Design Patterns

Take home points for today:
learn of the existence of several patterns (and kinds of patterns)
have a high-level sense of how to distinguish/choose them based on when they're meant to be used

http://sourcemaking.com/design_patterns

Programming
“on your own” vs “with help”

- Think about what it feels like to solve a problem for the very first time...
- You need to think about the problem from scratch
- You have to test very thoroughly, and maybe won’t even think of all possible cases to test!

Programming
“on your own” vs “with help”

- Now think about what it’s like when someone tells you a possible solution
- That solution includes a lot of knowledge, experimentation, and testing!
- This saves a *lot* of time!!!

Design patterns: the ultimate help

- Industry noticed this “on your own” versus “with help” phenomenon too.
- A community developed where people could describe their solutions to commonly encountered problems, and others could benefit
- They called these solutions “Design Patterns”

Essentially...

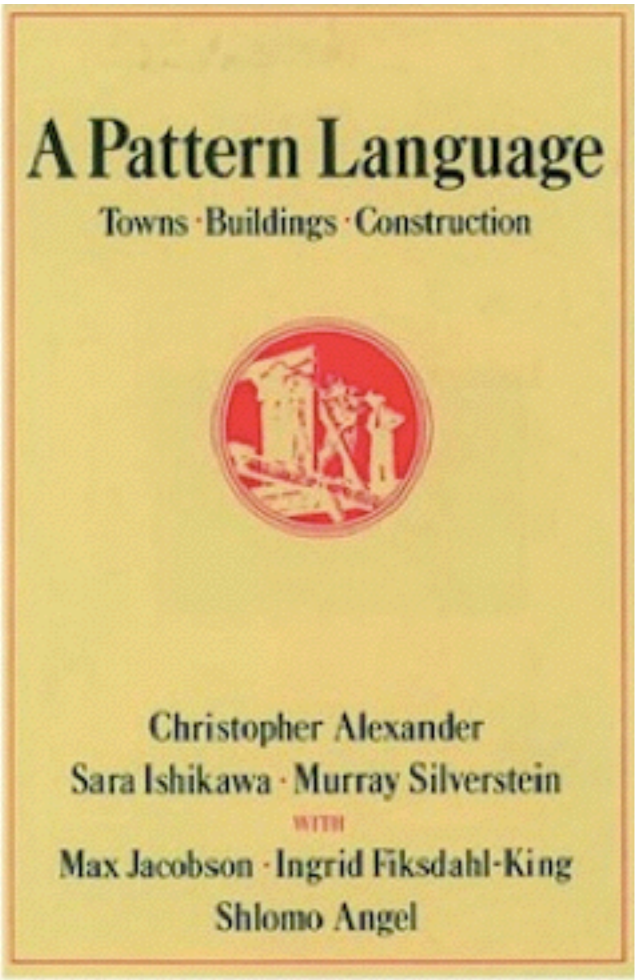
- A Design Pattern is:

A Tried and True Solution
To a Common Problem

- Basically, smart people, who have done this a lot, are making a suggestion!

The “Design Patterns” name

- The original use of the term “Design Patterns” really comes from Architecture (building architecture) from Christopher Alexander
- These were architectural idioms, to guide architectural design (a house is composed of a kitchen, bathroom, bedrooms etc... to be placed in certain basic configurations)



A Pattern Language

Towns · Buildings · Construction

Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

The Design Patterns “format”

- Alexander provided a nice way to describe these patterns:
    - PROBLEM (whats wrong?)
    - CONTEXT (what’s happening around the problem)
    - SOLUTION (what to do)

Every Day Patterns...

You can have a "pattern" for anything where expert advice is helpful!

- **A Good Dinner Party**
  - **PROBLEM**: You want your guests to have fun and be impressed by your prowess as a host, chef and sommelier.
  - **SOLUTION (abridged)**:
    - 1. invite people who will like each other. Or, if you want a more lively party, people who will not like each other.
    - 2. greet your guests and introduce them to everyone who is already there.
    - 3. serve dinner, then dessert, then drinks, avoiding allergies of your guests.
  - **WARNINGS**: might lead to grogginess the next morning. Do not apply this pattern the night before the use of the "Write an Exam" pattern.

What about Software?

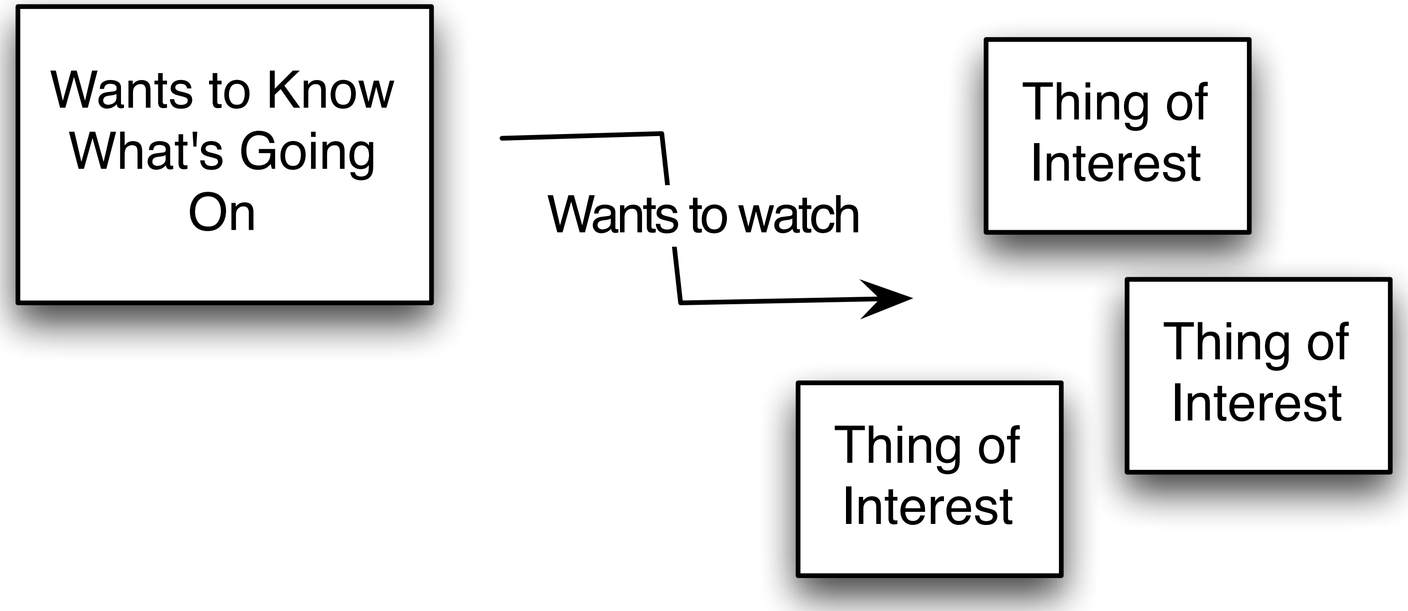
- Software Design Patterns are good ideas .... *For Software Development*
- They give advice for tricky issues that often arise when building software programs.

Design patterns adopted

Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

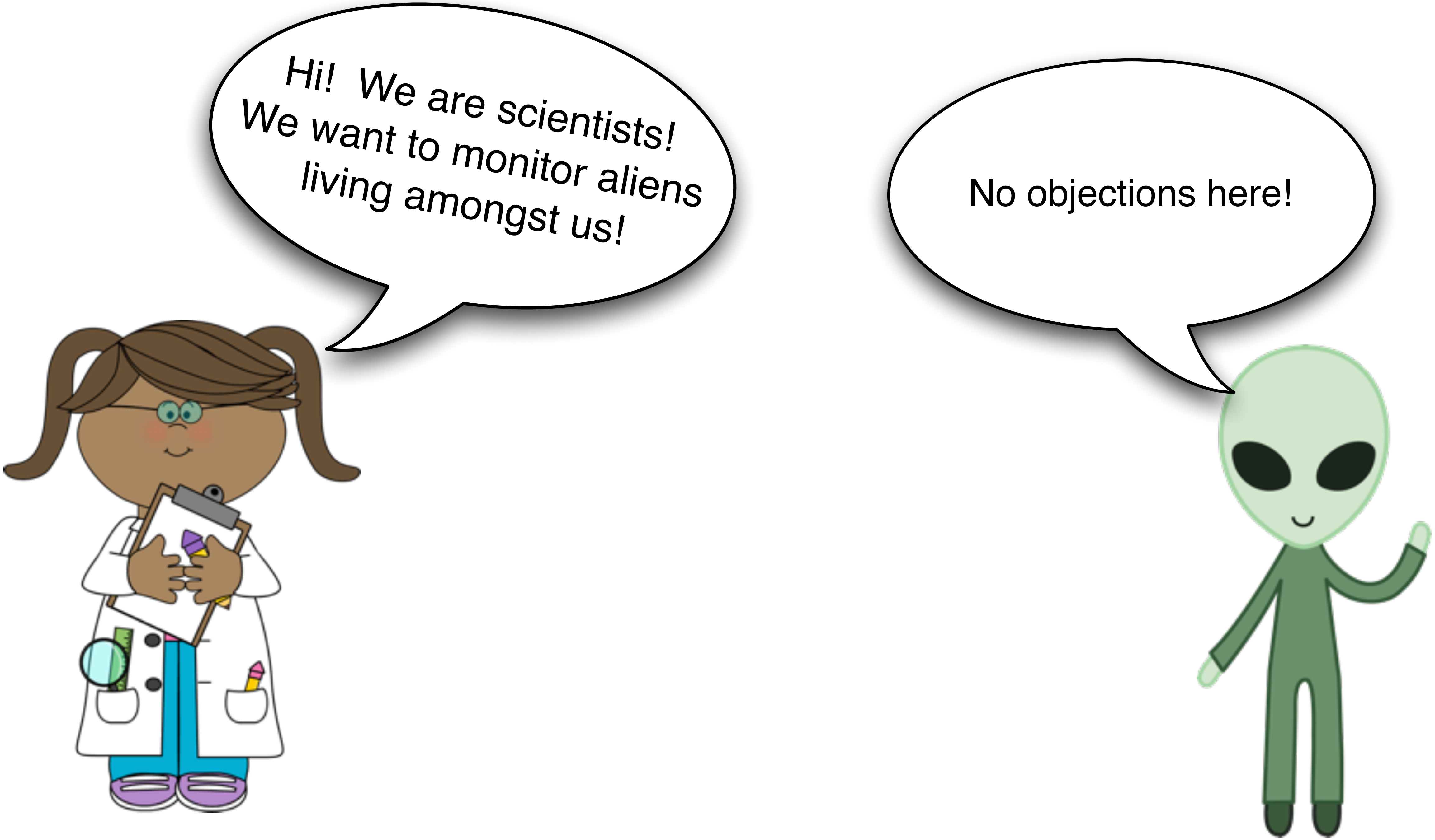
- Software engineers adopted a similar approach for describing common solutions to common problems
  - PROBLEM: intent, motivation, applicability
  - SOLUTION: structure, participants, collaborations, implementation
  - CONSEQUENCES: warnings, known uses, related patterns

For instance:

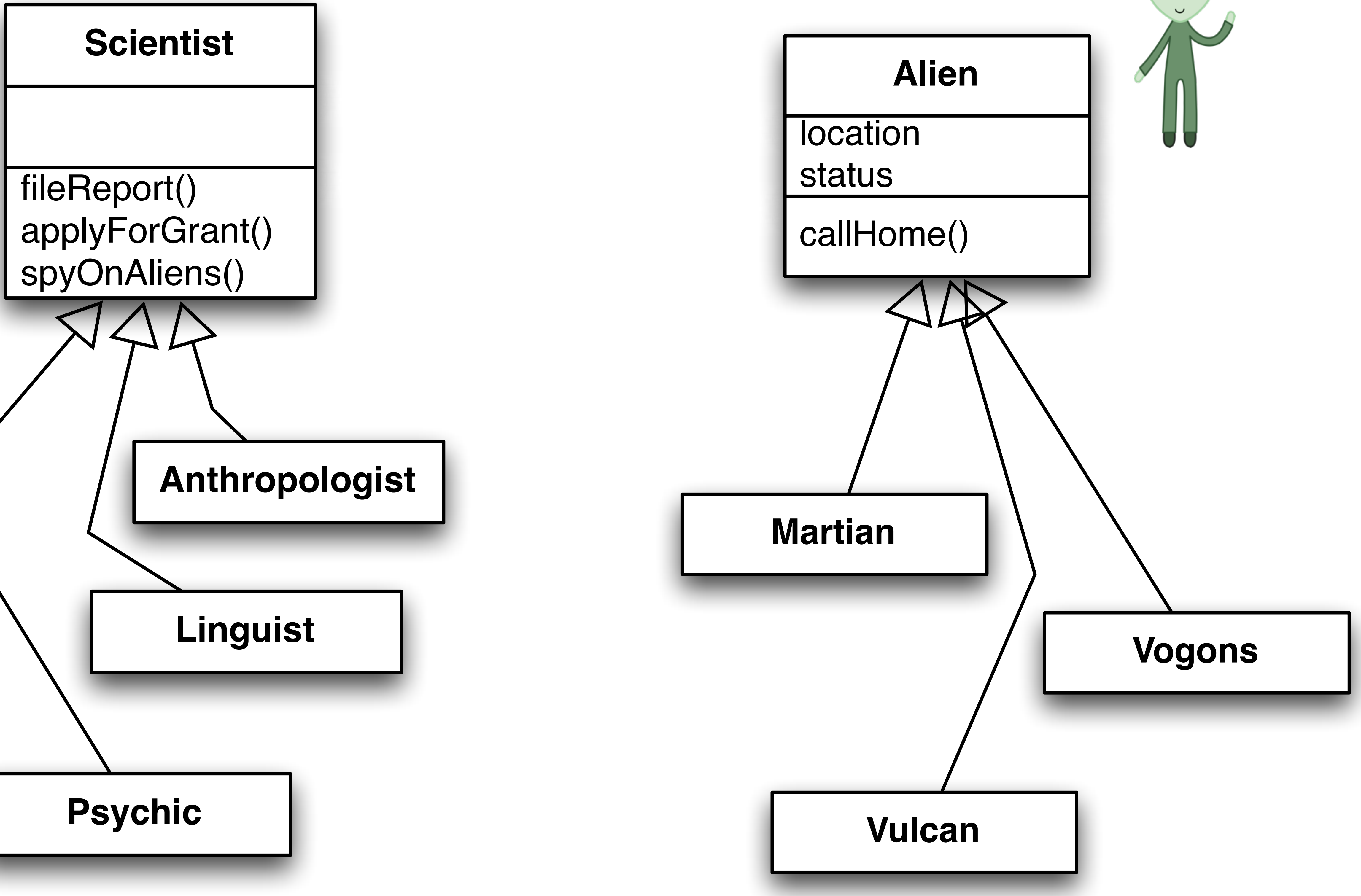


- A very common problem is monitoring, where an object is interested in the status of another.
- In software, this might be...
  - a game controller interested in players in a game
  - sessions in an internet application interested in a central database
  - central process maintaining concurrent threads

We can relate to that issue...


Hi! We are scientists! We want to monitor aliens living amongst us!
No objections here!

Our Basic Design...



At the moment, the scientists are haphazardly spying on the aliens.

How could we design a better protocol?

- Have all the aliens send a signal every time something happens?
- Have them write to a log file?
- Have them send a message when they're in trouble?
- There are so many options! Which is best?

Thankfully... there's a pattern for that!

The Observer Pattern

Observer

where the "**Observer**" watches the "**Subject**"

Subject

yep - this is 210 review

The Observer Pattern

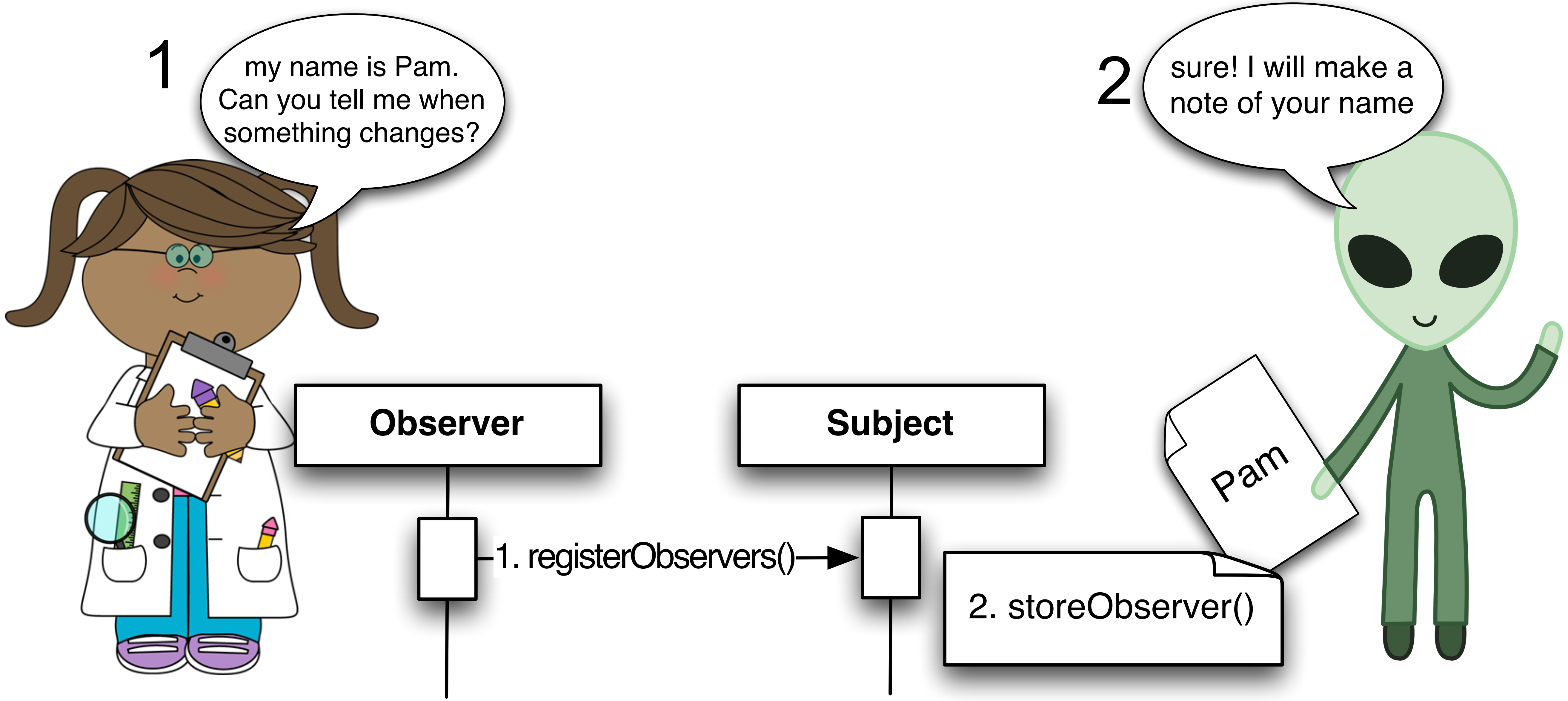
- PROBLEM: an object (the “observer”) wants to watch the status updates of another object (the “subject”)

- SOLUTION: the observer registers with the subject for updates; the subject sends notices to the observer

- CONSEQUENCES: the observer might be overloaded with updates. If so, tailor the design.

Step 1: Register

- The observer has to register with the subject for updates.
- The subject puts them in some kind of list or record, to contact later.


1
my name is Pam. Can you tell me when something changes?
2
sure! I will make a note of your name
Pam

Observer

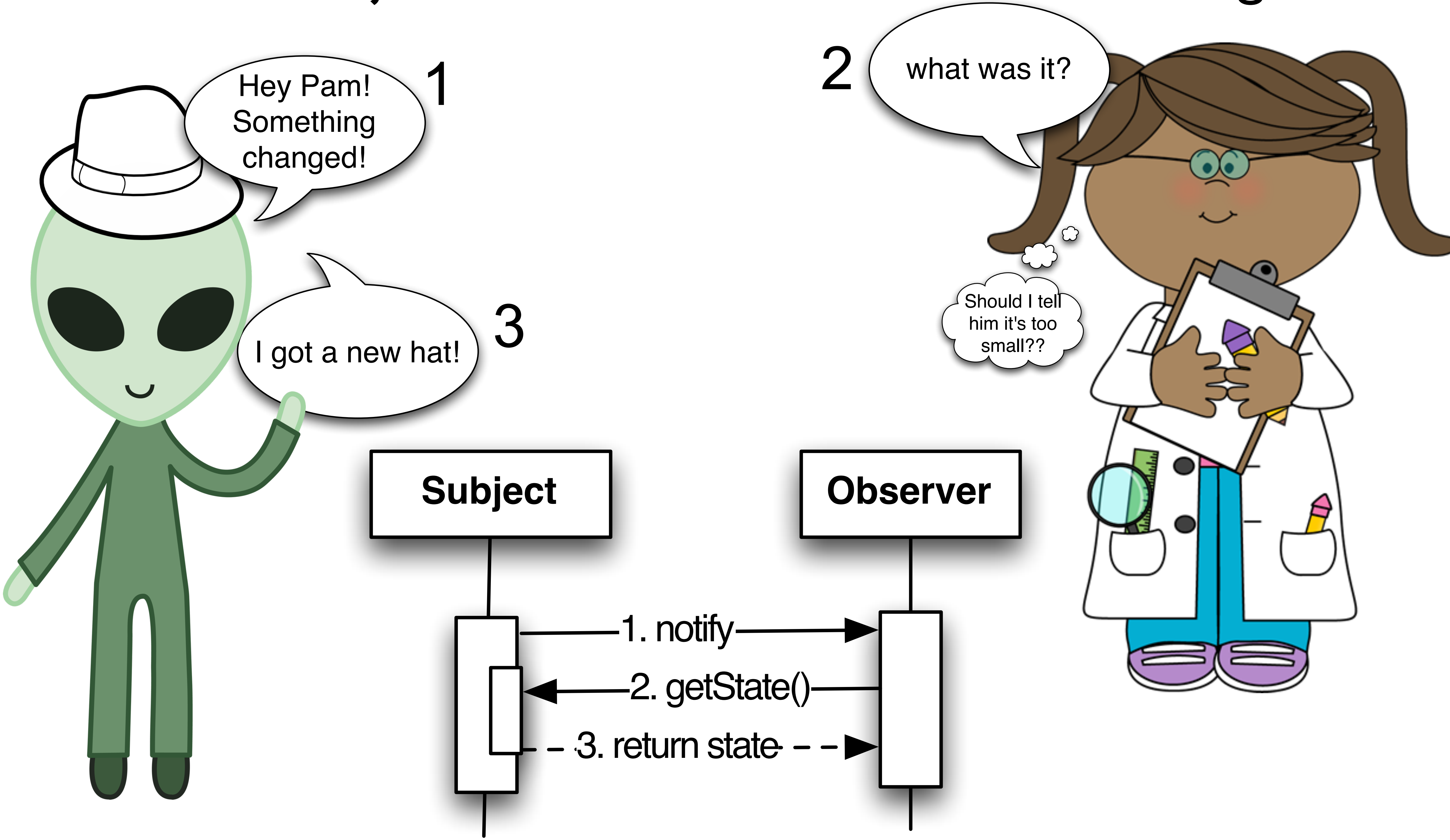
Subject

1. registerObservers()

2. storeObserver()

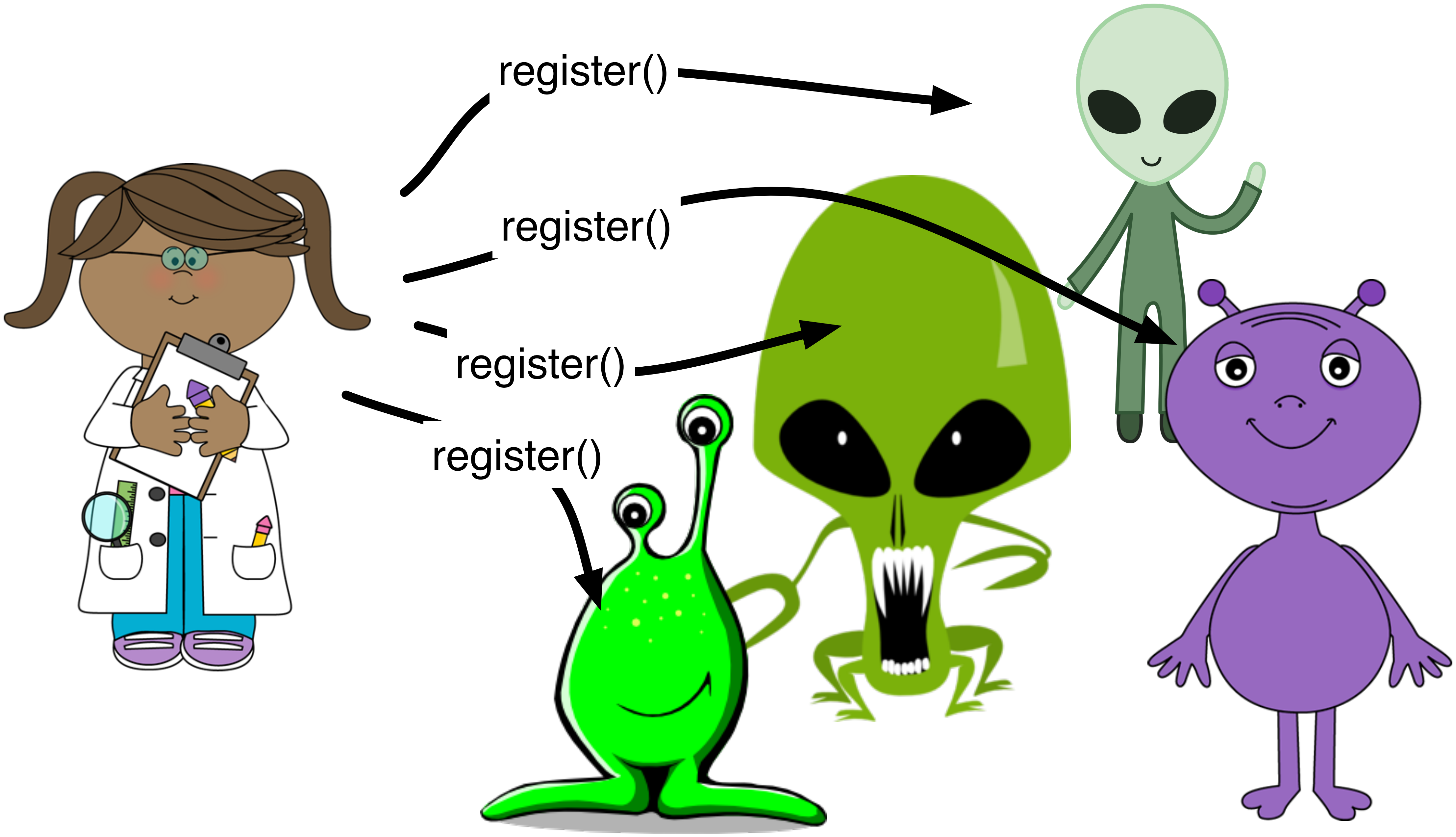
Step 2. Notify Observers!

- The Subject notifies the Observer of a change.



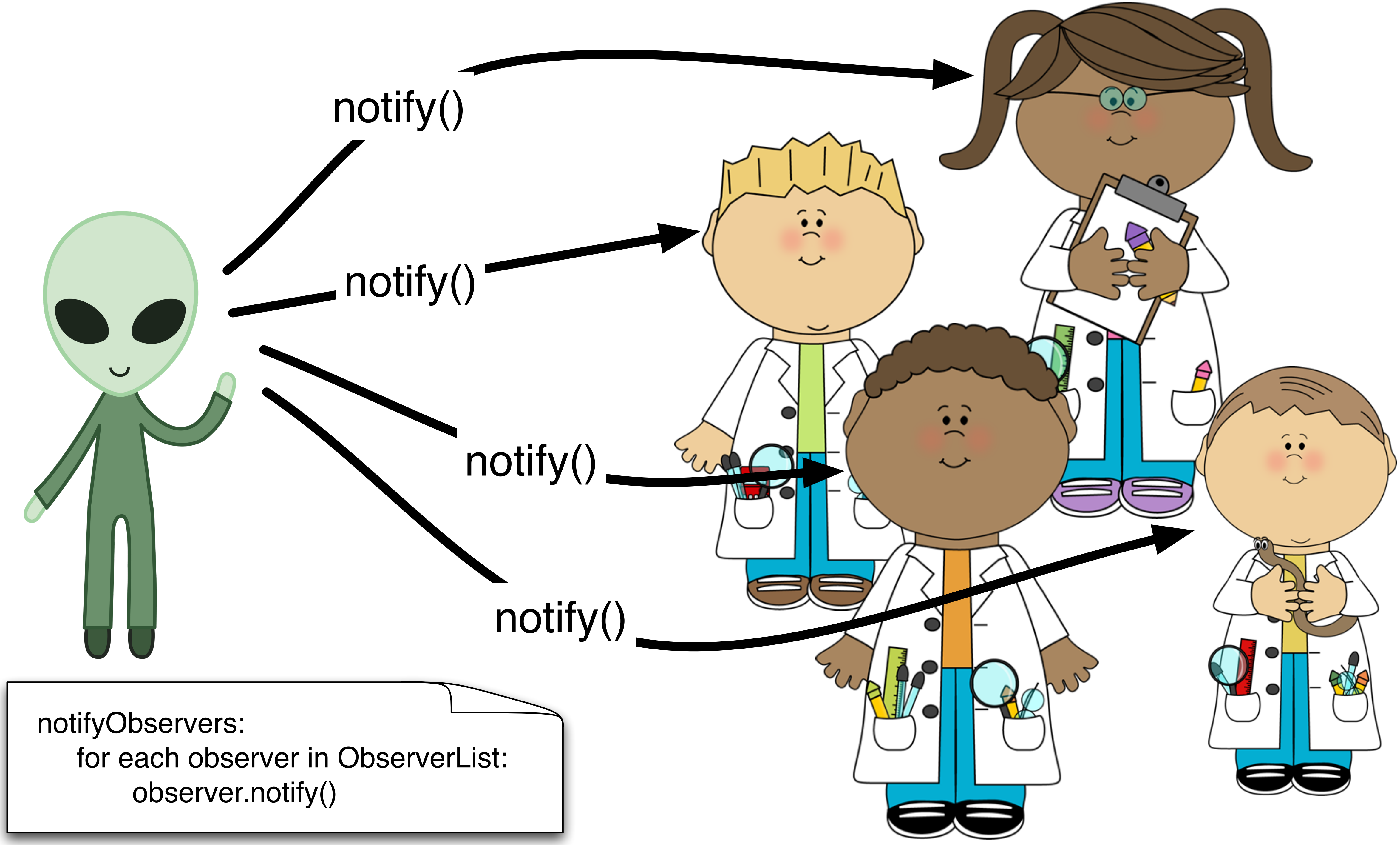
Variation: Lots of Subjects

Observer registers with each one and then waits for them each to notify of changes



Variation: Lots of Observers

The Subject loops through their list of Observers, notifying each one of every change

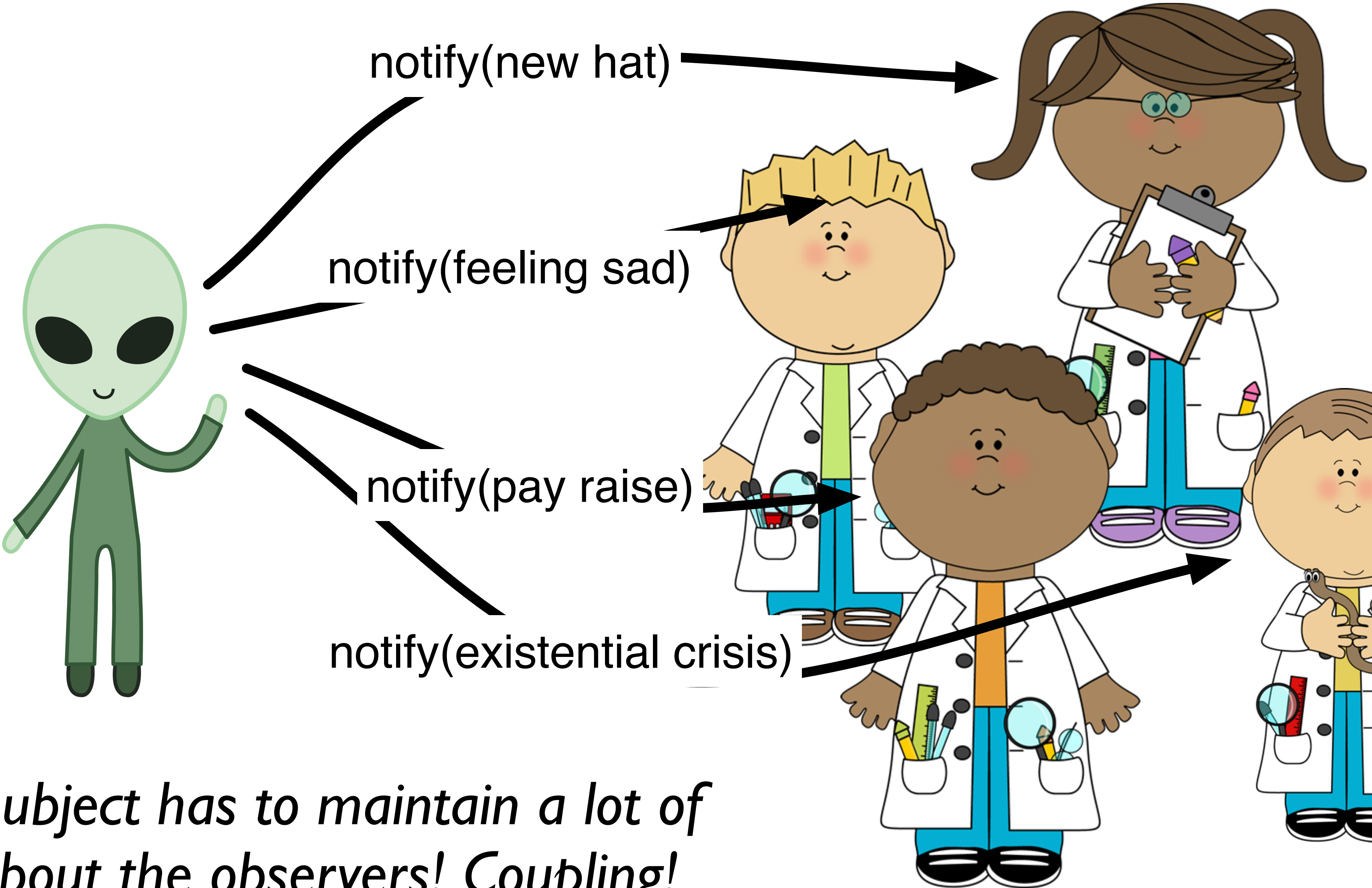


```
notifyObservers:
    for each observer in ObserverList:
        observer.notify()
```

If the Observer is overloaded...

There are some more efficient options for notification

Tailored Push Model: the subject only tells what it knows the observer wants to hear

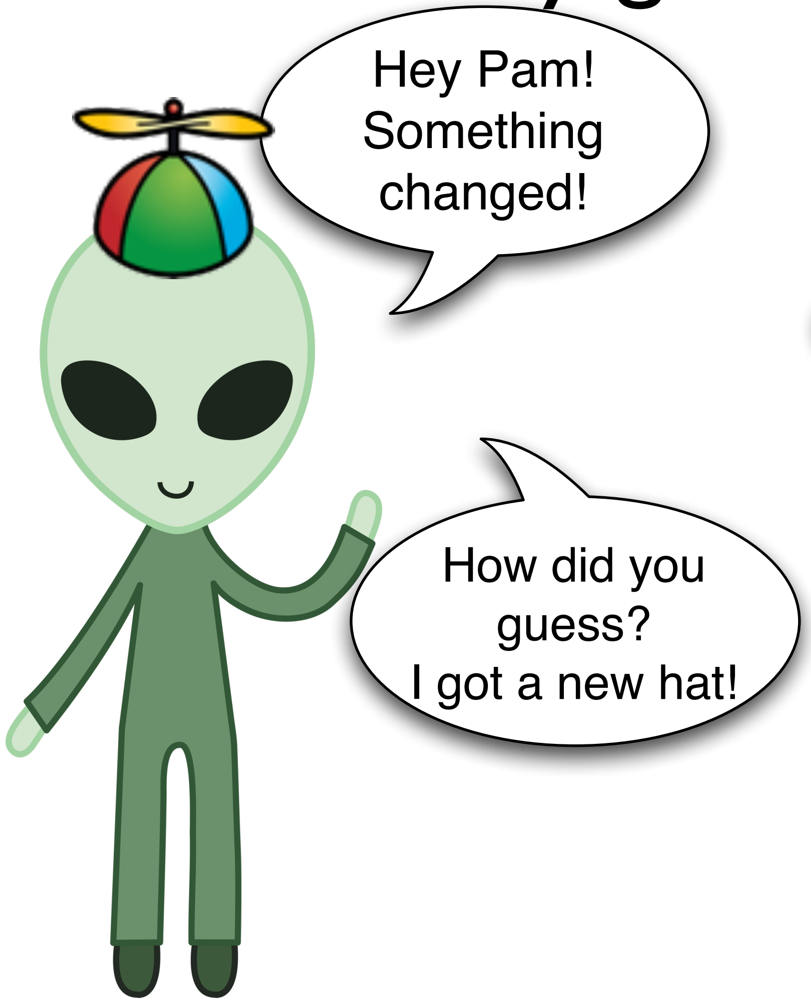


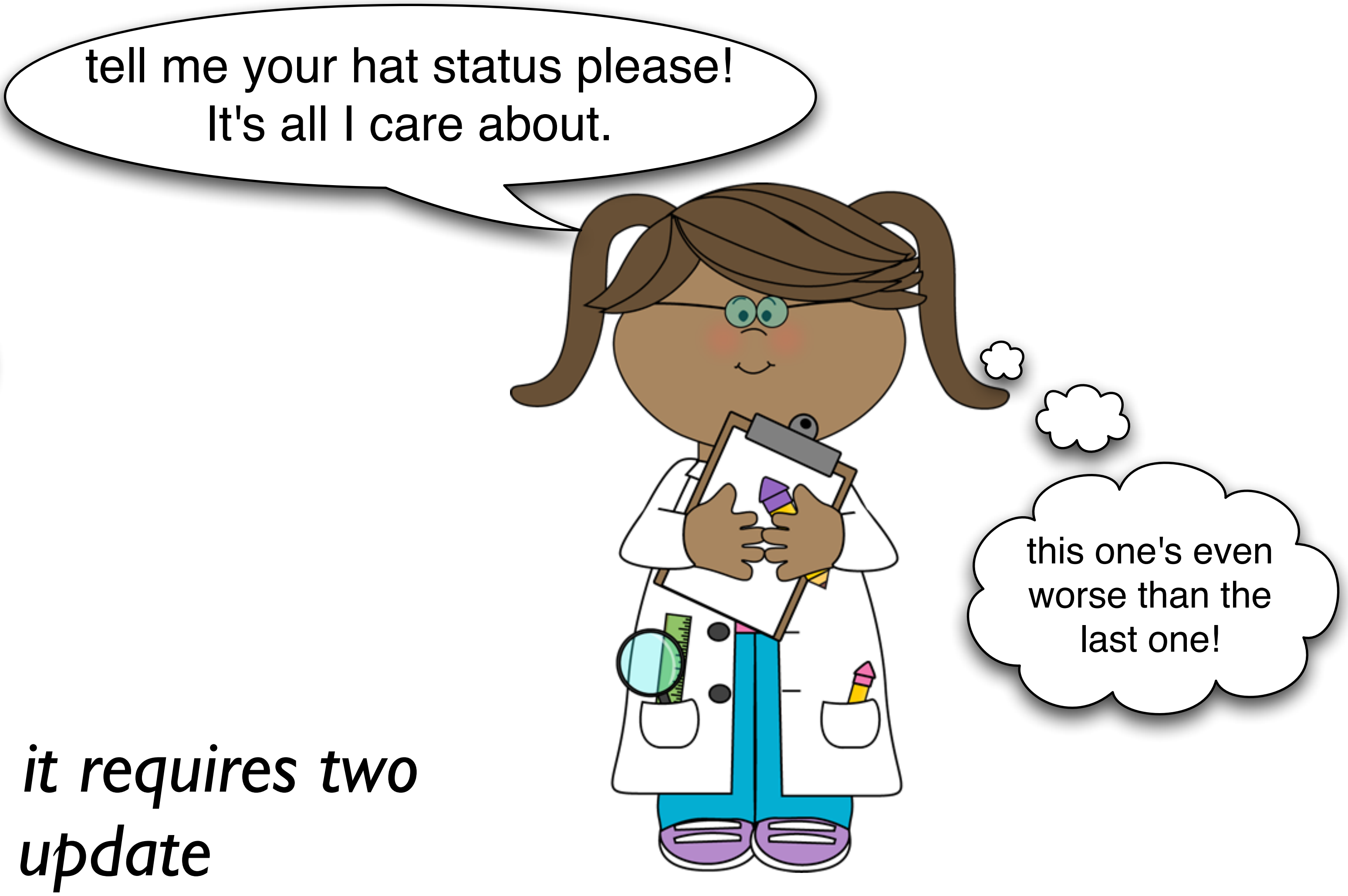
but now the subject has to maintain a lot of knowledge about the observers! Coupling!

If the Observer is overloaded...

There are some more efficient options for notification

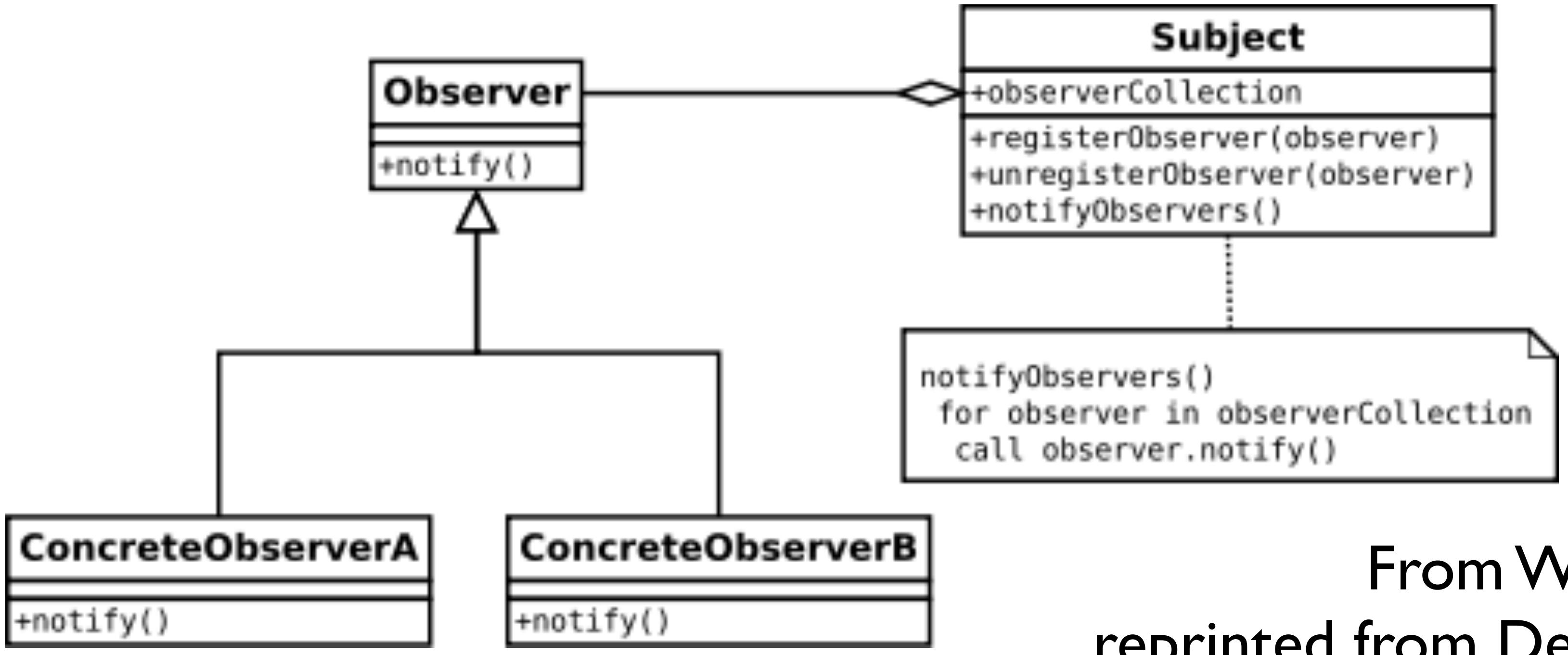
Pull Model: the subject sends a basic "notify" message, but the observer only gets status details of interest.


Hey Pam! Something changed!
How did you guess? I got a new hat!

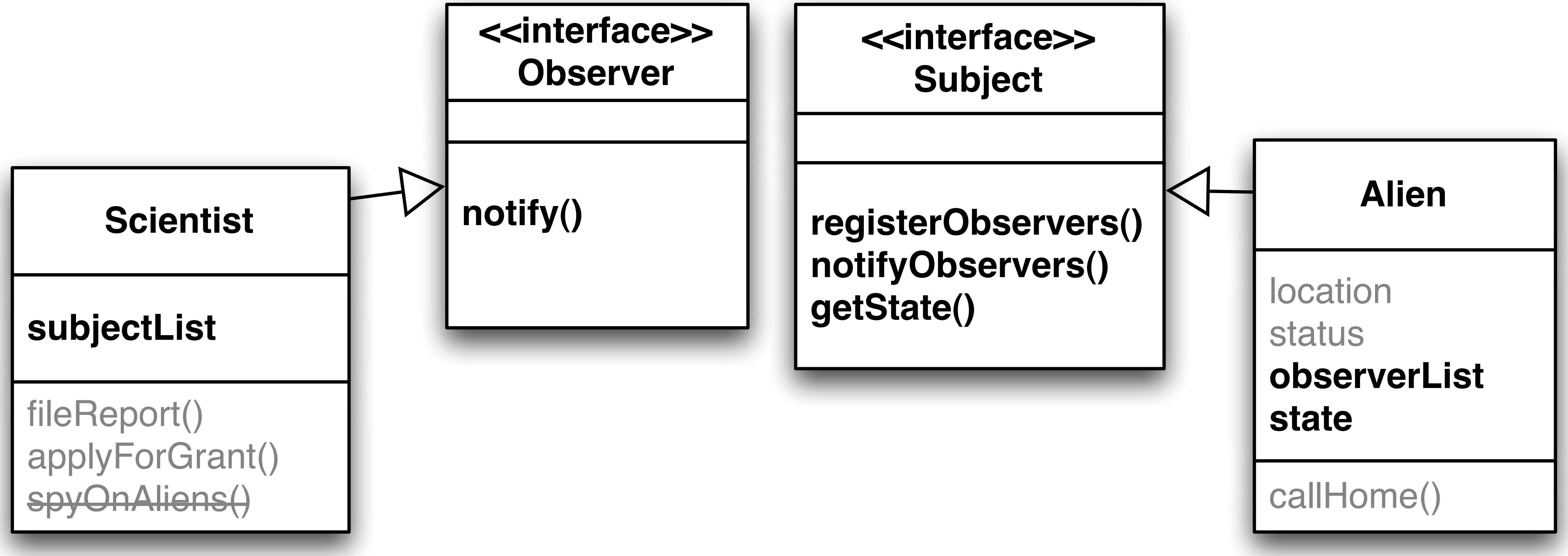

tell me your hat status please! It's all I care about.
this one's even worse than the last one!

but this can be slow, because it requires two calls to get the status update

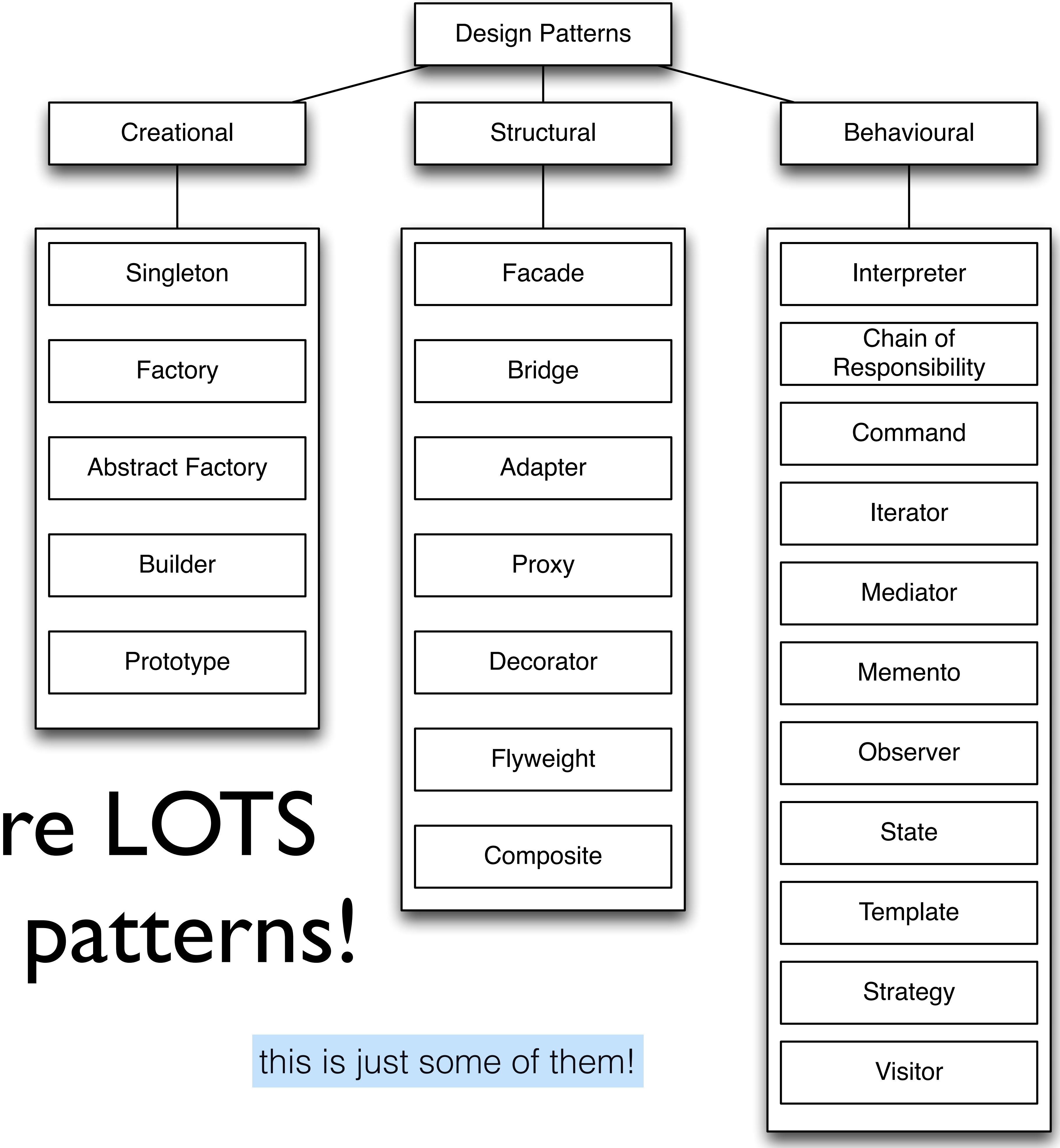
So what's the overall design?



From Wikipedia,
reprinted from Design Patterns book



And there are patterns for other situations too!



- Design Patterns
  - Creational
    - Singleton
    - Factory
    - Abstract Factory
    - Builder
    - Prototype
  - Structural
    - Facade
    - Bridge
    - Adapter
    - Proxy
    - Decorator
    - Flyweight
    - Composite
  - Behavioural
    - Interpreter
    - Chain of Responsibility
    - Command
    - Iterator
    - Mediator
    - Memento
    - Observer
    - State
    - Template
    - Strategy
    - Visitor

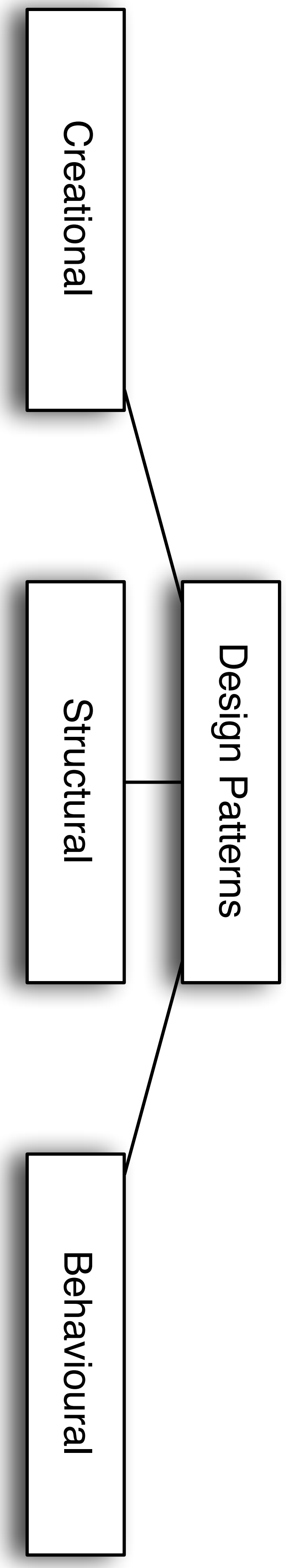
There are LOTS of design patterns!

this is just some of them!

> creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

> structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities

> behavioural design patterns are design patterns that identify common communication patterns between objects and realize these patterns



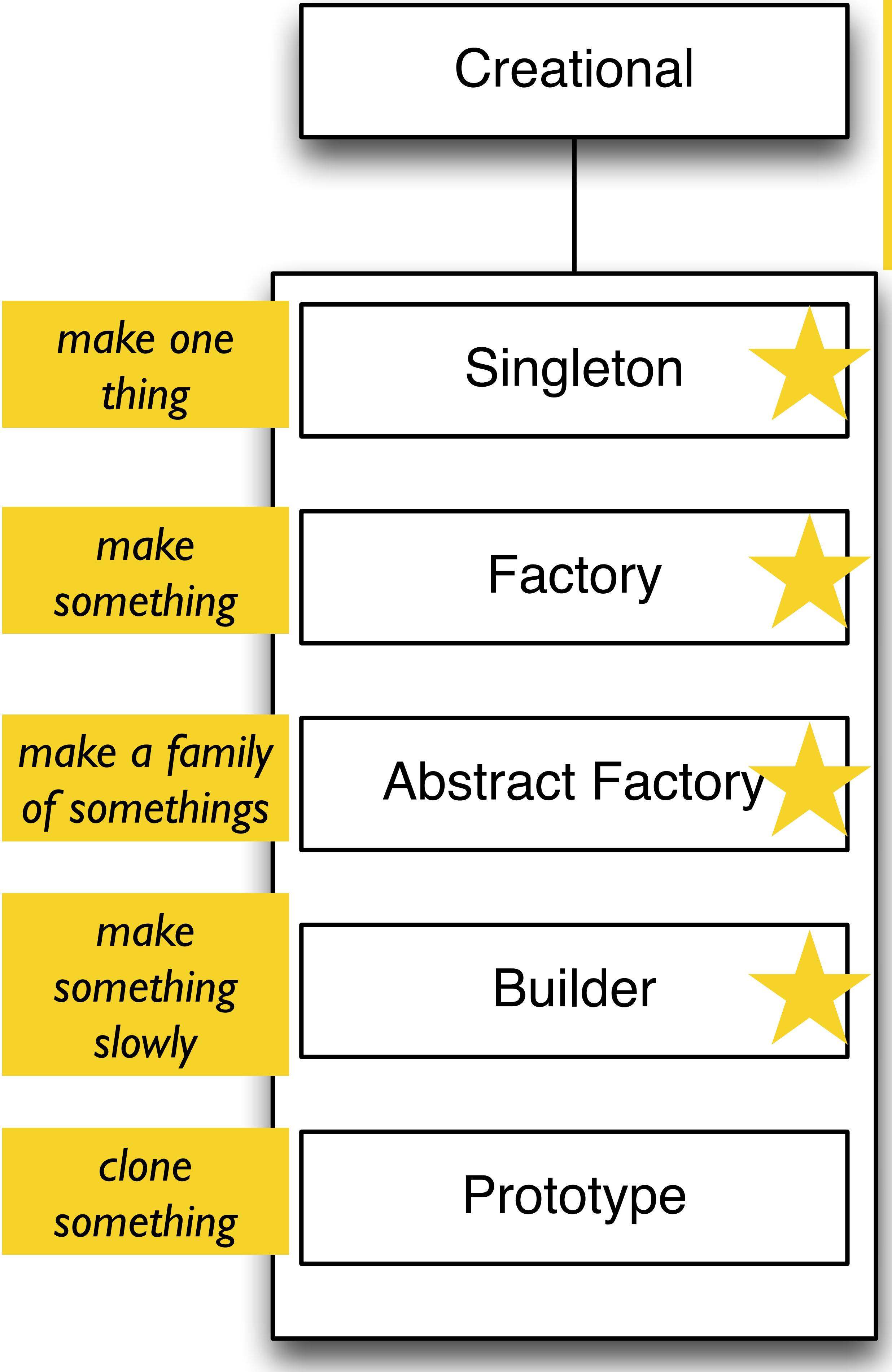
Design Patterns

Creational

Structural

Behavioural

Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed. (GoF, p136)



Creational

make one thing

Singleton

One instance of a class, and *only* one instance!

make something

Factory

AKA Factory Method pattern (because it's a method) A "make it" method that returns objects of another class, made all in one go, but with some collaboration needed right then.

make a family of somethings

Abstract Factory

A class (that makes use of factories) that can be extended to make families of objects that all need to share a consistent theme.

make something slowly

Builder

Allows incremental creation of an object over the running of a system with the object being produced at the end of a long customisation process.

clone something

Prototype

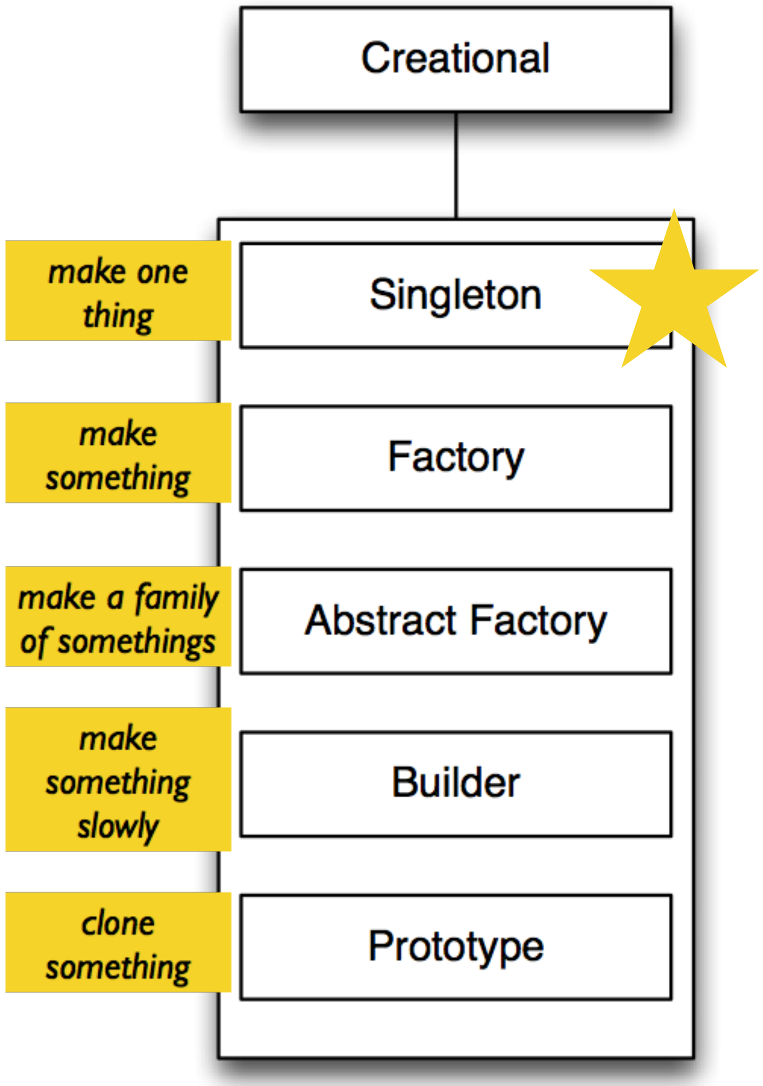
Allows production of clones of a particular object instance.

Sample Problem

- You need to create a class to manage preferences. In order to maintain consistency, there should only ever be one instance of this class. How can you ensure that only one instance of a class is instantiated?

(Question: How could your preferences become inconsistent if your class was instantiated more than once?)

Singleton



Creational

make one thing — Singleton

make something — Factory

make a family of somethings — Abstract Factory

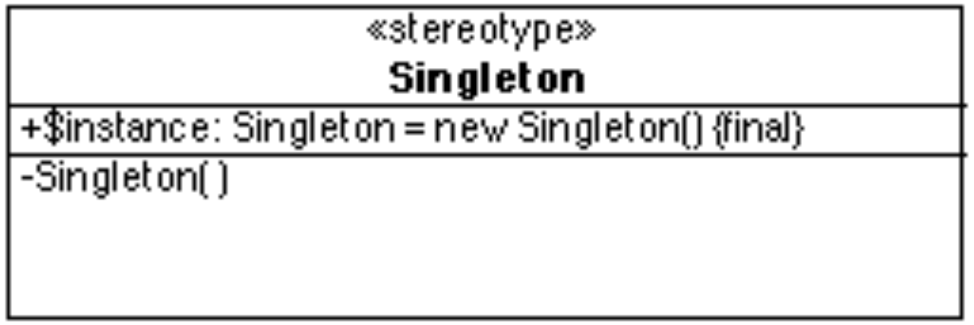
make something slowly — Builder

clone something — Prototype

Name: Singleton

Intent: Make sure a class has a single point of access and is globally accessible (*i.e. Filesystem, Display, PreferenceManager*…)

Participants & Structure:



«stereotype»
Singleton

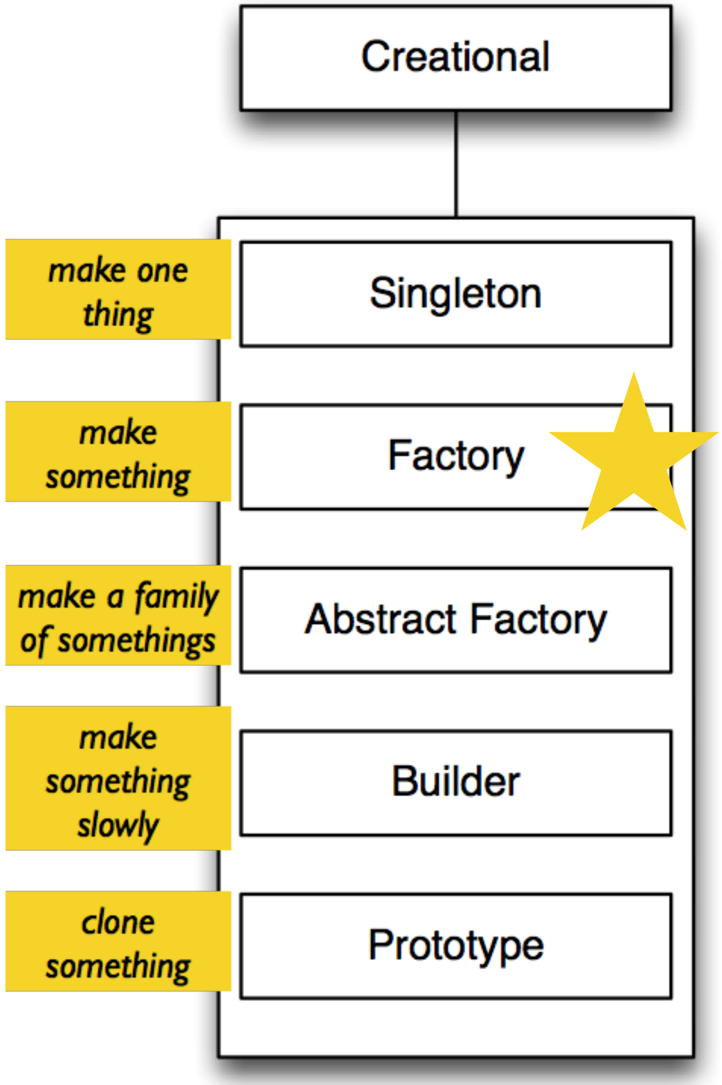
+$instance: Singleton = new Singleton() {final}

-Singleton( )

Singleton Example

```
private static Singleton uniqueInstance = null;
public static Singleton getInstance() {
        if (uniqueInstance == null)
                uniqueInstance = new Singleton();
        return uniqueInstance;
}
// Make sure constructor is private!
private Singleton() {…}
```

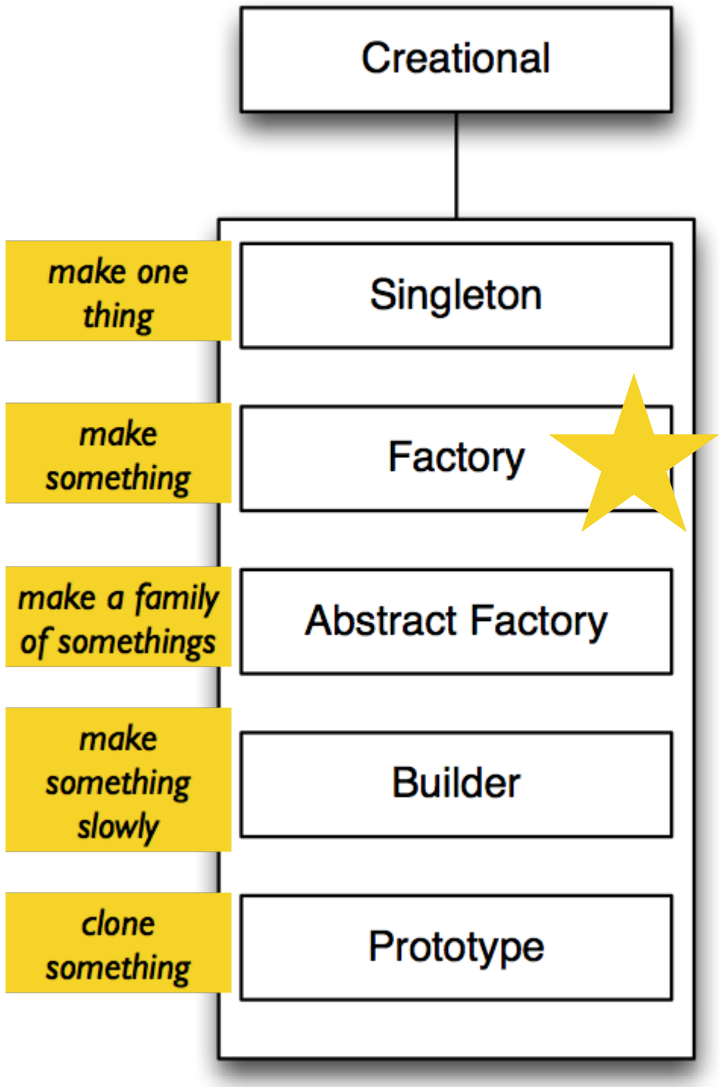
otherwise anyone can make one!



We want to make an object, but its construction is complicated, and is practically a "responsibility" in and of its own!

Seems like we need some separation of responsibilities…

Simple Factory



Creational

make one thing Singleton

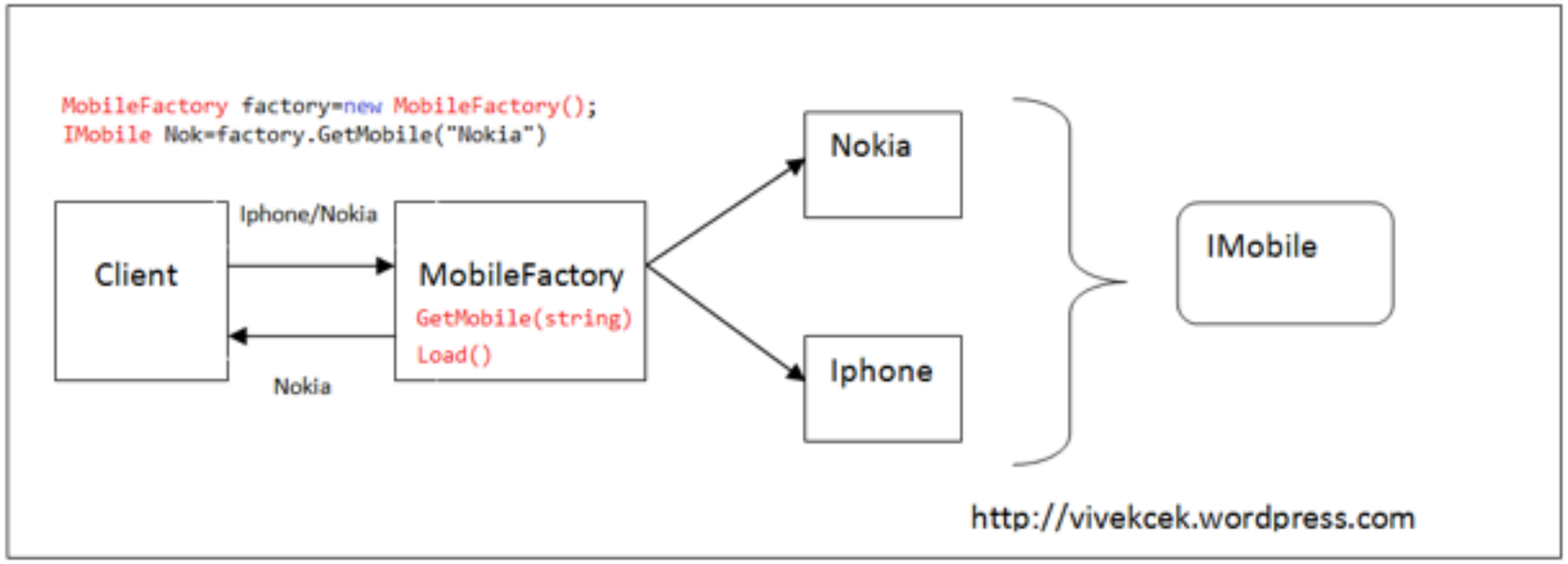
make something Factory

make a family of somethings Abstract Factory

make something slowly Builder

clone something Prototype

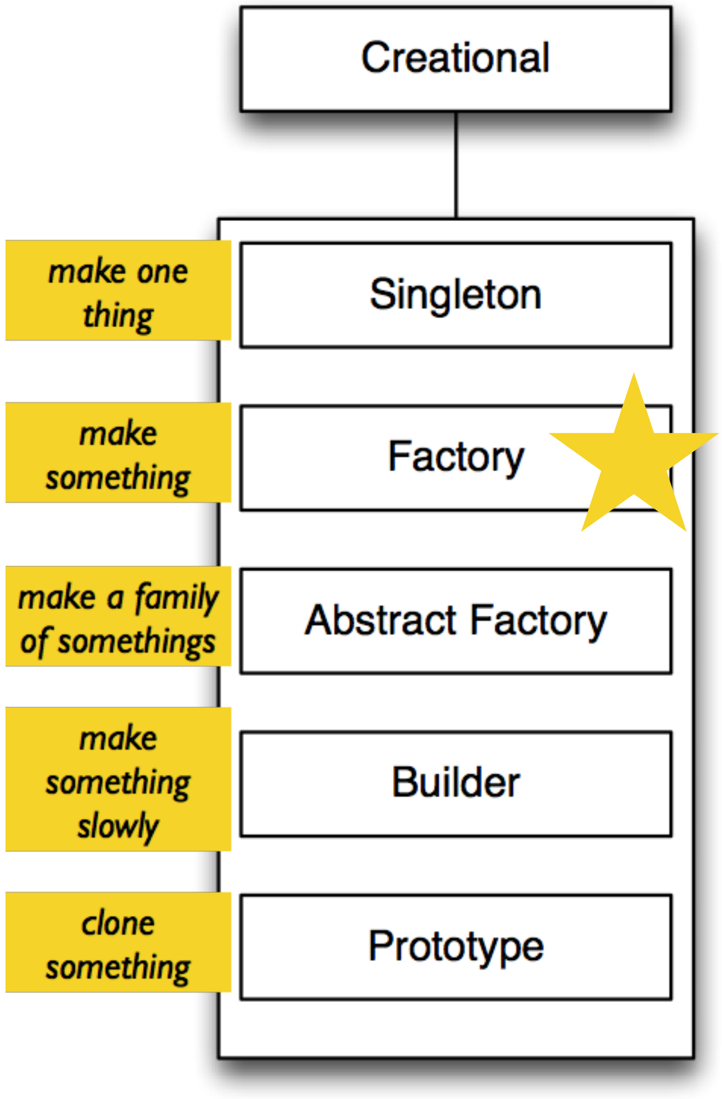
Creates objects without exposing the instantiation logic to the client.
Refers to the newly created object through a common interface



```
MobileFactory factory=new MobileFactory();
IMobile Nok=factory.GetMobile("Nokia")
```

http://vivekcek.wordpress.com

Factories can be used when:
1. The creation of an object makes reuse impossible without significant duplication of code.
2. The creation of an object requires access to information or resources that should not be contained within the composing class.
3. The lifetime management of the generated objects must be centralized to ensure a consistent behavior within the application.



Creational

make one thing — Singleton

make something — Factory

make a family of somethings — Abstract Factory

make something slowly — Builder

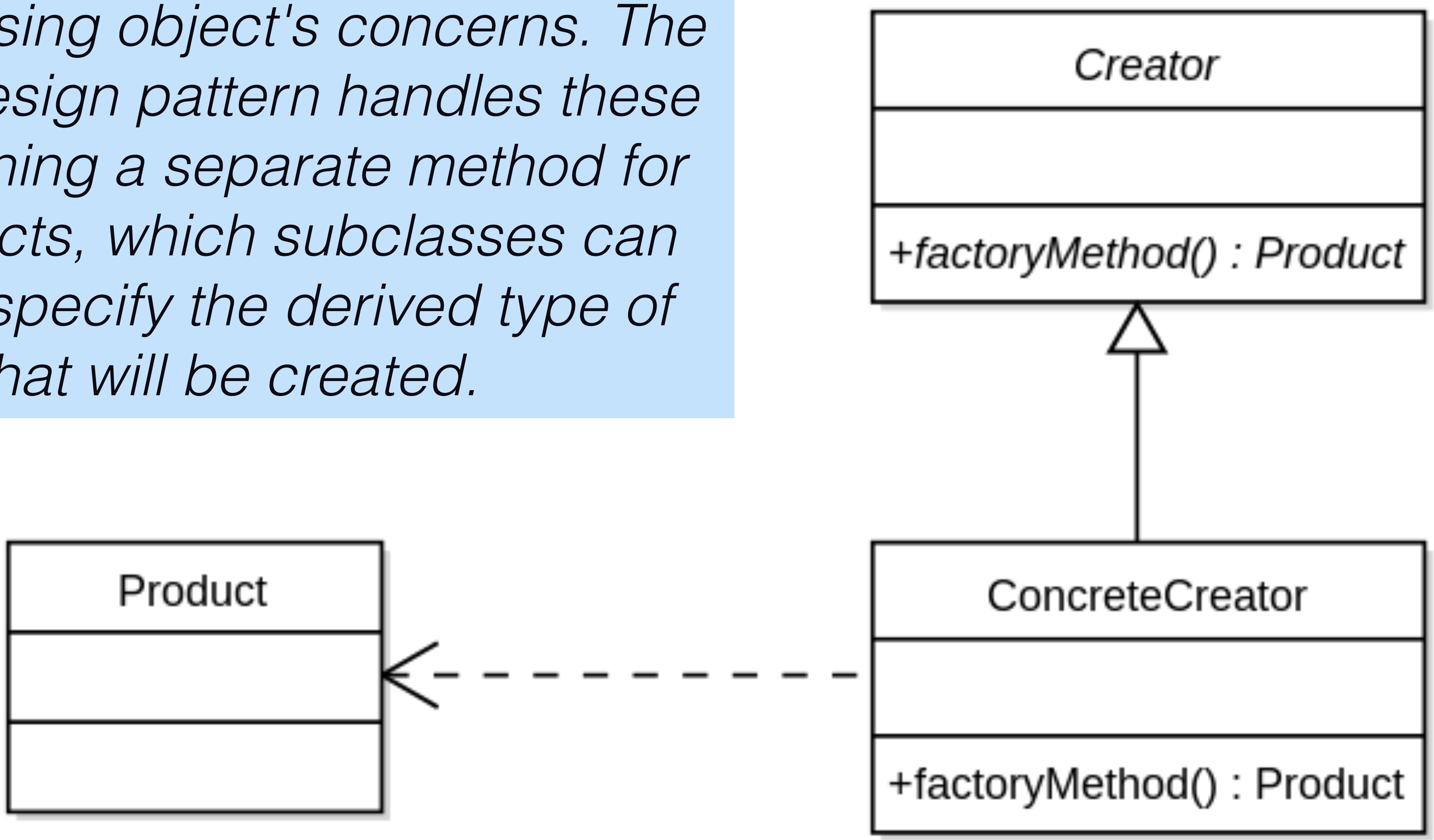
clone something — Prototype

But now I want to be able to decide later (at runtime or compile time) what's actually created!

Factory Method Pattern

> *Creating an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.*

http://en.wikipedia.org/wiki/Factory_method

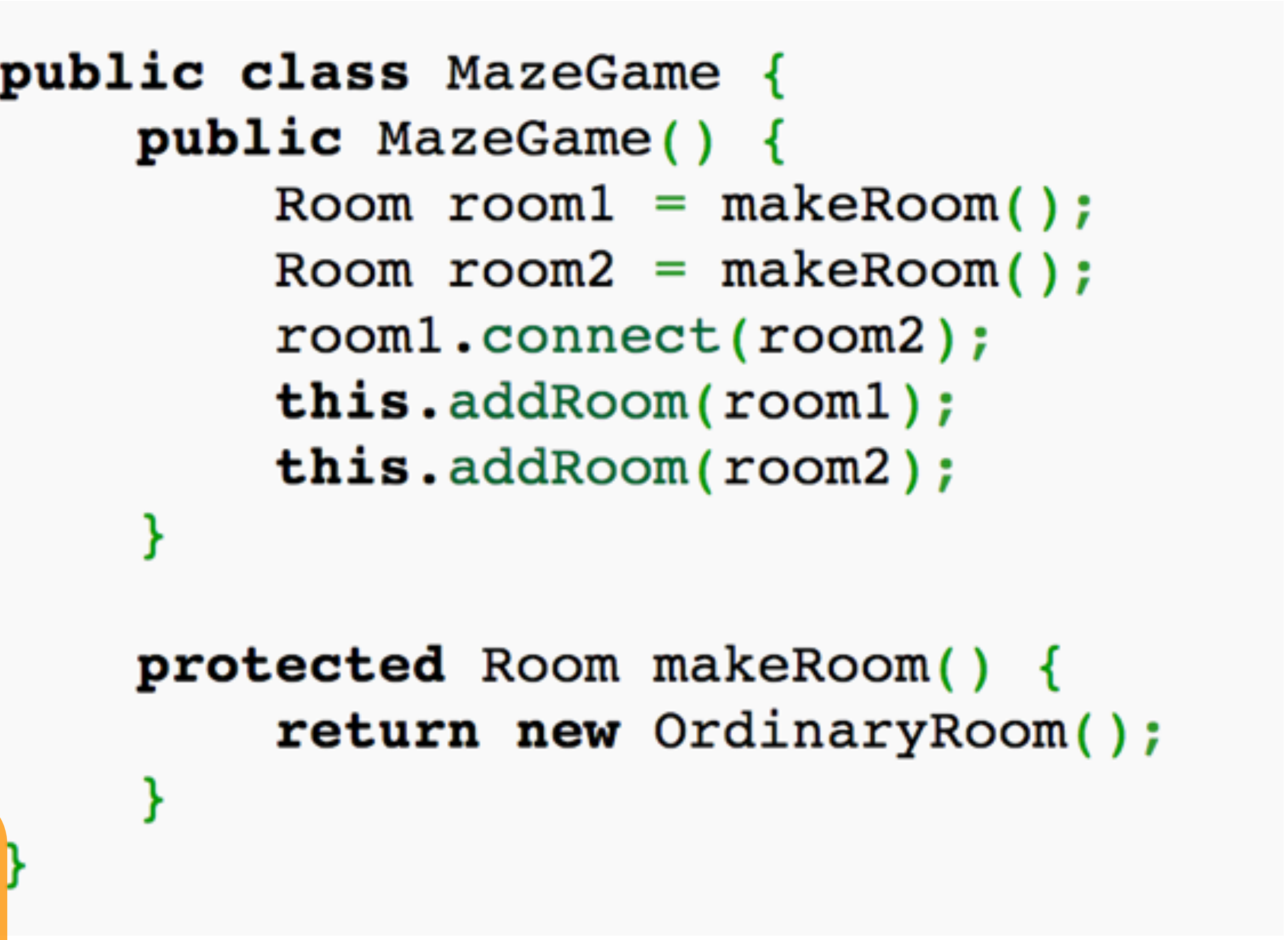


Factory Method Pattern

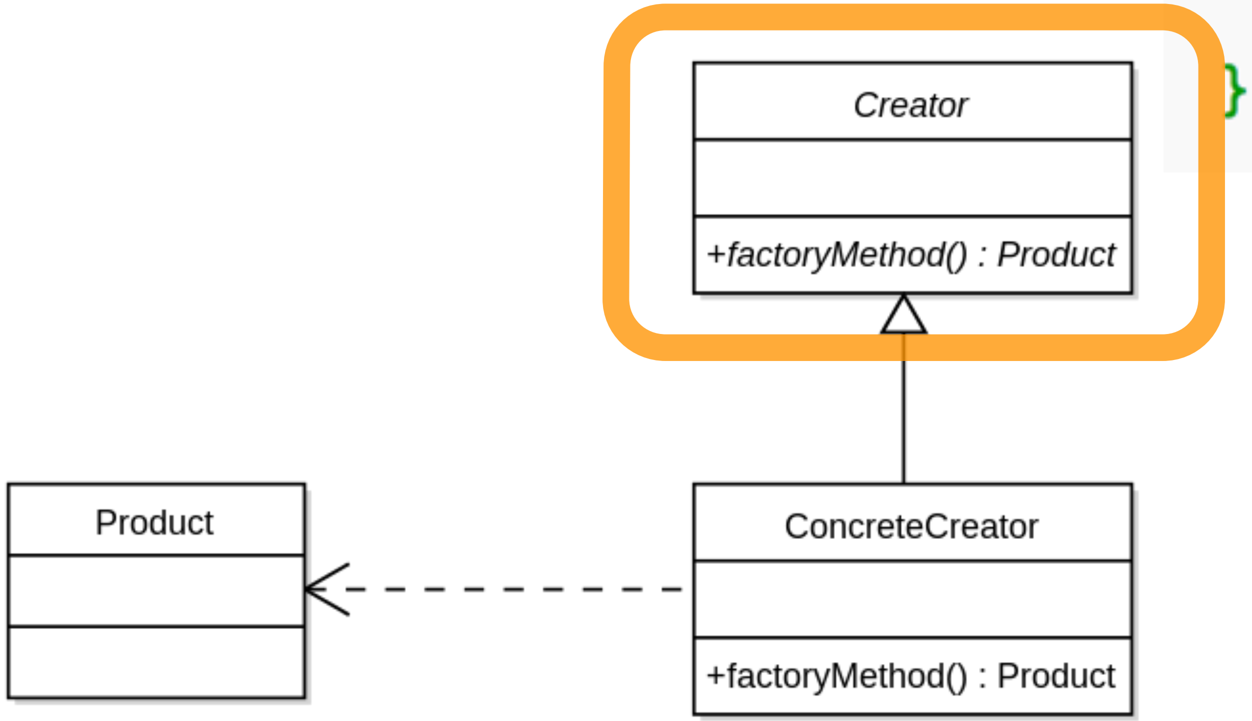
http://en.wikipedia.org/wiki/Factory_method

the factory method pattern is a creational pattern which uses simple factory methods to deal with the problem of creating objects without specifying the exact class of object that will be created.

A maze game may be played in two modes, one with regular rooms that are only connected with adjacent rooms, and one with magic rooms that allow players to be transported at random



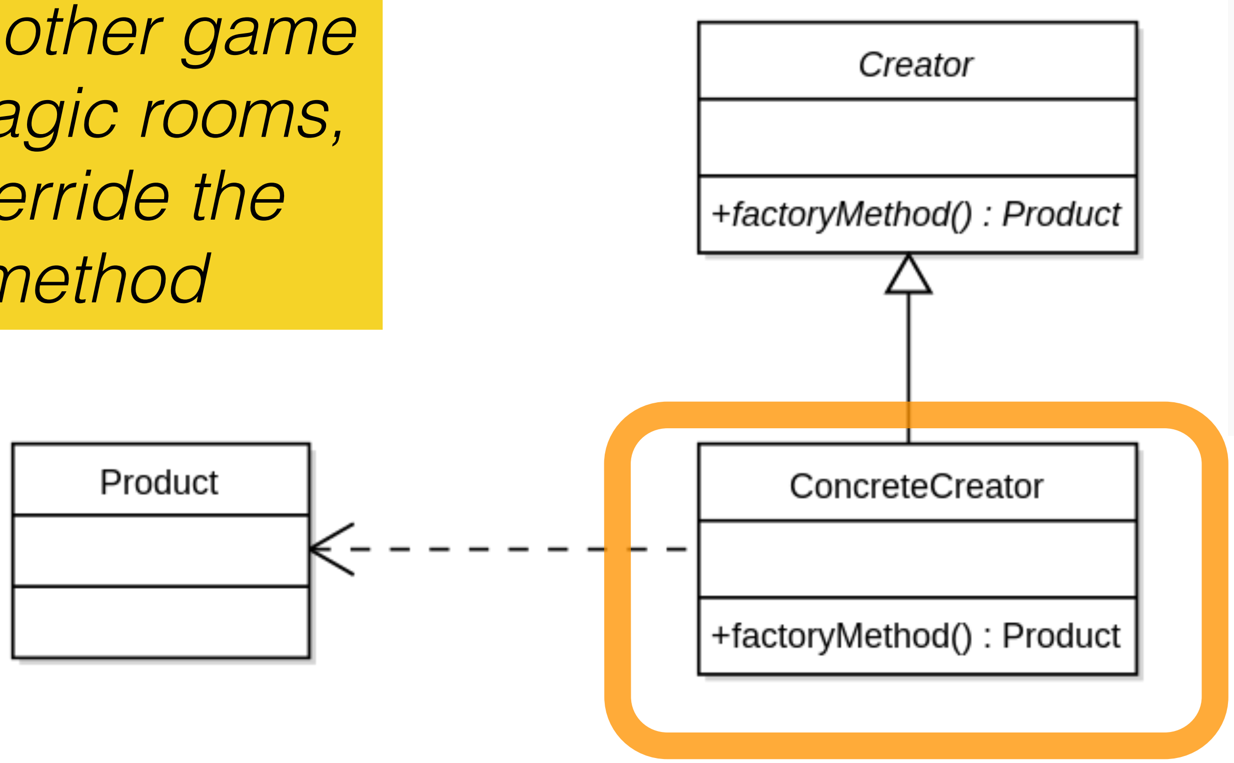
```
public class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
```

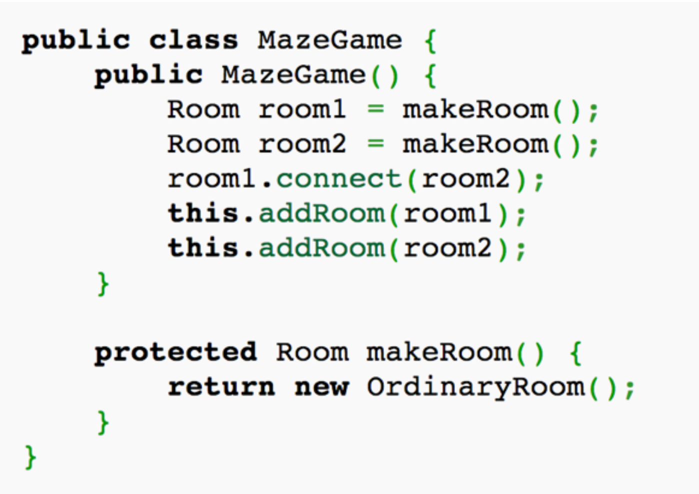


Factory Method Pattern

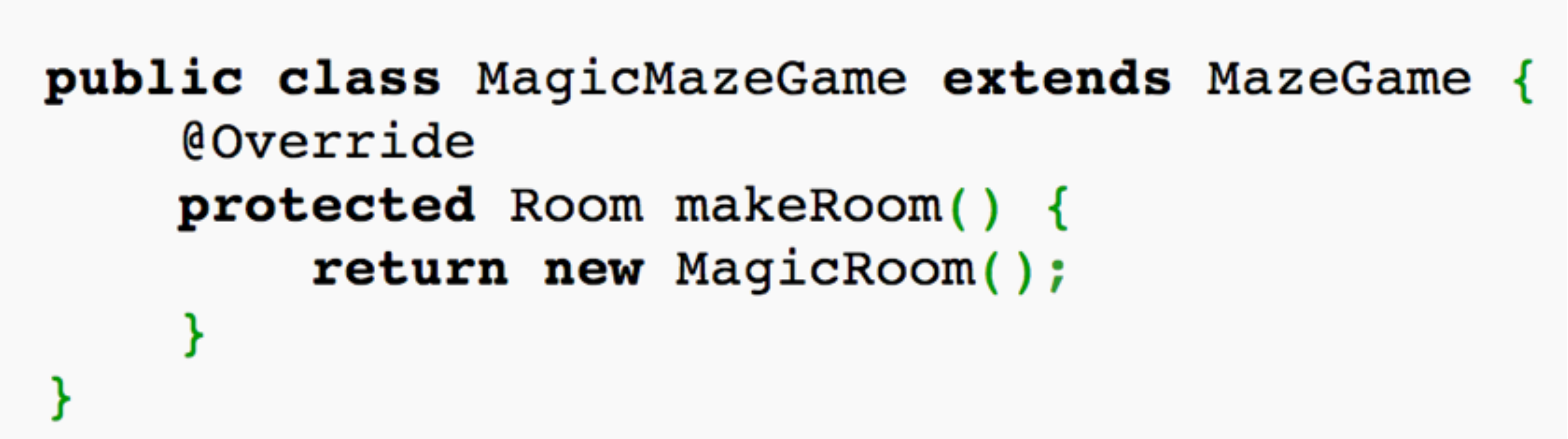
http://en.wikipedia.org/wiki/Factory_method

To implement the other game mode that has magic rooms, it suffices to override the makeRoom method

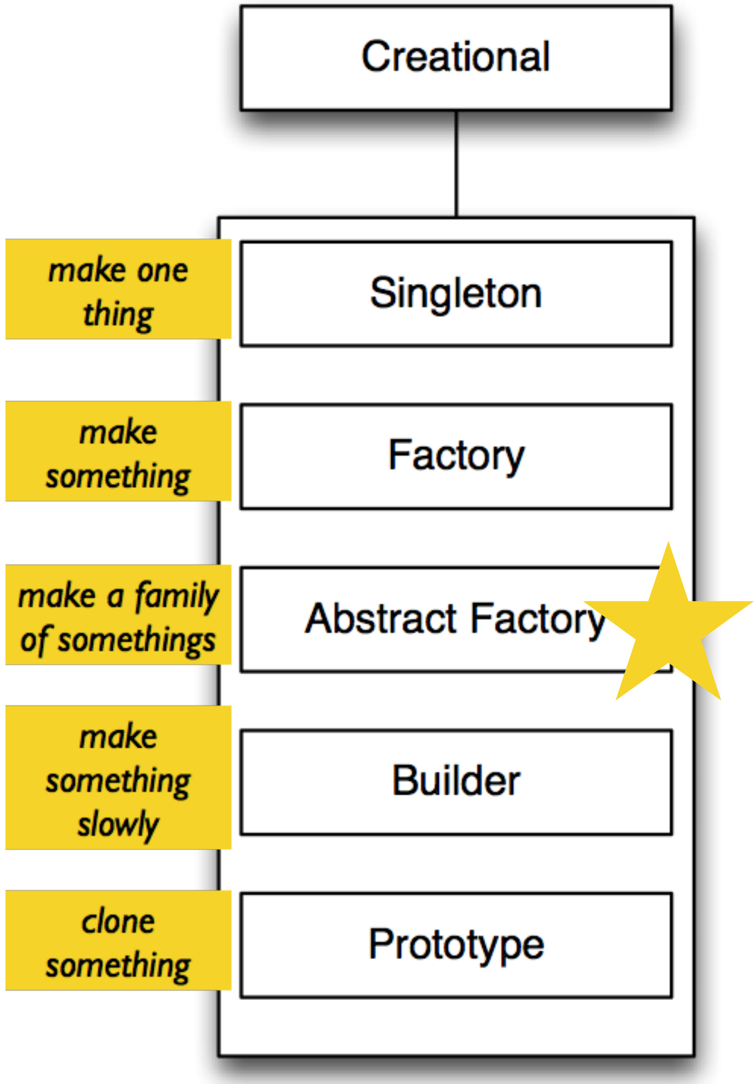




```
public class MazeGame {
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        this.addRoom(room1);
        this.addRoom(room2);
    }

    protected Room makeRoom() {
        return new OrdinaryRoom();
    }
}
```



```
public class MagicMazeGame extends MazeGame {
    @Override
    protected Room makeRoom() {
        return new MagicRoom();
    }
}
```



We want to make themes of classes — and we don't want to have to keep track of all the factory methods by hand … so what do we do?

Seems like we need some more abstraction…

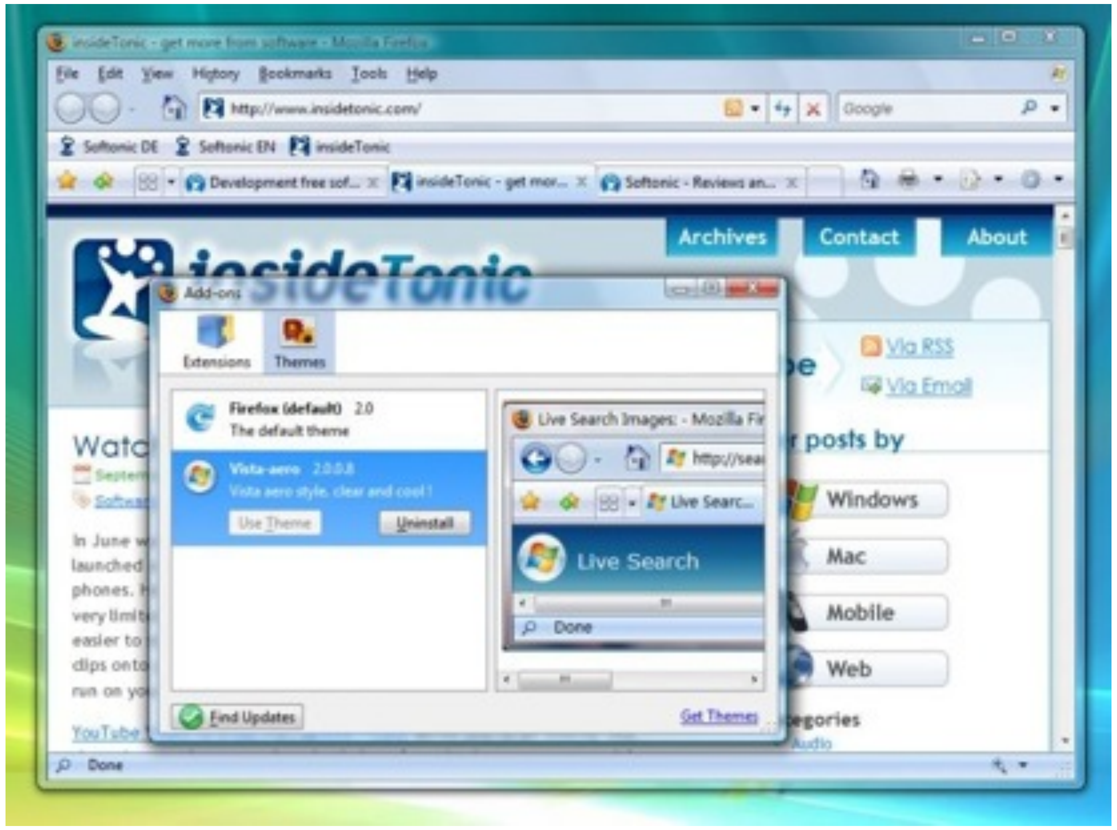
Abstract Factory Intent

- “Provide an interface for creating families of related or dependent objects without specifying their concrete classes.”
  - enforce Client to create sets of related objects
  - ensure Client doesn’t mix incompatible objects

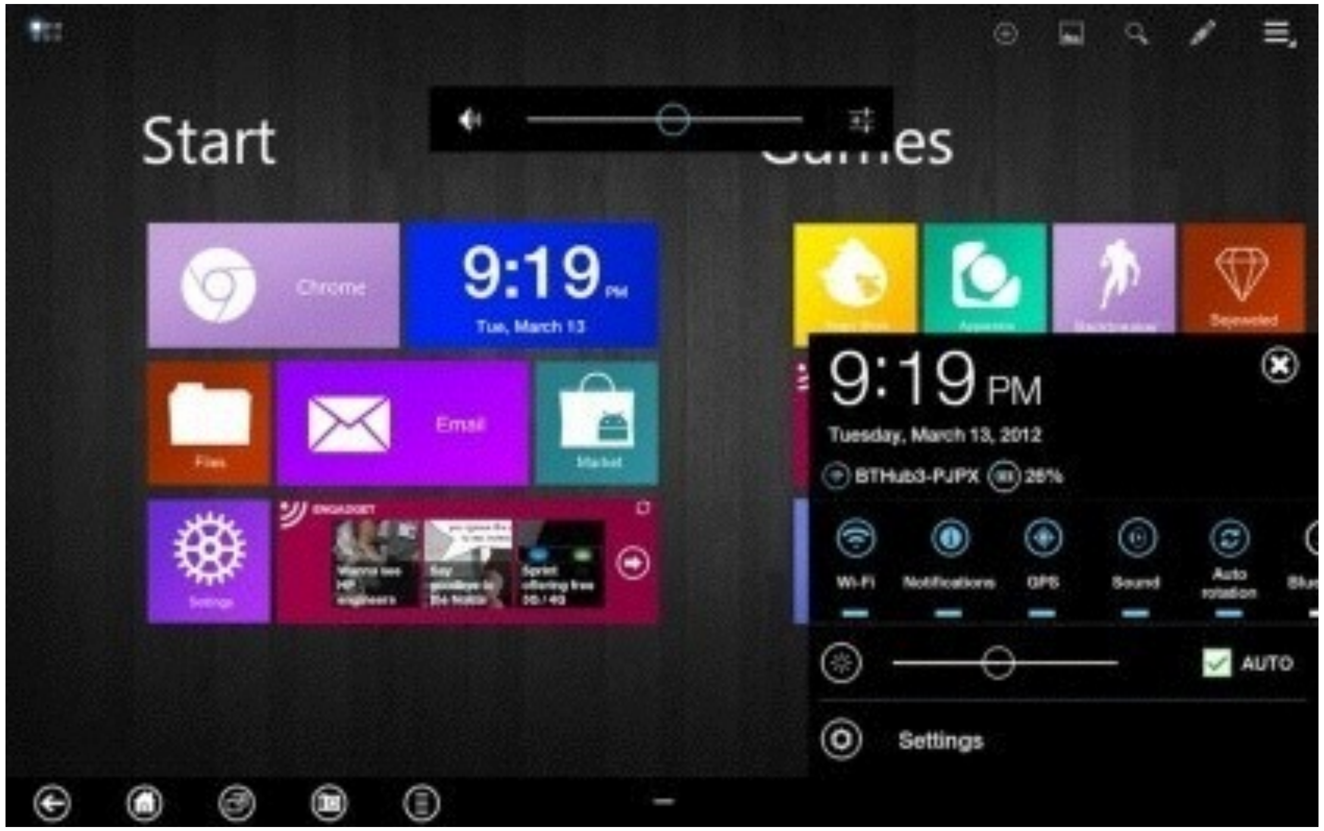
Abstract Factory Example

- Programmers use GUI toolkits like Swing to create and compose widgets
- Metro and Aero are two “desktop themes” which give a GUI a different look and feel
- Want to provide a WidgetFactory which will ensure that all GUI elements are created according to a single theme

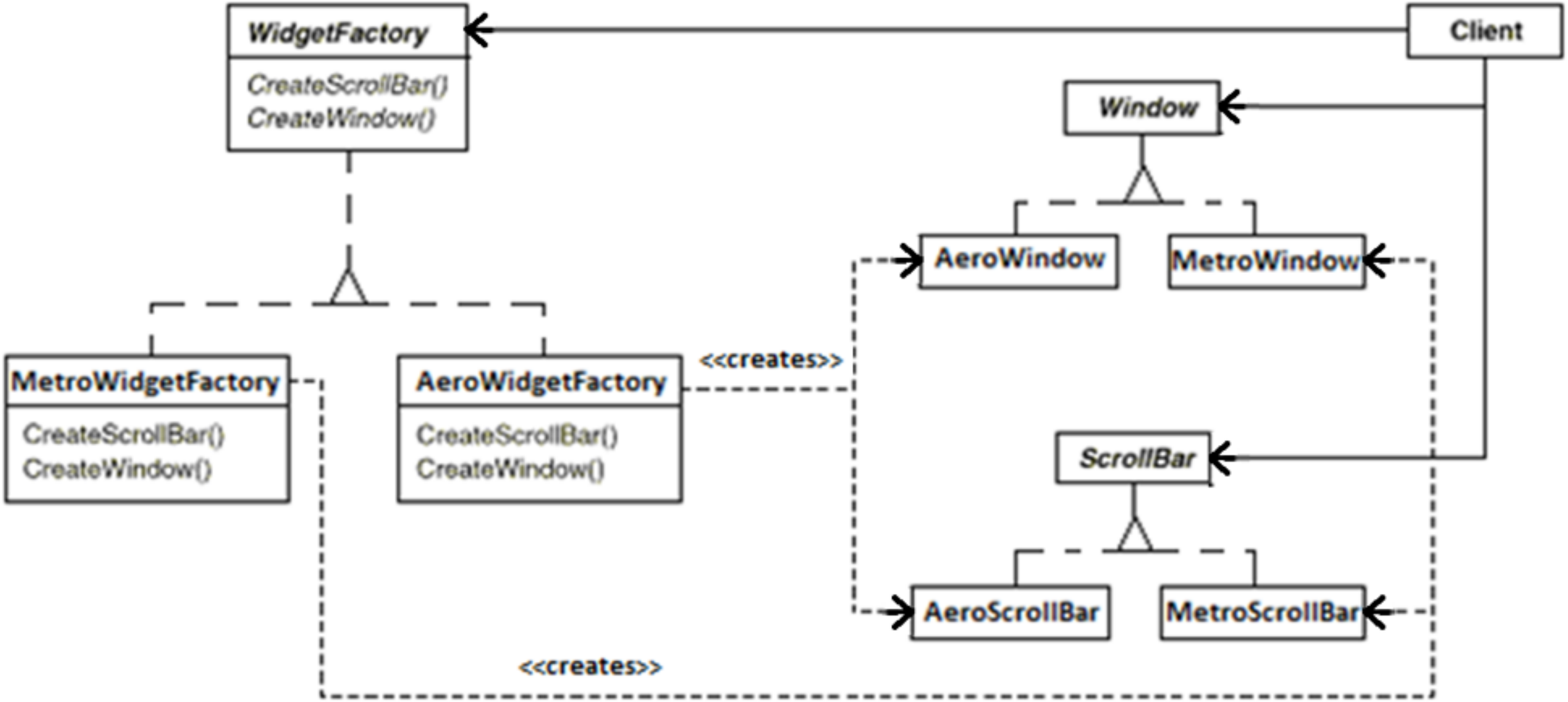
Aero


insideTonic - get more from software - Mozilla Firefox
File Edit View History Bookmarks Tools Help
http://www.insidetonic.com/
Softonic DE
Softonic EN
insideTonic
Development free sof...
insideTonic - get mor...
Softonic - Reviews an...
Archives
Contact
About
insideTonic
Add-ons
Extensions
Themes
Firefox (default) 2.0
The default theme
Vista-aero 2.0.0.8
Vista aero style, clear and cool !
Use Theme
Uninstall
Live Search Images - Mozilla Fi
Live Search
Done
Find Updates
Get Themes
Via RSS
Via Email
posts by
Windows
Mac
Mobile
Web
egories
Done

Metro

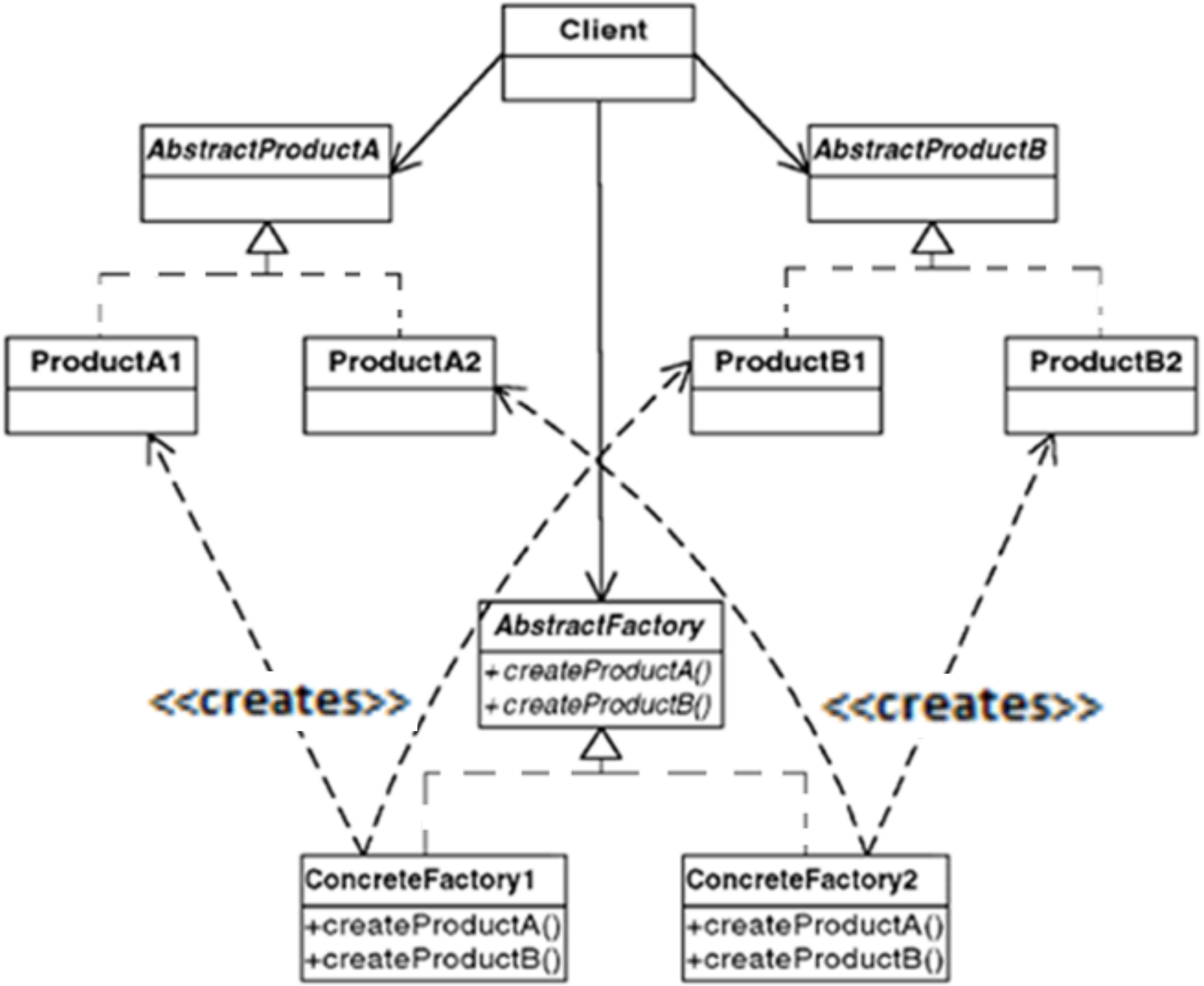

Start
Games
Chrome
9:19 PM
Tue, March 13
Files
Email
Market
Settings
ENGADGET
9:19 PM
Tuesday, March 13, 2012
BTHub3-PJPX
26%
Wi-Fi
Notifications
GPS
Sound
Auto rotation
AUTO
Settings

Abstract Factory Example



Note: as indicated, dotted line with arrow-head shows "creates" relationship

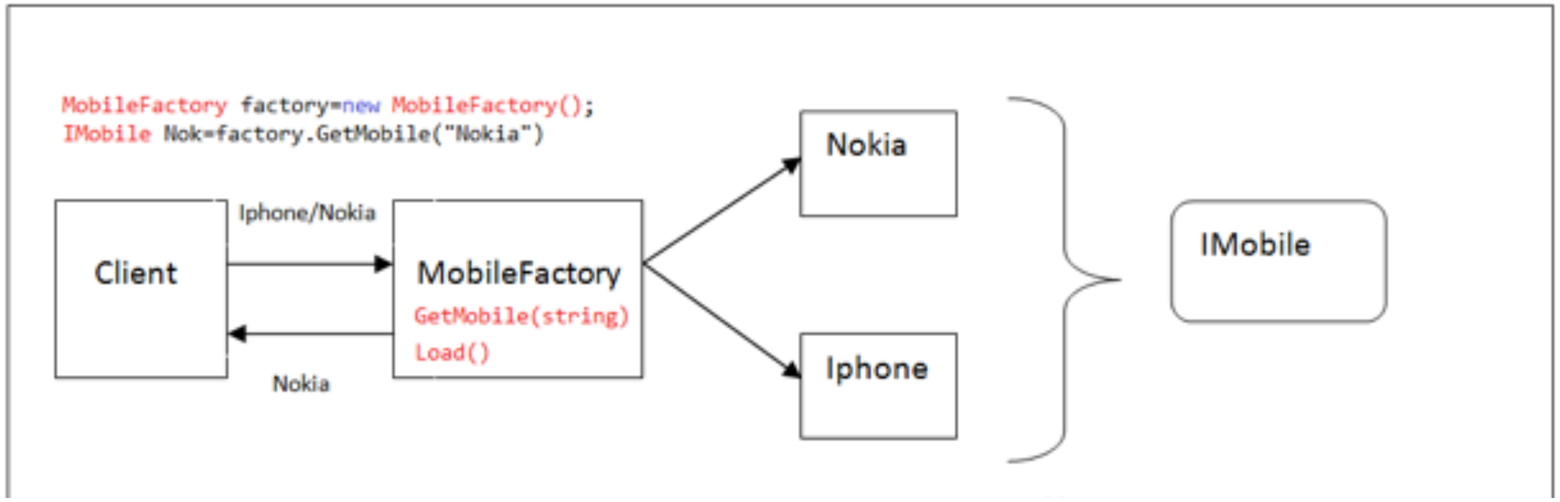
Generic Abstract Factory Structure



Abstract Factory Participants

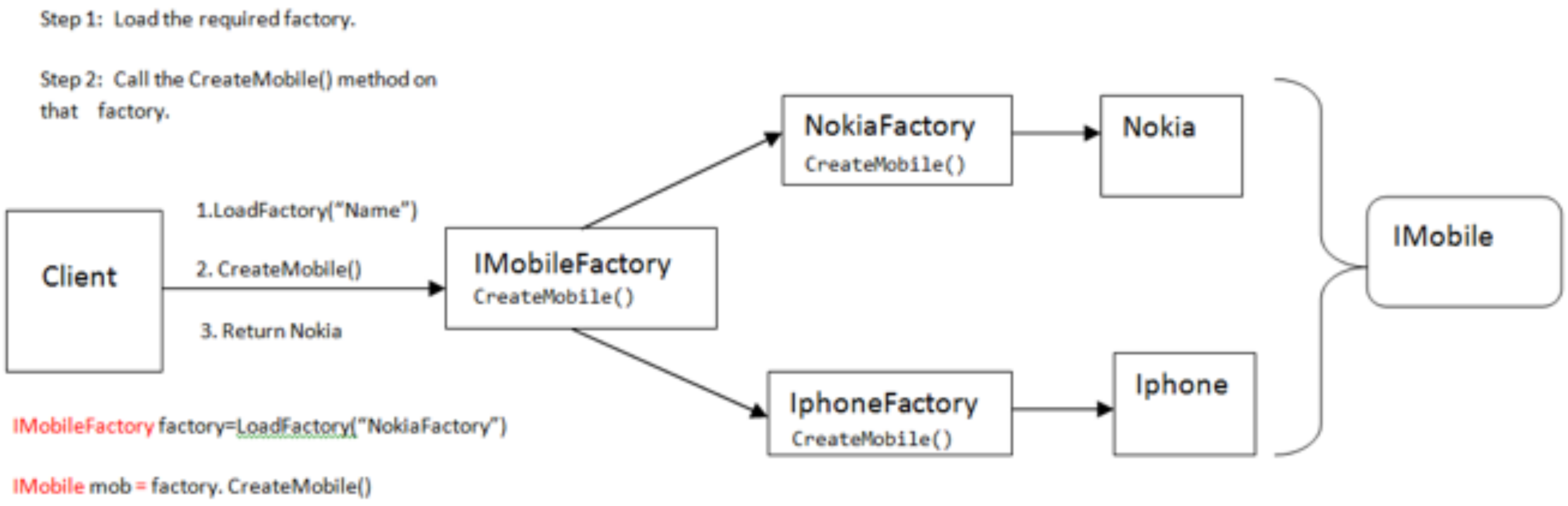
- Client
  - uses only interfaces declared by `AbstractFactory` and `AbstractProduct` classes
- AbstractFactory
  - declares an interface for operations that create abstract product objects
- ConcreteFactory
  - implements the operations to create concrete product objects
- AbstractProduct
  - declares an interface for a type of product object
- ConcreteProduct
  - corresponding to concrete factory
  - implements the `AbstractProduct` interface

simple factory



```
MobileFactory factory=new MobileFactory();
IMobile Nok=factory.GetMobile("Nokia")
```

factory method

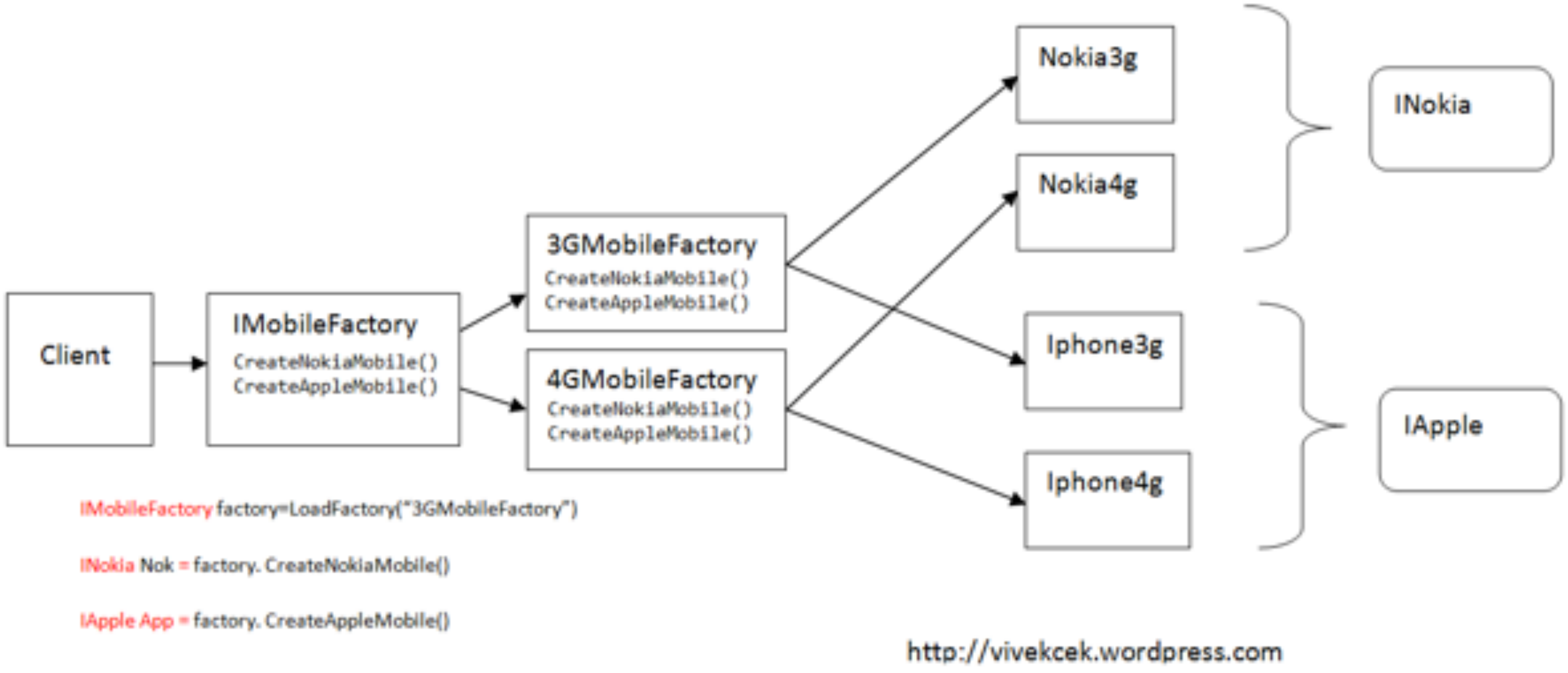


Step 1: Load the required factory.

Step 2: Call the CreateMobile() method on that factory.

```
IMobileFactory factory=LoadFactory("NokiaFactory")

IMobile mob = factory. CreateMobile()
```

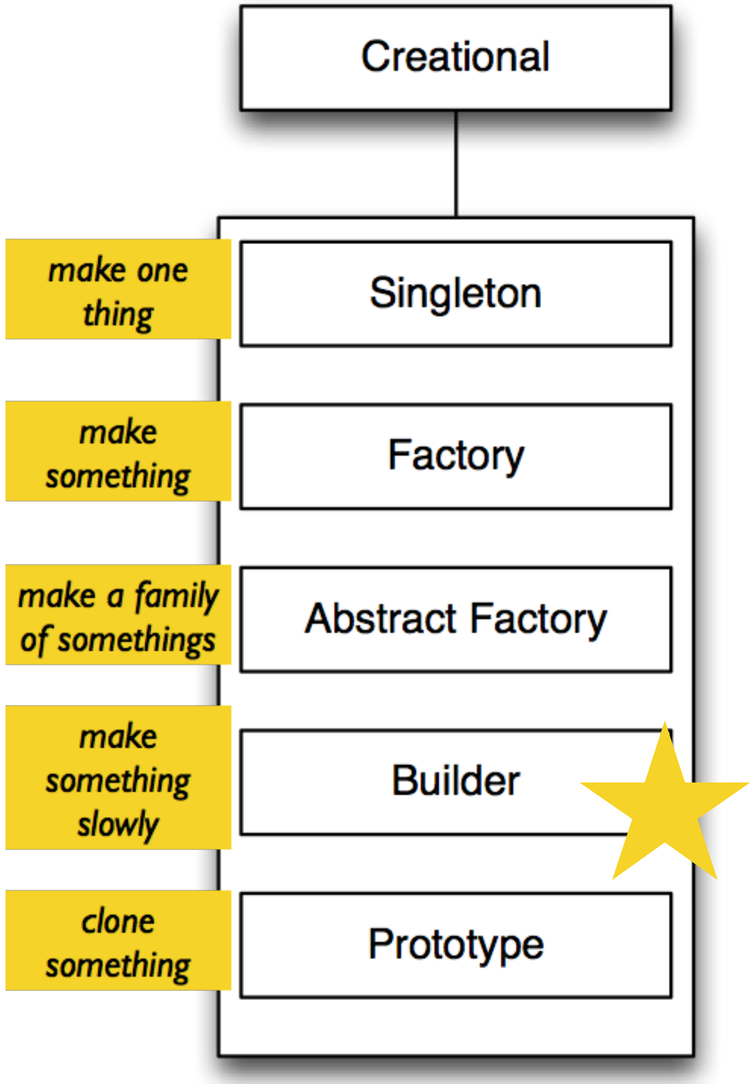
abstract factory



```
IMobileFactory factory=LoadFactory("3GMobileFactory")

INokia Nok = factory. CreateNokiaMobile()

IApple App = factory. CreateAppleMobile()
```

http://vivekcek.wordpress.com

http://vivekcek.wordpress.com/2013/03/17/simple-factory-vs-factory-method-vs-abstract-factory-by-example/



But I don't want to make a bunch of themey objects, I want to make one complicated object over time that needs a lot of other objects to help!

Builder (Creational pattern)

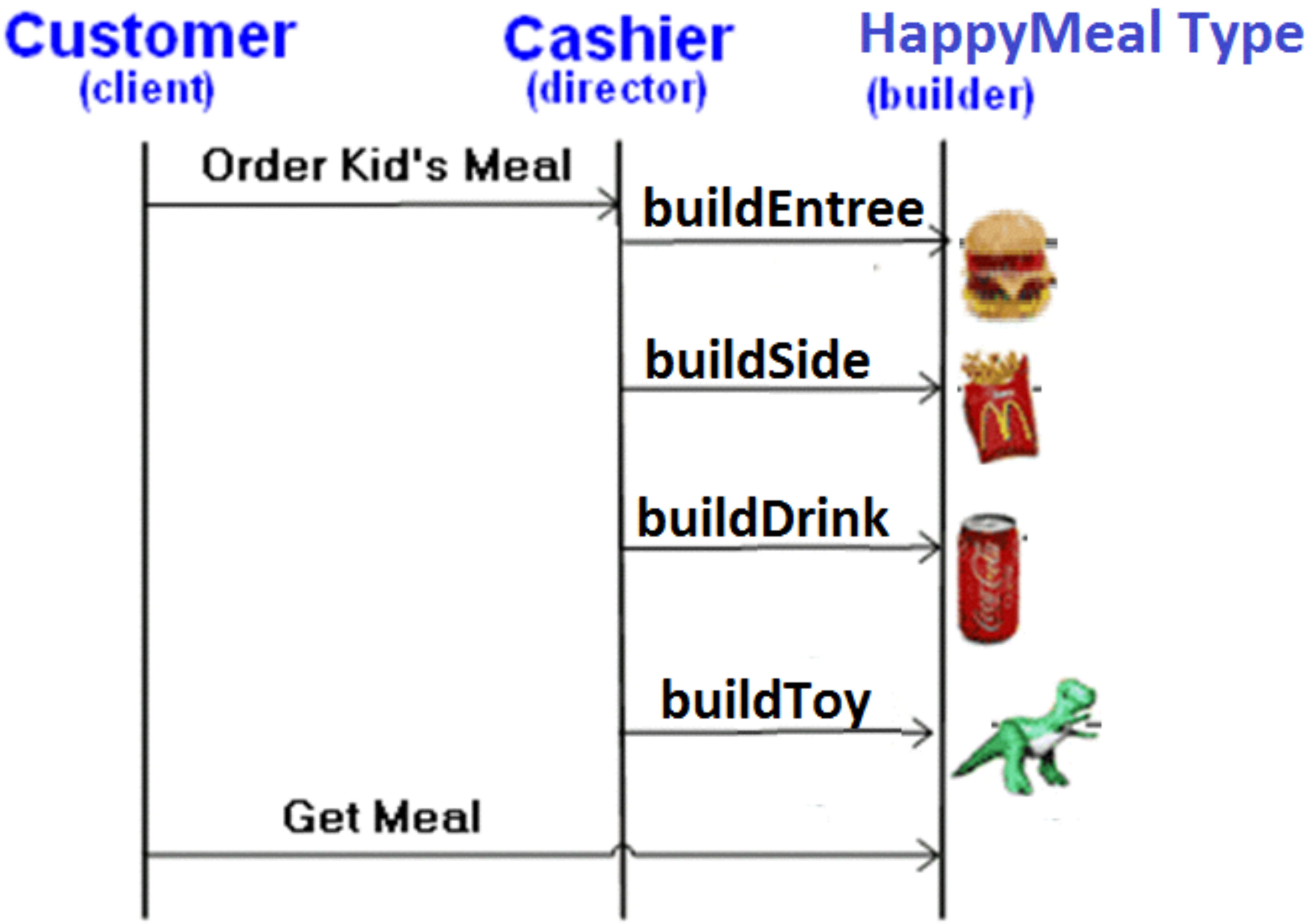
Intent: Separate the construction of a complex object from its representation so that the same construction process can create different representations

•Use the Builder pattern when:

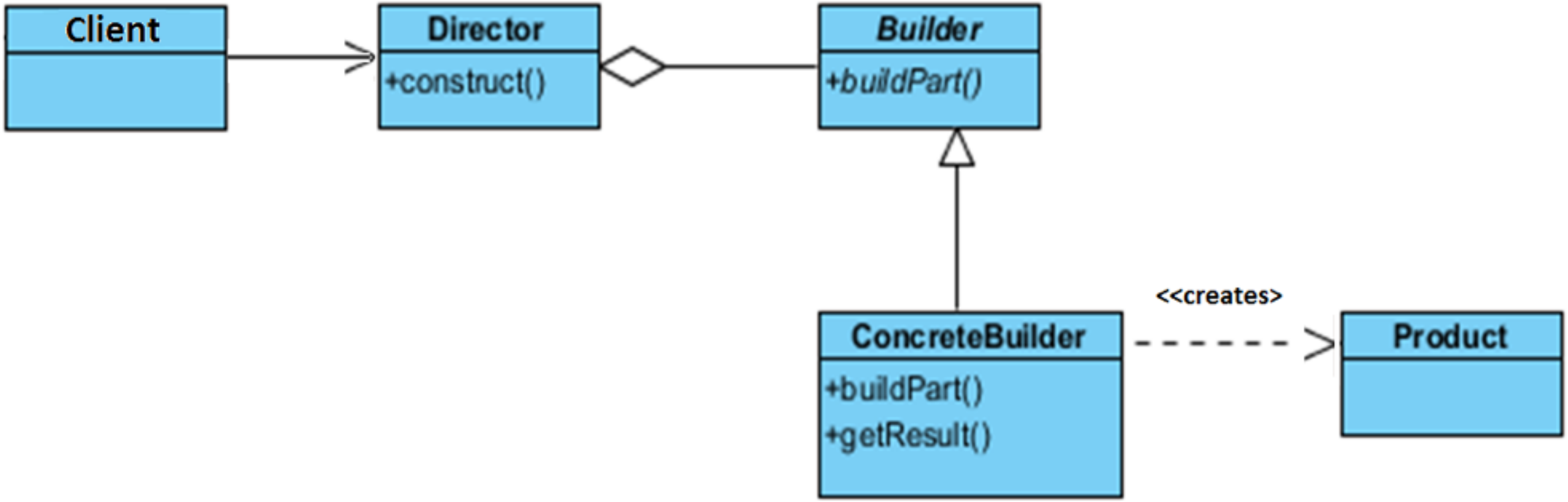
- the algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled
- the construction process must allow different representations for the object that is constructed

Reference: Design Patterns, Gamma, et. al., Addison Wesley, 1995, pp 97-98

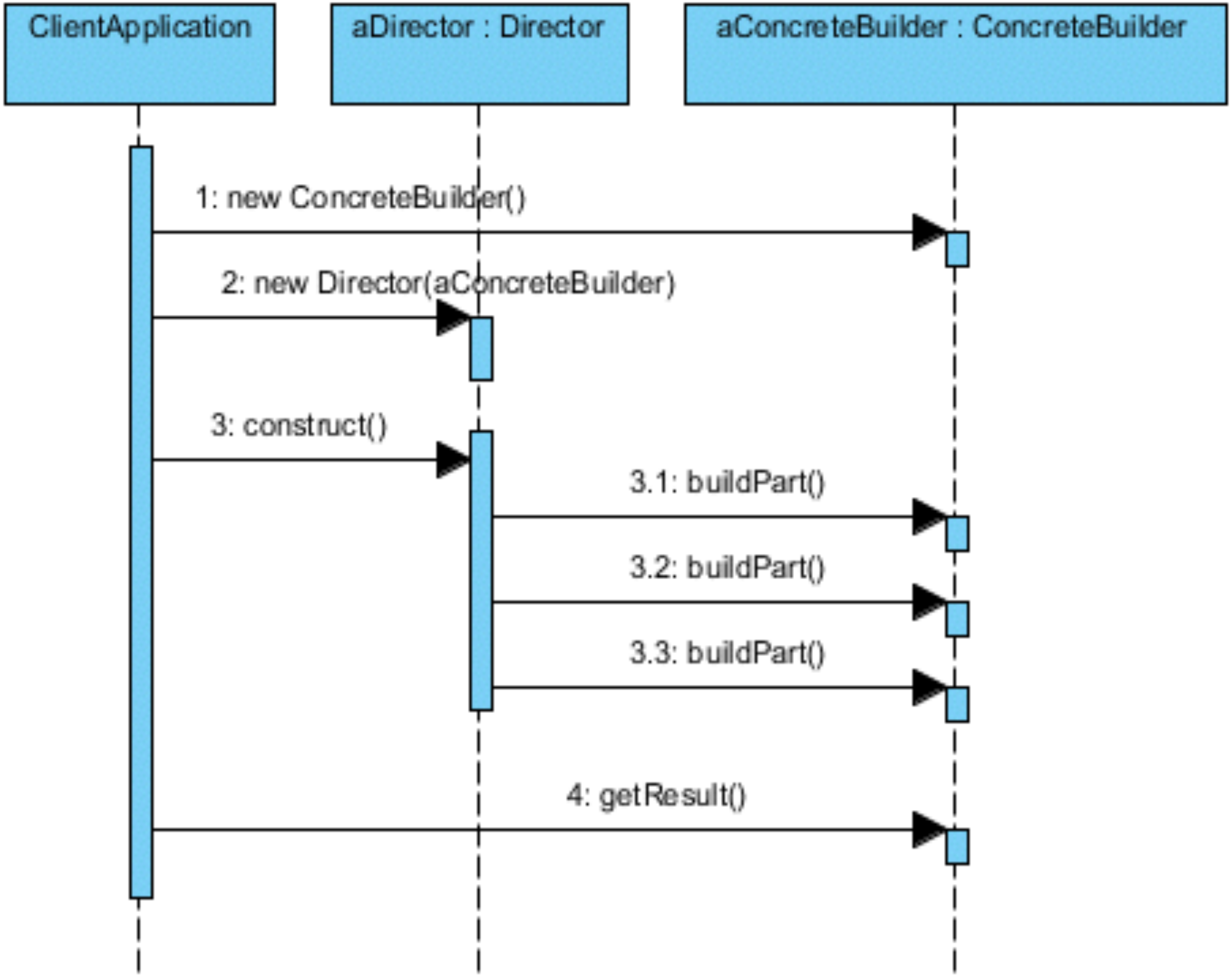
Builder Metaphor



UML Structure

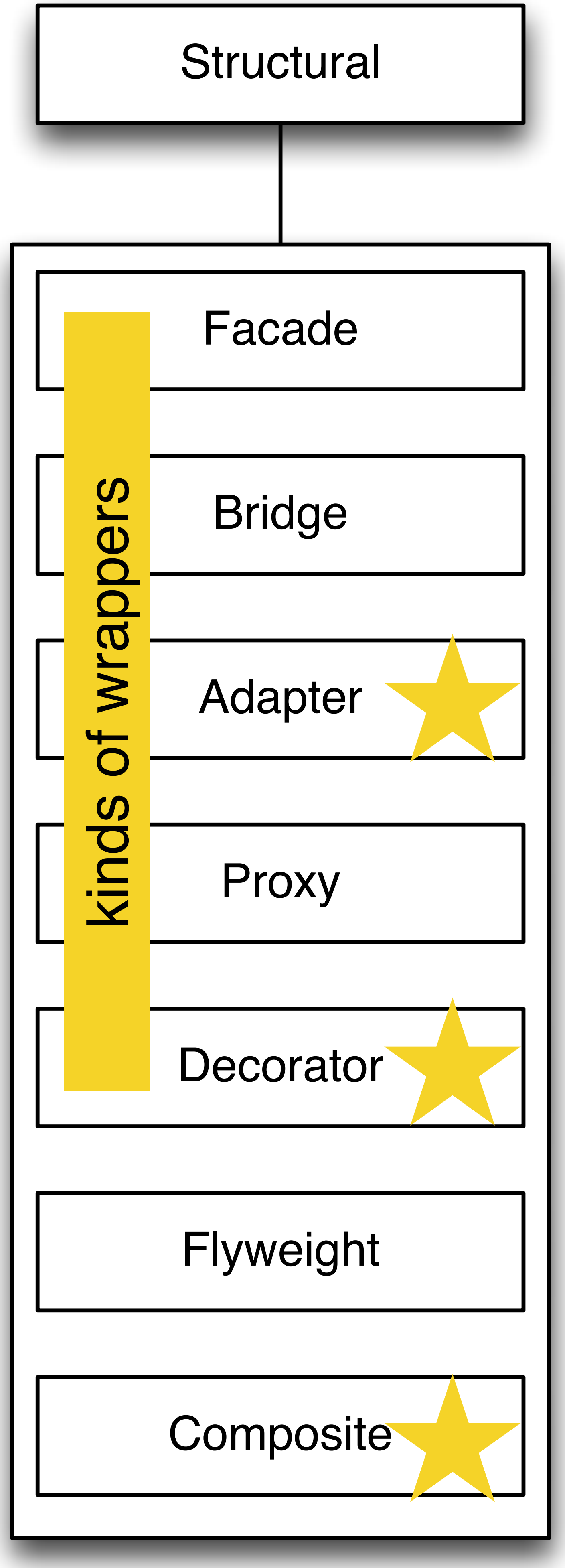


Collaborations



Creational Patterns Summary

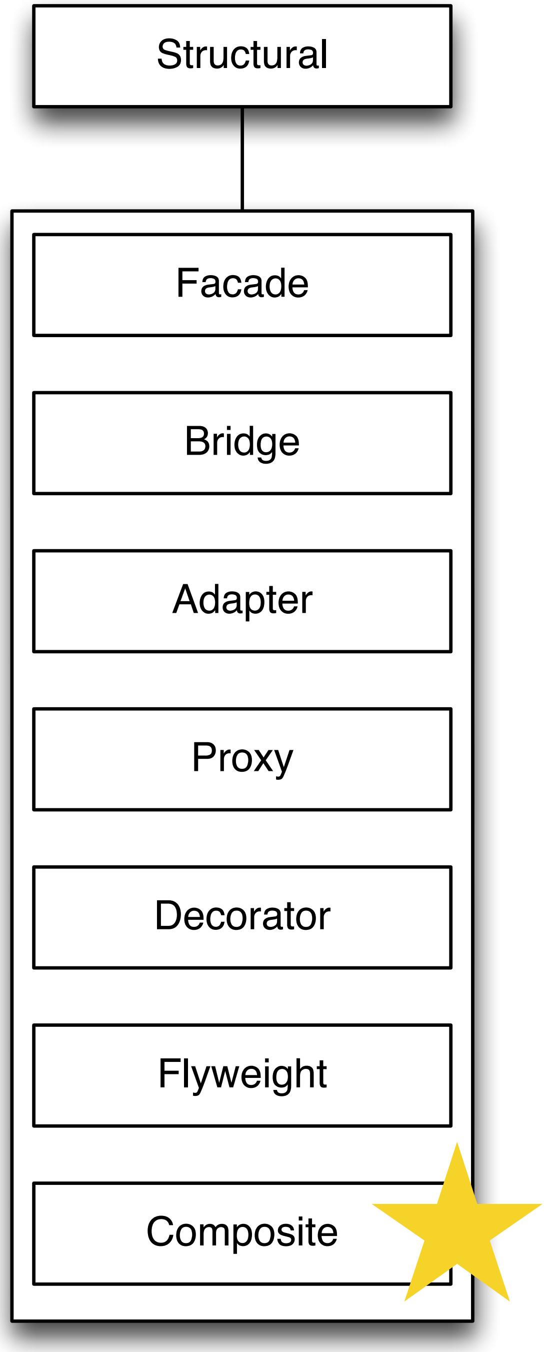
- Factory Method
  - hide the details of sub-class specific implementations from Client
- Abstract Factory
  - enforce creation of related family objects
- Builder
  - Multi-step object creation
  - Separate the construction steps from the low-level representation of the constructed object



Structural

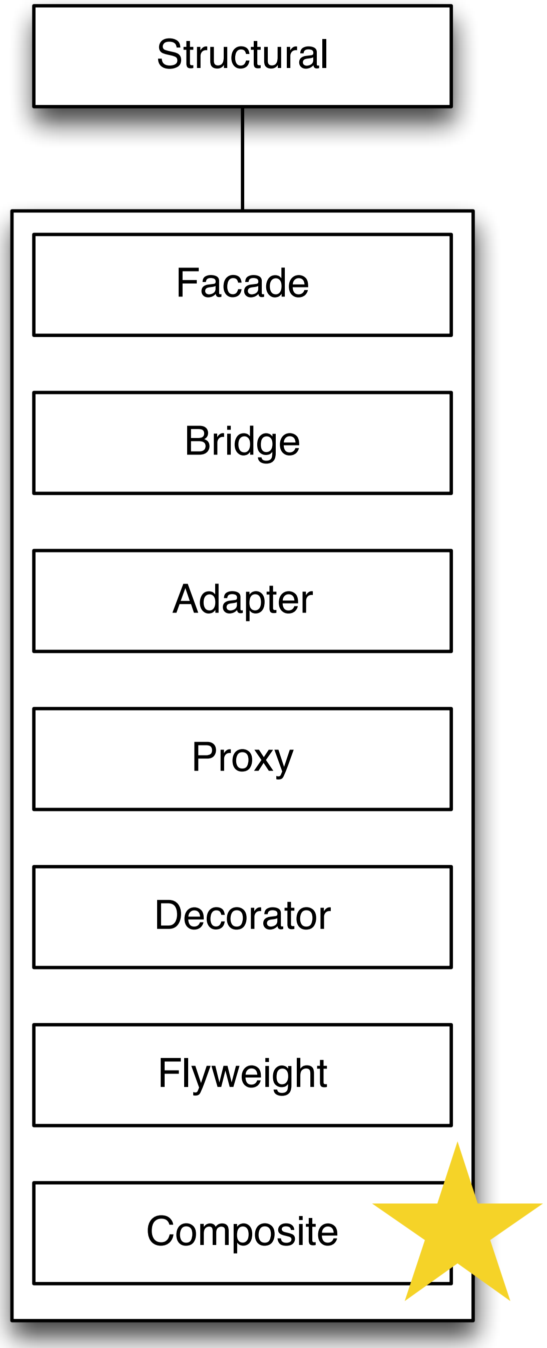
kinds of wrappers

Facade	Simple(r) interface to a class. Kind of an object container.
Bridge	A link between two hierarchies, both of which you’re developing
Adapter ★	A link between two interfaces, at least one of which is out of your control
Proxy	A link to hide the fact that your object defers to a remote service
Decorator ★	Takes a proxy, and adds to its behaviour (decorates it)
Flyweight	sharing data with other objects to avoid duplication and to minimise memory usage
Composite ★	a tree structure where leaves and branches can be treated the same.

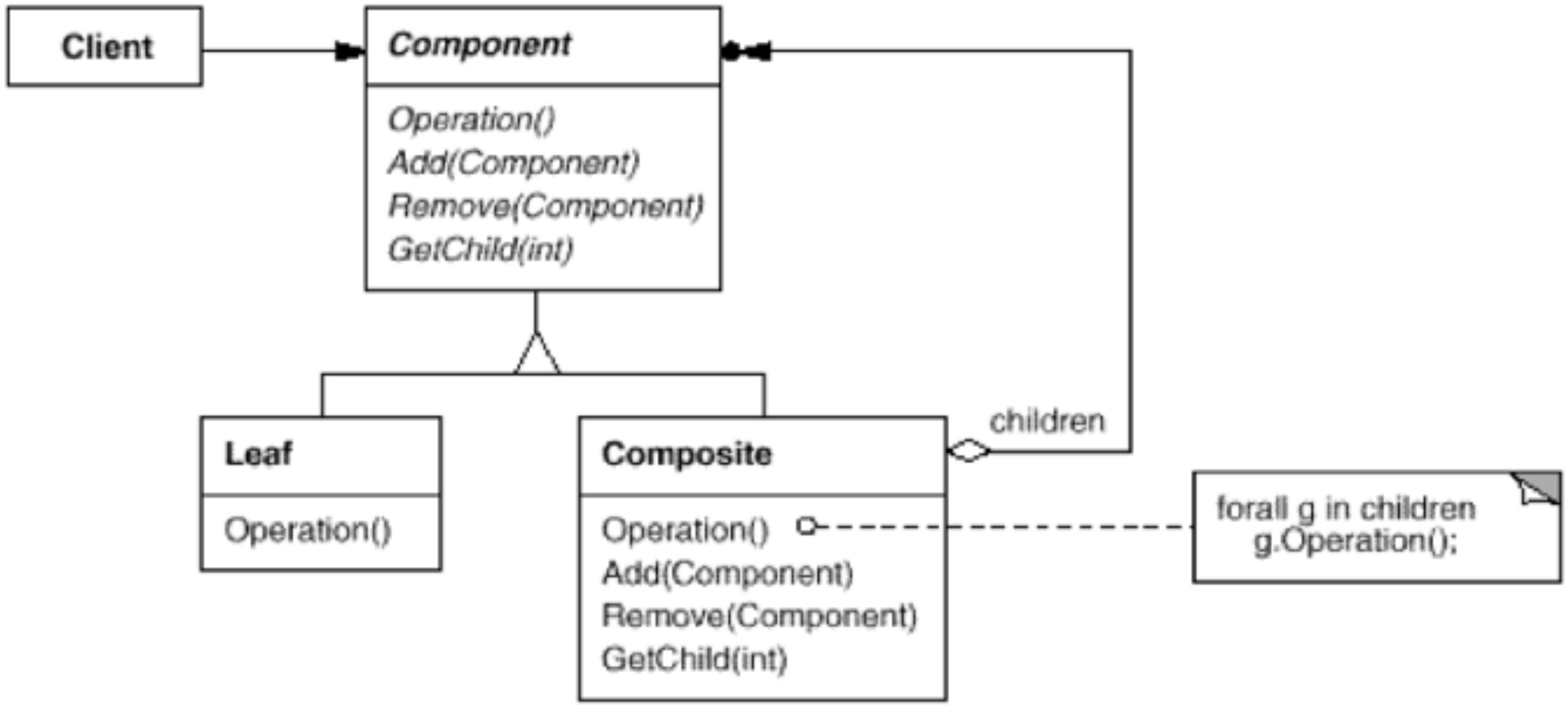


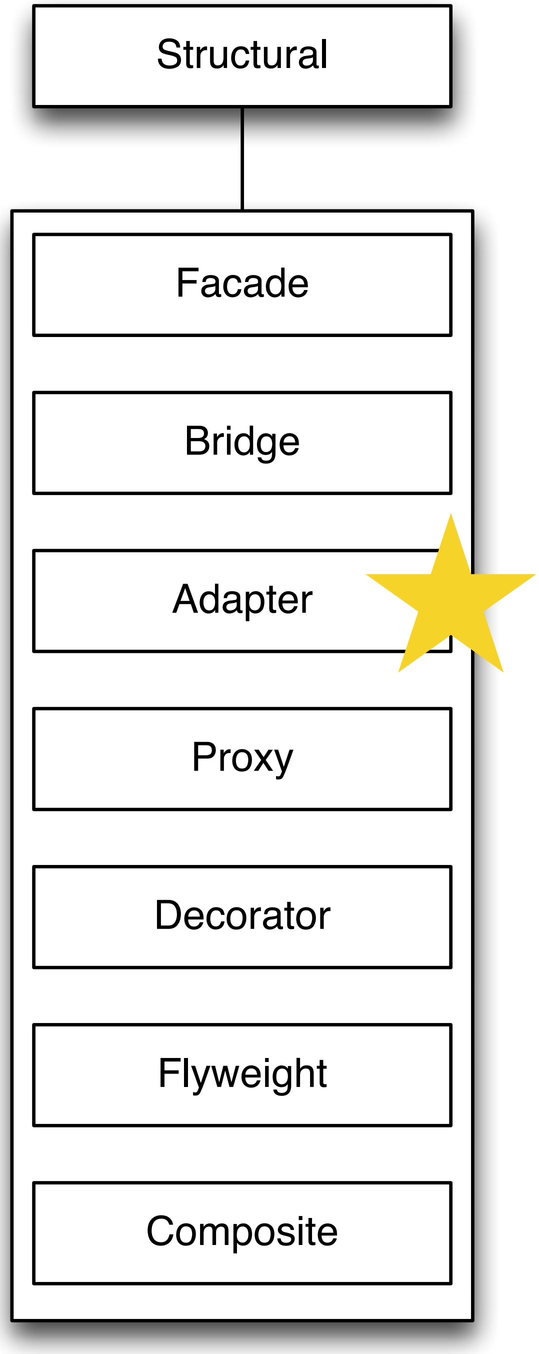
I want to be able to have a hierarchical organisation, but don't want to write special code for containers and leaf nodes…

Composite



- **Name:** Composite
- **Intent:** Compose objects into tree structures. Lets clients treat individual objects and compositions uniformly.
- **Participants & Structure:**





Structural

Facade

Bridge

Adapter

Proxy

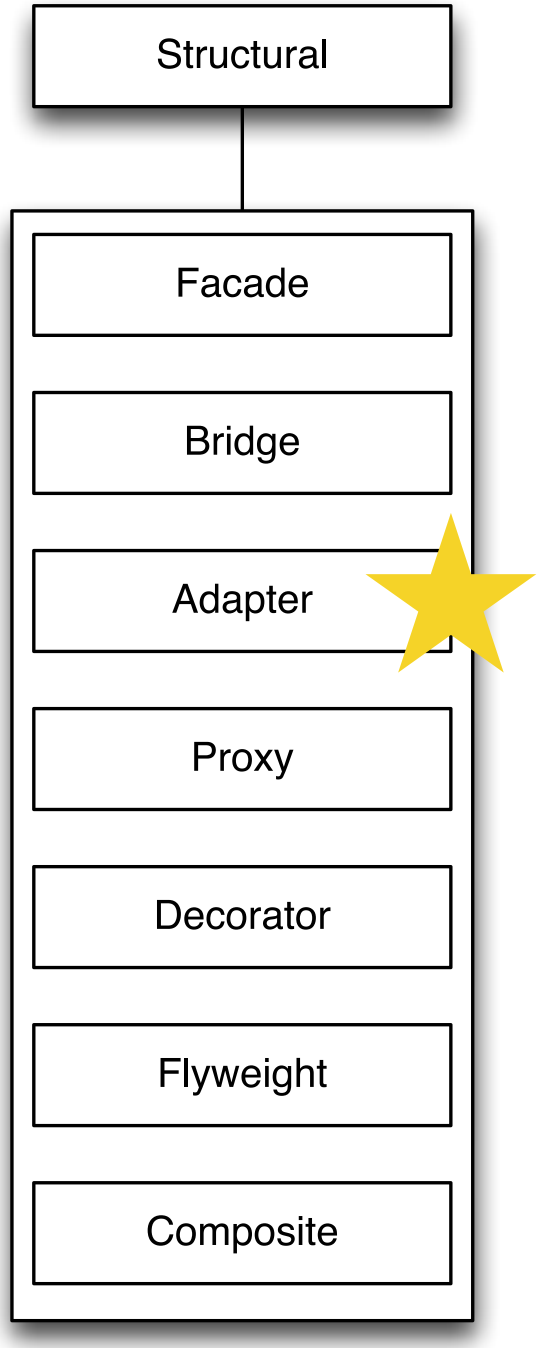
Decorator

Flyweight

Composite

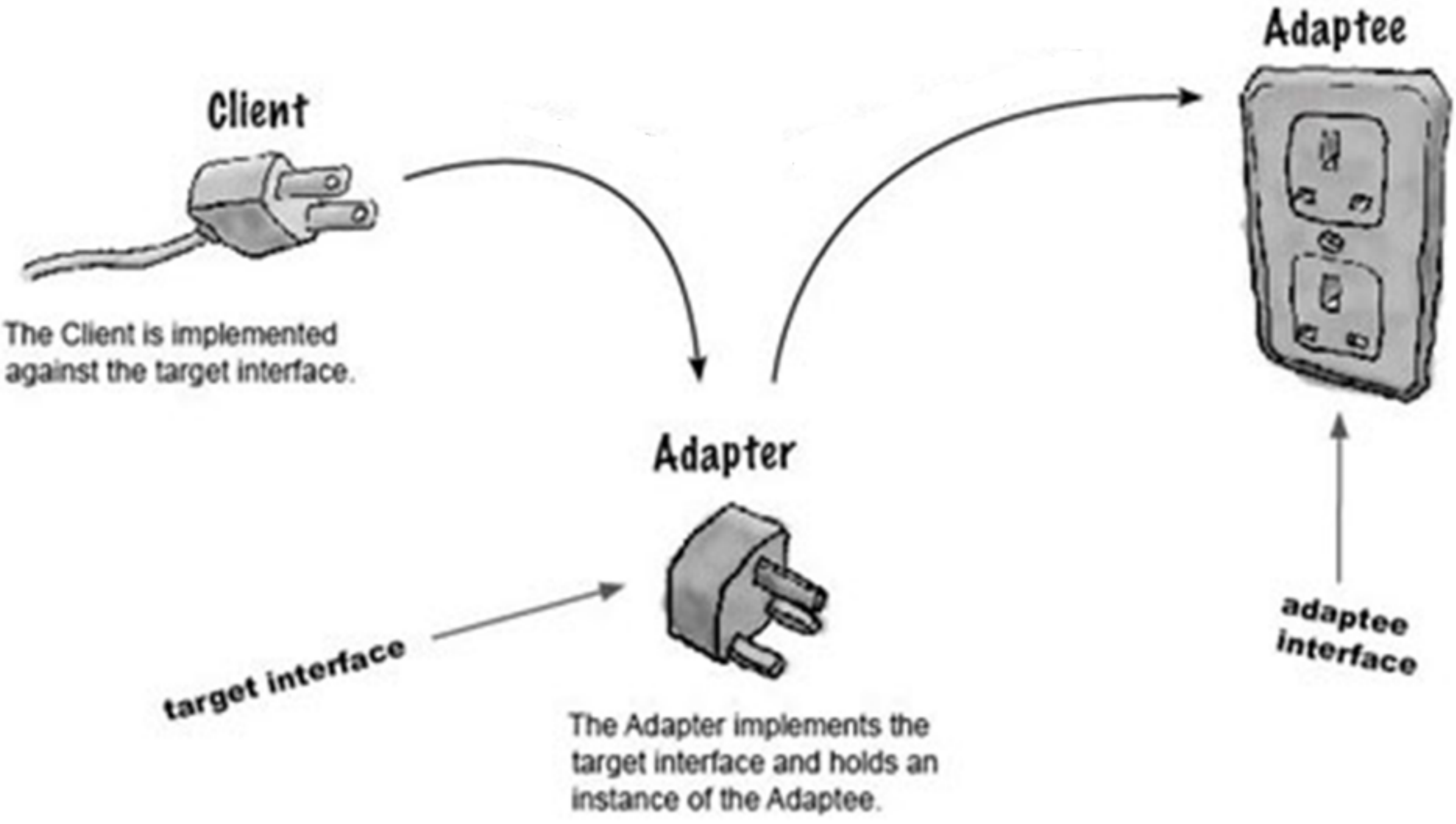
I have an existing class that I want to use, but I don’t want to modify its source code…

Adapter



- Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
- We can not change the library interface, since we may not have its source code
- Even if we did have the source code, we probably should not change the library for each domain-specific application

Metaphor: Electric Adapters



The Client is implemented against the target interface.

The Adapter implements the target interface and holds an instance of the Adaptee.

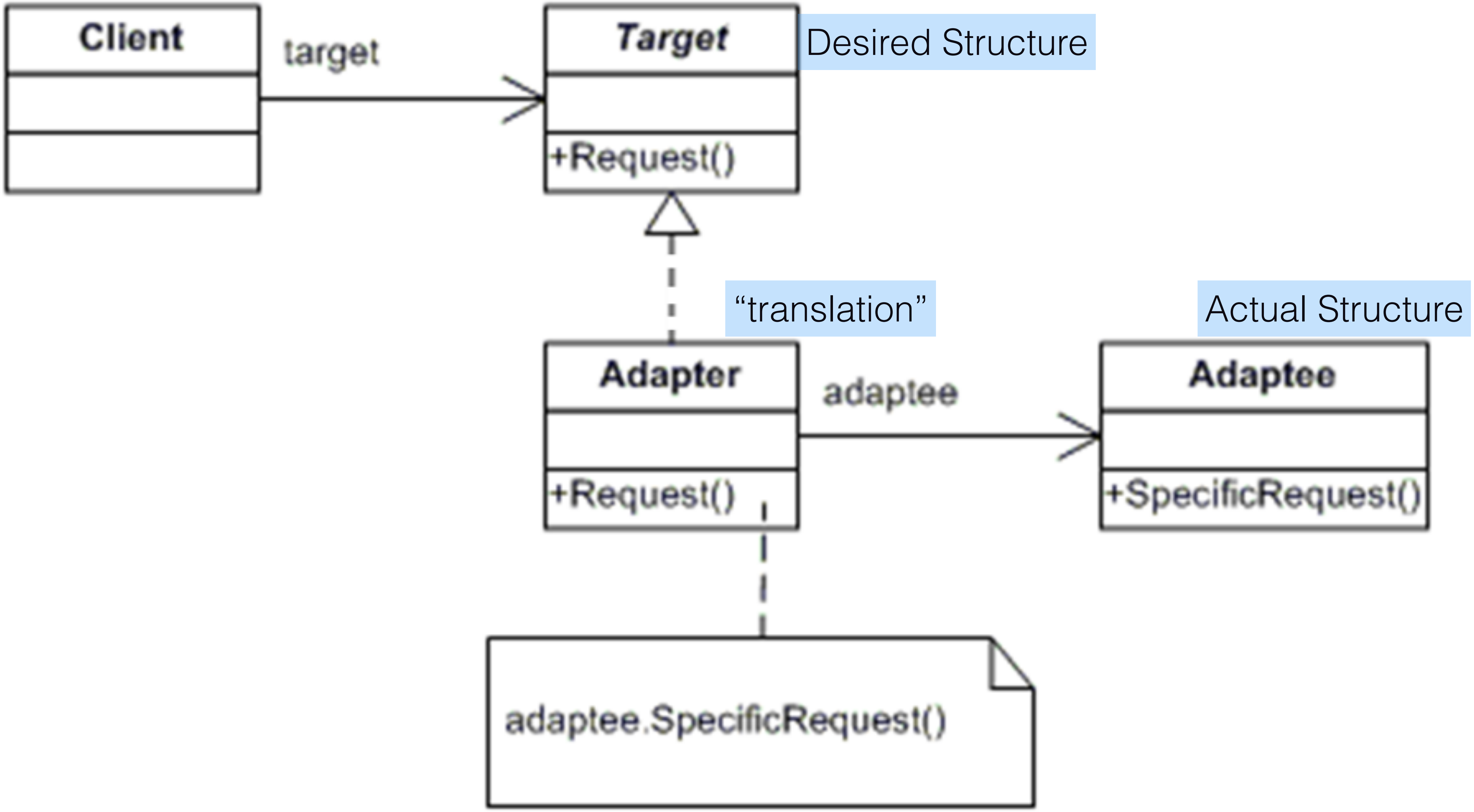
Adapter: Intent

- Convert the interface of a class into another interface clients expect.
- Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Wrap an existing class with a new interface.
- Also Known As Wrapper

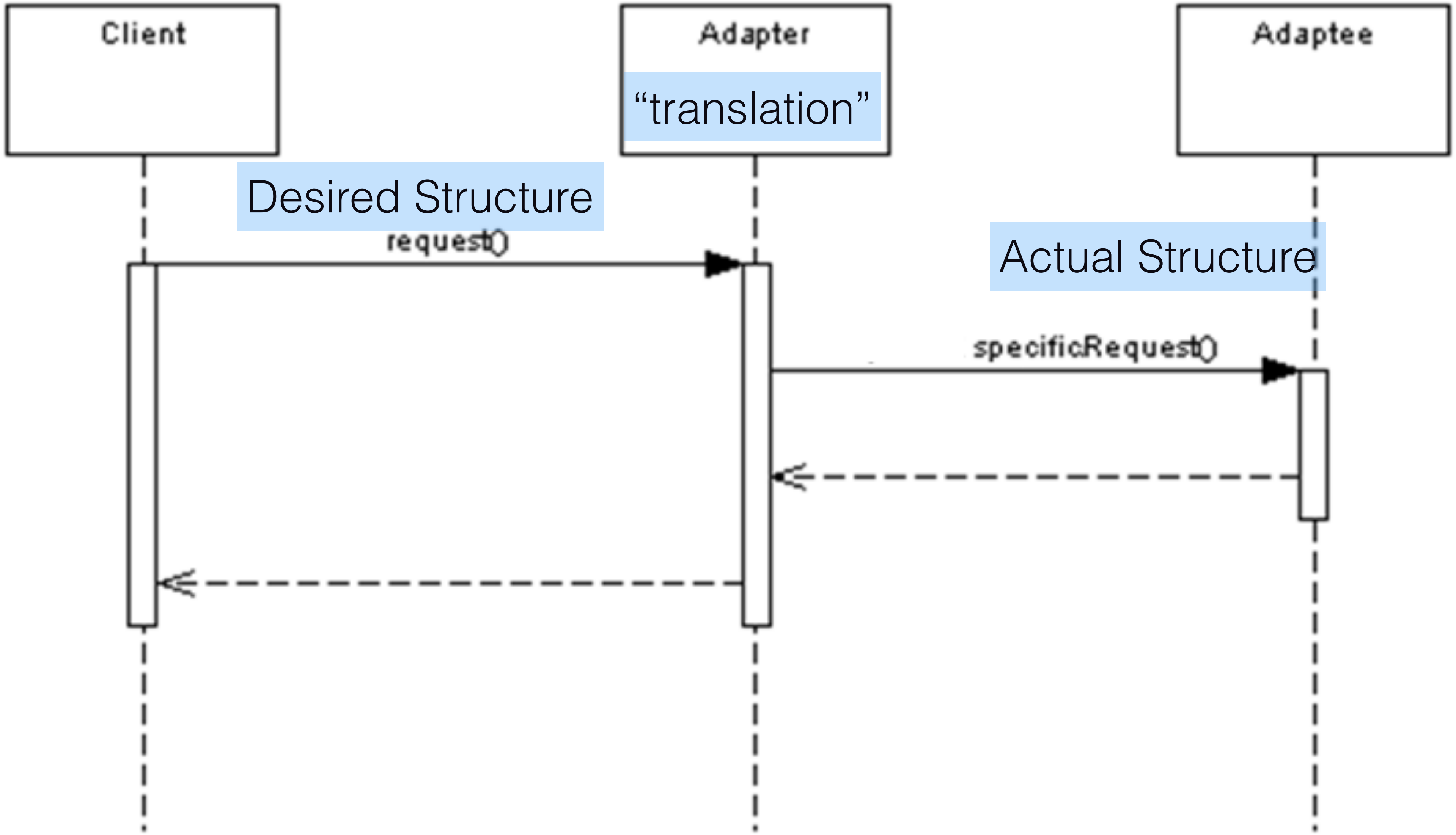
Participants

- **Target**
  - defines the domain-specific interface that Client uses.
- **Adapter**
  - adapts the interface Adaptee to the Target interface.
- **Adaptee**
  - defines an existing interface that needs adapting.
- **Client**
  - collaborates with objects conforming to the Target interface.

Generic Adapter Structure



Collaboration



Example Adaptee

Consider that we have a third party library that provides sorting functionality through it's NumberSorter class.

This is our adaptee, a third party implementation of a number sorter that deals with Lists, not arrays.

```
public class NumberSorter
{
            public void sort(List<Integer> numbers)
            {
                        //sort numbers (details elided)
                        …
            }
}
```

Example Target

Our Client deals with primitive arrays rather than Lists.

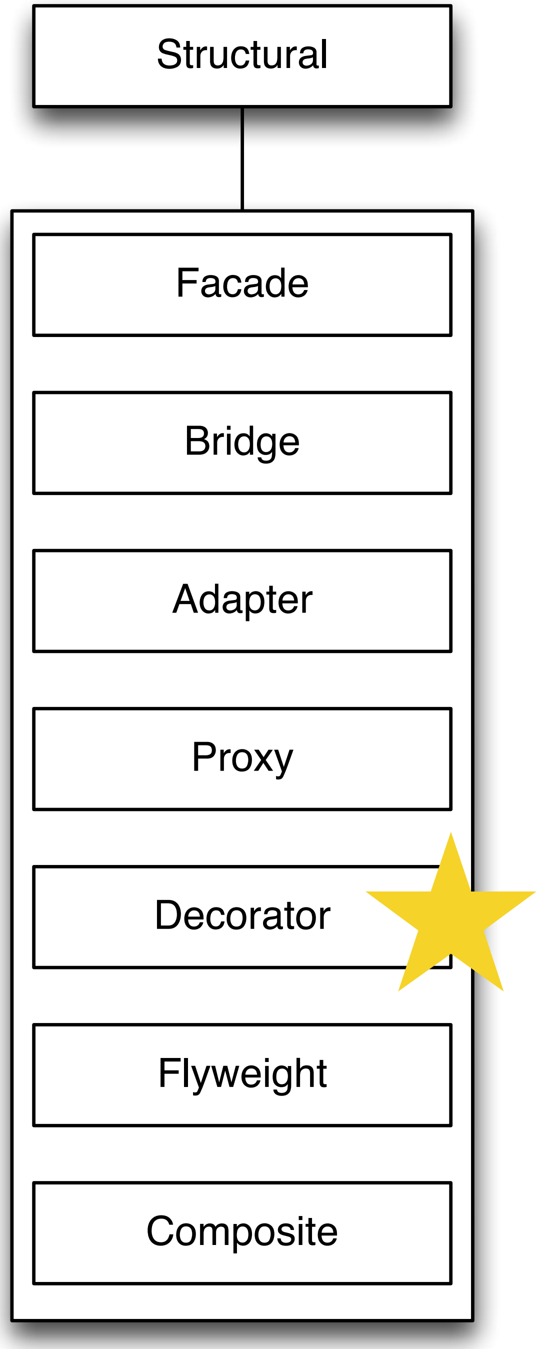
We've provided a Sorter interface that expects the client input. This is our target.

```
//this is our Target interface
public interface Sorter
{
        public int[] sort(int[] numbers);
}
```

Example Adapter

Finally, the SortListAdapter implements our target interface and deals with our adaptee, NumberSorter

```
public class SortListAdapter implements Sorter
{
        NumberSorter sorter = new NumberSorter();

        @Override
        public int[] sort(int[] numbers)
        {
                //convert the array to a List (conversion detail elided)
                 List<Integer> numberList  = convertArrayToList(numbers);

                sorter.sort(numberList);

                //convert the list back to an array and return (conversion detail elided)
                int[] sortedArray = convertListToArray(numberList);
                return sortedArray;
        }
}
```



Structural

Facade

Bridge

Adapter

Proxy

Decorator

Flyweight

Composite

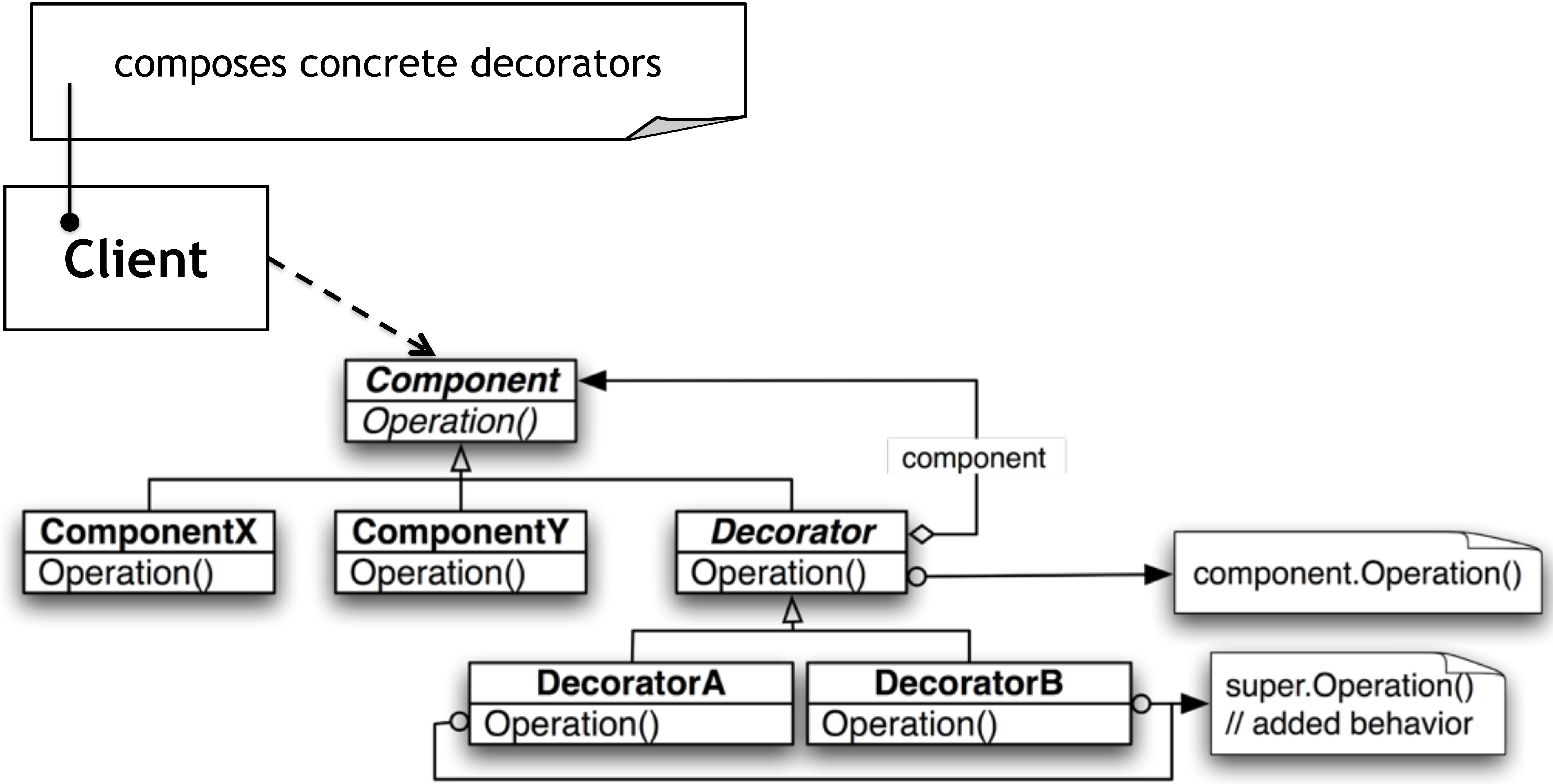
I have a thing, and I want to be able to customise it on the fly. But I don't want to make a million little classes each with the right settings!

Decorator

Intent:

•Client-specified customization of features for object

•Attach additional responsibilities to an object dynamically (at run-time)



composes concrete decorators

Client

Component

Operation()

component

ComponentX

Operation()

ComponentY

Operation()

Decorator

Operation()

component.Operation()

DecoratorA

Operation()

DecoratorB

Operation()

super.Operation()
// added behavior

Participants

- Component
  - the interface of a core object
- Concrete component
  - an implementation of the component interface
- Decorator
  - abstract class which wraps another Component
- Concrete Decorator
  - an implementation of Decorator which adds new behavior before/after core behavior
    - through delegation
  - optionally adds new methods/properties

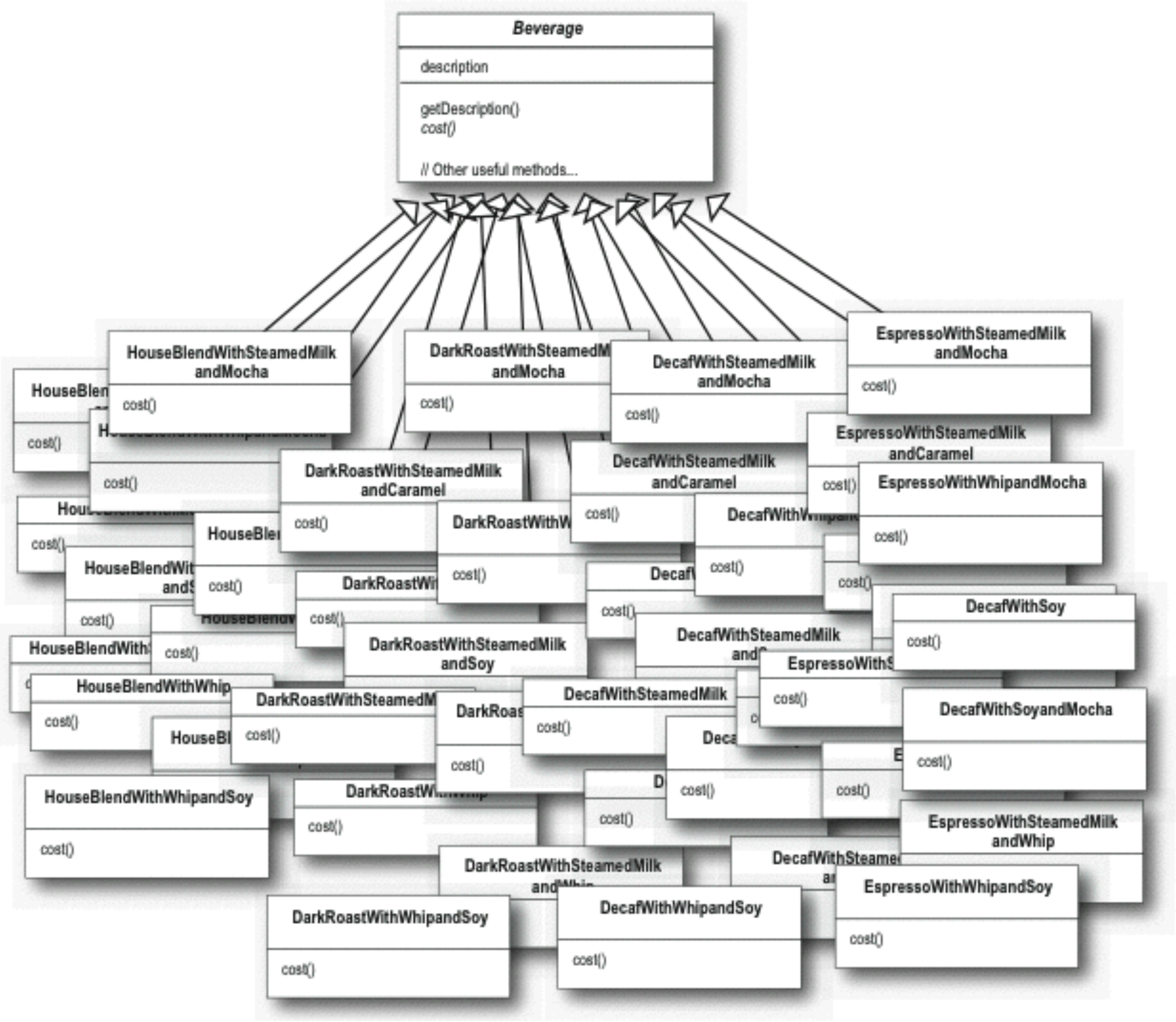
Sample problem

You need to implement a point-of-sale system for a coffee shop. The coffee shop has some basic beverages, but customers can customize their drinks by choosing what kind of milk they want, if they want flavoured syrup, etc.

You could create a class for each drink, but there are so many possible combinations that the number of classes would quickly get out of hand.

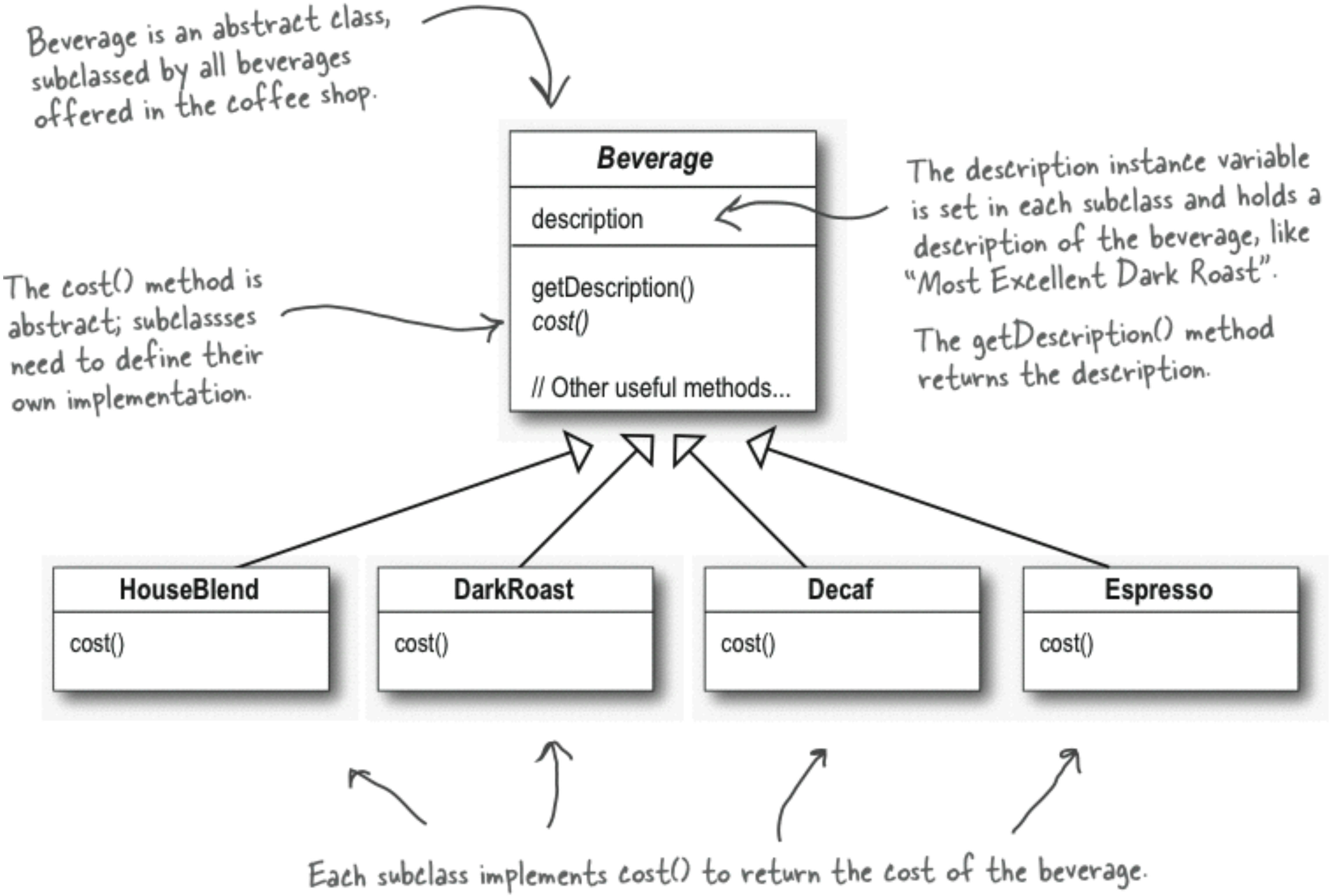
http://oreilly.com/catalog/hfdesignpat/chapter/ch03.pdf

Solving this problem with inheritance


Beverage
description
getDescription()
cost()
// Other useful methods...
HouseBlendWithSteamedMilk andMocha
cost()
DarkRoastWithSteamedMilk andMocha
cost()
DecafWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andMocha
cost()
EspressoWithSteamedMilk andCaramel
EspressoWithWhipandMocha
DarkRoastWithSteamedMilk andCaramel
DecafWithSteamedMilk andCaramel
DarkRoastWithSteamedMilk andSoy
DecafWithSteamedMilk
DecafWithSoy
DecafWithSoyandMocha
HouseBlendWithWhip
HouseBlendWithWhipandSoy
EspressoWithSteamedMilk andWhip
EspressoWithWhipandSoy
DarkRoastWithSteamedMilk andWhip
DarkRoastWithWhipandSoy
DecafWithWhipandSoy

Freeman, et al. *Design Patterns, Head First*

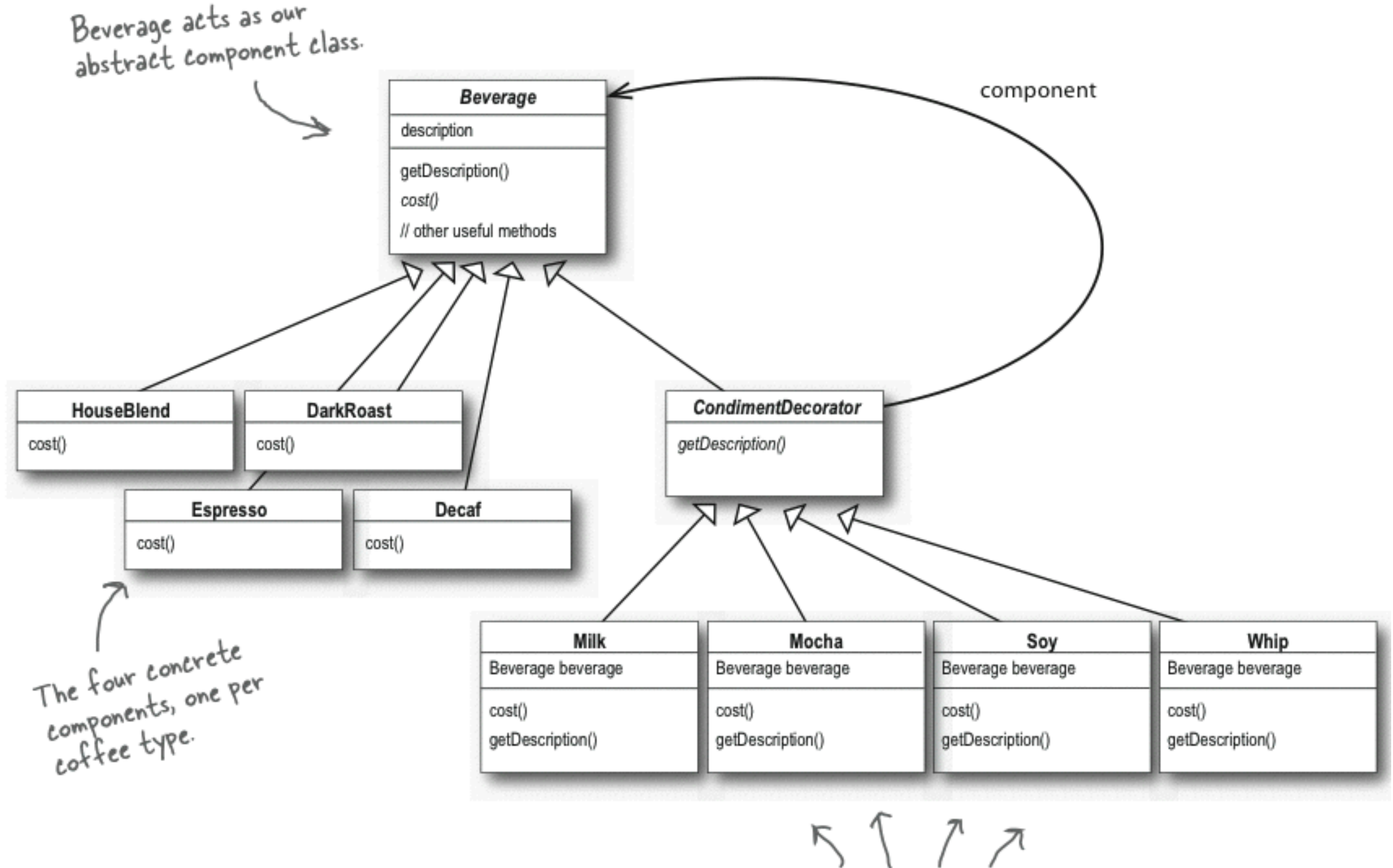
Solving this problem with Decorators



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.

Freeman, et al. *Design Patterns, Head First*

Solving this problem with Decorators



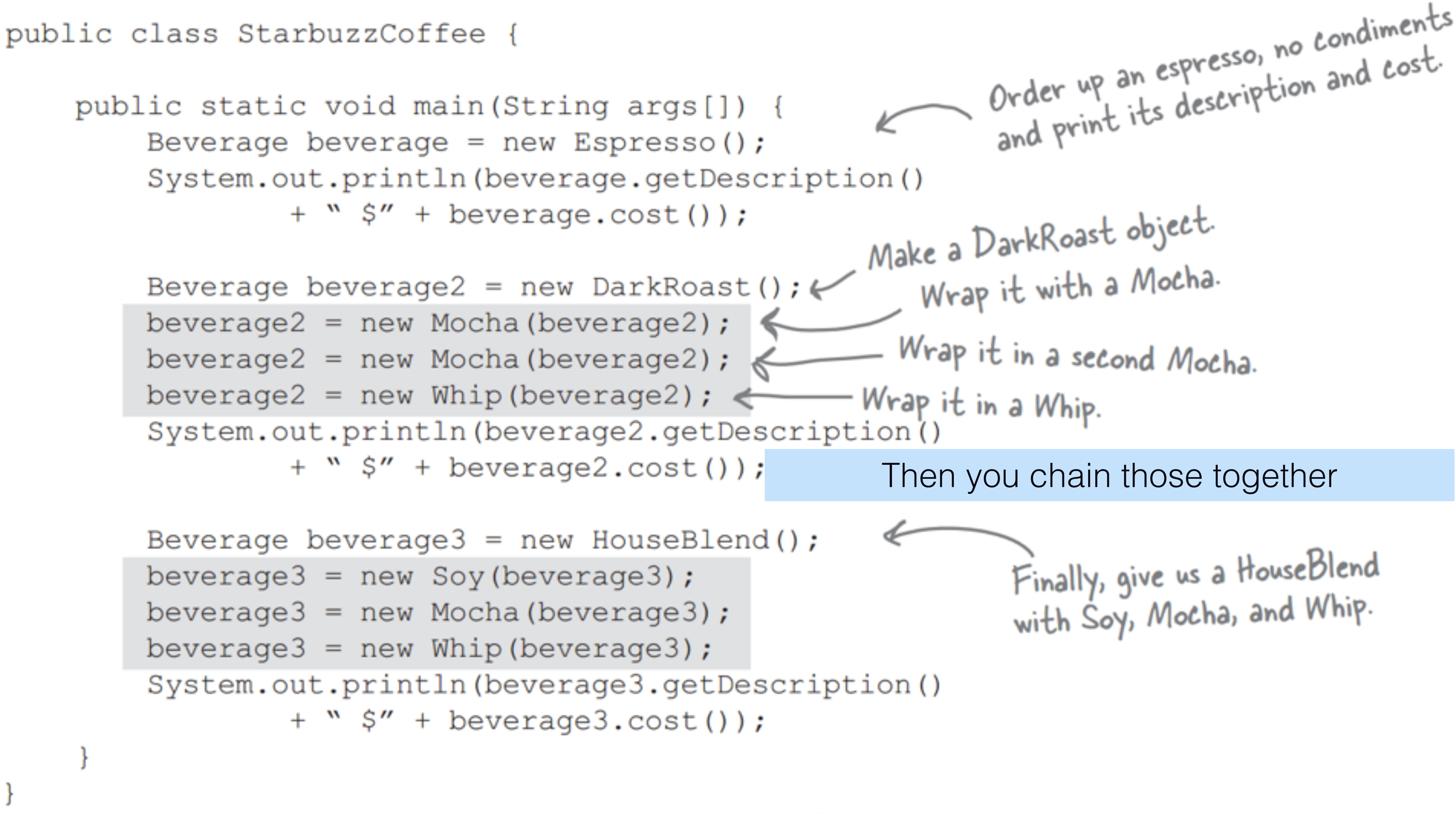
Beverage acts as our abstract component class.

The four concrete components, one per coffee type.

And here are our condiment decorators; notice they need to implement not only cost() but also getDescription(). We'll see why in a moment...

Freeman, et al. *Head First Design Patterns*

Solving this problem with Decorators



```
public class StarbuzzCoffee {

    public static void main(String args[]) {
        Beverage beverage = new Espresso();
        System.out.println(beverage.getDescription()
                + “ $” + beverage.cost());

        Beverage beverage2 = new DarkRoast();
        beverage2 = new Mocha(beverage2);
        beverage2 = new Mocha(beverage2);
        beverage2 = new Whip(beverage2);
        System.out.println(beverage2.getDescription()
                + “ $” + beverage2.cost());

        Beverage beverage3 = new HouseBlend();
        beverage3 = new Soy(beverage3);
        beverage3 = new Mocha(beverage3);
        beverage3 = new Whip(beverage3);
        System.out.println(beverage3.getDescription()
                + “ $” + beverage3.cost());
    }
}
```

Order up an espresso, no condiments and print its description and cost.

Make a DarkRoast object.

Wrap it with a Mocha.

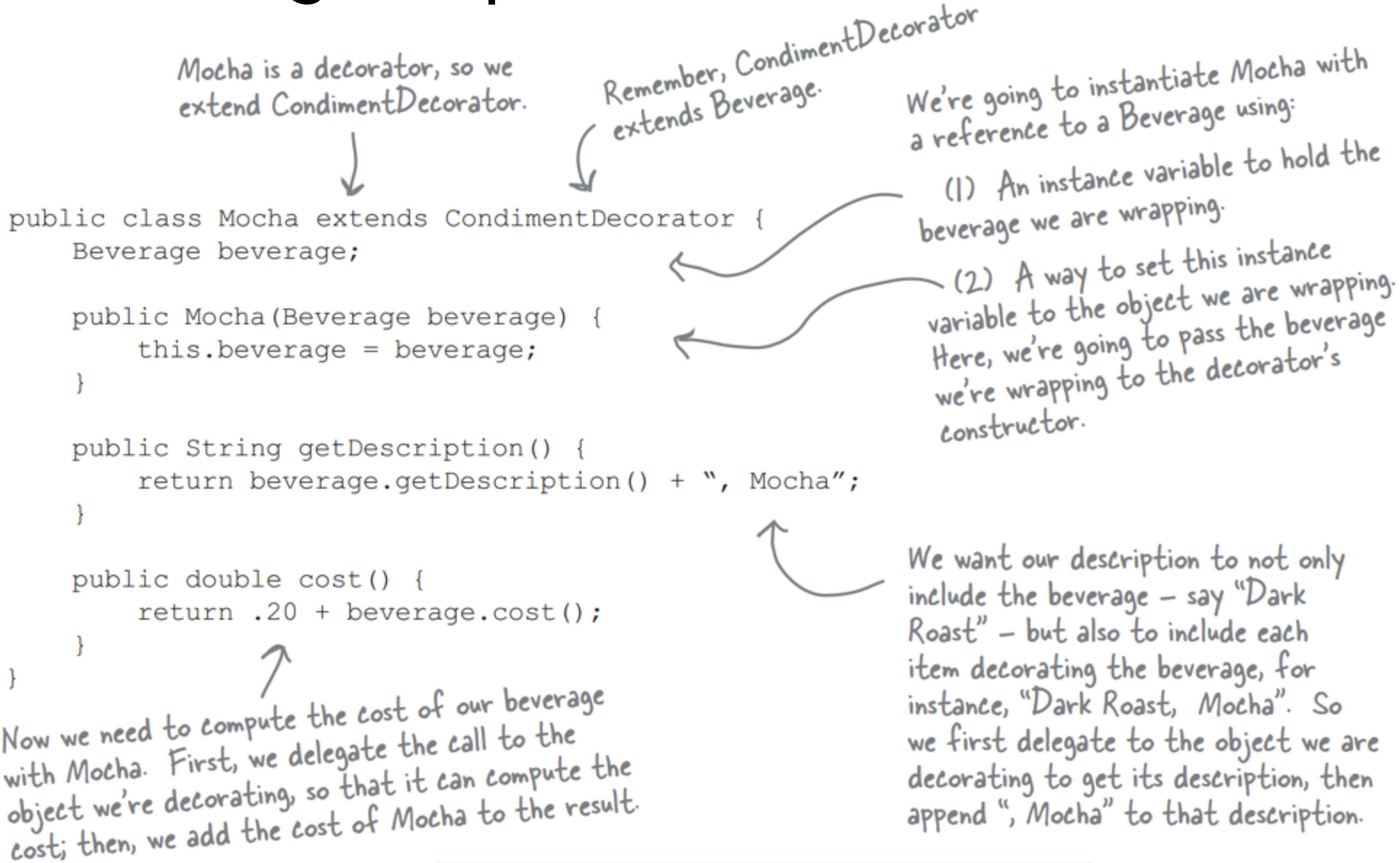
Wrap it in a second Mocha.

Wrap it in a Whip.

Then you chain those together

Finally, give us a HouseBlend with Soy, Mocha, and Whip.

Solving this problem with Decorators



Mocha is a decorator, so we extend CondimentDecorator.

Remember, CondimentDecorator extends Beverage.

We're going to instantiate Mocha with a reference to a Beverage using:

(1) An instance variable to hold the beverage we are wrapping.

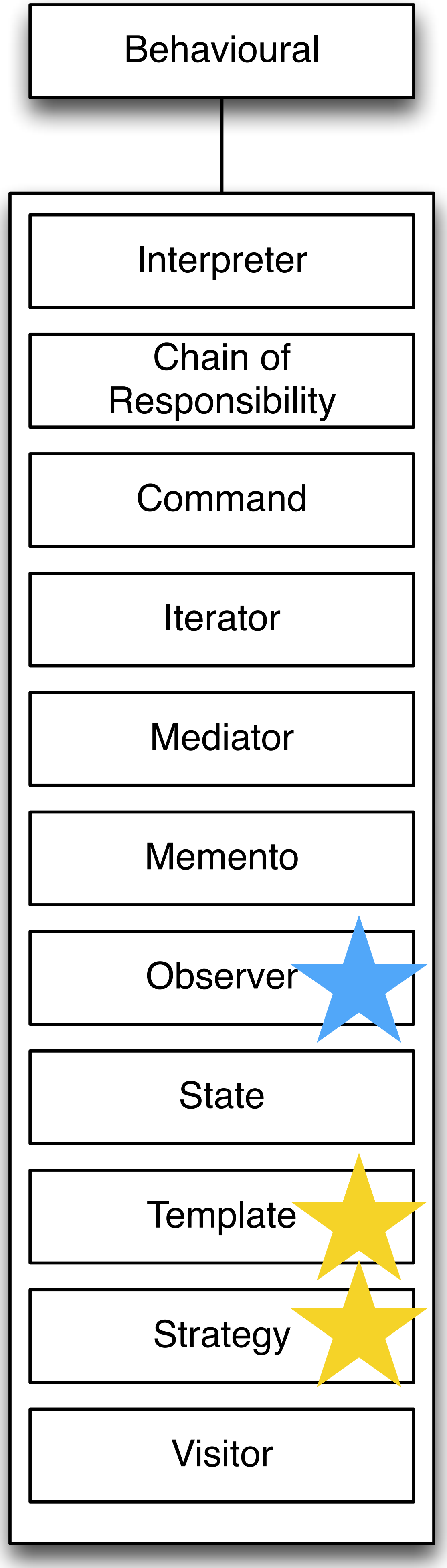
(2) A way to set this instance variable to the object we are wrapping. Here, we're going to pass the beverage we're wrapping to the decorator's constructor.

```
public class Mocha extends CondimentDecorator {
    Beverage beverage;

    public Mocha(Beverage beverage) {
        this.beverage = beverage;
    }

    public String getDescription() {
        return beverage.getDescription() + ", Mocha";
    }

    public double cost() {
        return .20 + beverage.cost();
    }
}
```

We want our description to not only include the beverage – say "Dark Roast" – but also to include each item decorating the beverage, for instance, "Dark Roast, Mocha". So we first delegate to the object we are decorating to get its description, then append ", Mocha" to that description.

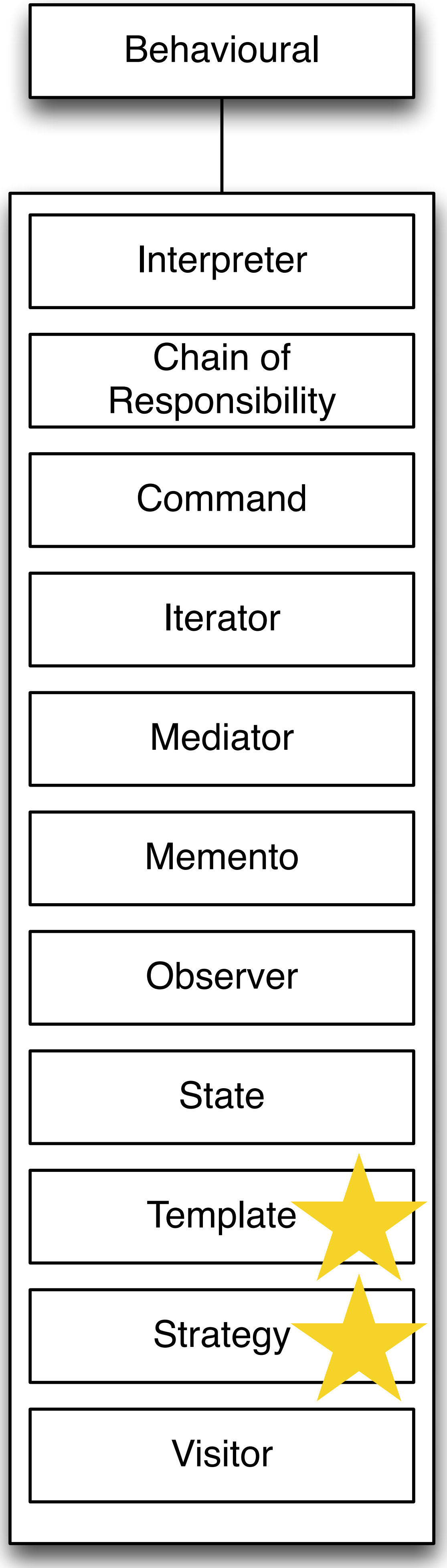
Now we need to compute the cost of our beverage with Mocha. First, we delegate the call to the object we're decorating, so that it can compute the cost; then, we add the cost of Mocha to the result.

and then "get description" and "cost" are computed down their subtrees



Behavioural

Interpreter	interpreting sentences in a grammar
Chain of Responsibility	a chain of objects, passing along (or not) command objects
Command	an object that encloses a generic command. Provides a nice generic “execute” method to fire the command whatever the underlying behaviour is.
Iterator	allows a collection to be iterated over by adding “next()” “hasNext()” and other iteration-friendly operations to their abstractions.
Mediator	acts as a communication hub for multiple objects
Memento	saves state; allows rollback
Observer ★	lets an object watch other objects (we just did this one)
State	strategy pattern but based on an object’s state
Template ★	defines the skeleton of a program, for others to fill in the details.
Strategy ★	if you want to change behaviour of an object at runtime
Visitor	iterate over a hierarchy to take both caller and callee type into account without a huge loop+case statement.



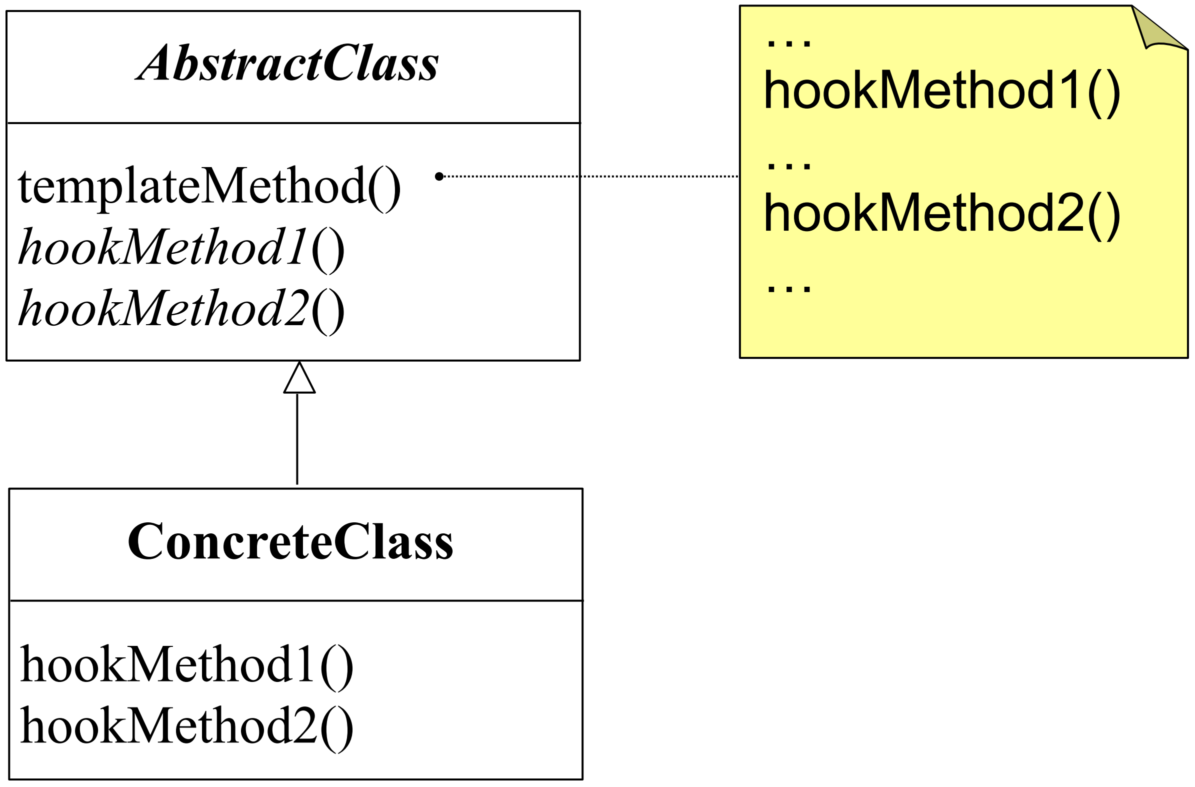
Behavioural

- Interpreter
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Template ★
- Strategy ★
- Visitor

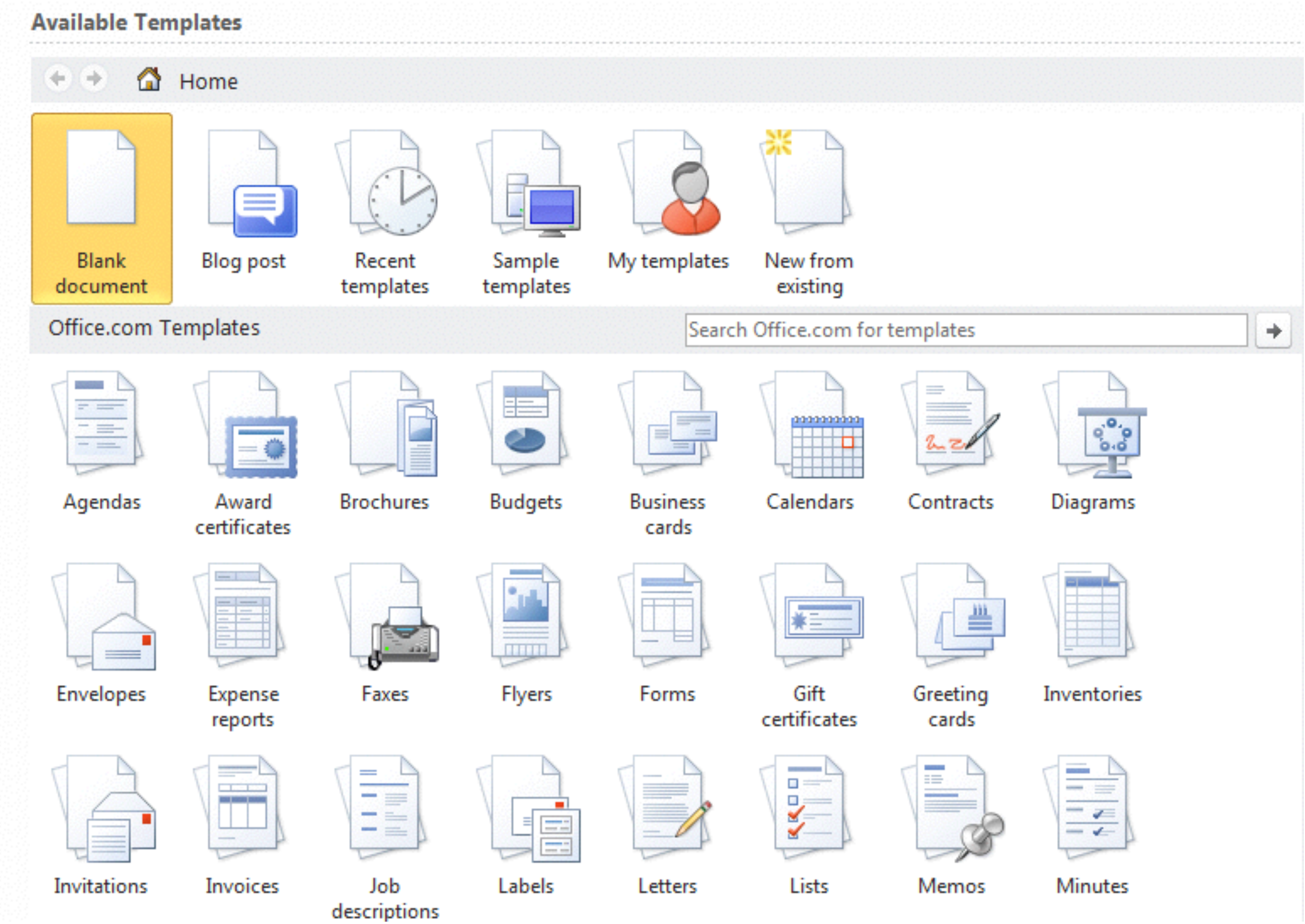
I want the behaviour of my system to follow certain steps, but I want clients to be able to define each of those steps for themselves

Template Method

- Intent
  - To define a skeleton algorithm by deferring some steps to subclasses
  - To allow the subclasses to redefine certain steps
- Generic Structure



Metaphor: Templates in Word



Example

- Want to provide custom rendering of output according to some visual pattern
- For real life example, consider example of web page rendering: header, body, footer, etc…

Input: “JAVA”

Output 1:

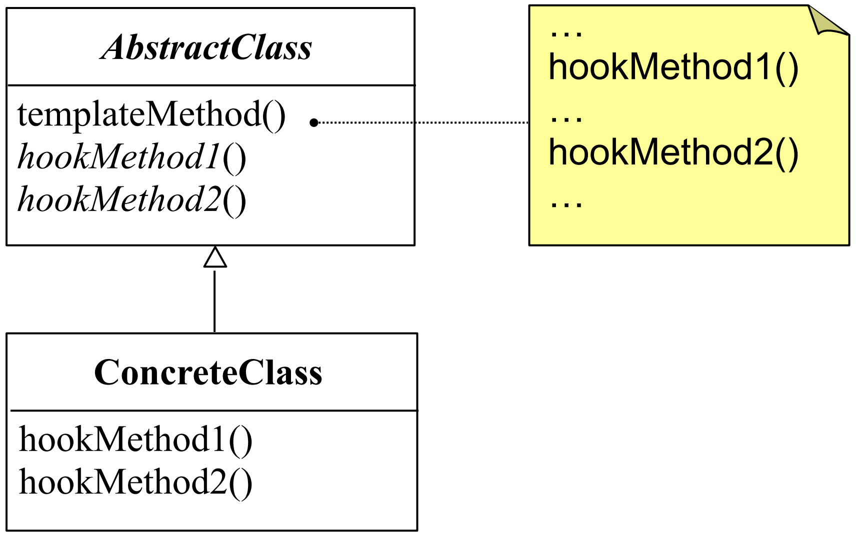
```
******
*JAVA*
******
```

Output 2:

```
~~~~~~
|JAVA|
------
```

Participants

- **AbstractClass**
  - defines abstract *primitive operations* that concrete subclasses define to implement steps of an algorithm
  - implements a template method defining the skeleton of an algorithm. The template method calls primitive operations as well as operations defined in AbstractClass or those of other objects.
- **ConcreteClass**
  - implements the primitive operations to carry out subclass-specific steps of the algorithm



Note: hookMethod sometimes called concreteMethod

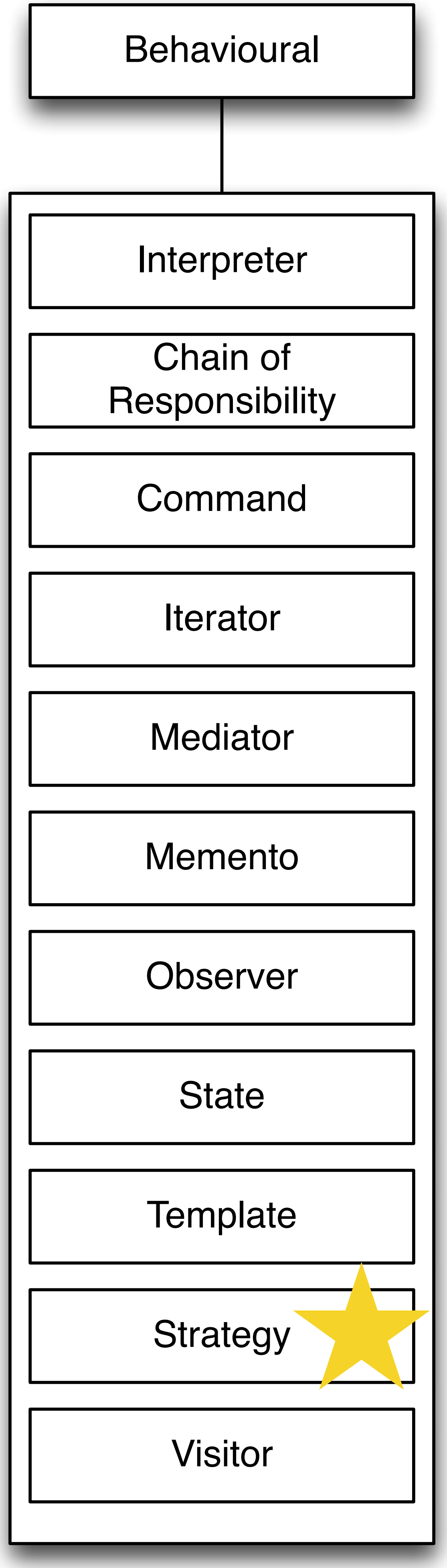
Template Method Example

```
public abstract class StringRenderer {
        //Template method
     public final void render(String str)
     {
         for(int i=0; i< str.length() + 2; i++)
             printTopCharacter();
             System.out.println();
             printLeftCharacter();
             System.out.print(str);
             printRightCharacter();
             System.out.println();
             for(int i=0; i < str.length() + 2; i++)
                  printBottomCharacter();
     }
        //"hooks"
        protected abstract void printTopCharacter();
        protected abstract void printLeftCharacter();
        protected abstract void printRightCharacter();
        protected abstract void printBottomCharacter();
}
```

abstract class

```
public class StarRenderer extends StringRenderer {
        protected void printTopCharacter() { System.out.print(“*”); }
        protected void printLeftCharacter() { System.out.print(“*”); }
        protected void printRightCharacter() { System.out.print(“*”); }
        protected void printBottomCharacter() { System.out.print(“*”); }
}
```

concrete class



Behavioural

Interpreter

Chain of Responsibility

Command

Iterator

Mediator

Memento

Observer

State

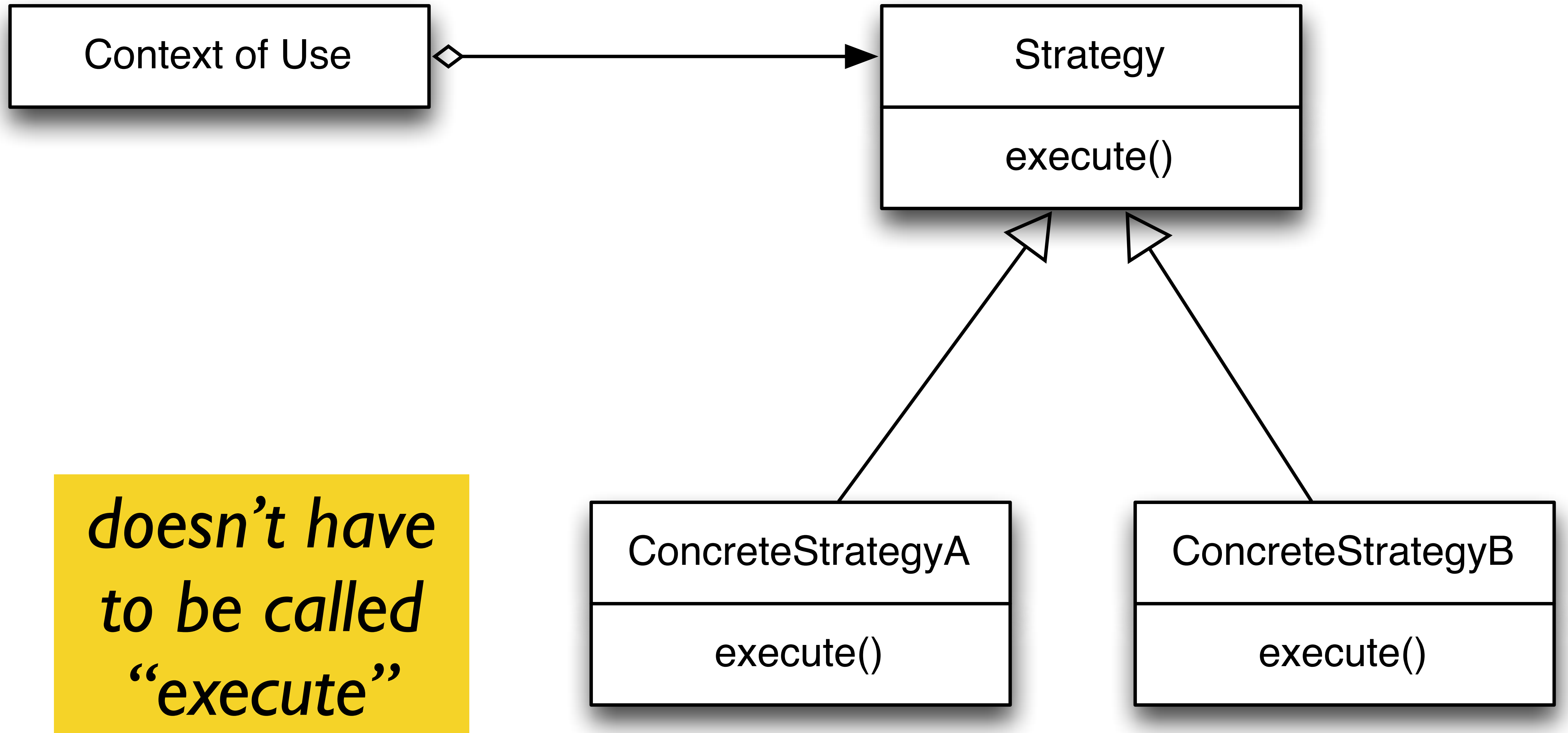
Template

Strategy ★

Visitor

I want my class to do something in particular, but I want the client to pick the strategy (or algorithm) it uses to do it

Strategy Pattern



doesn't have to be called "execute"

```
class Doer:
    method do_it(strategy):
        strategy.execute()
```

```
class FastWay is_a Strategy:
    method execute():
        //do it fast!
```

```
class SlowWay is_a Strategy:
    method execute():
        //do it slowly!
```

```
if user selects "fast"
    Doer.do_it(new FastWay)
if user selects "slow"
    Doer.do_it (new SlowWay)
```

Strategy pattern lets clients override the algorithm (strategy) used for certain parts of the program's computation while the code is running

To Sum Up

A lot of patterns are very related. Many patterns use one another to facilitate their work. Some are useful in very slightly different situations.

In industry, patterns are not dogma — patterns are advice.
But for our purposes, we aim to be able to distinguish them, and apply them correctly.