Design patterns III : Structural Patterns

J.Serrat

Object-oriented programming, MatCAD

Index

Overview

- Behavioural patterns characterize the ways in which classes or objects interact and distribute responsibility
- Creational patterns concern the process of object creation
- Structural patterns deal with the composition of classes or objects to form larger structures (composite, bridge) or to realize new functionality (adapter ...)

Composite

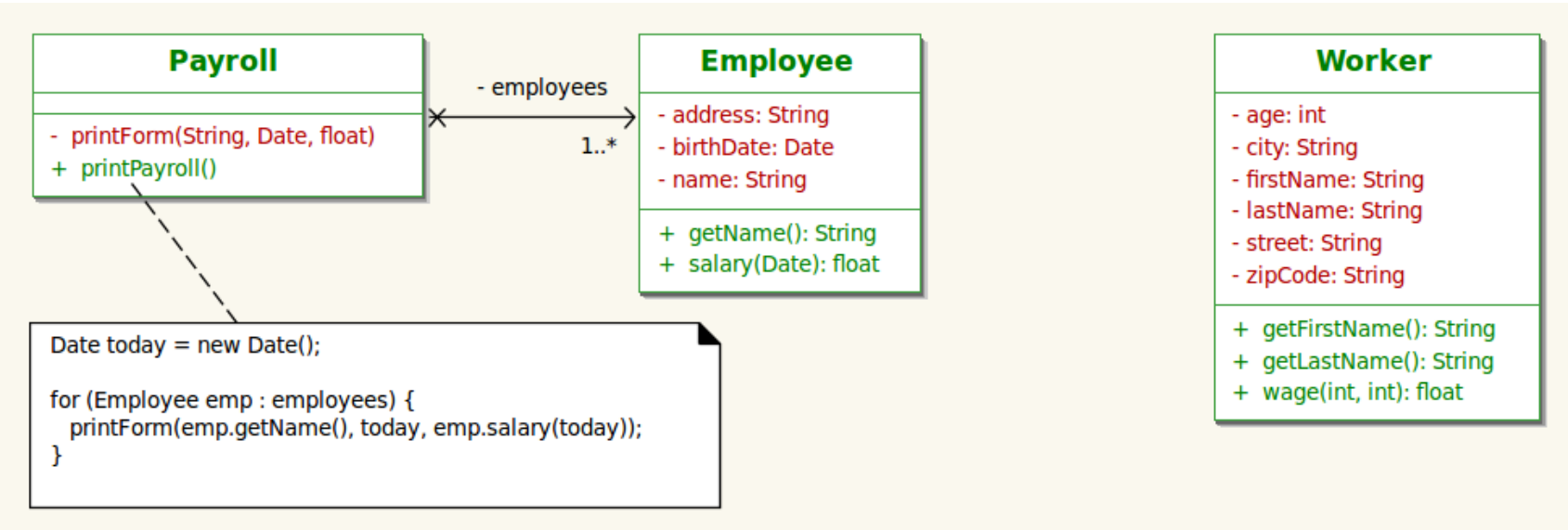
See part I

Adapter

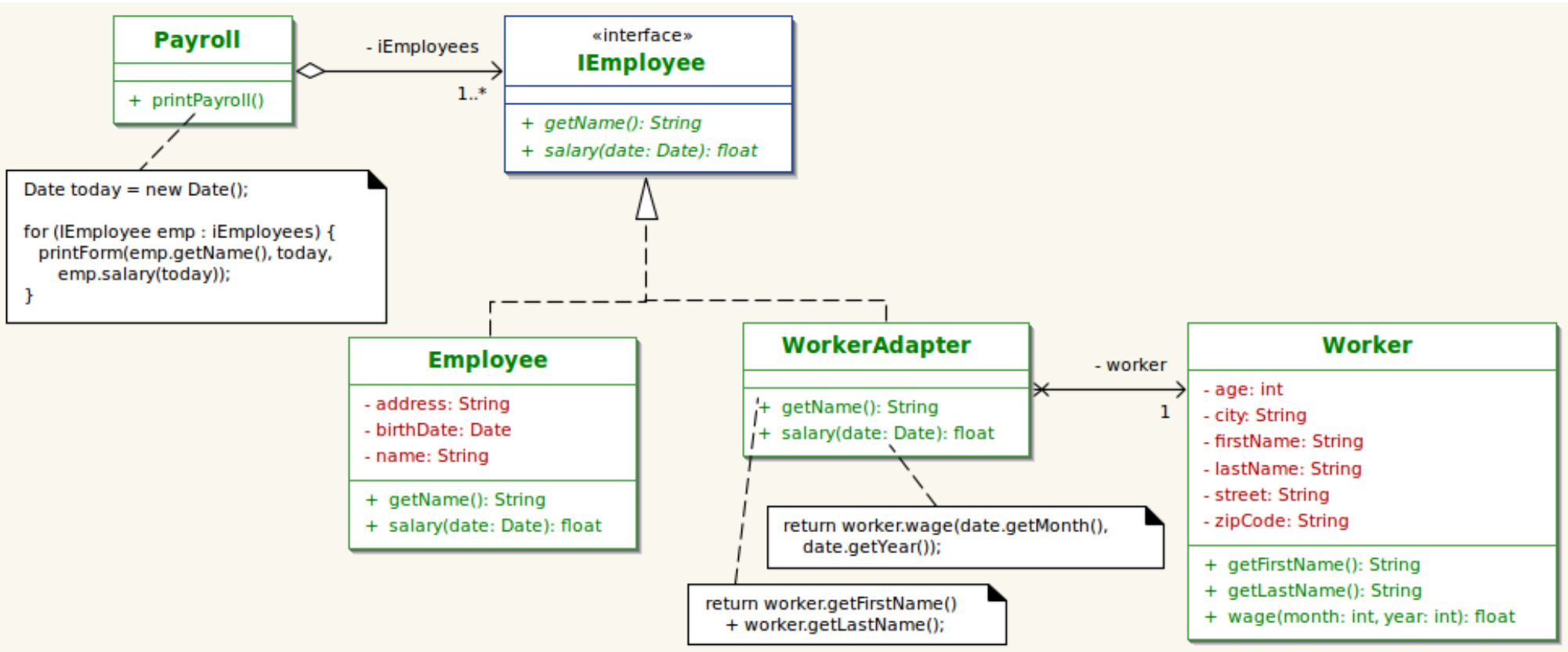
Motivating example

- a company A buys company B, A's payroll application must be expanded with B's employees
- luckily, both applications are written in same OO language
- of course, names of classes, methods and signatures are different
  - A: `Employee`, `getName()`, `salary(Date)`
  - B: `Worker`, `getFirstName()`, `getLastName()`, `wage(int month, int year)`
- at the moment, *reuse* legacy code of B payroll application with minimum effort

Adapter


Payroll
- printForm(String, Date, float)
+ printPayroll()
- employees
1..*
Employee
- address: String
- birthDate: Date
- name: String
+ getName(): String
+ salary(Date): float
Worker
- age: int
- city: String
- firstName: String
- lastName: String
- street: String
- zipCode: String
+ getFirstName(): String
+ getLastName(): String
+ wage(int, int): float
Date today = new Date();
for (Employee emp : employees) {
printForm(emp.getName(), today, emp.salary(today));
}

Adapter


Payroll
+ printPayroll()
- iEmployees
1..*
«interface»
IEmployee
+ getName(): String
+ salary(date: Date): float
Date today = new Date();
for (IEmployee emp : iEmployees) {
printForm(emp.getName(), today,
emp.salary(today));
}
Employee
- address: String
- birthDate: Date
- name: String
+ getName(): String
+ salary(date: Date): float
WorkerAdapter
+ getName(): String
+ salary(date: Date): float
- worker
1
Worker
- age: int
- city: String
- firstName: String
- lastName: String
- street: String
- zipCode: String
+ getFirstName(): String
+ getLastName(): String
+ wage(month: int, year: int): float
return worker.wage(date.getMonth(),
date.getYear());
return worker.getFirstName()
+ worker.getLastName();

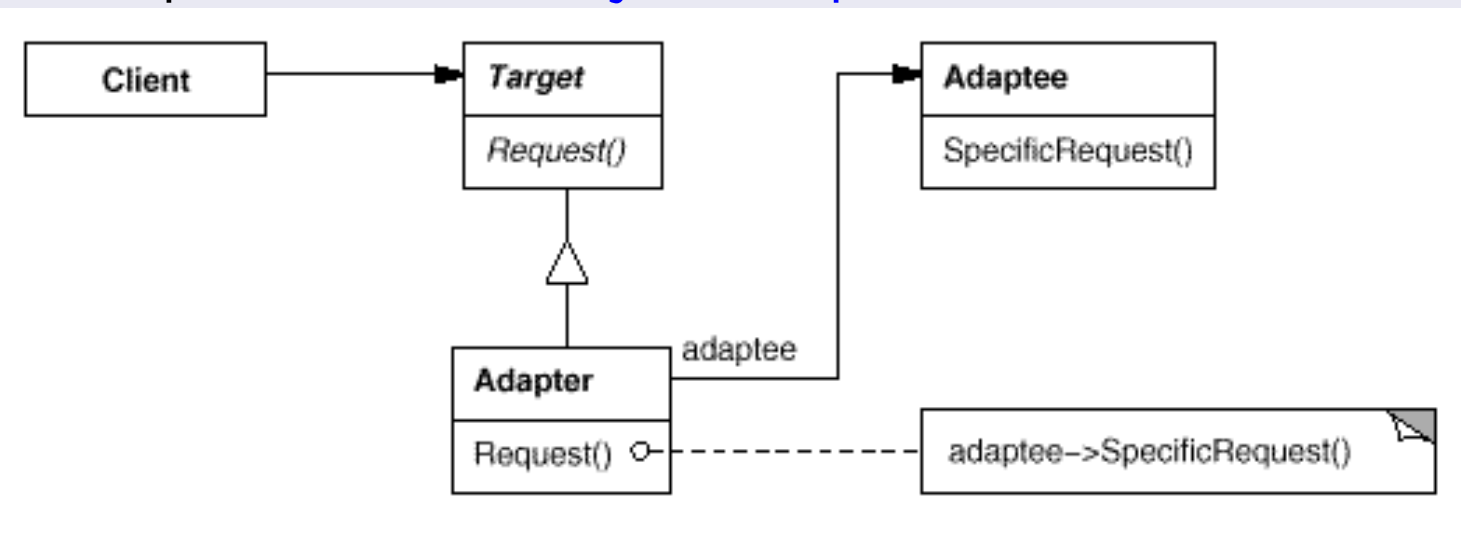
Adapter

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure

Object adapter : relies on object composition



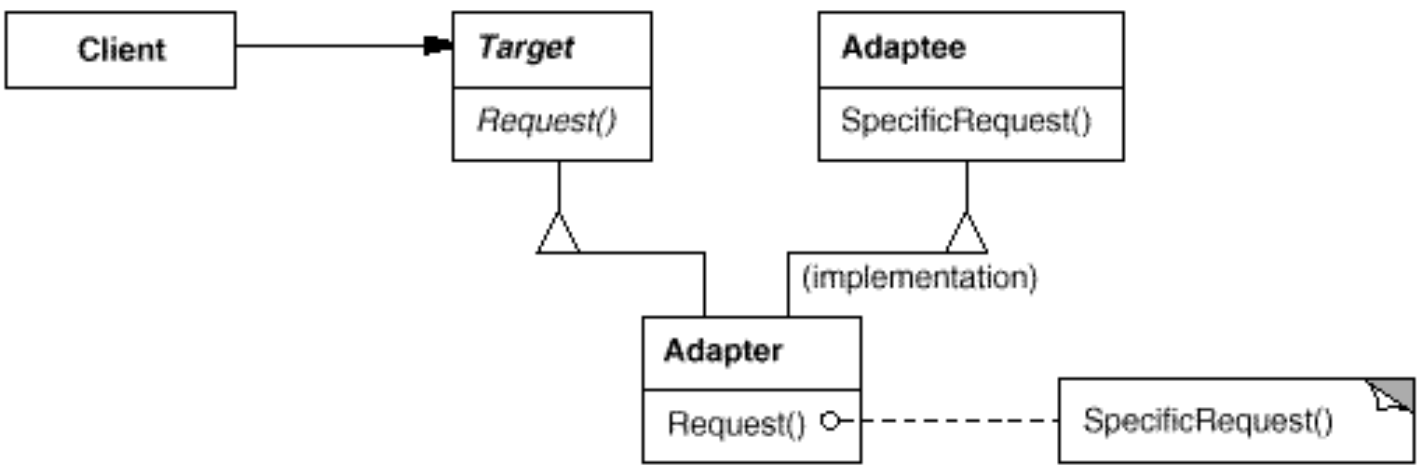
Adapter

Intent

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Structure

Class adapter : relies on multiple inheritance



Adapter

An object adapter

- lets a single Adapter work with many Adaptees—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Adapter

A class adapter

- adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
- introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

Adapter

Applicability

- want to use an existing class but its interface does not match the one you need
- want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
- (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one

Adapter

When no `Target` interface is present we have a wrapper method:

- method delegation: call a second method with little or no additional computation
- adapts an existing class or object to a different interface for reusing existing code

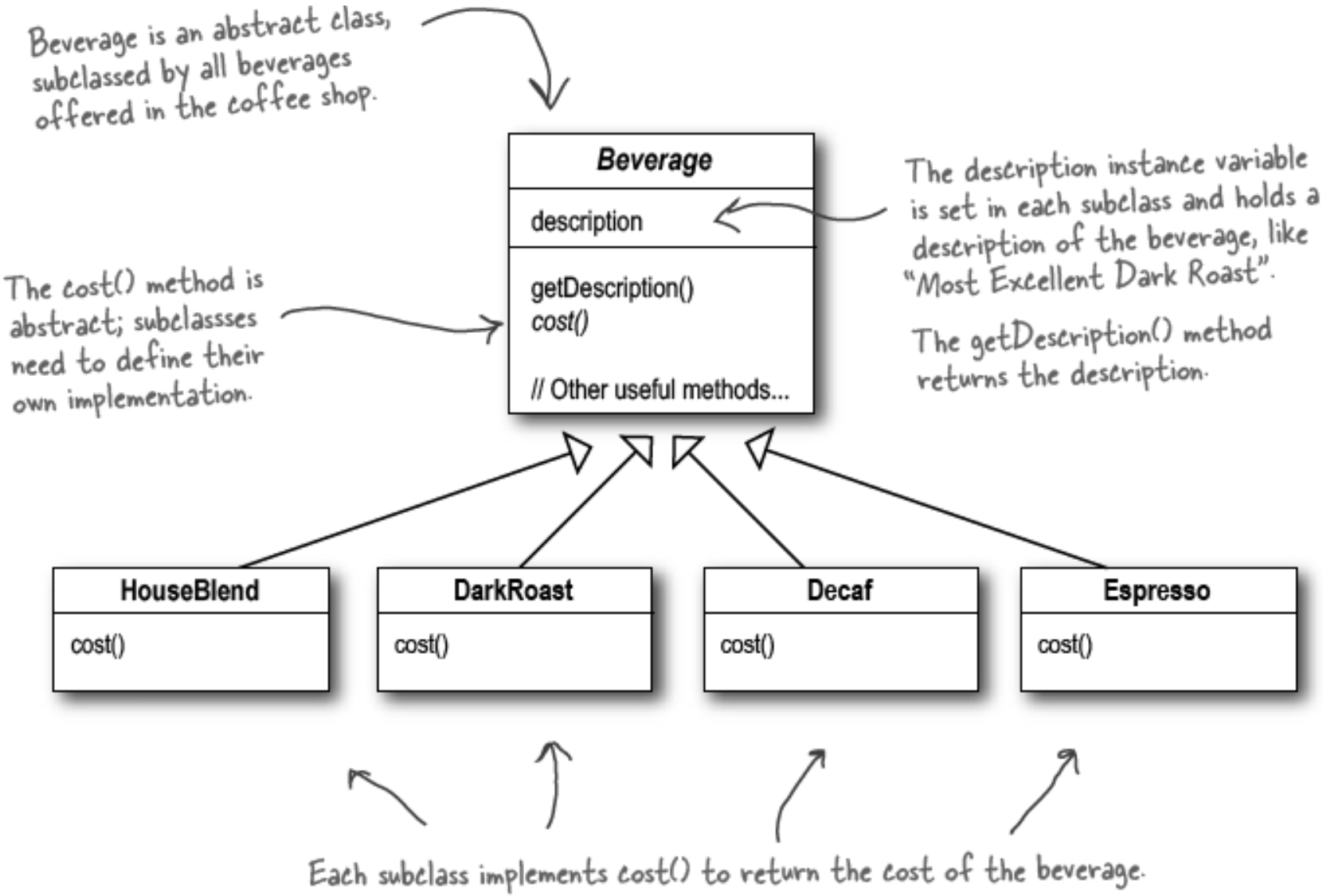
Decorator

Motivating example[1]

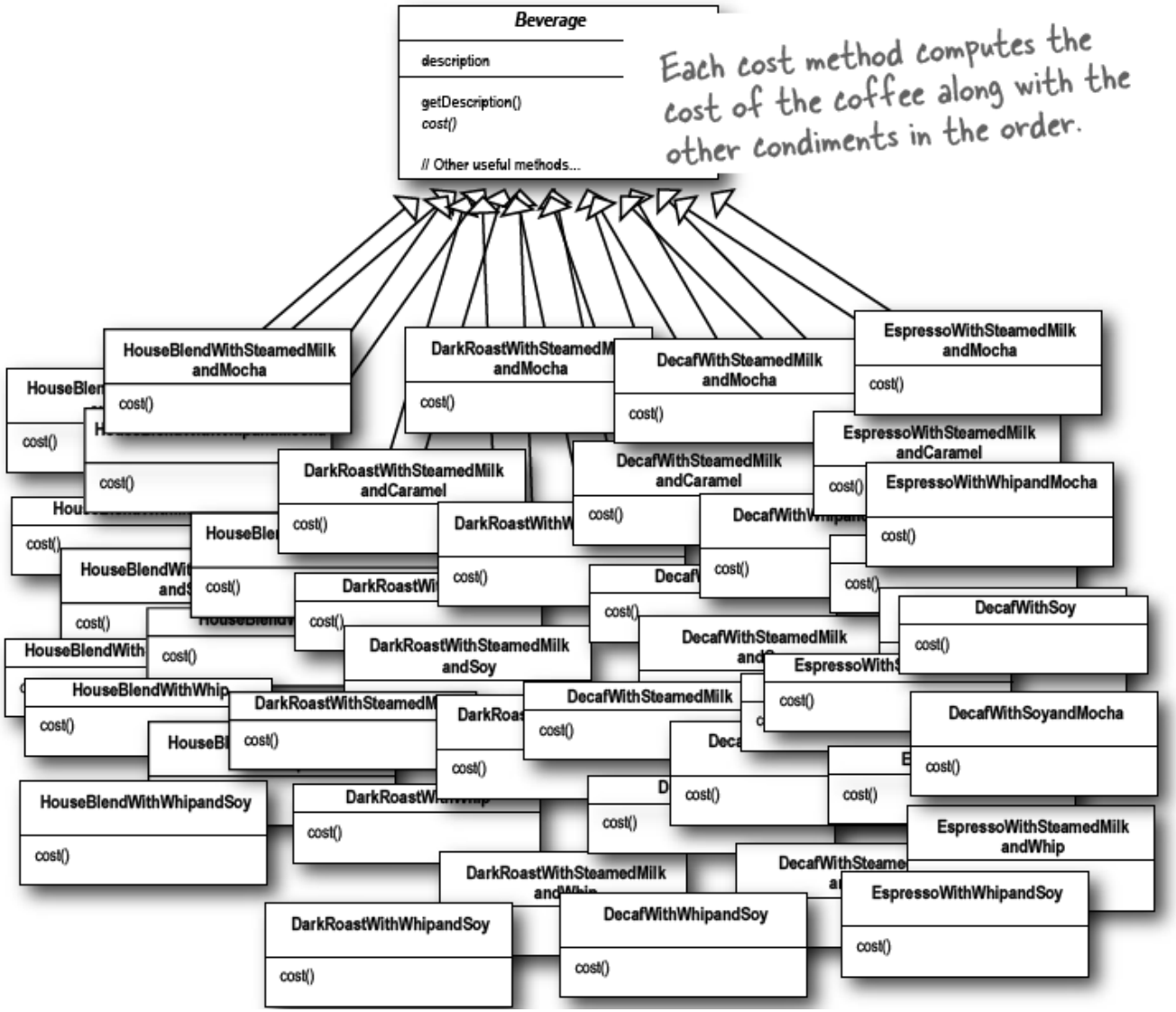
- *Starbuzz Coffee* started selling a few coffee beverage types: house blend, dark roast, decaf, espresso
- a few classes could then represent their offerings
- later on, more and more condiments you can optionally order were introduced : steamed milk, soy milk, mocha, caramel, whip ...
- a customer can order none or any number of them served with a beverage
- Starbuzz charges a bit for each of these, so they really need to get them built into their order system

[1] Head-first design patterns

Decorator


Beverage is an abstract class, subclassed by all beverages offered in the coffee shop.
Beverage
description
getDescription()
cost()
// Other useful methods...
The description instance variable is set in each subclass and holds a description of the beverage, like "Most Excellent Dark Roast".
The getDescription() method returns the description.
The cost() method is abstract; subclassses need to define their own implementation.
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()
Each subclass implements cost() to return the cost of the beverage.

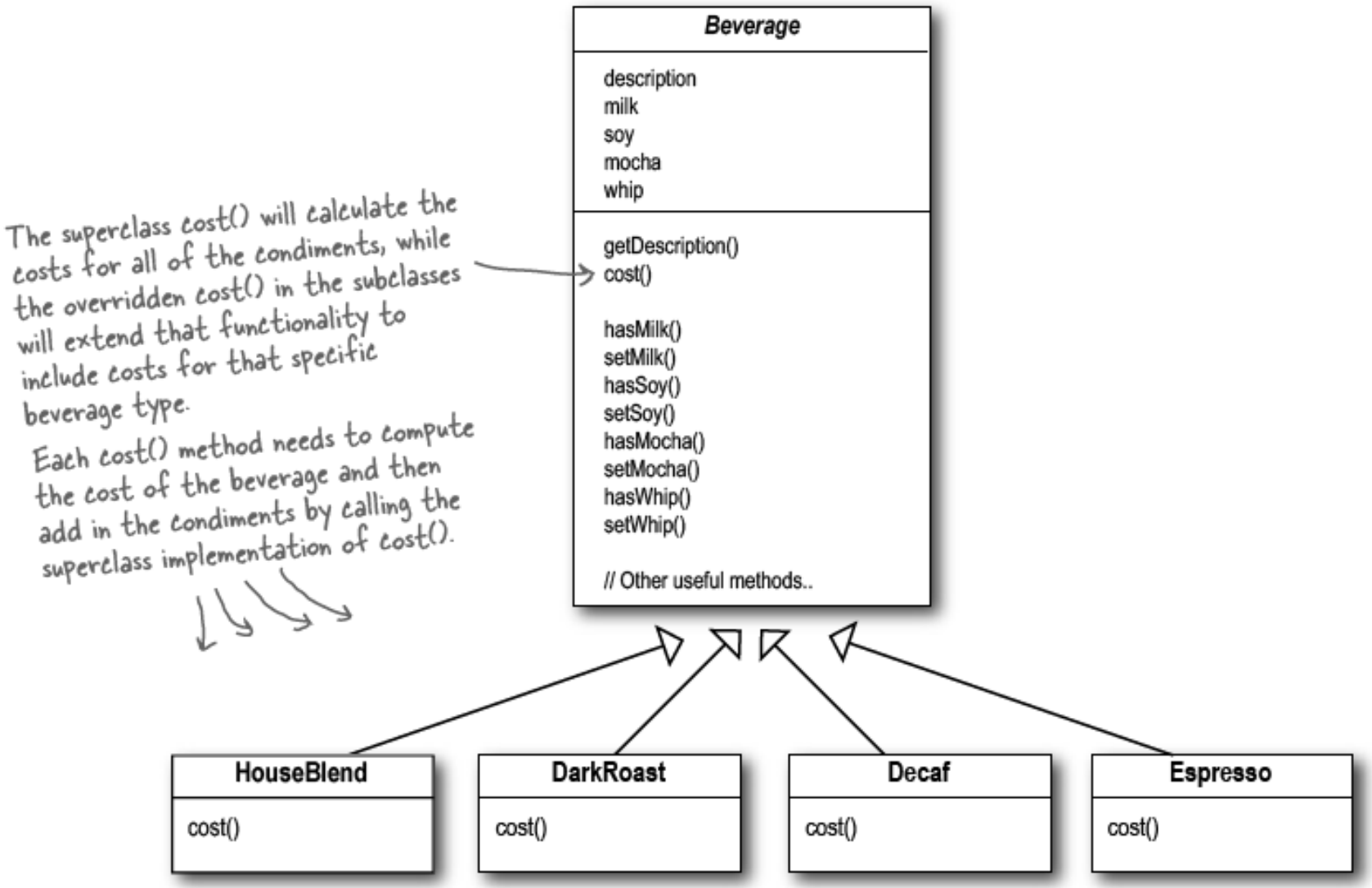
Decorator


Beverage
description
getDescription()
cost()
// Other useful methods...
Each cost method computes the cost of the coffee along with the other condiments in the order.
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
DecafWithSteamedMilk andMocha
EspressoWithSteamedMilk andMocha
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
EspressoWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DecafWithSoy
DarkRoastWithSteamedMilk andSoy
DecafWithSoyandMocha
HouseBlendWithWhipandSoy
EspressoWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
DecafWithWhipandSoy
EspressoWithWhipandSoy

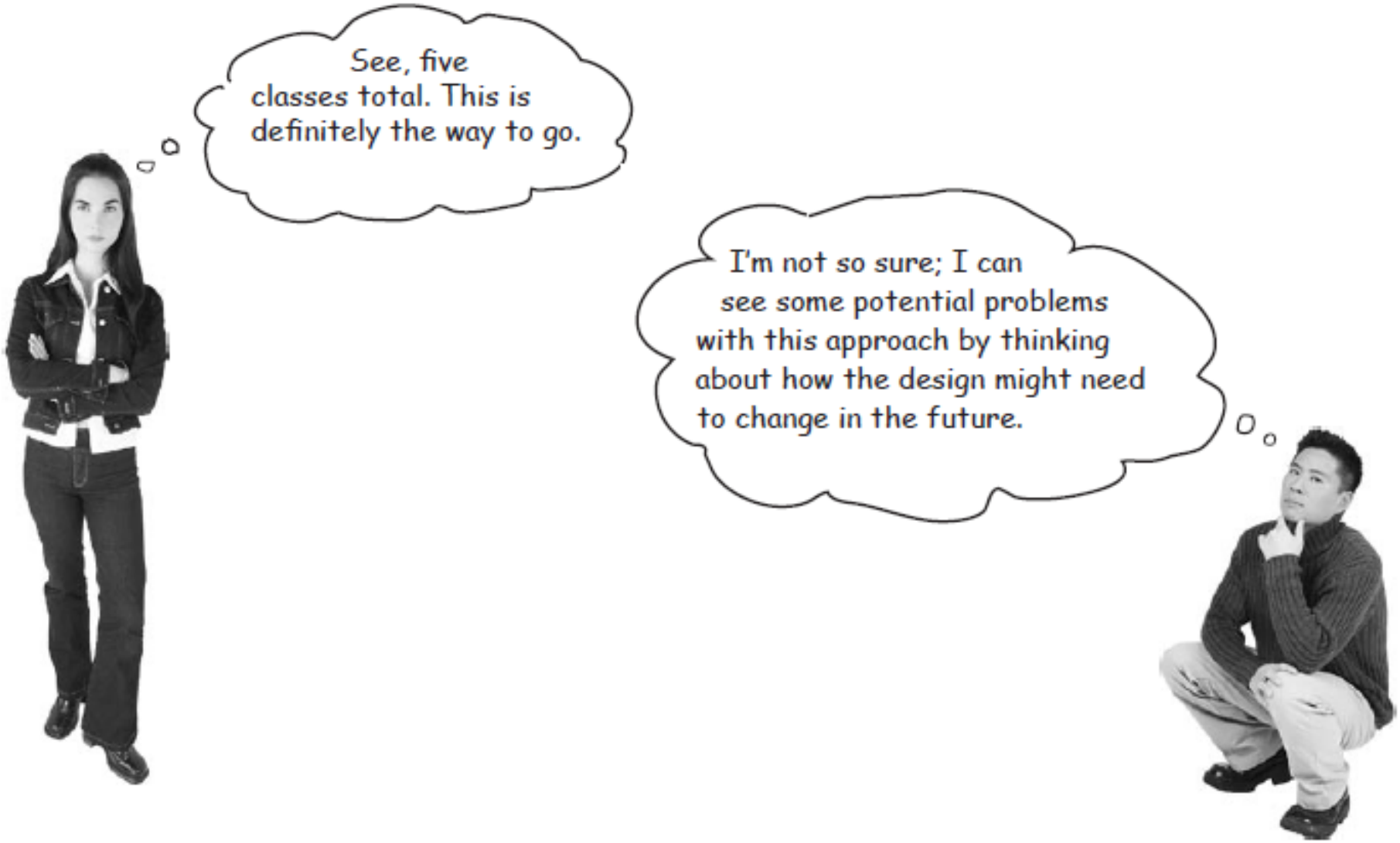
Decorator


This is stupid; why do we need all these classes? Can't we just use instance variables and inheritance in the superclass to keep track of the condiments?

Decorator


Beverage
description
milk
soy
mocha
whip
getDescription()
cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()
// Other useful methods..
The superclass cost() will calculate the costs for all of the condiments, while the overridden cost() in the subclasses will extend that functionality to include costs for that specific beverage type.
Each cost() method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of cost().
HouseBlend
cost()
DarkRoast
cost()
Decaf
cost()
Espresso
cost()

Decorator


See, five classes total. This is definitely the way to go.
I'm not so sure; I can see some potential problems with this approach by thinking about how the design might need to change in the future.

Decorator

Problems:

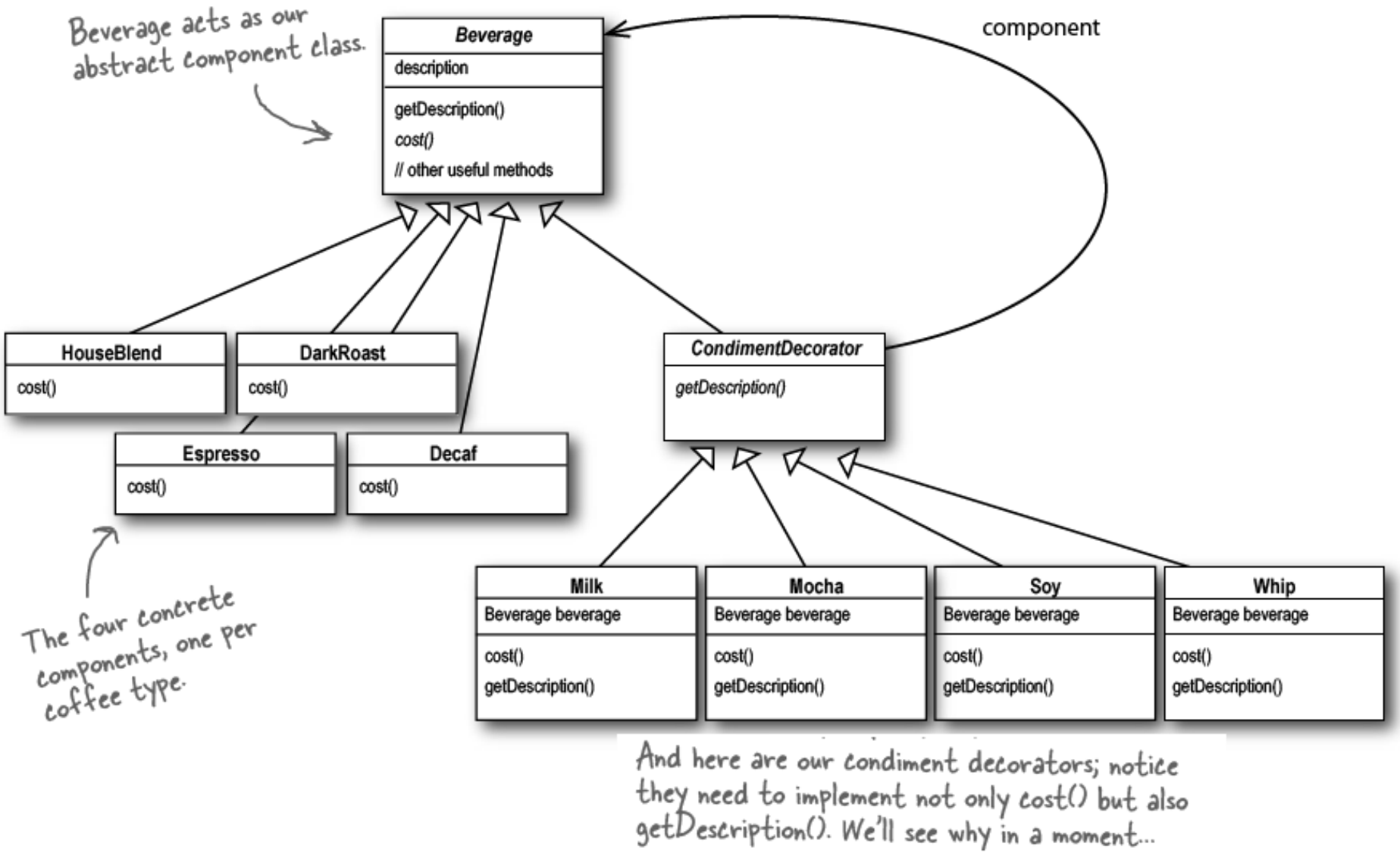
- price changes for *condiments* force changes in `Beverage` class
- a new condiment $\longrightarrow$ new method in `Beverage` and change `cost()` by adding a new `if`
- what if we do not allow some condiments for a beverage (iced tea + whip) ?
- how to represent *double* mocha ?

Decorator

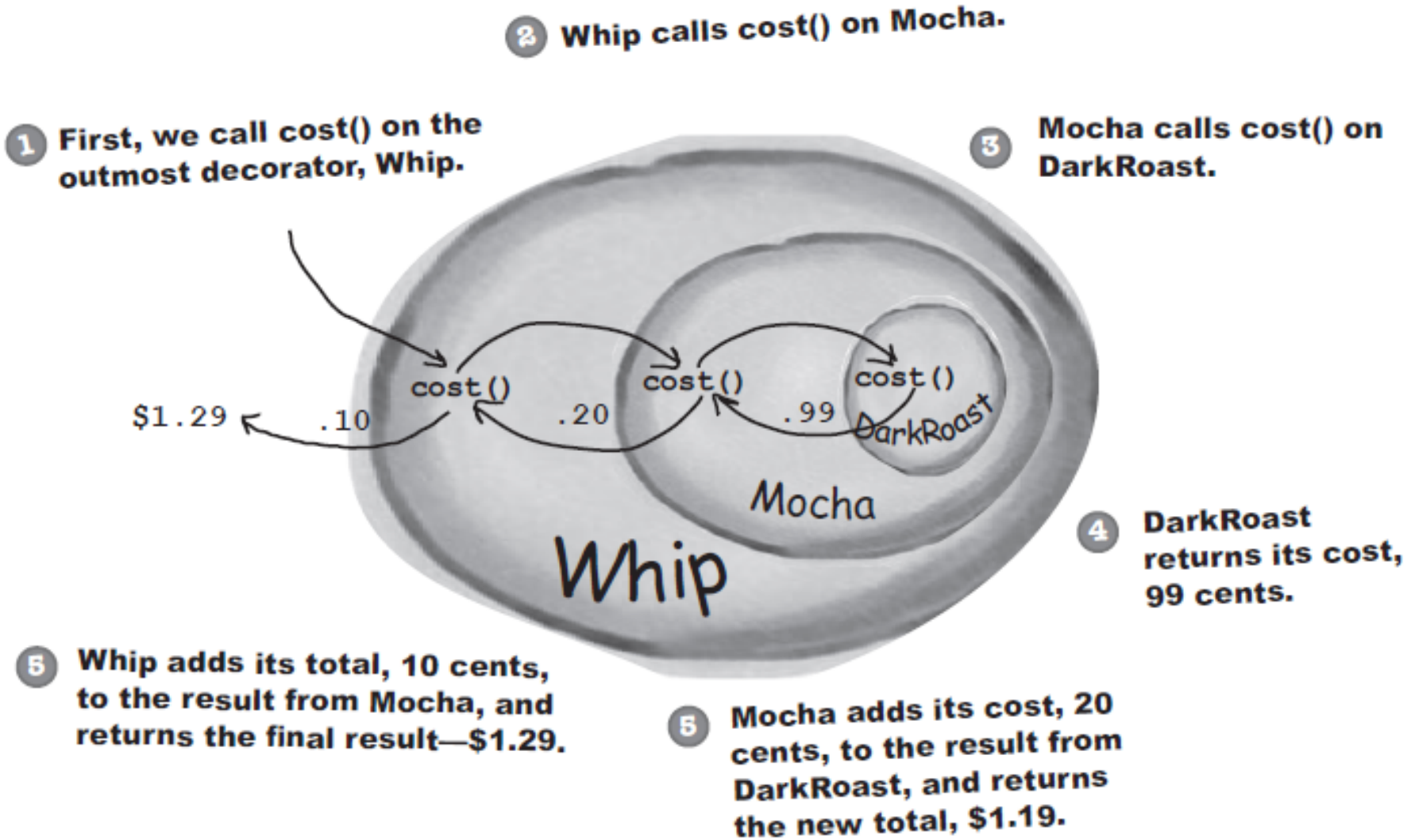
Solution decorator:

- "decorate" = wrap a beverage object with one or more condiments *objects*
- make a list of condiments and put at the end a beverage
- each condiment knows its own price and that it has to be added to the price of beverage or another condiment *it contains*, recursively

Decorator


Beverage acts as our abstract component class.
Beverage
description
getDescription()
cost()
// other useful methods
component
HouseBlend
cost()
DarkRoast
cost()
Espresso
cost()
Decaf
cost()
CondimentDecorator
getDescription()
The four concrete components, one per coffee type.
Milk
Beverage beverage
cost()
getDescription()
Mocha
Beverage beverage
cost()
getDescription()
Soy
Beverage beverage
cost()
getDescription()
Whip
Beverage beverage
cost()
getDescription()
And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

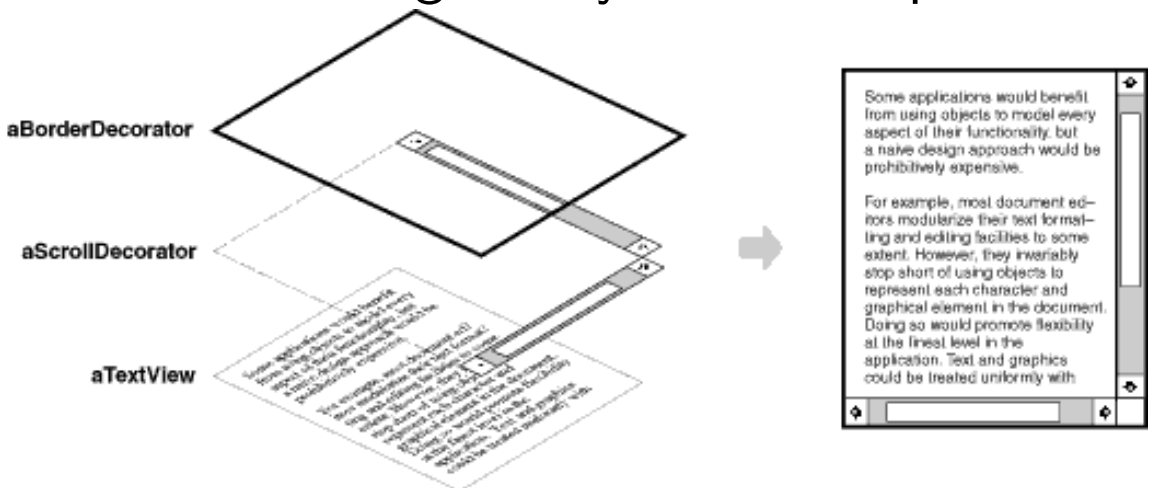
Decorator


2 Whip calls cost() on Mocha.
1 First, we call cost() on the outmost decorator, Whip.
3 Mocha calls cost() on DarkRoast.
cost()
cost()
cost()
$1.29
.10
.20
.99
DarkRoast
Mocha
Whip
4 DarkRoast returns its cost, 99 cents.
5 Whip adds its total, 10 cents, to the result from Mocha, and returns the final result—$1.29.
5 Mocha adds its cost, 20 cents, to the result from DarkRoast, and returns the new total, $1.19.

Decorator

Second example[2]

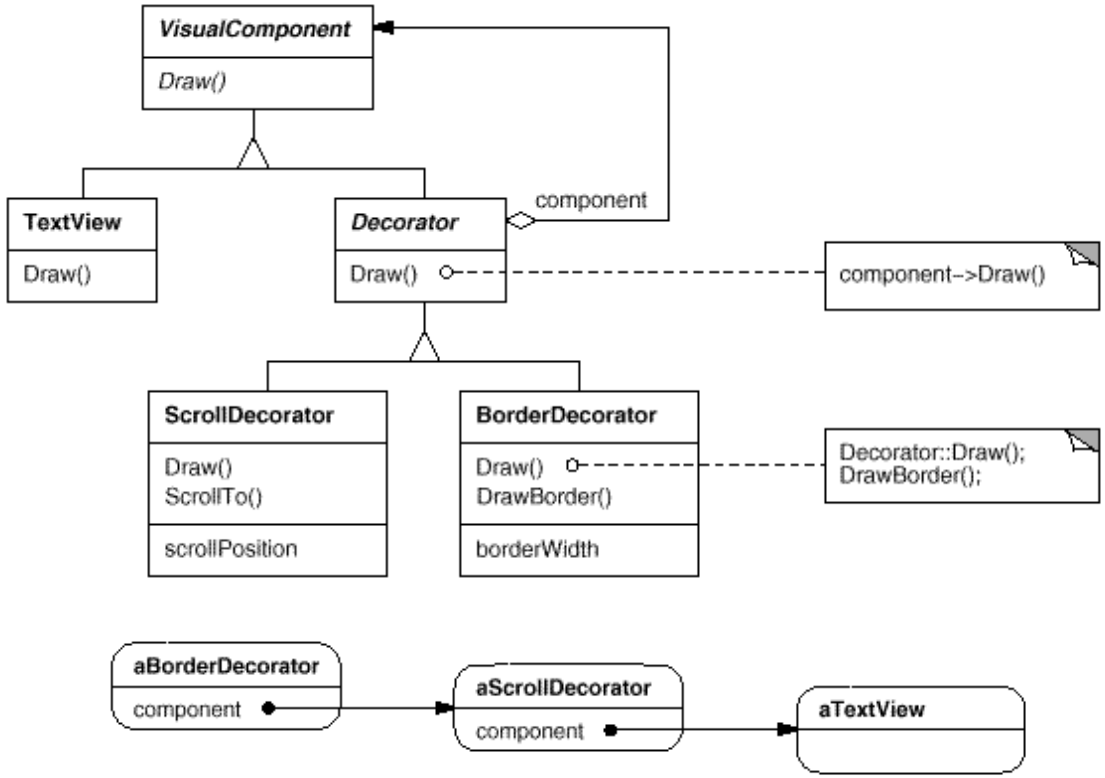
- a GUI toolkit should let you add properties like borders or behaviours like scrolling to any visual component



- inheritance is not convenient because user can not choose whether to add or not: the visual component *class* has already inherited it
- we may want to add scrollbars or borders at run time

[2] Gamma et al.

Decorator



- it is possible to decorate any `VisualComponent` with one or more decorators
- `Decorator` subclasses *extend* a VisualComponent's `draw()` by drawing themselves and forward draw request

Decorator

Intent

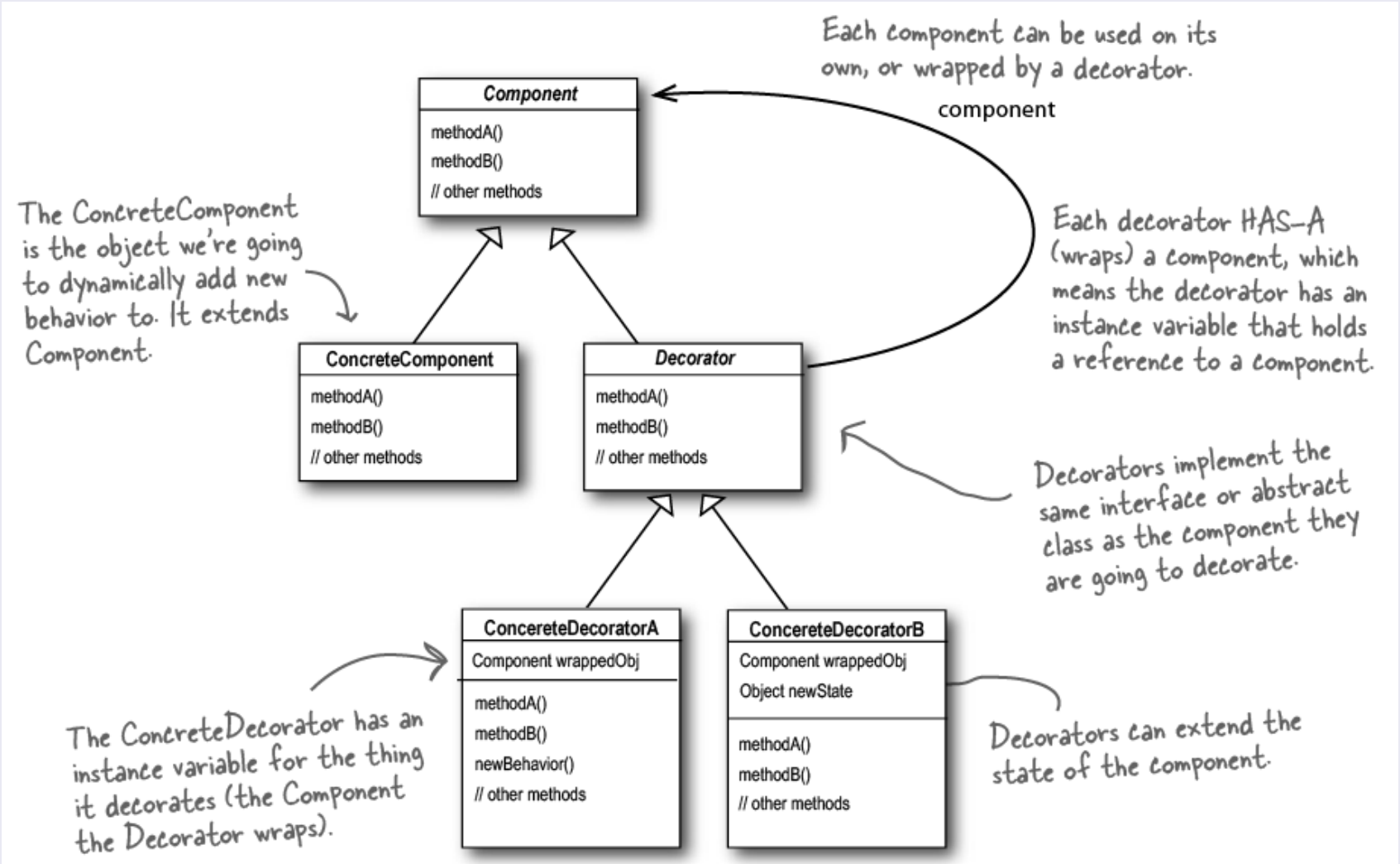
Attach additional responsibilities to an individual object dynamically, not to an entire class (inheritance). Decorators provide a flexible alternative to subclassing for extending functionality.

Applicability

- add responsibilities to individual objects dynamically and without affecting other objects
- for responsibilities that can be withdrawn
- when extension by subclassing is impractical because would produce explosion of subclasses

Decorator

Structure



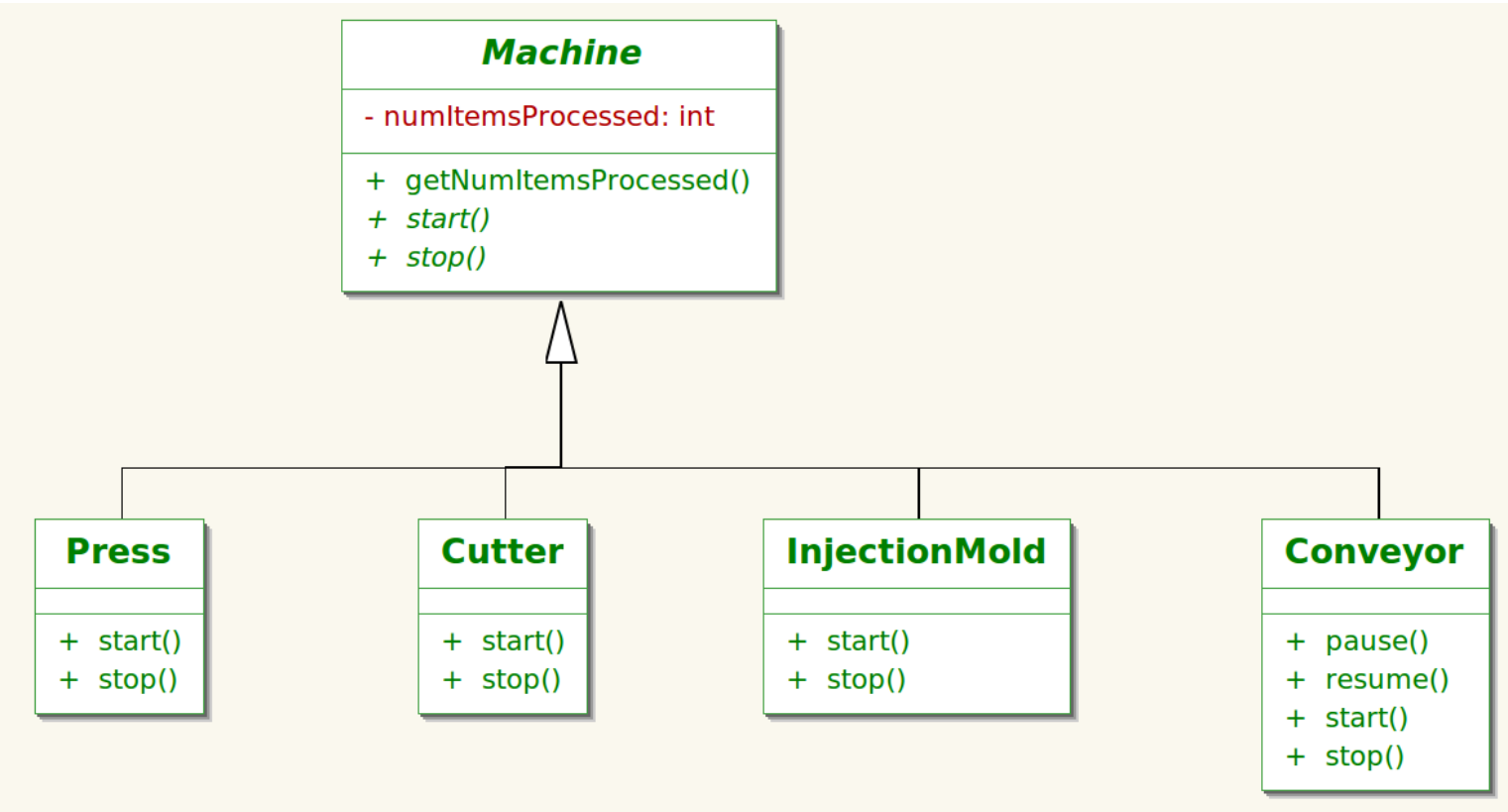
Bridge

Motivating example

- an industrial plant has machines of several types: press, cutter, injection molding, conveyor belt
- machines start/stop is software controlled
- at the moment every machine is thus directly operated by calling its `start()` and `stop()` methods
- start/stop is different depending on the machine type: presses start their associated feeder and output conveyors, cutters wait for a worker to press a confirmation button, etc.
- any machine can be asked for the number of processed items since last start operation

Bridge

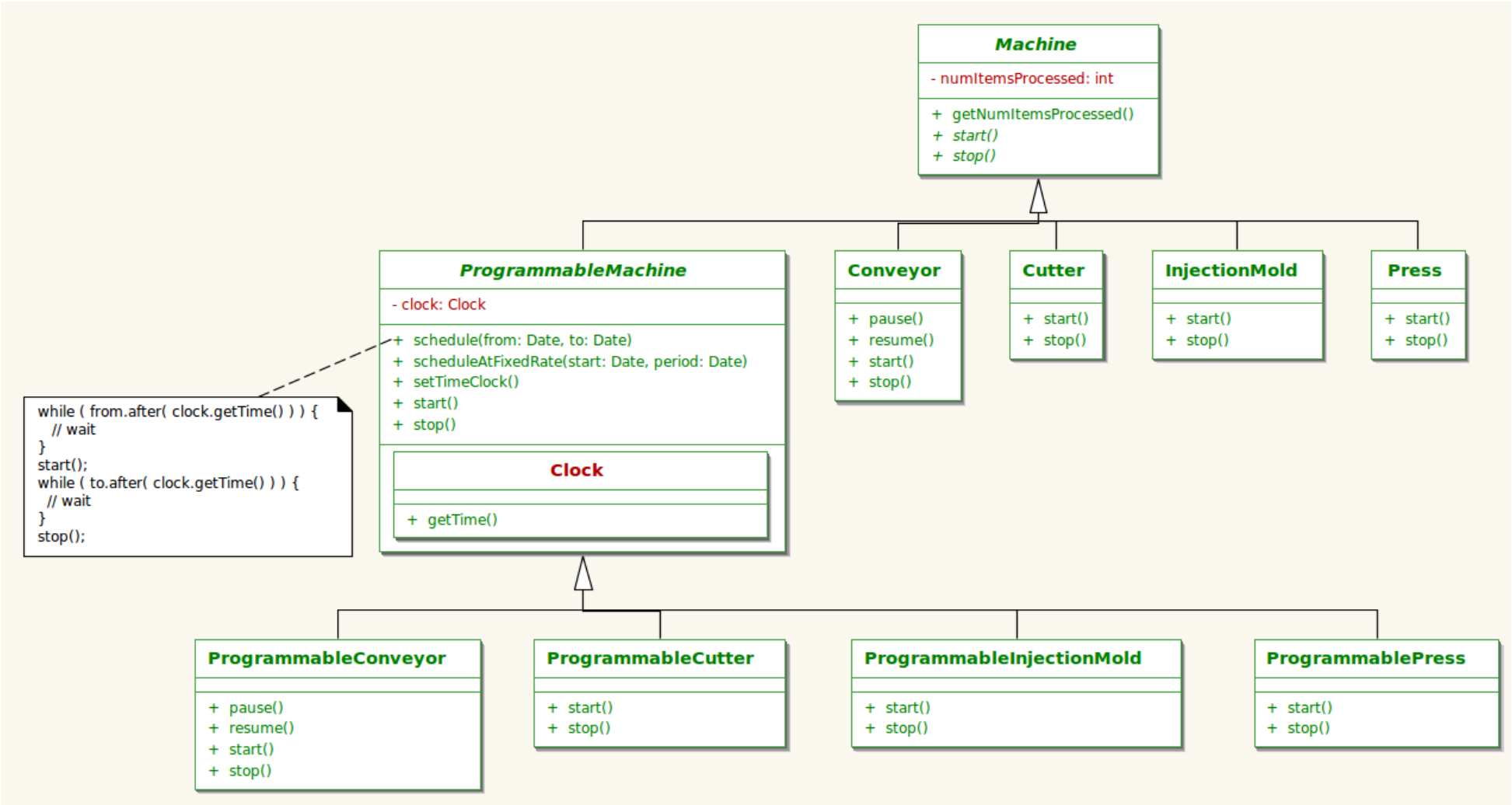
An abstraction, `Machine`, has one of several possible implementations, `Press`, `Cutter` which may override or extend `start()` and `stop()`



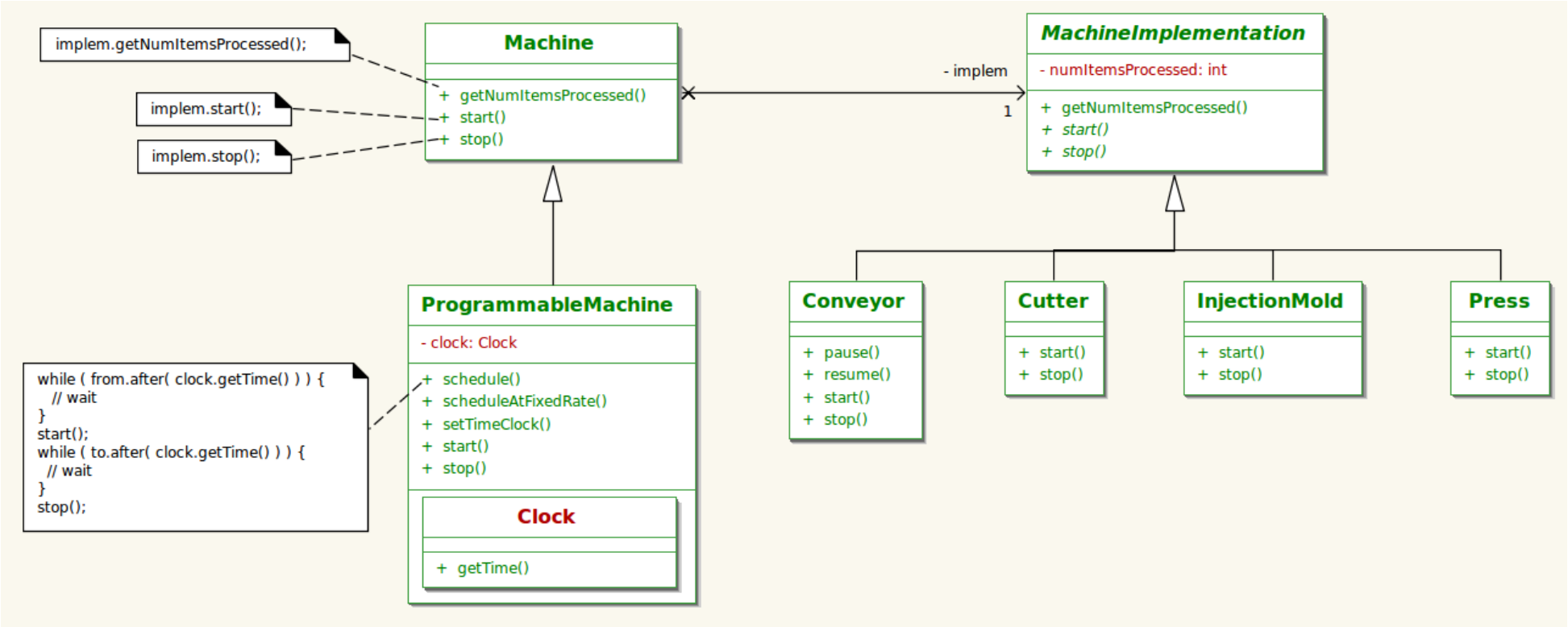
Bridge

- Later on, new press, cutters, conveyor belts are bought which can be programmed to start and stop at given times, and even to do it periodically
- We are forced to add many new classes: every combination of $\{$non-programmable, programmable$\} \times \{$press, cutter, injection molding, conveyor belt$\}$ because start/stop are different *and* may have or not `schedule()` capability
- Inheritance binds an implementation to the abstraction *permanently*, which makes it difficult to modify, extend, and reuse abstractions and implementations *independently*

Bridge


Machine
- numItemsProcessed: int
+ getNumItemsProcessed()
+ start()
+ stop()
ProgrammableMachine
- clock: Clock
+ schedule(from: Date, to: Date)
+ scheduleAtFixedRate(start: Date, period: Date)
+ setTimeClock()
+ start()
+ stop()
Clock
+ getTime()
Conveyor
+ pause()
+ resume()
+ start()
+ stop()
Cutter
+ start()
+ stop()
InjectionMold
+ start()
+ stop()
Press
+ start()
+ stop()
while (from.after(clock.getTime())) {
// wait
}
start();
while (to.after(clock.getTime())) {
// wait
}
stop();
ProgrammableConveyor
+ pause()
+ resume()
+ start()
+ stop()
ProgrammableCutter
+ start()
+ stop()
ProgrammableInjectionMold
+ start()
+ stop()
ProgrammablePress
+ start()
+ stop()

Bridge



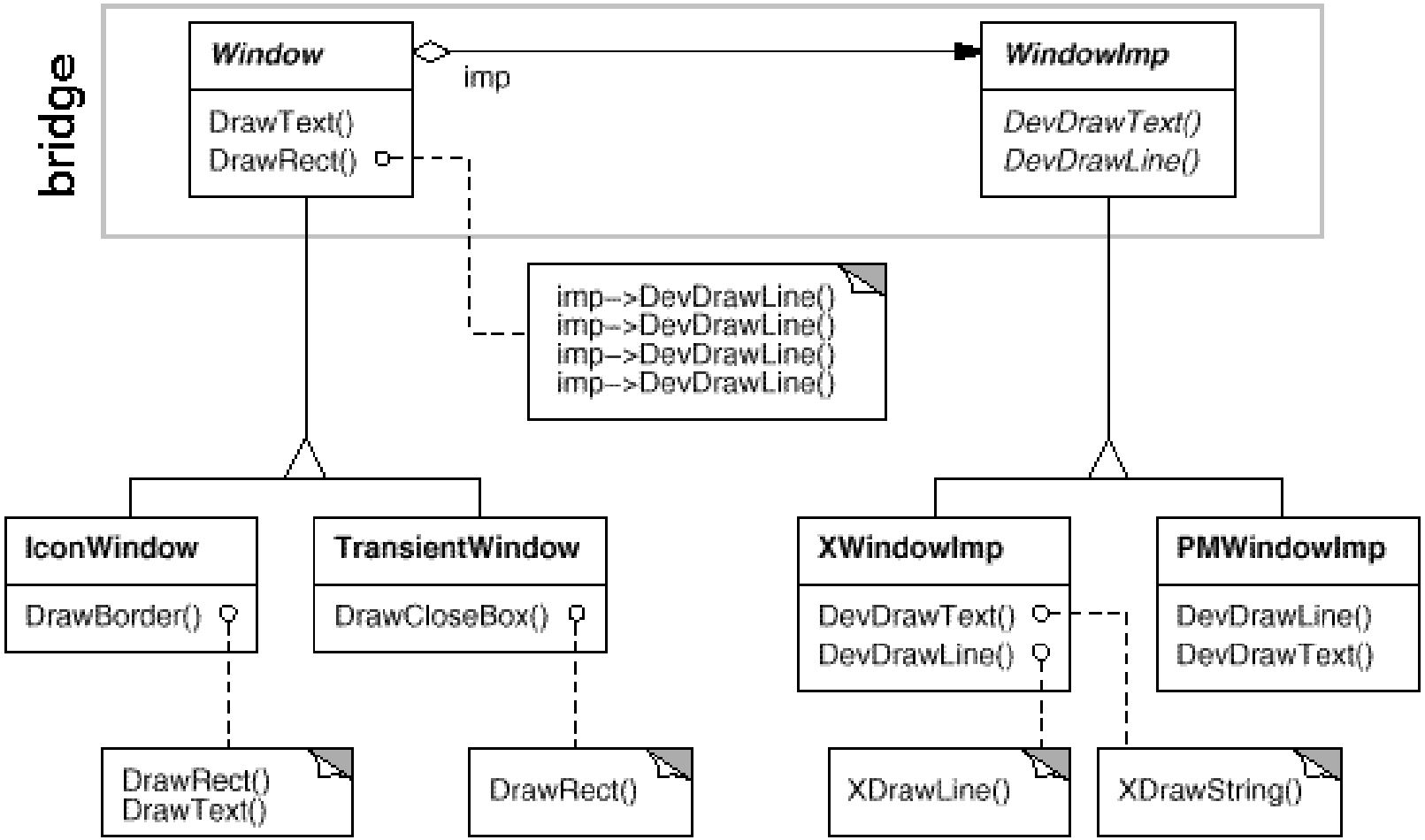
Operations on `Machine` subclasses are implemented in terms of abstract operations from the `MachineImplementation` interface. This decouples the window abstractions from implementations.

Bridge

Second example

- a multi-platform library for GUI: X-Windows (Linux), PM-Window (obsolete)
- several types of windows: `IconWindow`, `TransientWindow`...
- all windows may need to `drawText` and `drawRectangle`
- drawing text and lines depends on the specific plaftorm
- do not want class explosion: {X-Windows, PM-Window } $\times$ {IconWindow, TransientWindow}

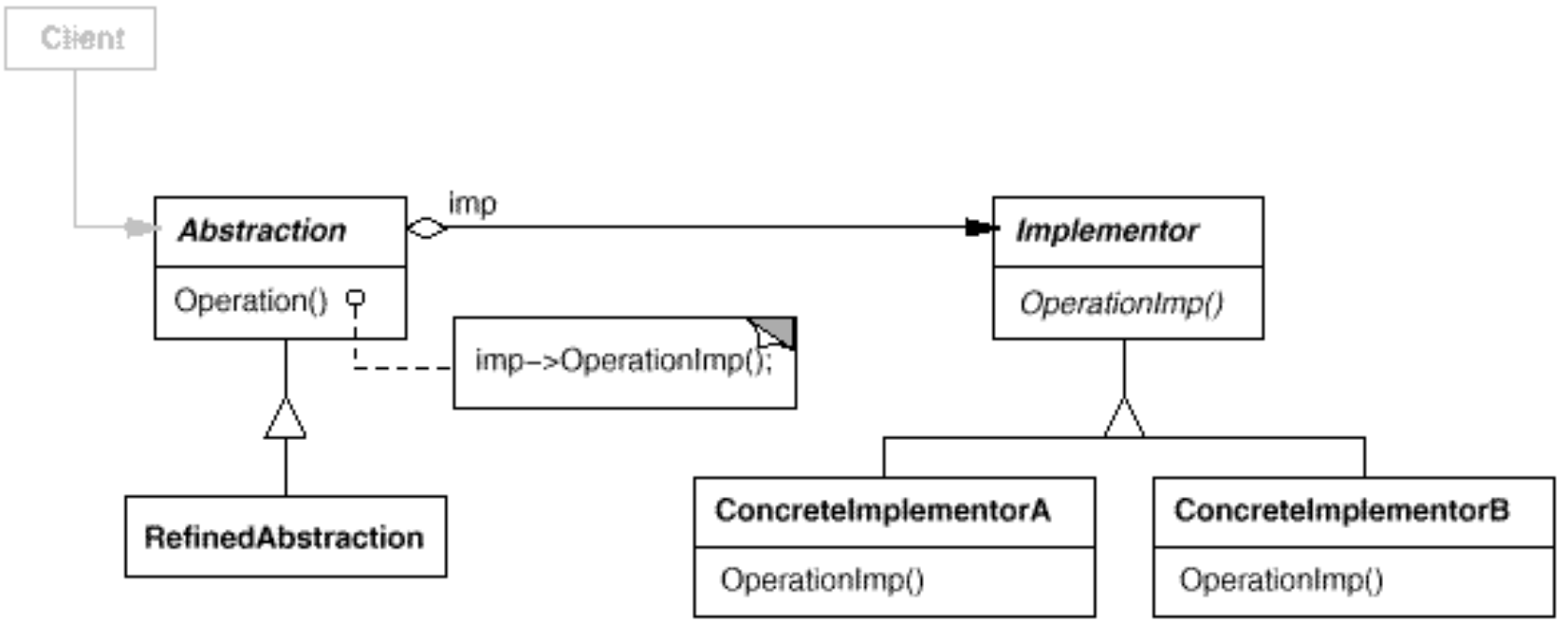
Bridge



Operations on `Window` subclasses are implemented in terms of abstract operations from the `WindowImp` interface. This decouples the window abstractions from implementations.

Bridge

Structure



Bridge

Intent

Decouple an abstraction from its implementation so that the two can vary independently.

Applicability

- avoid a permanent binding between an abstraction and its implementation, e.g. selected at run-time
- both the abstractions and their implementations should be independently extensible by subclassing
- changes in the implementation of an abstraction should have no impact on clients
- explosion of classes in inheritance

Proxy

Motivating example

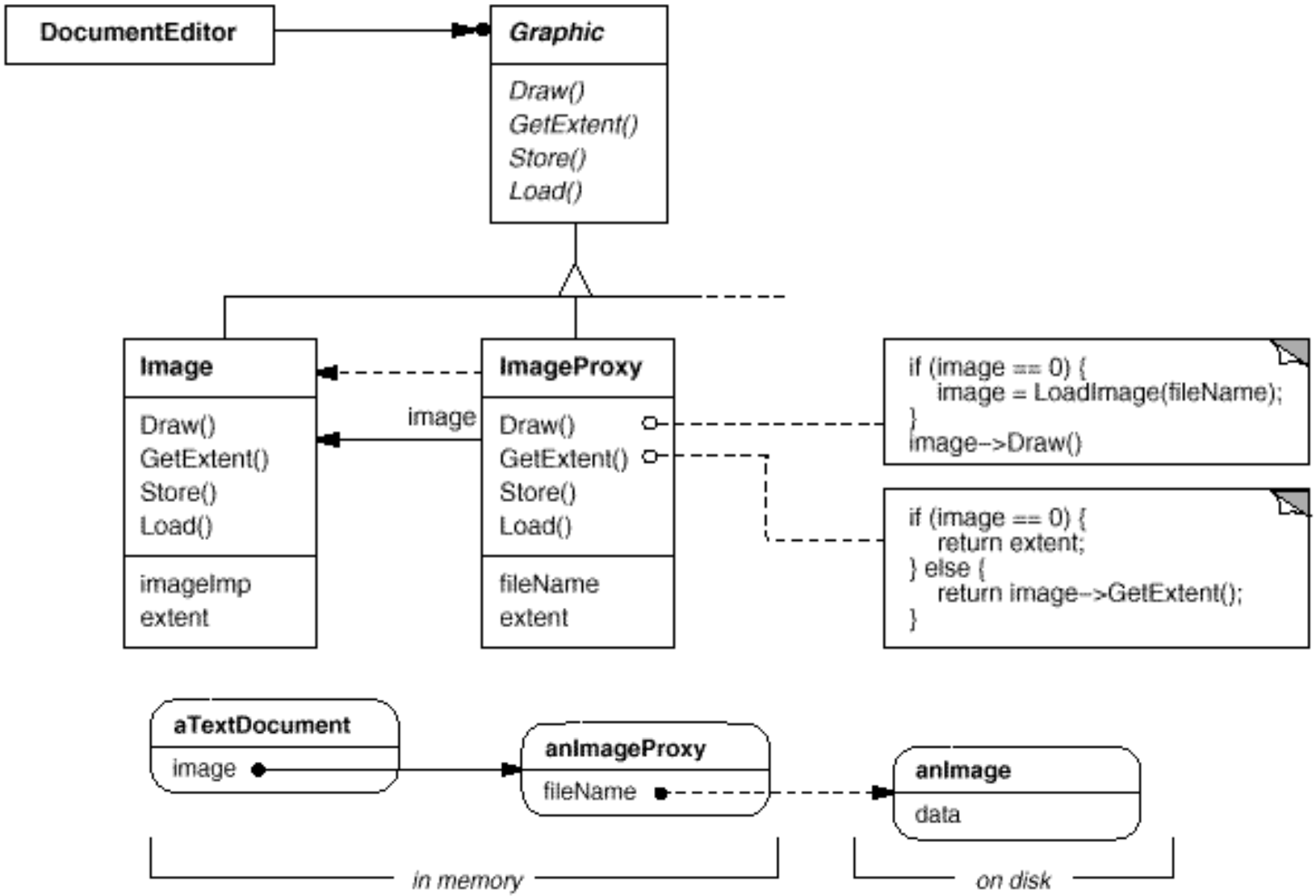
- a document editor like Word can embed graphical objects
- some graphical objects, like large images, can be expensive to create (read from file, web address)
- opening a document should be fast $\rightarrow$ we should avoid creating *then* all the expensive objects
- anyway, not all of them will be visible at the same time
- creating object images on demand: when an image becomes visible
- but we may need some of their properties, like width and height, to format the document *before* rendering them
- client code shouldn't depend on whether the image has been fully loaded or not

Proxy

A solution

- use another object, an image proxy, that acts as a surrogate for the real image
- the proxy acts just like the image : same methods and properties (width, height)
- takes care of instantiating the real image when it's required
- keeps a reference to it
- from then on, forwards incoming messages to the real image object

Proxy

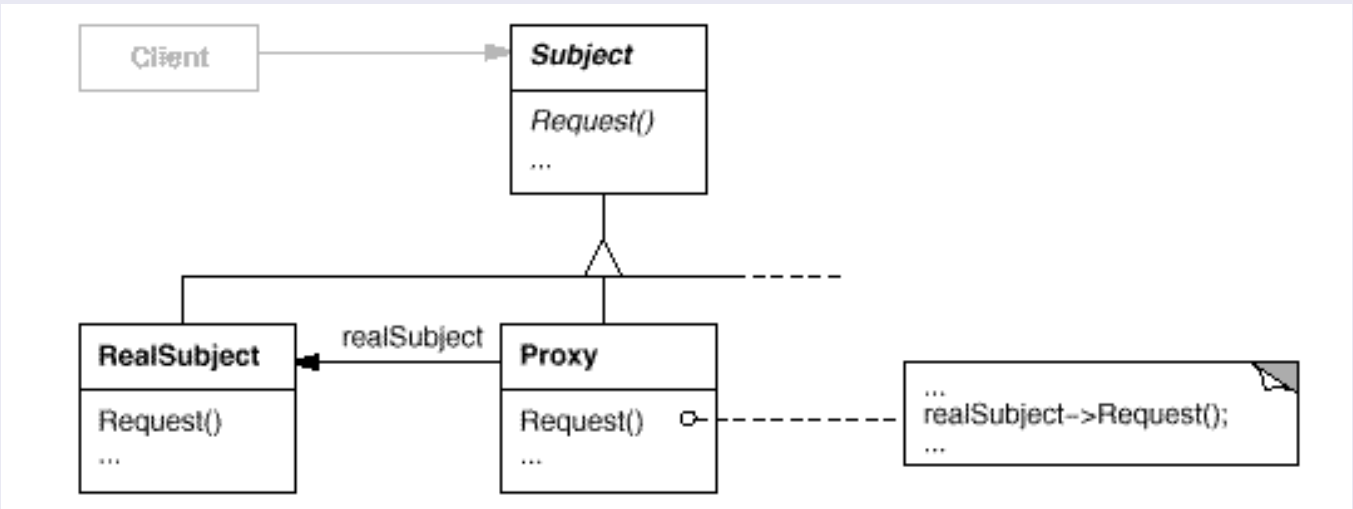

DocumentEditor
Graphic
Draw()
GetExtent()
Store()
Load()
Image
Draw()
GetExtent()
Store()
Load()
imageImp
extent
image
ImageProxy
Draw()
GetExtent()
Store()
Load()
fileName
extent
if (image == 0) {
image = LoadImage(fileName);
}
image->Draw()
if (image == 0) {
return extent;
} else {
return image->GetExtent();
}
aTextDocument
image
anImageProxy
fileName
anImage
data
in memory
on disk

Proxy

Intent

Provide a surrogate or placeholder for another object to control access to it. There is a need for a more versatile reference to an object than a simple pointer.

Structure



Proxy

Applicability

- a virtual proxy creates expensive objects on demand (like ImageProxy), defer the cost of its creation and initialization until we actually need it
- a protection proxy controls access to the original object, for instance because of client objects have different access rights
- a smart reference proxy performs additional actions like
  - count the number of references to the real object so that it can be freed automatically when there are no more references
  - load a persistent object into memory when it's first referenced.
  - check that the real object is locked before it's accessed to ensure that no other object can change it

Façade

Exercise

Flyweight

Exercise

You should know

- intent and structure of each pattern
- similarity and differences of Adapter, Decorator, Proxy: forwarding messages with different purpose
- similarity and differences between Bridge and Strategy, State
- what each pattern encapsulates

Pattern	Aspects that vary
Composite	structure and composition of an object
Adapter	interface to an object
Façade	interface to a subsystem
Decorator	responsibilities of an object without subclassing
Bridge	implementation of an object
Proxy	how an object is accessed; its location
Flyweight	storage costs of objects