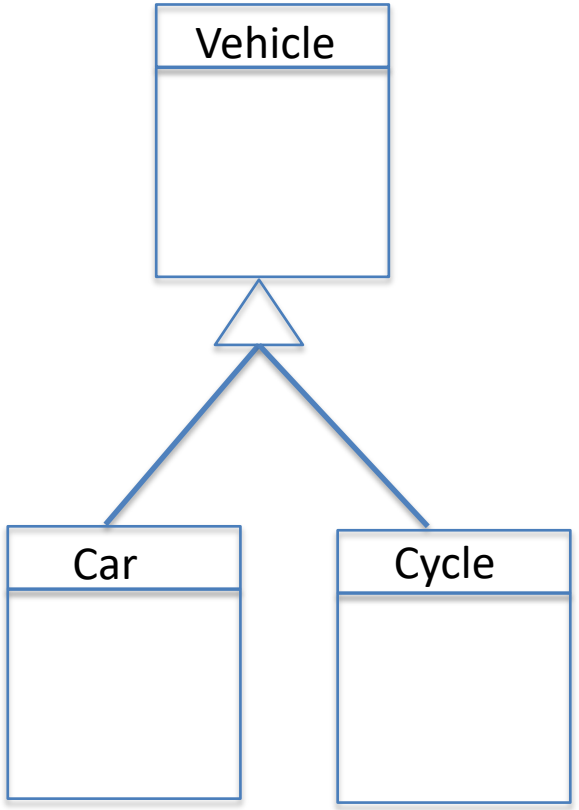


- In a **composition**, the owned objects are usually destroyed when the owning object is destroyed

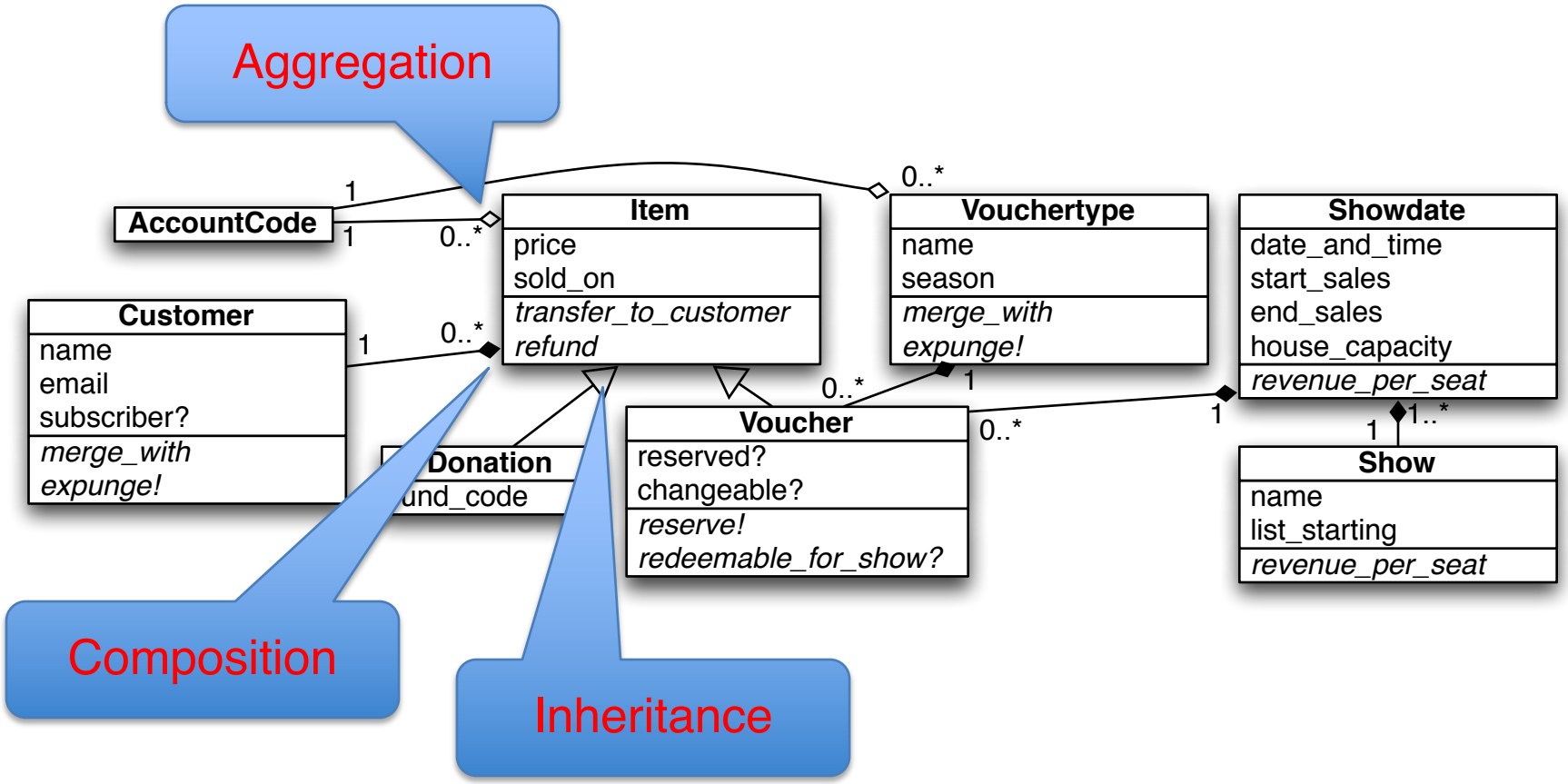


Generalization/Inheritance

- Inheritance between classes
- Denoted by open triangle on superclass
- All arrows point in the direction of code dependency

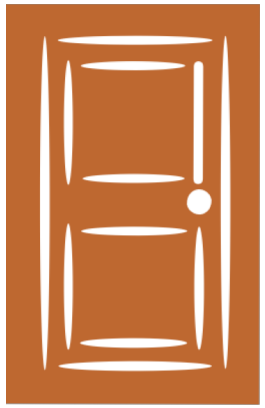


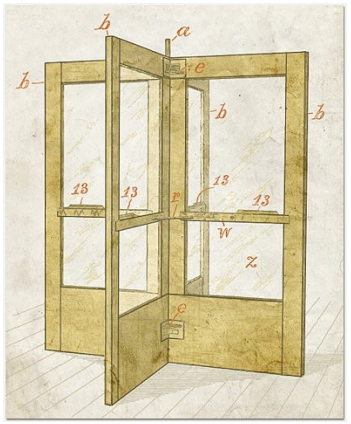
Relationships



Inheritance vs. Containment

Inheritance





- Inherit properties of base class
  - E.g., door vs. specific doors
  - Polymorphism: operations that adjust at runtime
- Use only when it simplifies design
  - Rich set of operations on the base class
  - Mapping to real-world inheritance
- Containment
  - Is containment a better choice than inheritance?

Inheritance vs. Containment

```
class Passenger {
   FullName name;
   Address address;
   PhoneNumber number;
}

class VIP extends Passenger {
   FrequentFlyerNumber account;
}
```

- Inheritance = “is a”
  - Class is a specialization of another class
  - Share common data and methods
- Containment = “has a”
  - Class is implemented with the help of another
  - Accesses are translated and forwarded

Liskov Substitution Principle (LSP)

Barbara Liskov

- LSP intuition
  - Subclass is specialized version of base class
  - All methods of subclass usable through base class interface without knowing the type
  - Base class can be replaced by a subclass, and client code will still be correct

"A method that works on an instance of *type T*, should also work on any *subtype of T*"

Liskov Substitution Principle (LSP)

Let q(x) be a property provable about objects x of type T.
Then q(y) should be provable for objects y of type S where S is a subtype of T.

⇒

Subtype must preserve supertype's invariants
Subtype not allowed to strengthen preconditions
Subtype not allowed to weaken postconditions

Liskov Substitution Principle (LSP)

```
public class Rectangle {
   private int width;
   private int height;

   public void setWidth(int width) {
      this.width = width;
   }

   public void setHeight(int height) {
      this.height = height;
   }

   public int getArea() { return
      width * height;
   }
}
```

```
public class Square extends Rectangle {

   public void setWidth(int width) {
      super.setWidth(width);
      super.setHeight(width);
   }

   public void setHeight(int height) {
      super.setWidth(height);
      super.setHeight(height);
   }
}

void initialize(Rectangle r) {
   r.setWidth(5); r.setHeight(10);
   assert(r.getArea() == 50);
}
```

Liskov Substitution Principle (LSP)

```
public class Rectangle {
   private int width;
   private int height;

   Rectangle(int width, int height) {
      this.height = height;
      this.width = width;
   }

   public int getArea() { return
      width * height;
   }
}
```

```
public class Square extends Rectangle {

   Square(int height, int width) throws
      BadParametersException {
      super(width, height);
      if (height != width) {
            throw new BadParametersException();
      }
   }
}
```

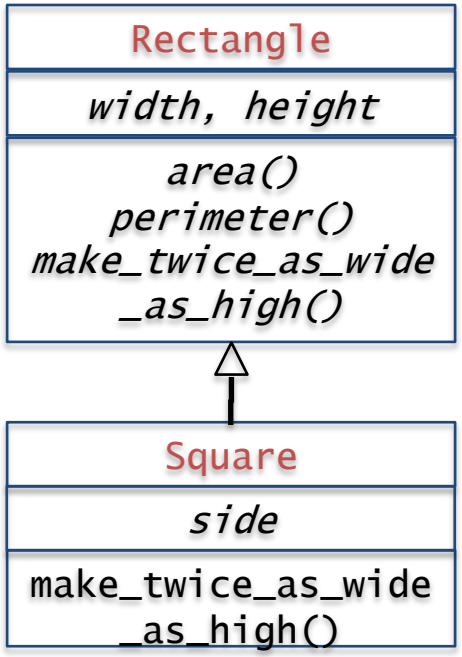
```
Rectangle r = new Rectangle(5,10);
```
✓

```
Square s = new Square(10,10);
```
✓

```
Square s = new Square(5,10);
```
BadParametersException

Contracts

- Composition vs. (misuse of) inheritance
- If can't express consistent assumptions about "contract" between class & collaborators, likely LSP violation



Inheritance

If a subclass won't take advantage of its parent's impl., it might not deserve to be a subclass at all

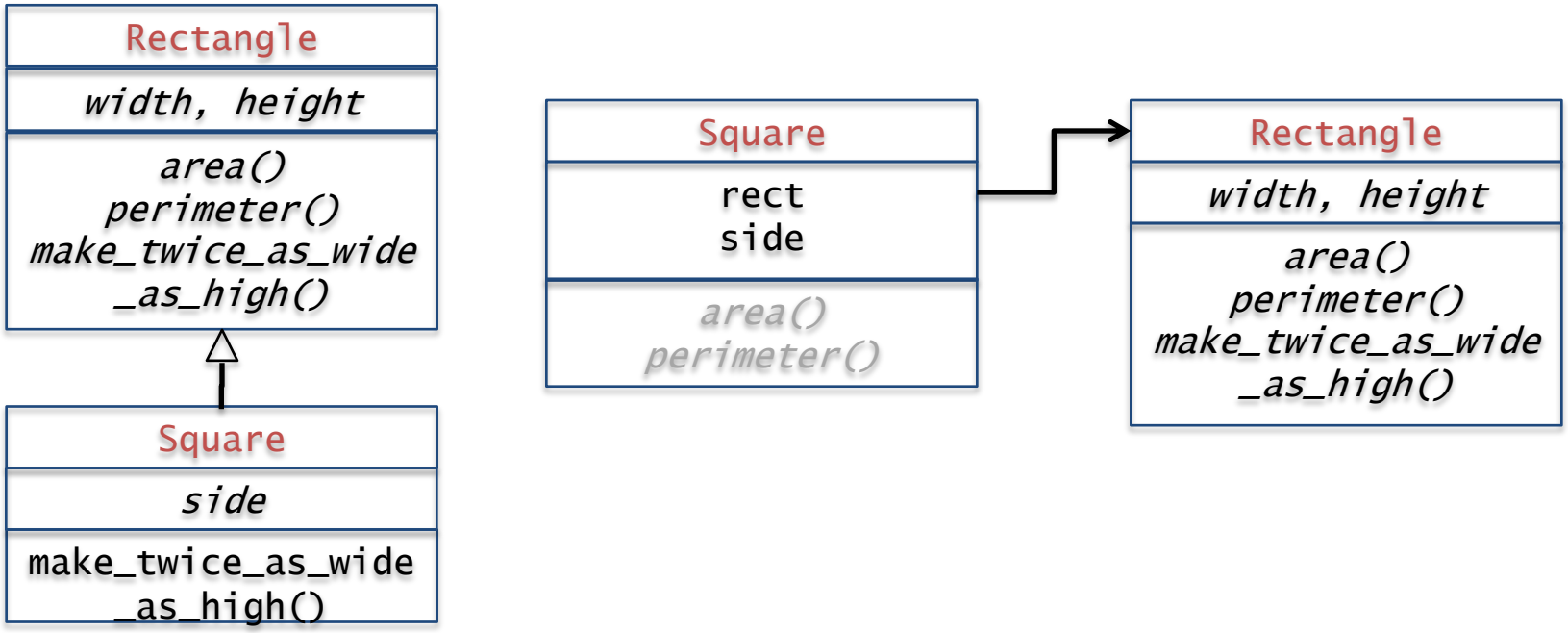
Symptoms

- Subclass destructively overrides a behavior inherited from the superclass

- Forces changes to the superclass to avoid the problem

LSP-Compliant Code

- Composition of classes rather than inheritance, achieving reuse through delegation rather than through subclassing



Design for Inheritance

Modifier	Classes or subclasses in package	Subclasses outside package	Classes outside package
`public`	Yes	Yes	Yes
`protected`	Yes	Yes	No
no modifier	Yes	No	No
`private`	No	No	No

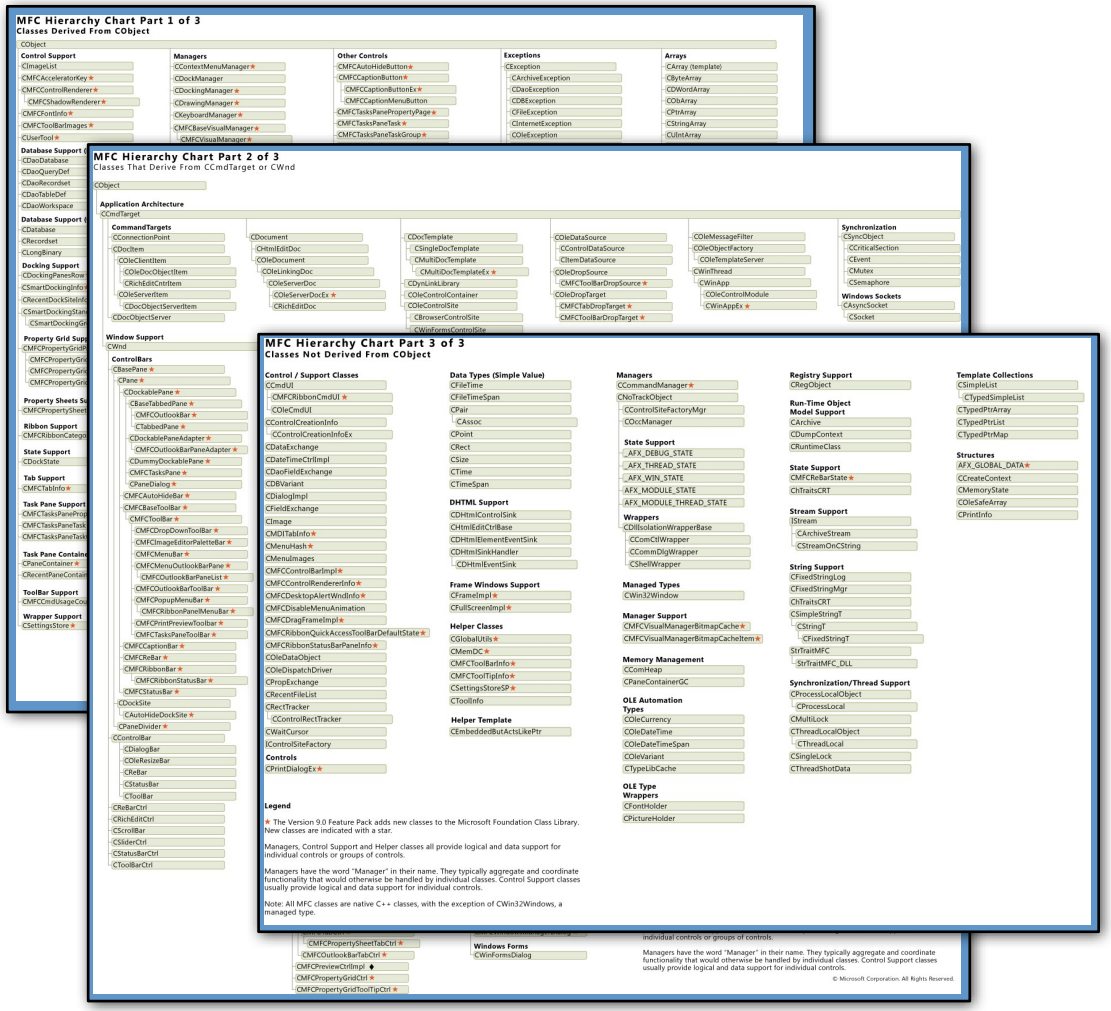
- Document overriding, or prohibit it
  - *use* `final` *for methods you don't want overridden*
  - *remember access rules for attributes and methods*
  - *use* `final` *class to prevent it being subclassed and force composition instead*

Inheritance and Encapsulation

```
public class CharacterSet {
   protected StringBuffer s;
   // invariant: s != null
   ...
}
public class PersistentCharacterSet extends CharacterSet {
   public loadFromFile(String name) {
      try {
         // load contents ...
      } catch (IOException e) {
         s = null;
      }
   }
}
```

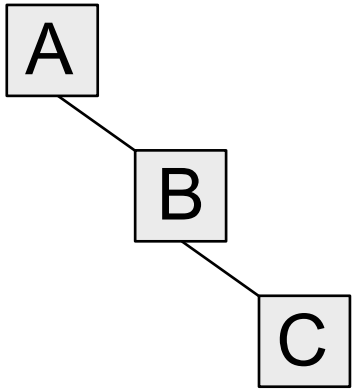
- Inheritance breaks encapsulation
  - In turn, this increases complexity and coupling
- Favor **private** over protected
  - Prevent subclass from violating invariants
  - If data access is necessary, use protected getter/setter methods that can protect invariants

Inheritance Hierarchies



- Avoid deep hierarchies
  - Max 3 levels of inheritance, max 7+-2 subclasses
  - Deep inheritance trees produce higher bug rates

Inheritance Hierarchies



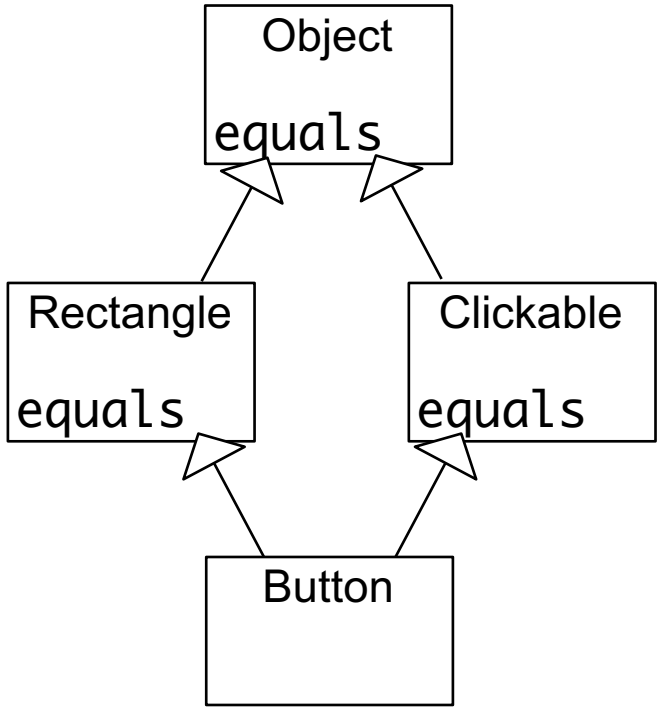
- Avoid deep hierarchies
  - Max 3 levels of inheritance, max 7+-2 subclasses
  - Deep inheritance trees produce higher bug rates
- Avoid linear hierarchies
  - Single derived class = warning sign for mistaken “designing ahead”
  - It’s better to design easy-to-change classes, and refactor later if needed

Inheritance Hierarchies

- Avoid deep hierarchies
  - Max 3 levels of inheritance, max 7+-2 subclasses
  - Deep inheritance trees produce higher bug rates
- Avoid linear hierarchies
  - Single derived class = warning sign for mistaken “designing ahead”
  - It’s better to design easy-to-change classes, and refactor later if needed
- Push common interfaces, data, and behavior as high up as possible

Multiple Inheritance

```
class Button extends Rectangle, Clickable {
   // ...
}
// ...
if (button.equals(obj))
// ...
```



- Hardly ever a good reason to do it
  - Even if your language allows it, avoid multiple inheritance
- Example problem
  - The “Diamond problem”

Interface Inheritance

```
public class JButton extends AbstractButton
implements Accessible, ImageObserver, ItemSelectable, MenuContainer,
Serializable, SwingConstants {
   // ...
}
```

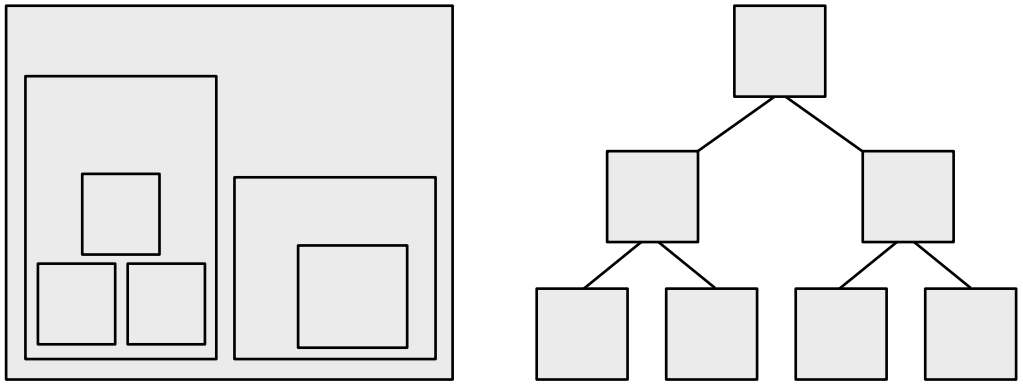
- Multiple inherited interfaces
  - E.g., in Java, C#
  - Only abstract methods, no implementation or fields

Mixins

```
abstract class AbsIterator {
 type T
 def hasNext: Boolean
 def next: T
}
trait RichIterator extends AbsIterator {
 def foreach(f: T => Unit) { while (hasNext) f(next) }
}

class StringIterator(s: String) extends AbsIterator {
 type T = Char
 private var i = 0
 def hasNext = i < s.length()
 def next = { val ch = s charAt i; i += 1; ch }
}
object StringIteratorTest {
 def main(args: Array[String]) {
  class Iter extends StringIterator(args(0)) with RichIterator
  val iter = new Iter
  iter foreach println
 }
}
```

- E.g., in Scala, Python, Perl, Ruby
- Inheritance of implementations from multiple mixins is allowed
- Orthogonal functionality, single purpose, not instantiable on their own

Containment vs. Inheritance

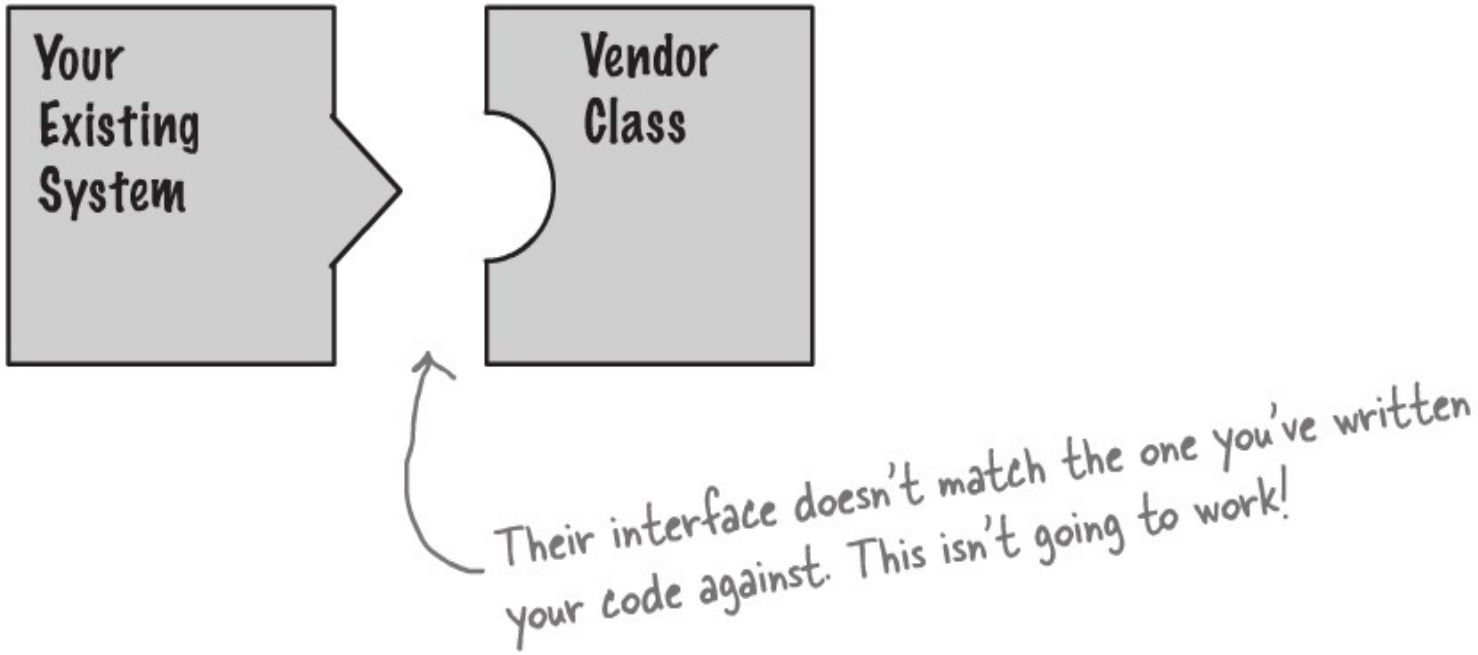


- Containment
  - Use when classes share common data but not behavior
  - Containing class controls the interface
  - Avoid excessive method forwarding
- Inheritance
  - Use if multiple classes share common behavior
  - Avoid if it violates the Liskov Substitution Principle
  - Only inherit what is truly shared
  - Base class controls interface and provides implementation

Adapter

Problem

- Class has different interface from what the caller/user expects



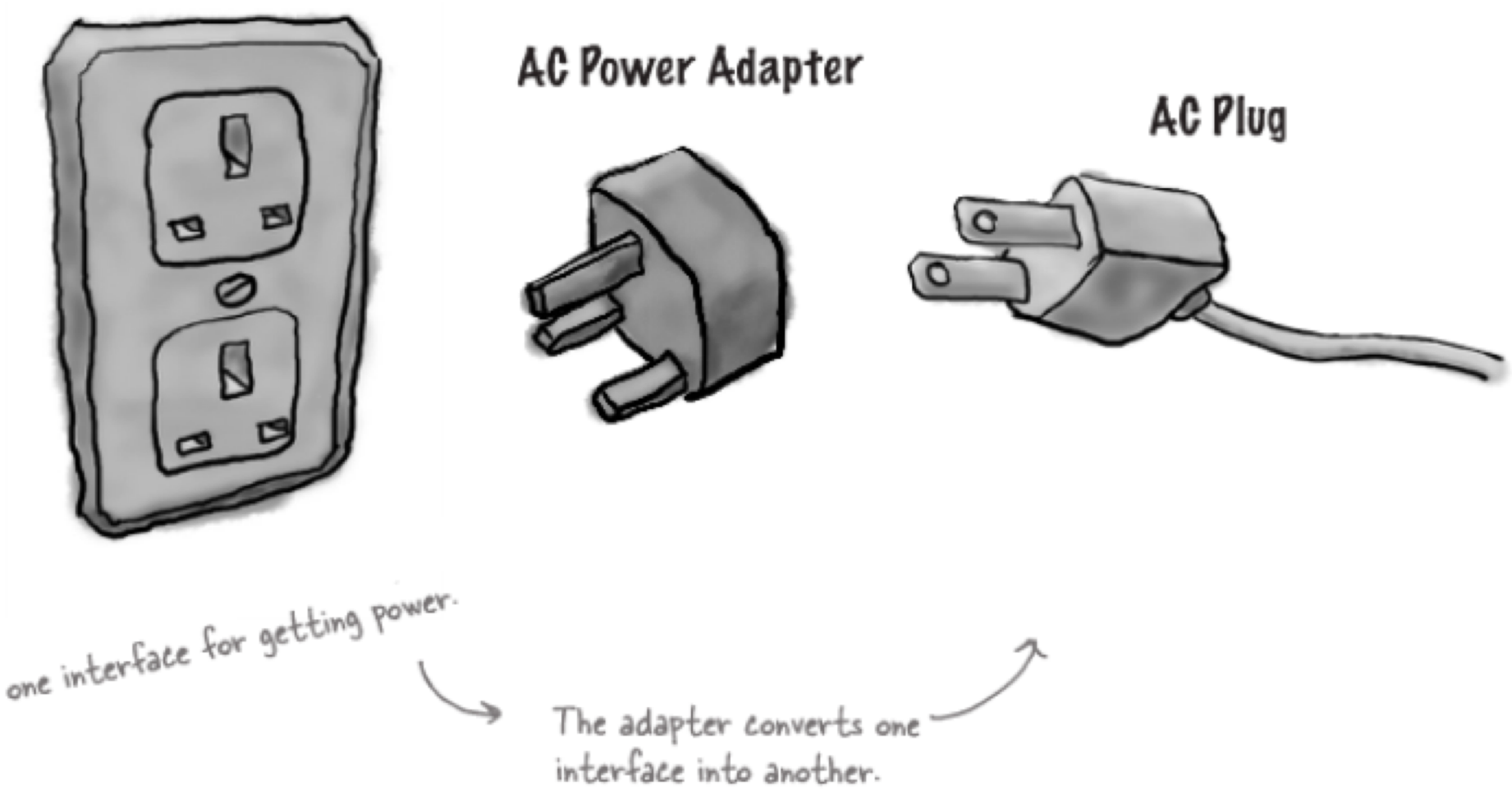
```
class Rectangle
{
    // grow or shrink rectangle by the given factor
    void setScale (float factor);

    // other operations
    float getArea();
    float getCircumference();
    ...
}
```

```
class NonScaleableRectangle
{
   void setWidth( float width ) { ... }
   void setHeight( float height ) { ... }
   ...
}
```

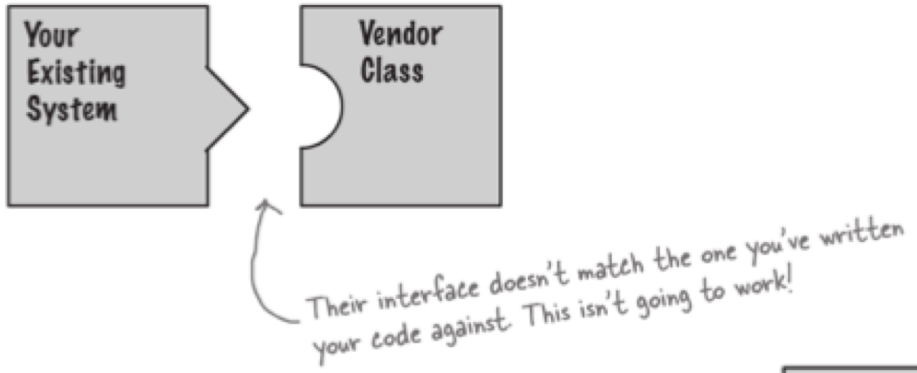
```
class myClass
{
    void myMethod (Rectangle rect) {
        ...
        rect.setScale(2);
        ...
    }
}
```

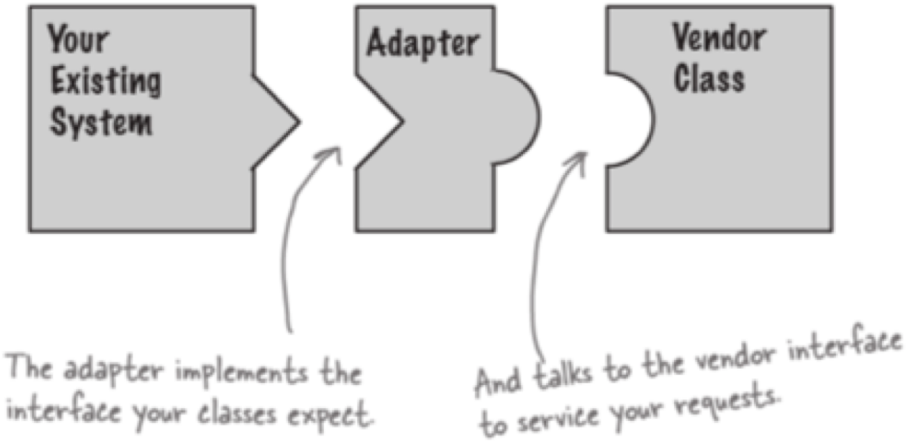
Our existing system (have only the .class, so you can't modify it).

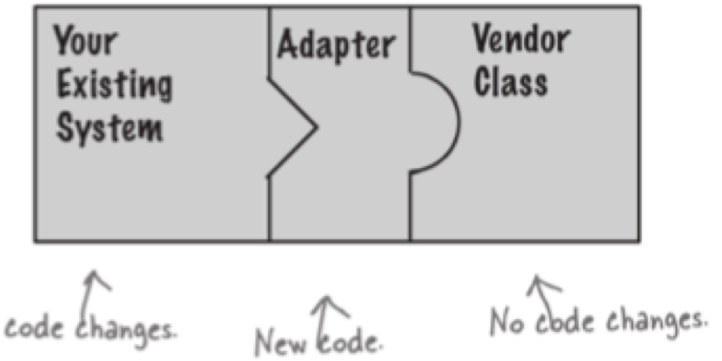
Solution Template



Solution Template







```
public interface Duck {
    public void quack();
    public void fly();
}

public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

Simple implementations: the duck just prints out what it is doing

Here's a concrete implementation of Turkey; like Duck, it just prints out its actions

Turkeys don't quack, they gobble.

```
public interface Turkey {
    public void gobble();
    public void fly();
}
```

Turkeys can fly, although they can only fly short distances

```
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }
    public void fly() {
        System.out.println("I'm flying a short
 distance");
    }
}
```

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i<5; i++) {
            turkey.fly();
        }
    }
}
```

First, you need to implement the interface of the type you're adapting to. This is the interface your client expects to see.

Next, we need to get a reference to the object that we are adapting; here we do that through the constructor.

Now we need to implement all the methods in the interface; the quack() translation between classes is easy: just call the gobble() method.

Even though both interfaces have a fly() method, Turkeys fly in short spurts – they can't do long-distance flying like ducks. To map between a Duck's fly() method and a Turkey's, we need to call the Turkey's fly() method five times to make up for it.

```
public class DuckTestDrive {
    public static void main(String[] args) {
        MallardDuck duck = new MallardDuck();

        WildTurkey turkey = new WildTurkey();
        Duck turkeyAdapter = new TurkeyAdapter(turkey);

        System.out.println("The Turkey says…");
        turkey.gobble();
        turkey.fly();

        System.out.println("\nThe Duck says…");
        testDuck(duck);

        System.out.println("\nThe TurkeyAdapter says…");
        testDuck(turkeyAdapter);
    }

    static void testDuck(Duck duck) {
        duck.quack();
        duck.fly();
    }
}
```

Let's create a Duck

and a Turkey

And then wrap the turkey in a TurkeyAdapter, which makes it look like a Duck

Then, let's test the Turkey: make it gobble, make it fly

Now let's test the duck by calling the testDuck() method, which expects a Duck object

Now the big test: we try to pass off the turkey as a duck

Here's our testDuck() method; it gets a duck and class its quack() and fly() methods

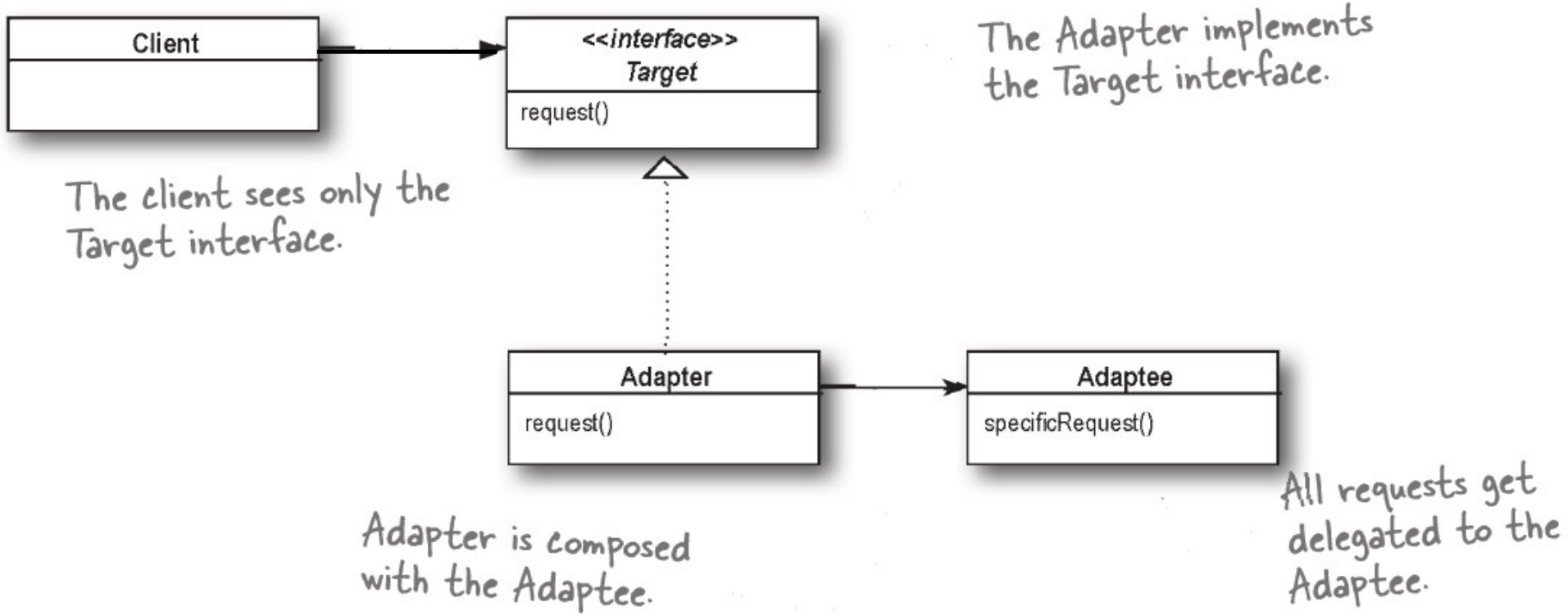
```
%java DuckTestDrive

The Turkey says...

Gobble gobble

I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

The Turkey gobbles and flies a short distance.

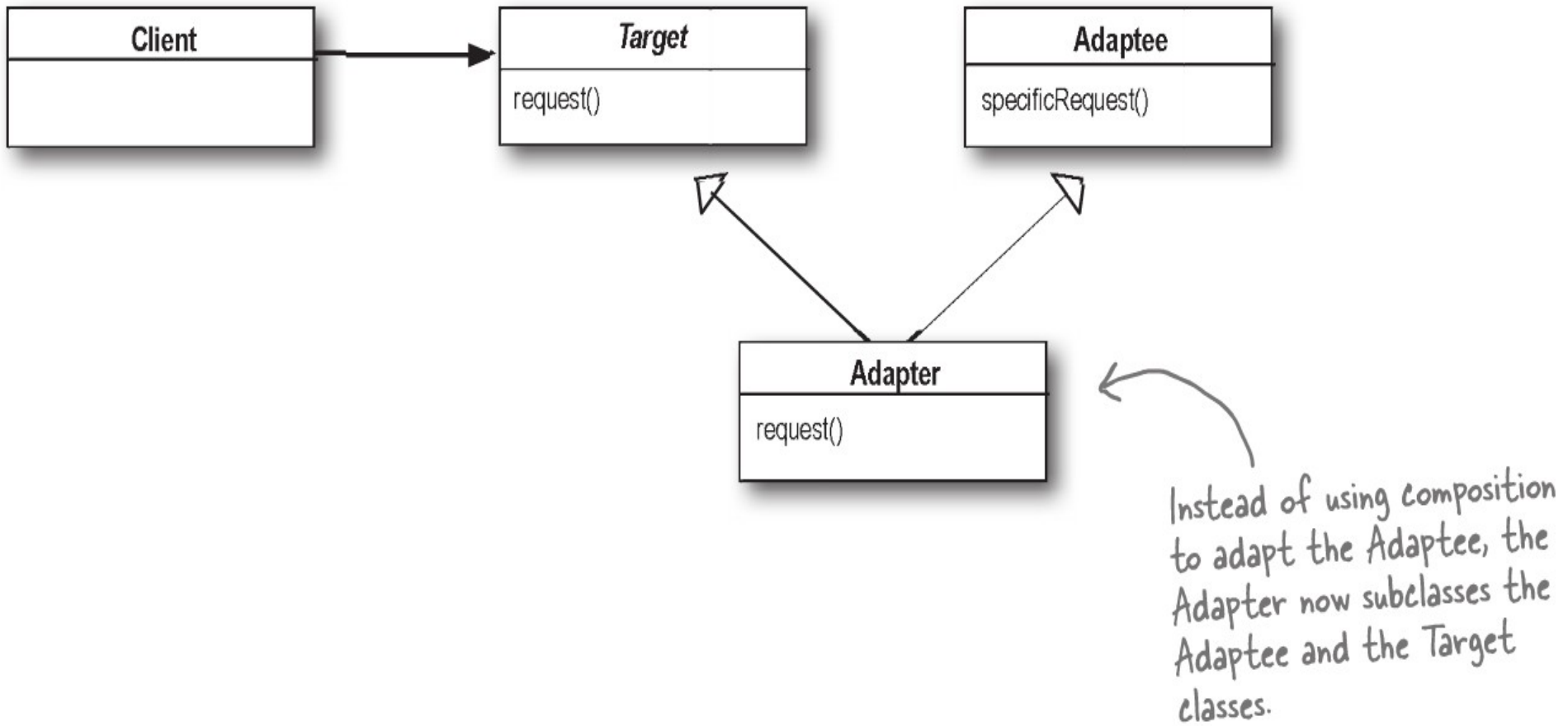
The Duck quacks and flies just like you'd expect.

And the adapter gobbles when quack() is called and flies a few times when fly() is called. The testDuck() method never knows it has a turkey disguised as a duck!

Object Adapter



Class Adapter



Tips & Tricks

- Despite its overhead, using an Adapter is often useful
- One Adapter can wrap multiple Adaptees
- Can develop two-way Adapters
  - Implement both the Target and the Adaptee interfaces
- Highly disciplined naming convention
  - <Adaptee>To<Target>Adapter (e.g., TurkeyToDuckAdapter)
- Spelling
  - Use Adapter, not Adaptor (but don't pick a fight over it)

Façade

Façade Pattern

- When the Adapter pattern not only converts an existing API but also simplifies it
- E.g., YourSpace provides many other YourSpace functions unrelated to email, but YourSpaceAdapter only adapts the email-specific part of that API, it’s sometimes called the Façade pattern.

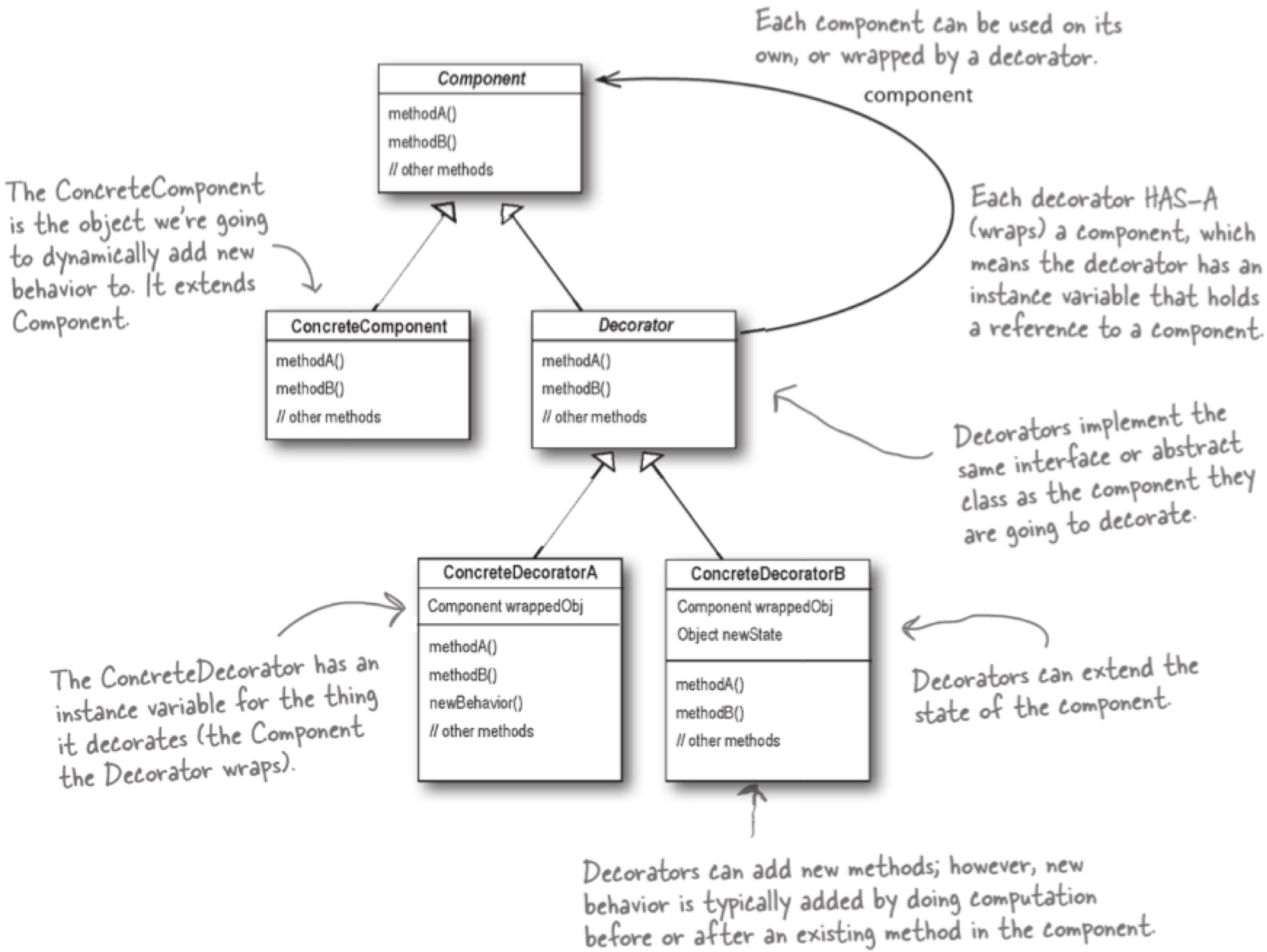
Decorator

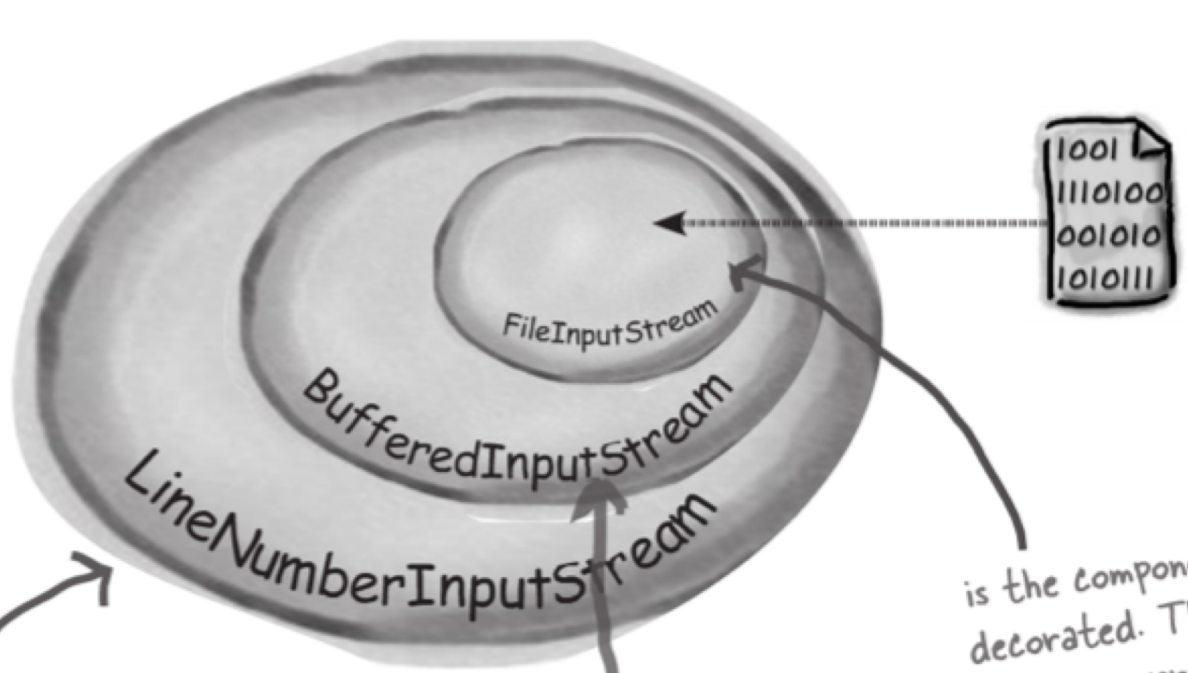
Problem

- Problem = augment functionality
  - Without changing the code that uses the class
  - Without using inheritance

Solution Template

- Keep interface the same
- Dynamically add/override behaviors underneath same interface

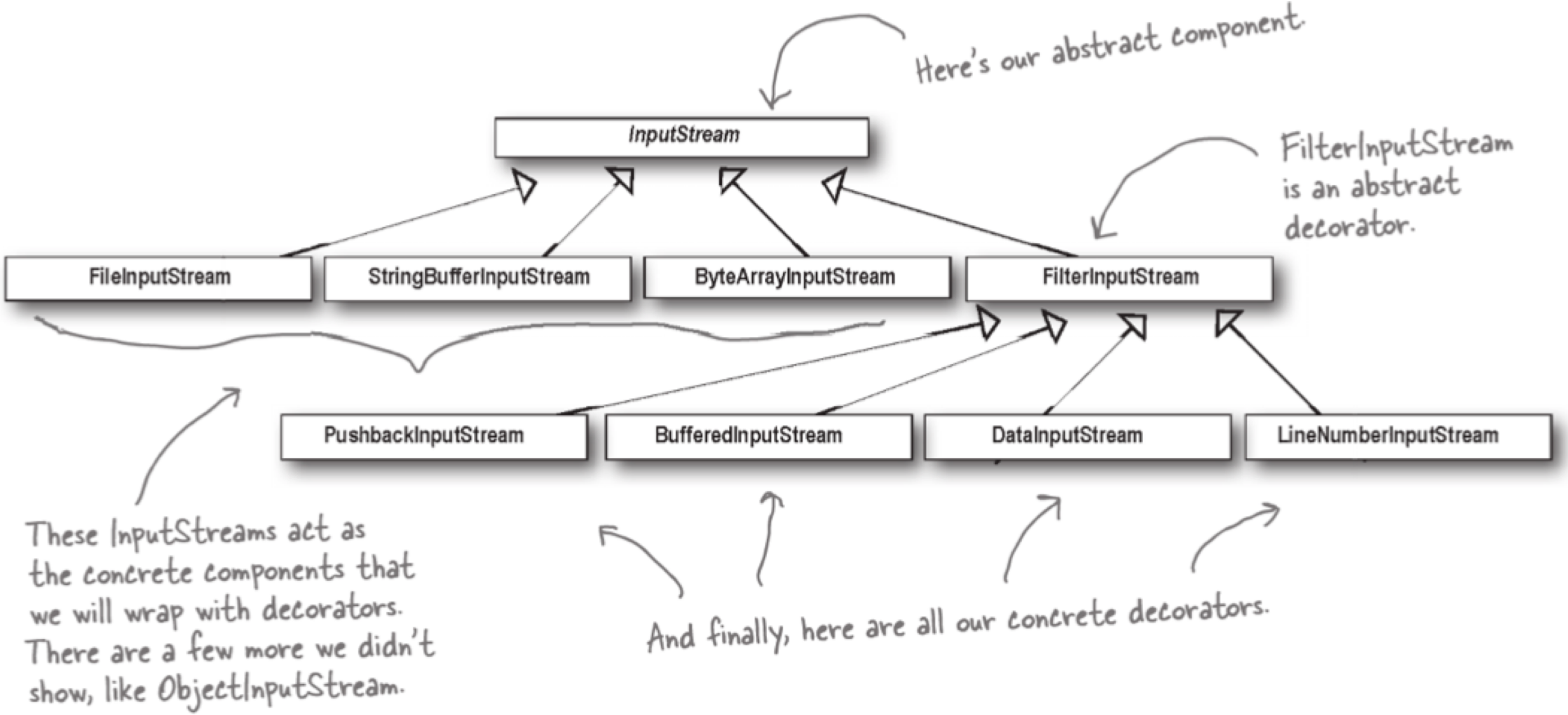

Each component can be used on its own, or wrapped by a decorator.
component
Component
methodA()
methodB()
// other methods
The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.
ConcreteComponent
methodA()
methodB()
// other methods
Decorator
methodA()
methodB()
// other methods
Each decorator HAS-A (wraps) a component, which means the decorator has an instance variable that holds a reference to a component.
Decorators implement the same interface or abstract class as the component they are going to decorate.
ConcreteDecoratorA
Component wrappedObj
methodA()
methodB()
newBehavior()
// other methods
The ConcreteDecorator has an instance variable for the thing it decorates (the Component the Decorator wraps).
ConcreteDecoratorB
Component wrappedObj
Object newState
methodA()
methodB()
// other methods
Decorators can extend the state of the component.
Decorators can add new methods; however, new behavior is typically added by doing computation before or after an existing method in the component.



FileInputStream is the component that's being decorated. The Java I/O library supplies several components, including FileInputStream, StringBufferInputStream, ByteArrayInputStream and a few others. All of these give us a base component from which to read bytes.

BufferedInputStream is a concrete decorator. BufferedInputStream adds behavior in two ways: it buffers input to improve performance, and also augments the interface with a new method readLine() for reading character-based input, a line at a time.

LineNumberInputStream is also a concrete decorator. It adds the ability to count the line numbers as it reads data.


Here's our abstract component.
InputStream
FilterInputStream is an abstract decorator.
FileInputStream
StringBufferInputStream
ByteArrayInputStream
FilterInputStream
PushbackInputStream
BufferedInputStream
DataInputStream
LineNumberInputStream
These InputStreams act as the concrete components that we will wrap with decorators. There are a few more we didn't show, like ObjectInputStream.
And finally, here are all our concrete decorators.

First, extend the FilterInputStream, the abstract decorator for all InputStreams.

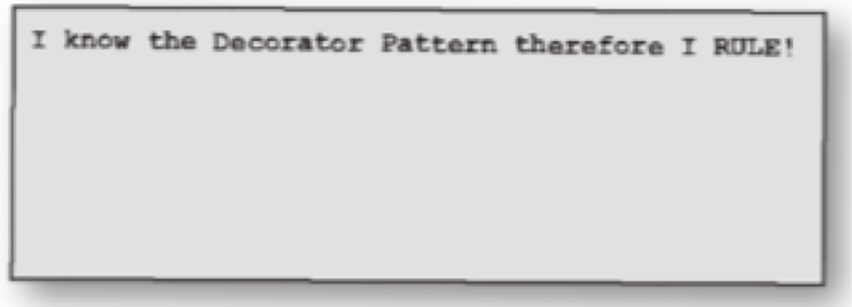
```
public class LowerCaseInputStream extends FilterInputStream {

    public LowerCaseInputStream(InputStream in) {
        super(in);
    }

    public int read() throws IOException {
        int c = super.read();
        return (c == -1 ? c : Character.toLowerCase((char)c));
    }

    public int read(byte[] b, int offset, int len) throws IOException {
        int result = super.read(b, offset, len);
        for (int i = offset; i < offset+result; i++) {
            b[i] = (byte)Character.toLowerCase((char)b[i]);
        }
        return result;
    }
}
```

Now we need to implement two read methods. They take a byte (or an array of bytes) and convert each byte (that represents a character) to lowercase if it's an uppercase character.

```
public class InputTest {
    public static void main(String[] args) throws IOException {
        int c;

        try {
            InputStream in =
                new LowerCaseInputStream(
                    new BufferedInputStream(
                        new FileInputStream("test.txt")));

            while((c = in.read()) >= 0) {
                System.out.print((char)c);
            }

            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Set up the FileInputStream and decorate it, first with a BufferedInputStream and then our brand new LowerCaseInputStream filter.

Just use the stream to read characters until the end of file and print as we go.




```
% java InputTest
i know the decorator pattern therefore i rule!
%
```

```
start_time = int(round(time.time() * 1000))
employees = Employee.get_all_employee_details()
time_diff = current_milli_time() — start_time
debug_log_time_diff.update({'FETCH_TIME': time_diff})
```

```
employees = Employee.get_all_employee_details()
```

```
start_time = int(round(time.time() * 1000))
employees = Employee.get_all_employee_details()
time_diff = current_milli_time() — start_time
debug_log_time_diff.update({'FETCH_TIME': time_diff})

employees = Employee.get_all_employee_details()
```

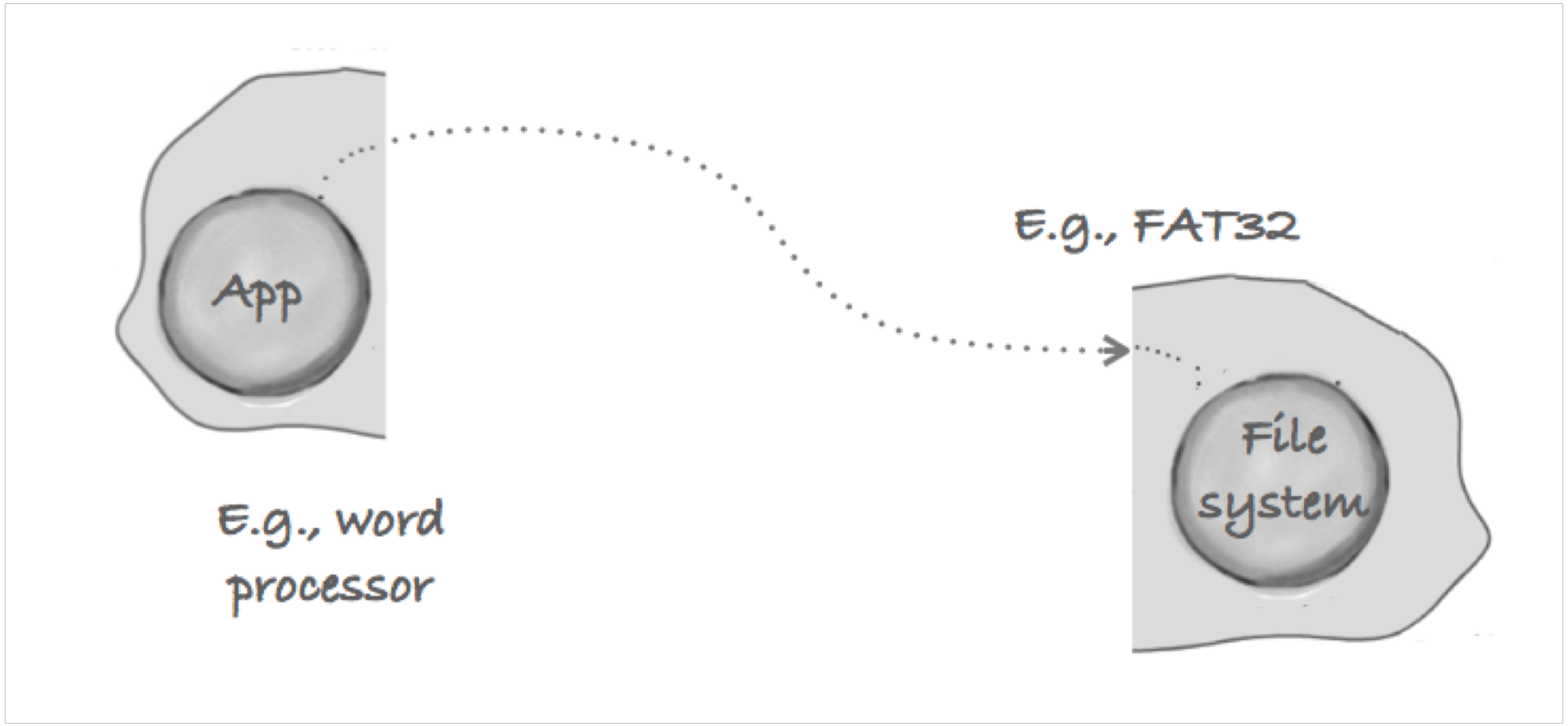
```
def timeit(method):
    def timed(*args, **kw):
        ts = time.time()
        result = method(*args, **kw)
        te = time.time()

        if 'log_time' in kw:
            name = kw.get('log_name', method.__name__.upper())
            kw['log_time'][name] = int((te - ts) * 1000)
        else:
            print '%r %2.2f ms' % \
                  (method.__name__, (te - ts) * 1000)
        return result

    return timed
```

```
@timeit
def get_all_employee_details(**kwargs):
    print 'employee details'
```

Caveats, Tips, and Tricks

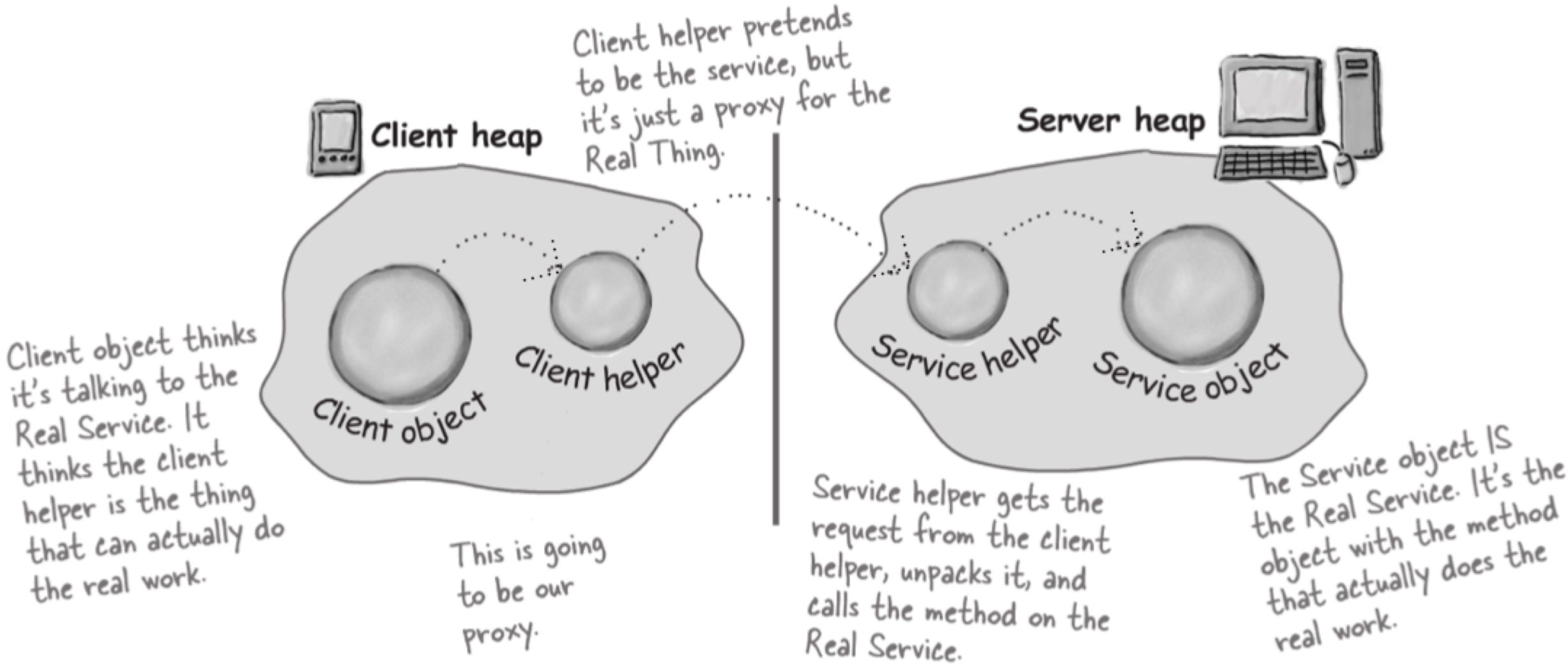
- Decorator pattern can lead to large number of small classes
- May not be able to tell Decorators apart => use Factories
- Instantiating the object is more complex => use Factories
- Peeking at wrapped Decorators is OK, but tread carefully
- Beware of introducing typing problems
- Can use Decorator to augment specific instances of an object

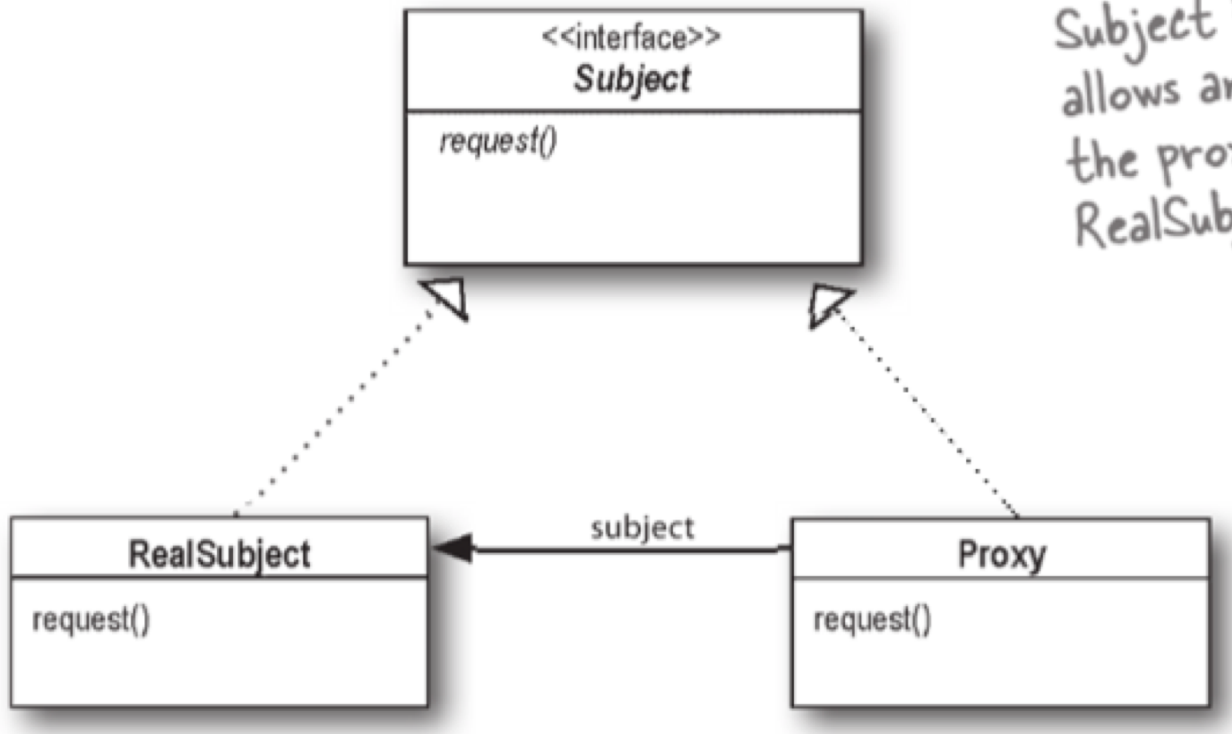
Proxy


App
E.g., word processor
E.g., FAT32
File system

Solution Template

- Create a surrogate object
  - Same interface as the “server” object (the file system)
  - Provides same functionality to “client”
  - Additionally mediates/controls access to “server”
- Client code need not change
  - Can employ a proxy on the server as well to avoid changing it


Client helper pretends to be the service, but it's just a proxy for the Real Thing.
Client heap
Server heap
Client object thinks it's talking to the Real Service. It thinks the client helper is the thing that can actually do the real work.
Client object
Client helper
This is going to be our proxy.
Service helper
Service object
Service helper gets the request from the client helper, unpacks it, and calls the method on the Real Service.
The Service object IS the Real Service. It's the object with the method that actually does the real work.



Both the Proxy and the RealSubject implement the Subject interface. This allows any client to treat the proxy just like the RealSubject.

The RealSubject is usually the object that does most of the real work; the Proxy controls access to it.

The Proxy often instantiates or handles the creation of the RealSubject.

The Proxy keeps a reference to the RealSubject so it can forward requests to the RealSubject when necessary.

Purposes of Proxy

- Isolate complexity
  - E.g., access resources over the network vs. locally
- Control access
  - E.g., locking/unlocking a shared resource, checking access permissions / policy
- Control behavior
  - E.g., fault injection proxies for testing robustness
- Improve performance
  - E.g., caching layer to speed up access

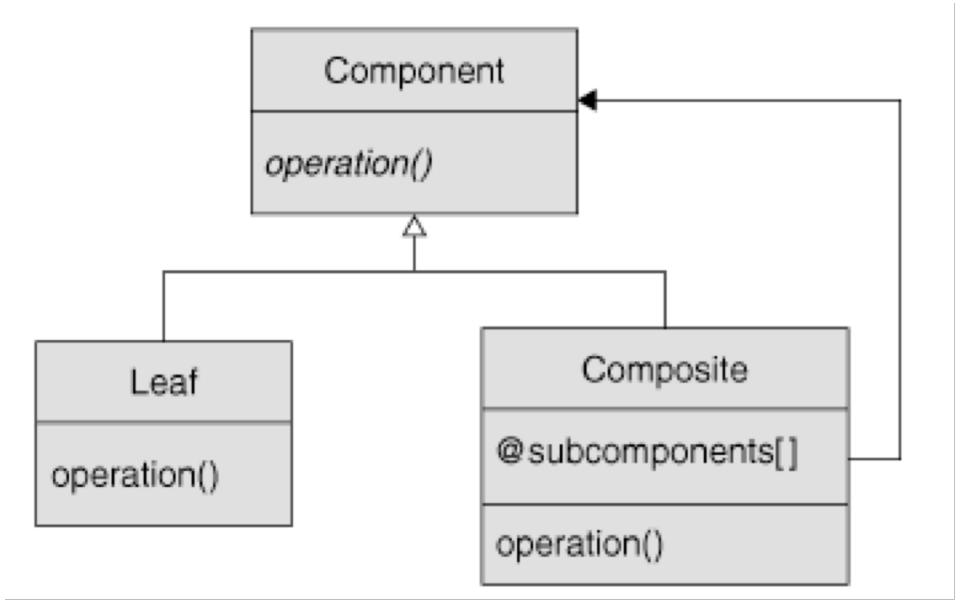
Tips & Tricks

- Proxies increase the number of classes
- Factories are often useful to return objects wrapped in proxies
- Java supports proxies directly in java.lang.reflect.
  - Since Java 5, remote proxies are easier to do
- Can overlap with other structural design patterns
  - Decorator looks similar, but it actually adds behavior (unlike Proxy)
  - Adapter also forwards requests to the “original”, but changes the interface

Composite

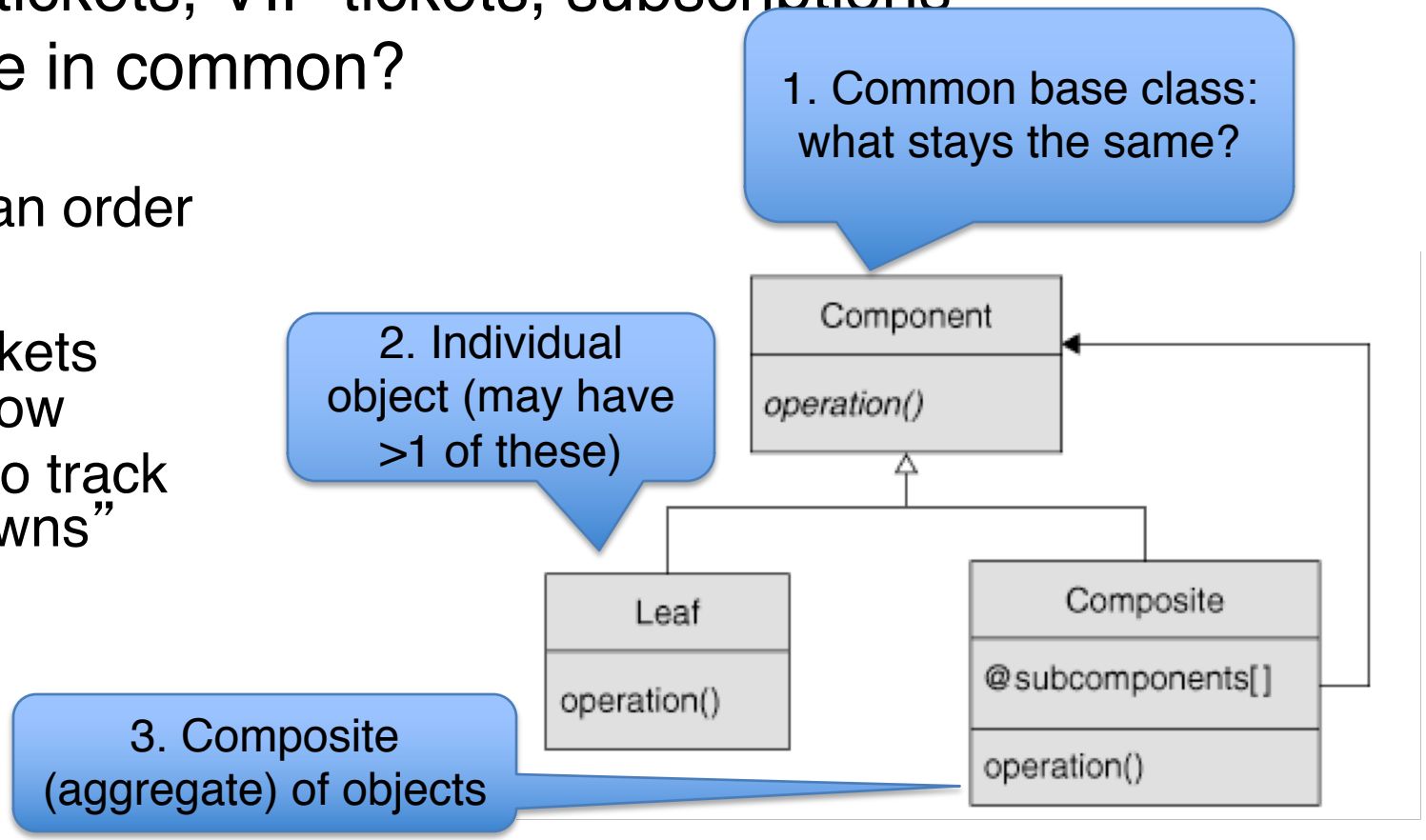
Composite

- What: component whose operations make sense on both individuals & aggregates
- Example: regular tickets, VIP tickets, subscriptions
- What do they have in common?
  - Has a price
  - Can be added to an order
- What's different?
  - Regular & VIP Tickets are for specific show
  - Subscription has to track which tickets it "owns"



Composite

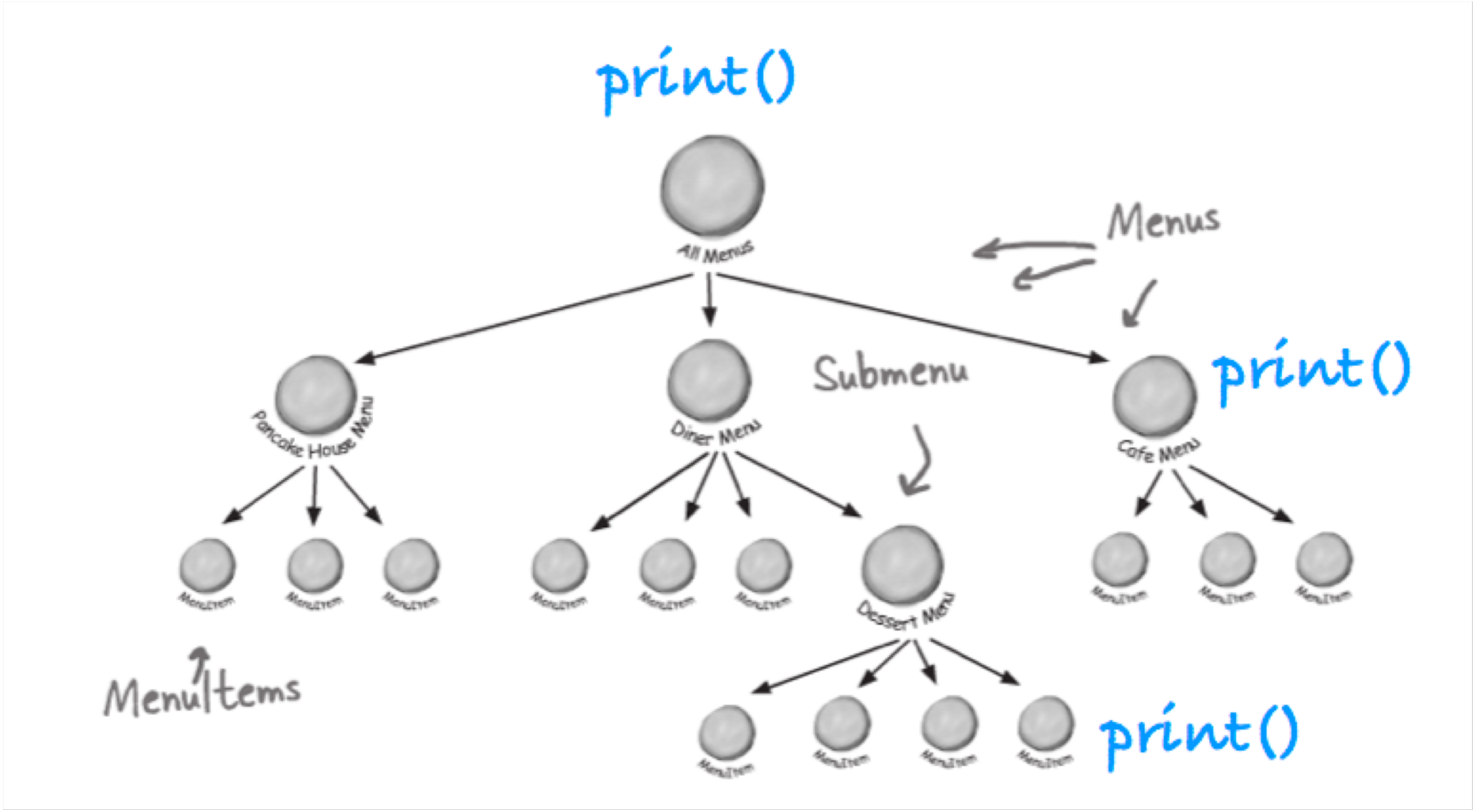
- What: component whose operations make sense on both individuals & aggregates
- Example: regular tickets, VIP tickets, subscriptions
- What do they have in common?
  - Has a price
  - Can be added to an order
- What's different?
  - Regular & VIP Tickets are for specific show
  - Subscription has to track which tickets it "owns"



Composite

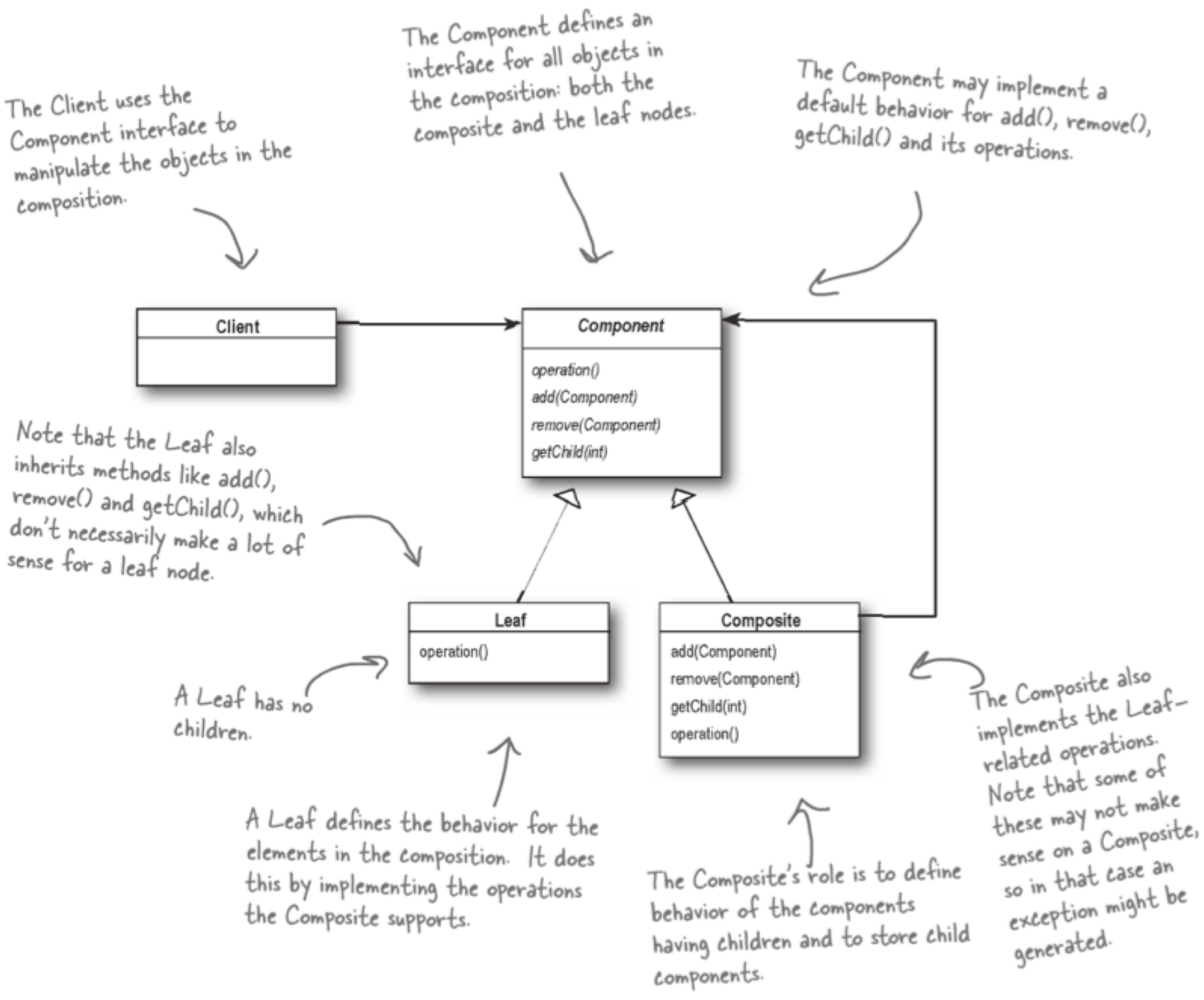
- Compose objects into tree structure to represent part-whole hierarchies.
- Composite lets client treat individual objects and compositions of objects uniformly
- Composite design pattern treats each node in two ways- Composite or leaf.
  - Composite means it can have other objects below it.
  - Leaf means it has no objects below it.

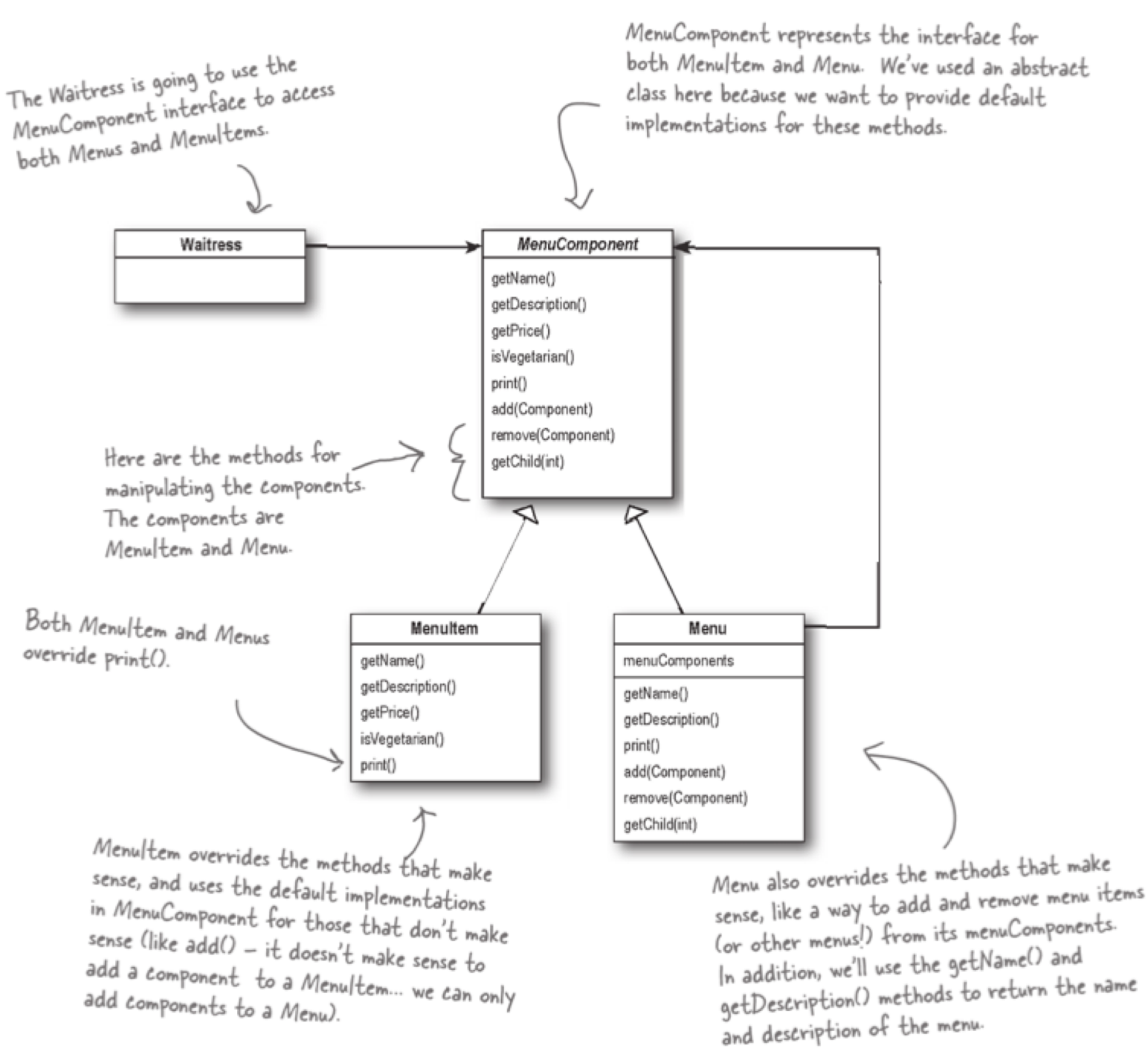
Problem


print()
All Menus
Menus
Pancake House Menu
Diner Menu
Submenu
Cafe Menu
print()
MenuItem
Dessert Menu
MenuItems
print()

Solution Template

- Compose one or more objects behind the same interface
  - Goal: operate on composite as if it was a fundamental type
  - This enables us to manipulate them all in the same way
- Black-box reuse
- Objects must be similar, and exhibit similar functionality


The Client uses the Component interface to manipulate the objects in the composition.
The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.
The Component may implement a default behavior for add(), remove(), getChild() and its operations.
Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Note that the Leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node.
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()
A Leaf has no children.
A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.
The Composite's role is to define behavior of the components having children and to store child components.
The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.


The Waitress is going to use the MenuComponent interface to access both Menus and MenuItems.
MenuComponent represents the interface for both MenuItem and Menu. We've used an abstract class here because we want to provide default implementations for these methods.
Waitress
MenuComponent
getName()
getDescription()
getPrice()
isVegetarian()
print()
add(Component)
remove(Component)
getChild(int)
Here are the methods for manipulating the components. The components are MenuItem and Menu.
Both MenuItem and Menus override print().
MenuItem
getName()
getDescription()
getPrice()
isVegetarian()
print()
Menu
menuComponents
getName()
getDescription()
print()
add(Component)
remove(Component)
getChild(int)
MenuItem overrides the methods that make sense, and uses the default implementations in MenuComponent for those that don't make sense (like add() – it doesn't make sense to add a component to a MenuItem... we can only add components to a Menu).
Menu also overrides the methods that make sense, like a way to add and remove menu items (or other menus!) from its menuComponents. In addition, we'll use the getName() and getDescription() methods to return the name and description of the menu.

MenuComponent provides default implementations for every method.

```
public abstract class MenuComponent {

    public void add(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public void remove(MenuComponent menuComponent) {
        throw new UnsupportedOperationException();
    }
    public MenuComponent getChild(int i) {
        throw new UnsupportedOperationException();
    }

    public String getName() {
        throw new UnsupportedOperationException();
    }
    public String getDescription() {
        throw new UnsupportedOperationException();
    }
    public double getPrice() {
        throw new UnsupportedOperationException();
    }
    public boolean isVegetarian() {
        throw new UnsupportedOperationException();
    }

    public void print() {
        throw new UnsupportedOperationException();
    }
}
```

Because some of these methods only make sense for MenuItems, and some only make sense for Menus, the default implementation is UnsupportedOperationException. That way, if MenuItem or Menu doesn't support an operation, they don't have to do anything, they can just inherit the default implementation.

We've grouped together the "composite" methods – that is, methods to add, remove and get MenuComponents.

Here are the "operation" methods; these are used by the MenuItems. It turns out we can also use a couple of them in Menu

print() is an "operation" method that both our Menus and MenuItems will implement, but we provide a default operation here.

```
public class MenuItem extends MenuComponent {
    String name;
    String description;
    boolean vegetarian;
    double price;

    public MenuItem(String name,
                    String description,
                    boolean vegetarian,
                    double price)
    {
        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }

    public double getPrice() {
        return price;
    }

    public boolean isVegetarian() {
        return vegetarian;
    }

    public void print() {
        System.out.print("  " + getName());
        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println("     -- " + getDescription());
    }
}
```

First we need to extend the MenuComponent interface.

The constructor just takes the name, description, etc. and keeps a reference to them all.

Here we're overriding the print() method in the MenuComponent class. For MenuItem this method prints the complete menu entry: name, description, price and whether or not it's veggie.

Menu is also a MenuComponent, just like MenuItem.

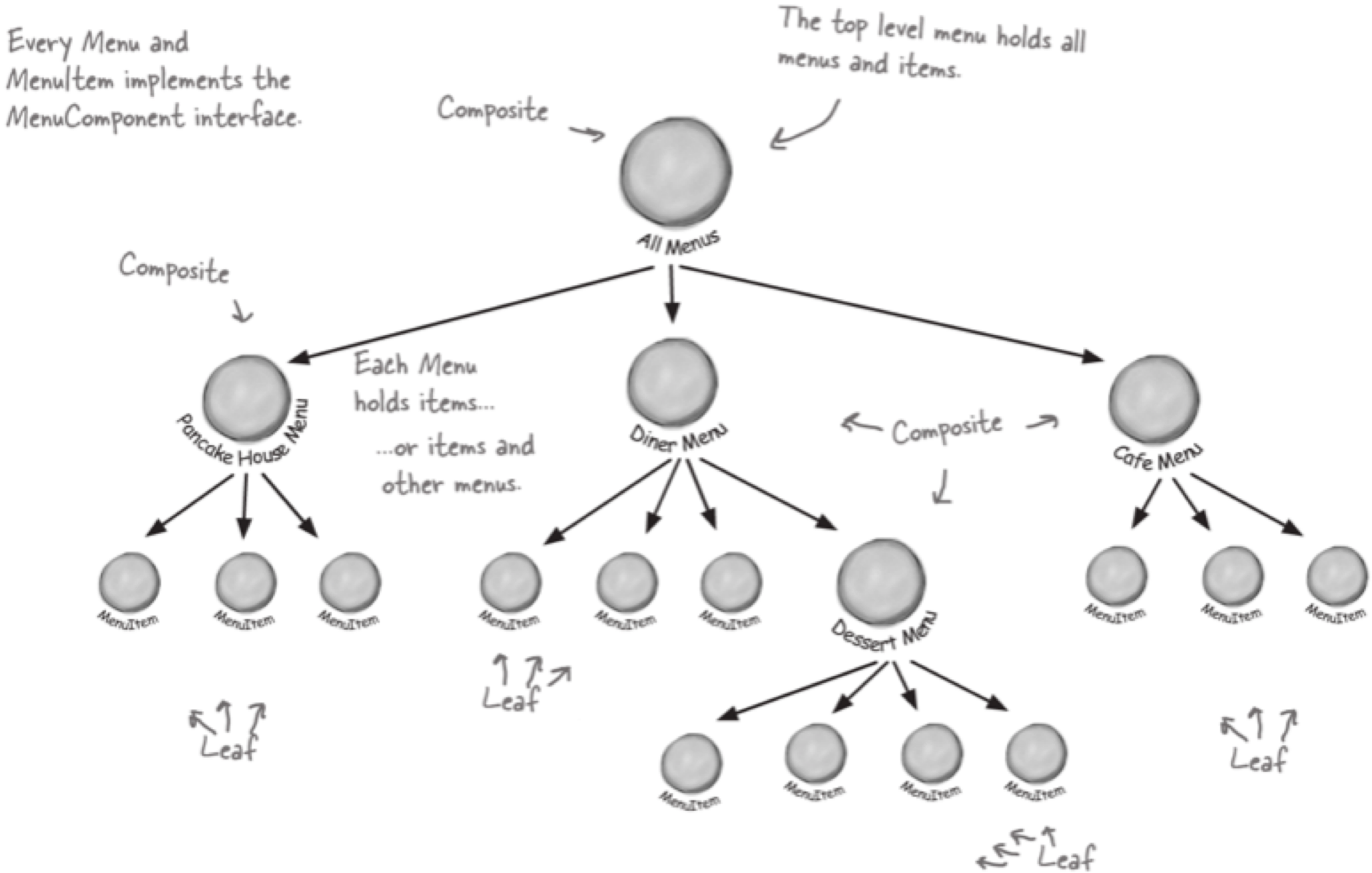
Menu can have any number of children of type MenuComponent, we'll use an internal ArrayList to hold these.

```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }

    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent);
    }

    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }

    public MenuComponent getChild(int i) {
        return (MenuComponent)menuComponents.get(i);
    }

    public String getName() {
        return name;
    }

    public String getDescription() {
        return description;
    }



    // ... print() comes later ...

}
```

Here's how you add MenuItems or other Menus to a Menu. Because both MenuItems and Menus are MenuComponents, we just need one method to do both.

You can also remove a MenuComponent or get a MenuComponent.

Here are the getter methods for getting the name and description.


Every Menu and MenuItem implements the MenuComponent interface.
The top level menu holds all menus and items.
Composite
All Menus
Composite
Pancake House Menu
Each Menu holds items...
...or items and other menus.
Diner Menu
Composite
Cafe Menu
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
Dessert Menu
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
MenuItem
Leaf
Leaf
Leaf
Leaf

```
public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // constructor code here

    // other methods here

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("---------------------");

        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}
```

iterate through all the Menu's components... those could be other Menus, or they could be MenuItems. Since both Menus and MenuItems implement print(), we just call print() and the rest is up to them.

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.

```
public class Waitress {
    MenuComponent allMenus;

    public Waitress(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }

    public void printMenu() {
        allMenus.print();
    }
}
```

Yup! The Waitress code really is this simple. Now we just hand her the top level menu component, the one that contains all the other menus. We've called that allMenus.

All she has to do to print the entire menu hierarchy – all the menus, and all the menu items – is call print() on the top level menu.

```
public class MenuTestDrive {
    public static void main(String args[]) {
        MenuComponent pancakeHouseMenu =
            new Menu("PANCAKE HOUSE MENU", "Breakfast");
        MenuComponent dinerMenu =
            new Menu("DINER MENU", "Lunch");
        MenuComponent cafeMenu =
            new Menu("CAFE MENU", "Dinner");
        MenuComponent dessertMenu =
            new Menu("DESSERT MENU", "Dessert of course!");

        MenuComponent allMenus = new Menu("ALL MENUS", "All menus combined");

        allMenus.add(pancakeHouseMenu);
        allMenus.add(dinerMenu);
        allMenus.add(cafeMenu);

        // add menu items here

        dinerMenu.add(new MenuItem(
            "Pasta",
            "Spaghetti with Marinara Sauce, and a slice of sourdough bread",
            true,
            3.89));

        dinerMenu.add(dessertMenu);

        dessertMenu.add(new MenuItem(
            "Apple Pie",
            "Apple pie with a flakey crust, topped with vanilla ice cream",
            true,
            1.59));

        // add more menu items here

        Waitress waitress = new Waitress(allMenus);

        waitress.printMenu();
    }
}
```

Let's first create all the menu objects.

We also need a top level menu that we'll name allMenus.

We're using the Composite add() method to add each menu to the top level menu, allMenus.

Now we need to add all the menu items, here's one example

And we're also adding a menu to a menu. All dinerMenu cares about is that everything it holds, whether it's a menu item or a menu, is a MenuComponent

Once we've constructed our entire menu hierarchy, we hand the whole thing to the Waitress, and as you've seen, it's easy as apple pie for her to print it out.

```
% java MenuTestDrive

ALL MENUS, All menus combined
---------------------

PANCAKE HOUSE MENU, Breakfast
---------------------
  K&B's Pancake Breakfast(v), 2.99
     -- Pancakes with scrambled eggs, and toast
  Regular Pancake Breakfast, 2.99
     -- Pancakes with fried eggs, sausage
  Blueberry Pancakes(v), 3.49
     -- Pancakes made with fresh blueberries, and blueberry syrup
  Waffles(v), 3.59
     -- Waffles, with your choice of blueberries or strawberries

DINER MENU, Lunch
---------------------
  Vegetarian BLT(v), 2.99
     -- (Fakin') Bacon with lettuce & tomato on whole wheat
  BLT, 2.99
     -- Bacon with lettuce & tomato on whole wheat
  Soup of the day, 3.29
     -- A bowl of the soup of the day, with a side of potato salad
  Hotdog, 3.05
     -- A hot dog, with saurkraut, relish, onions, topped with cheese
  Steamed Veggies and Brown Rice(v), 3.99
     -- Steamed vegetables over brown rice
  Pasta(v), 3.89
     -- Spaghetti with Marinara Sauce, and a slice of sourdough bread

DESSERT MENU, Dessert of course!
---------------------
  Apple Pie(v), 1.59
     -- Apple pie with a flakey crust, topped with vanilla icecream
  Cheesecake(v), 1.99
     -- Creamy New York cheesecake, with a chocolate graham crust
  Sorbet(v), 1.89
     -- A scoop of raspberry and a scoop of lime

CAFE MENU, Dinner
---------------------
  Veggie Burger and Air Fries(v), 3.99
     -- Veggie burger on a whole wheat bun, lettuce, tomato, and fries
  Soup of the day, 3.69
     -- A cup of the soup of the day, with a side salad
  Burrito(v), 4.29
     -- A large burrito, with whole pinto beans, salsa, guacamole
%
```

Here's all our menus... we printed all this just by calling print() on the top level menu

The new dessert menu is printed when we are printing all the Diner menu components

Summary of Composite

- Multiple objects used in the same way => Composite
  - Do you have nearly identical code to handle each of them?
  - Objects appear in a tree structure capturing a whole-part relationship
- Objects must be able to implement the same interface
- Objective is to simplify client code