Universität Zürich^UZH

Institut für Informatik

Software Engineering HS'16

Lecture: Design Patterns

Thomas Fritz & Martin Glinz

Many thanks to Philippe Beaudoin, Gail Murphy, David Shepherd, Neil Ernst and Meghan Allen

Reading!

For this lecture: (all required)

- Composite Design Pattern
  http://sourcemaking.com/design_patterns/composite
- Mediator Design Pattern
  http://sourcemaking.com/design_patterns/mediator
- Facade Design Pattern
  http://sourcemaking.com/design_patterns/facade

Design Patterns Overview

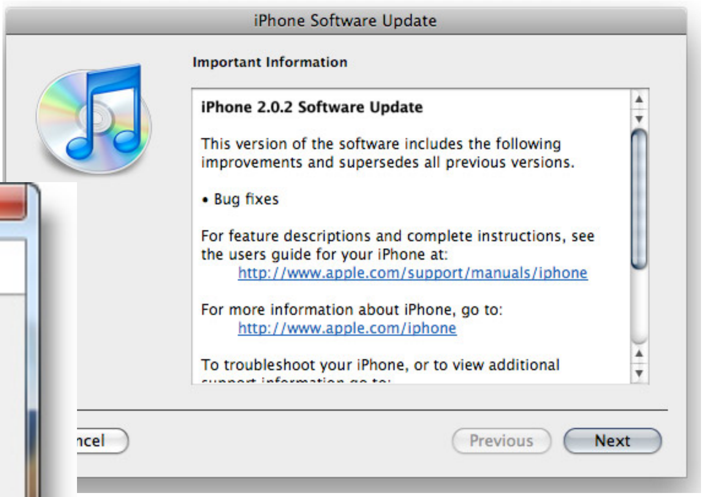
- Introduction to design patterns
- How to use design patterns
- Components of a pattern
- Various patterns
  - Creational
  - Structural
  - Behavioral
- Integrating Patterns

Learning Goals

By the end of this unit, you will be able to:

- **Explain** why design patterns are useful and some **caveats** to consider when using them
- Clearly and concisely **describe**, give **examples** of software situations in which you'd use, explain the key **benefit** of, and **drawbacks** or special considerations for the presented design patterns

Software Updates



Important Information

iPhone 2.0.2 Software Update

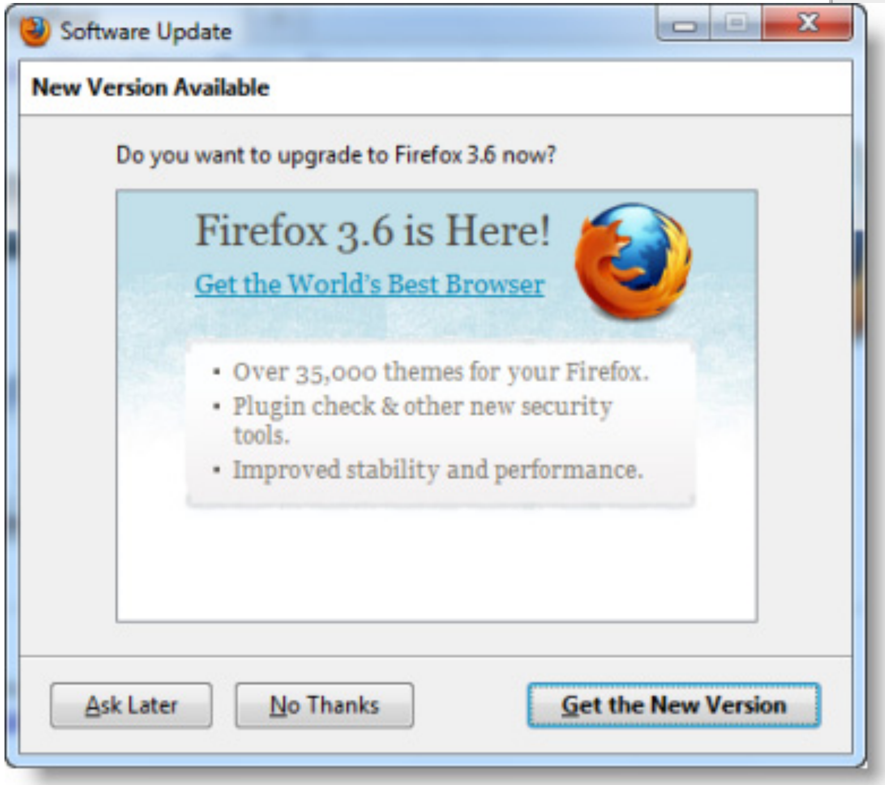
This version of the software includes the following improvements and supersedes all previous versions.

• Bug fixes

For feature descriptions and complete instructions, see the users guide for your iPhone at:
http://www.apple.com/support/manuals/iphone

For more information about iPhone, go to:
http://www.apple.com/iphone

To troubleshoot your iPhone, or to view additional



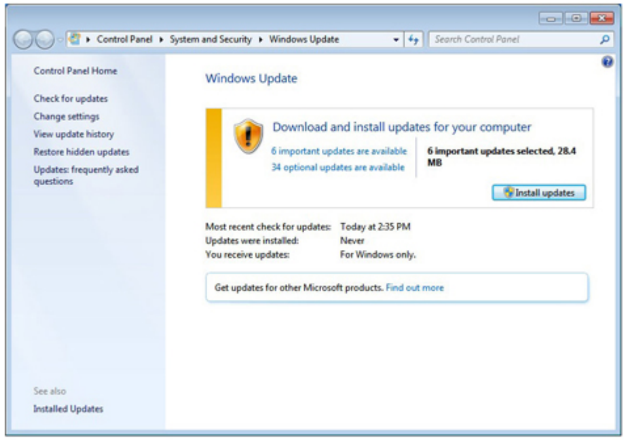
New Version Available

Do you want to upgrade to Firefox 3.6 now?

Firefox 3.6 is Here!

Get the World's Best Browser

- Over 35,000 themes for your Firefox.
- Plugin check & other new security tools.
- Improved stability and performance.



Windows Update

Download and install updates for your computer

6 important updates are available

34 optional updates are available

6 important updates selected, 28.4 MB

Most recent check for updates: Today at 2:35 PM
Updates were installed: Never
You receive updates: For Windows only.

Get updates for other Microsoft products. Find out more

Design Challenges

- Designing software with good modularity is hard!
- Designs often emerge from a lot of trial and error

Are there solutions to common recurring problems?

Essentially...

- A Design Pattern is:

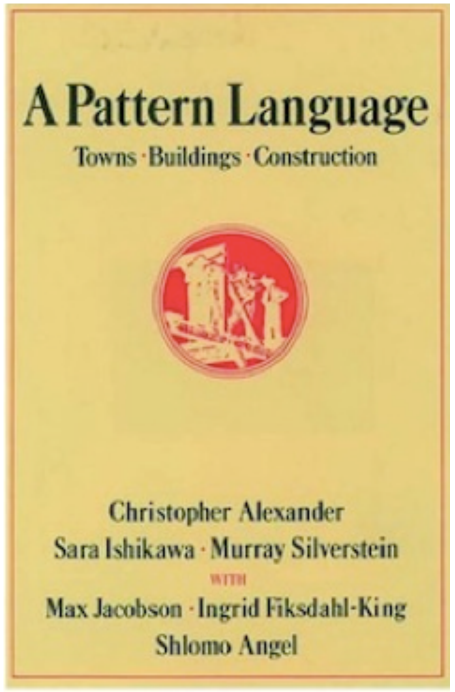
A Tried and True Solution
To a Common Problem

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

The “Design Patterns” name

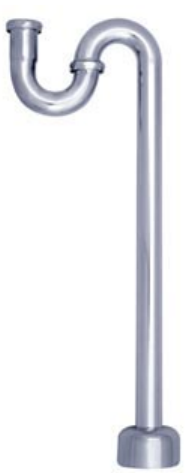
- original use really comes from (building) architecture from Christopher Alexander

- It was used for 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

Real World Pattern Examples

- Problem:
  sink stinks
- Pattern:
  S-trap



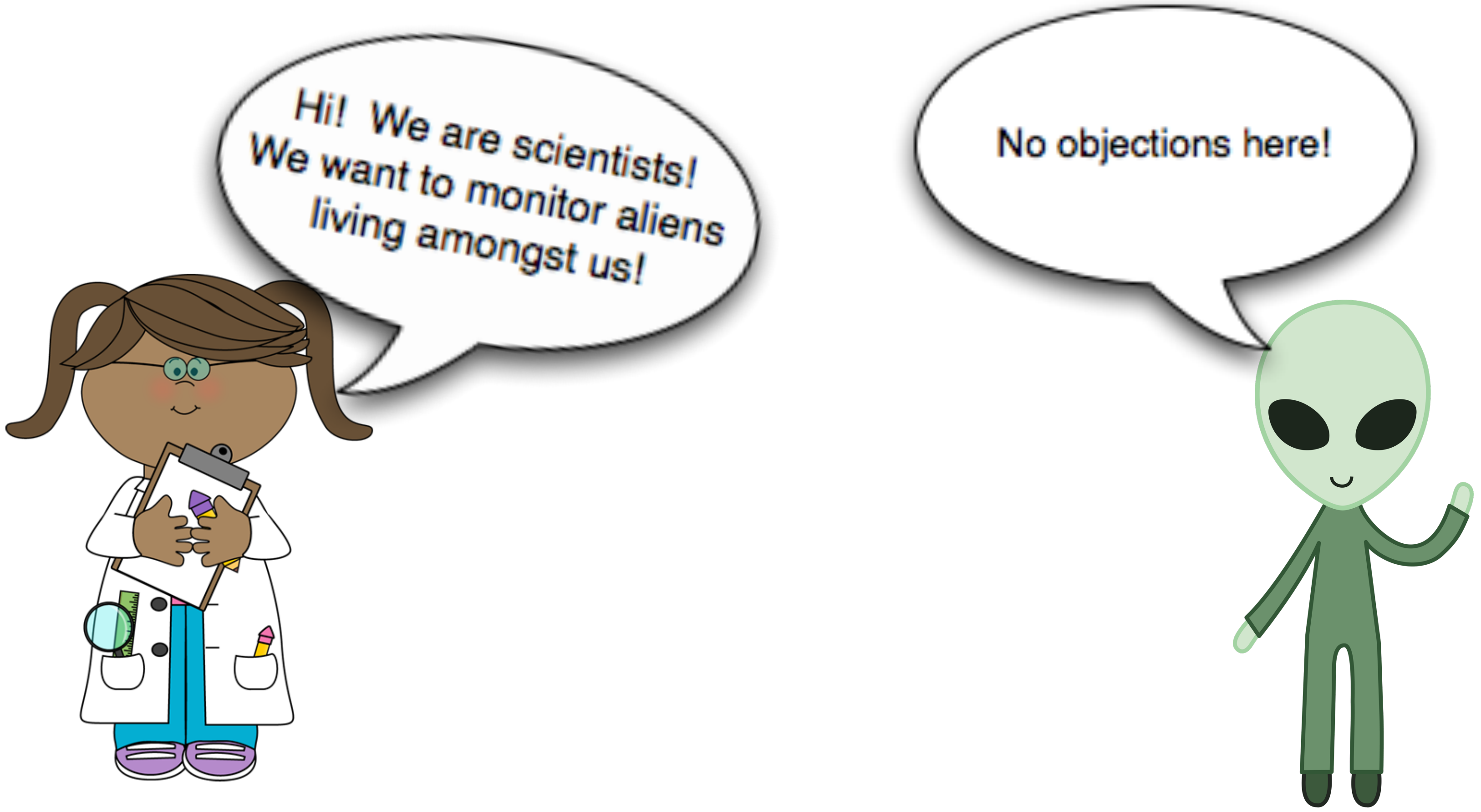
- Problem:
  highway crossing
- Pattern:
  clover leaf

Design Patterns

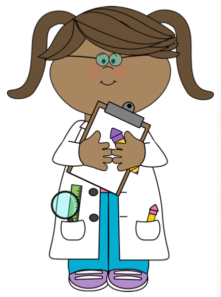
In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.

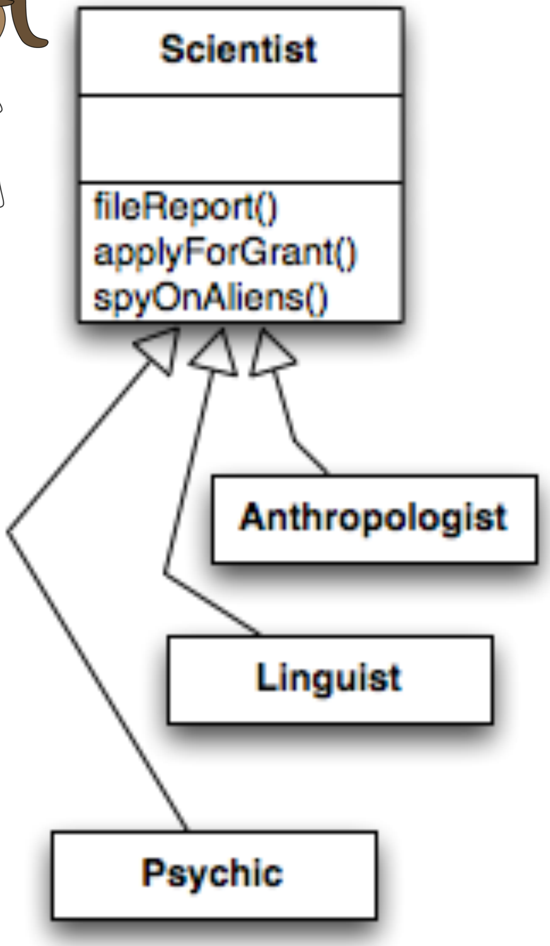
- A design pattern is a description or **template** for how to solve a problem
- Not a finished design
- Patterns capture design expertise and allow that expertise to be transferred and reused
- Patterns provide common design **vocabulary**, improve **communication**, ease **implementation** & documentation

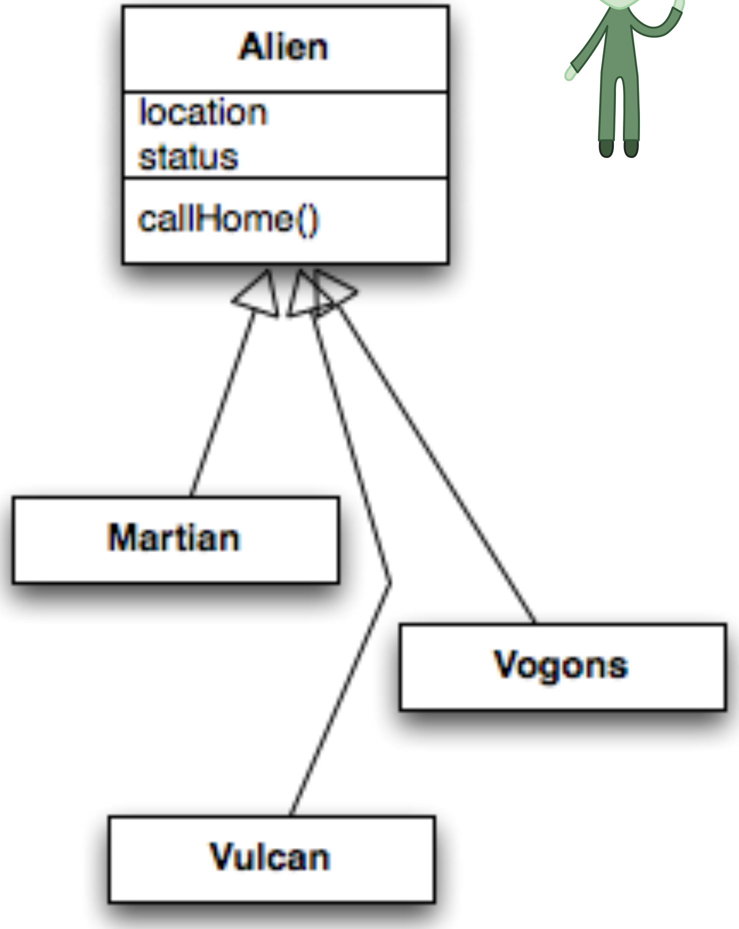
Back to the Update problem


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

A basic design…







Update Example cont'd

Current design very haphazard spying on aliens

- You'd need to know the exact alien you're spying on, aliens would need to know the exact scientist and what they want…

How to design a better protocol?

- Have all 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 one is best?

Updates – Observer Design Pattern

Observer

where the “**Observer**” watches the “**Subject**”

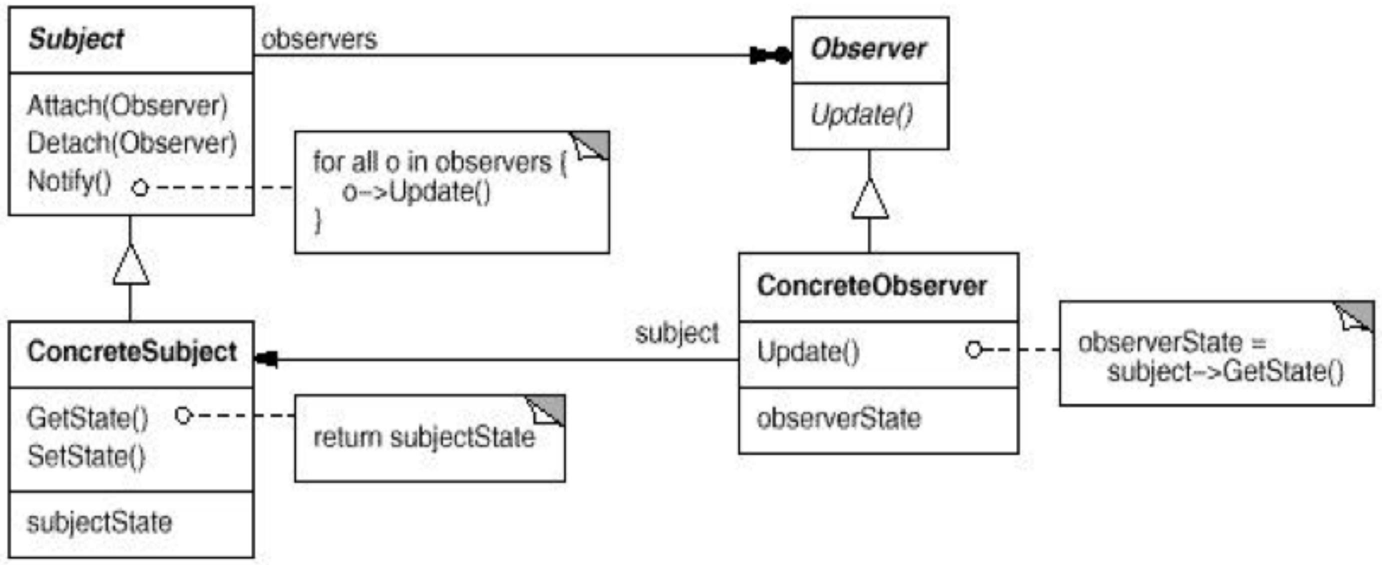
Subject

Observer Design Pattern

Name: Observer

Intent: Ensure that, when an object changes state, all its dependents are notified and updated automatically.

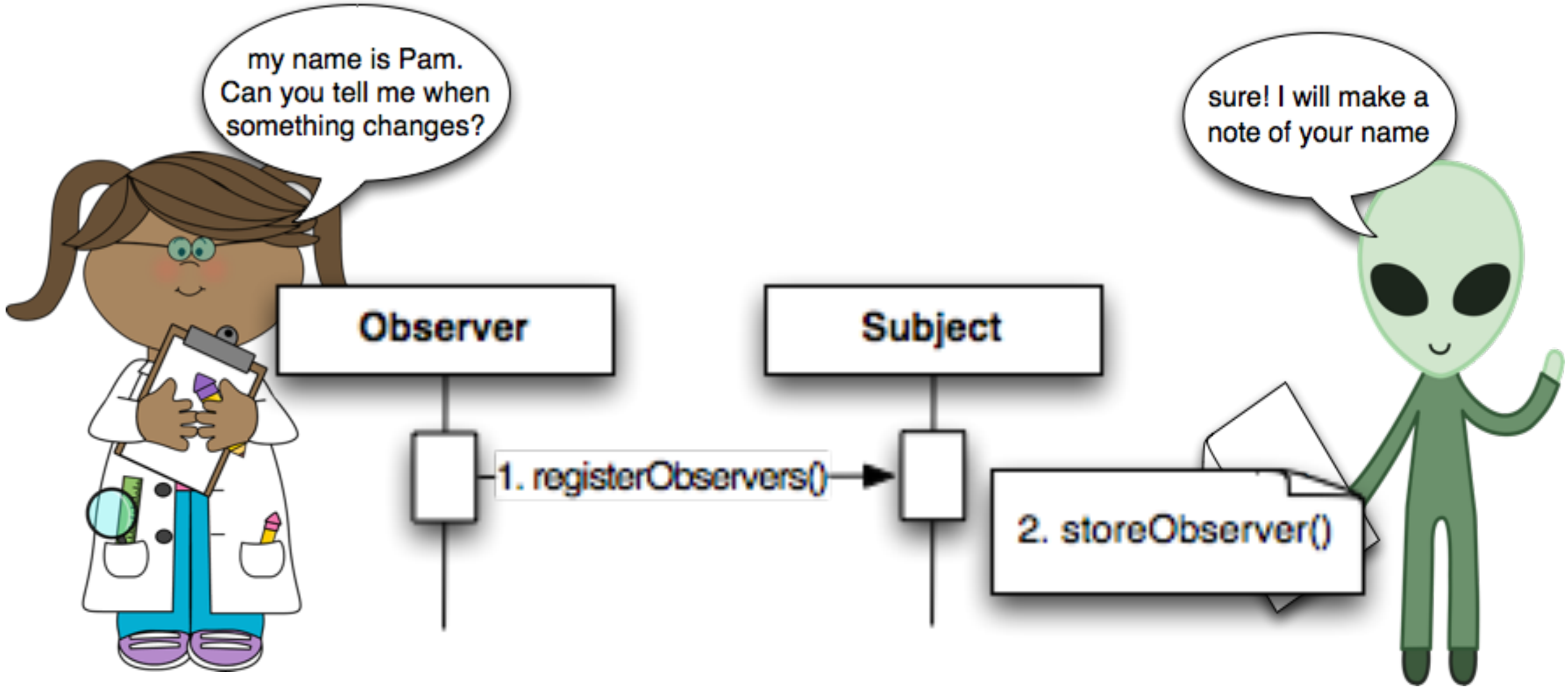
Participants & Structure:



Step 1: Observer Registers

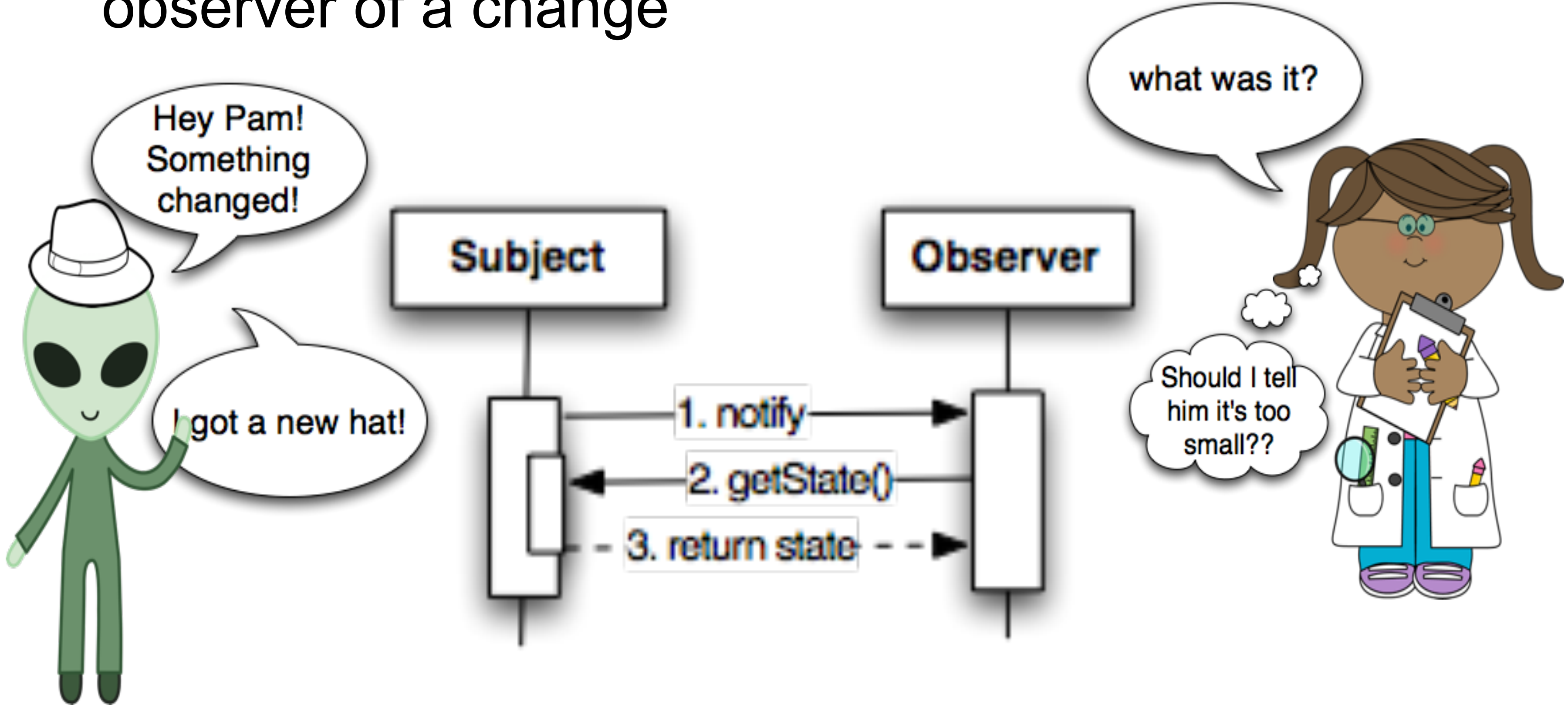
The observer has to register/attach with the subject for updates.

Subject stores them in a list/record to contact them later.


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

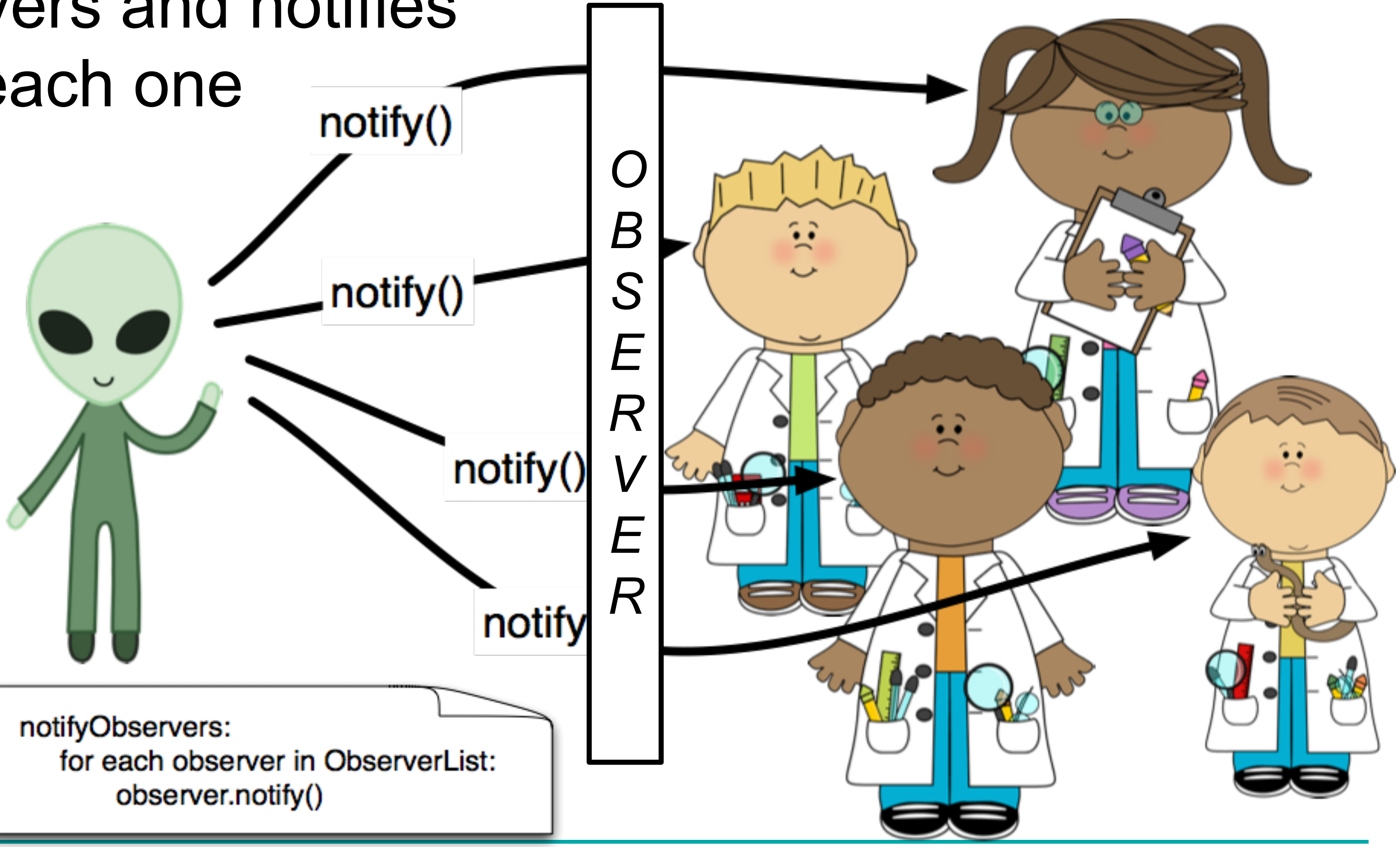
Step 2: Notify Observers

subject notifies the observer of a change



Variation: lots of Observers

Subject loops through list of observers and notifies each one



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

Observer DP (cont'd)

- I need the ***professor*** to be notified when a ***student*** joins his/her class
- I want the ***display*** to update when the size of a ***window*** is changed
- I need the ***schedule view*** to update when the ***database*** is changed

Design patterns are reusable!

Real world example

Newspaper subscriptions

- The newspaper company publishes newspapers.
- You subscribe to a particular paper, and every time there's a new paper it is delivered to you.
- At some point in the future, you can unsubscribe and the papers won't be delivered anymore.
- While the newspaper company is in business, people, hotels and other businesses constantly subscribe and unsubscribe to the newspaper.

In this example, who is the Observer and who is the Subject?

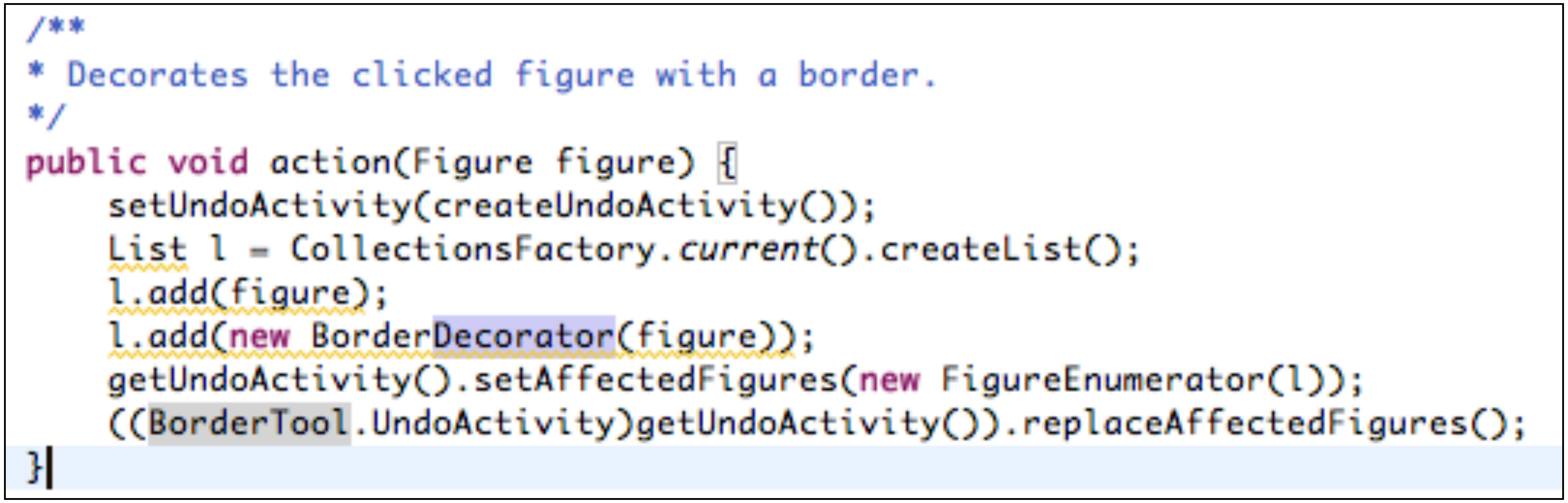
example from Head First Design Patterns

How to use Design Patterns?

- Part “Craft”
  - Know the patterns
  - Know the problem they can solve
- Part “Art”
  - Recognize when a problem is solvable by a pattern
- Part “Science”
  - Look up the pattern
  - Correctly integrate it into your code

Knowing the patterns helps understanding code

- The pattern sometimes convey a lot of information
- Try understanding this code:



```java
/**
 * Decorates the clicked figure with a border.
 */
public void action(Figure figure) {
    setUndoActivity(createUndoActivity());
    List l = CollectionsFactory.current().createList();
    l.add(figure);
    l.add(new BorderDecorator(figure));
    getUndoActivity().setAffectedFigures(new FigureEnumerator(l));
    ((BorderTool.UndoActivity)getUndoActivity()).replaceAffectedFigures();
}
```

- Key is to know the *Abstract Factory* and *Decorator* patterns!

Design patterns also provide a shared vocabulary.

Dev 1: “I made a Broadcast class. It keeps track of all of its listeners and anytime it has new data it sends a message to each listener. The listeners can join the Broadcast at any time or remove themselves from the Broadcast. It’s really dynamic and loosely-coupled!”

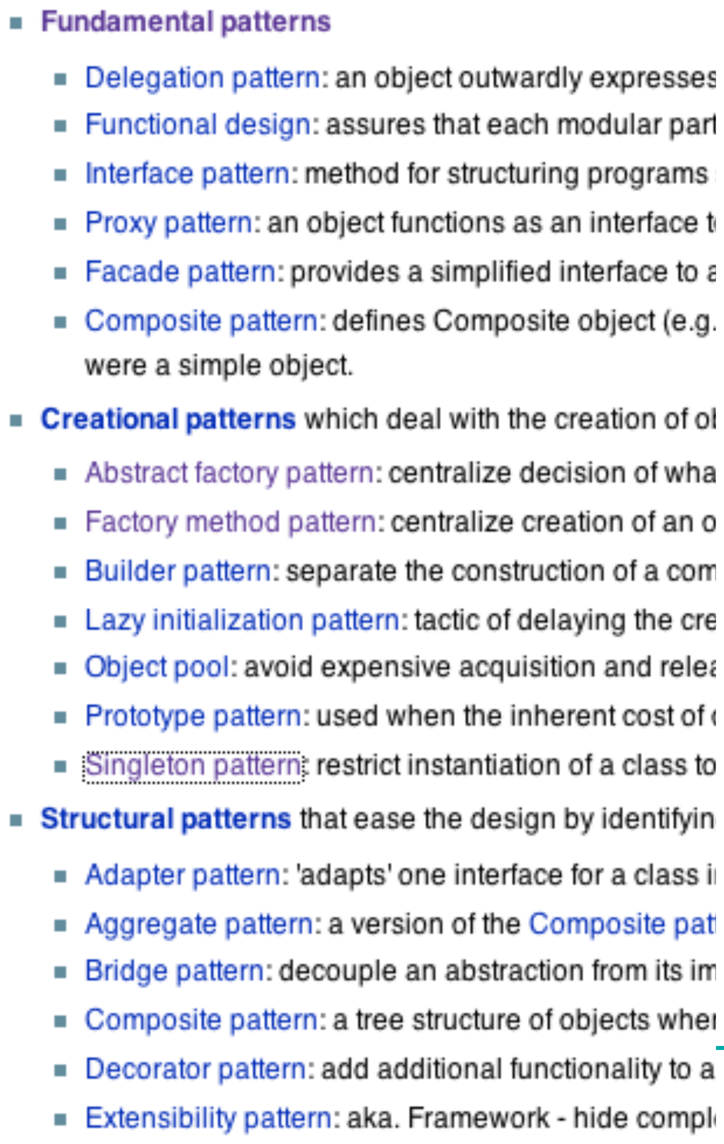
Dev 2: “Why didn’t you just say you were using the Observer pattern?”

example from Head First Design Patterns

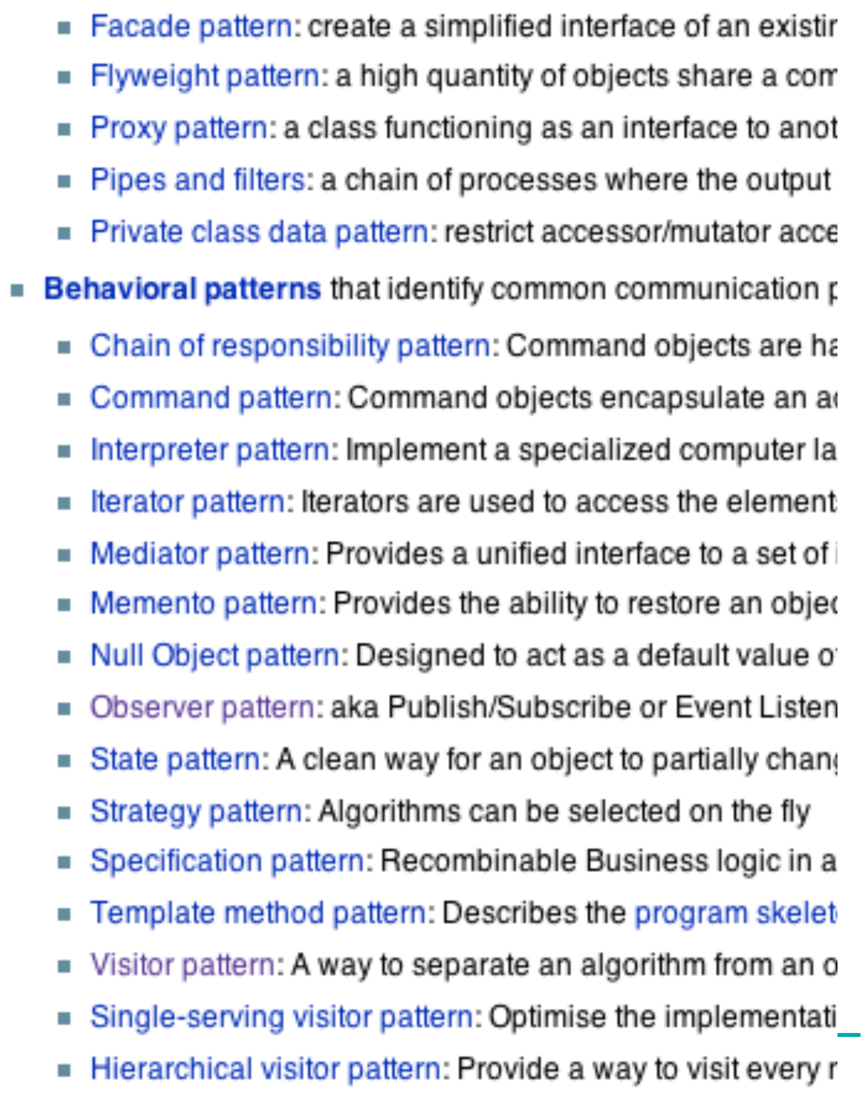
Components of a pattern

- Pattern Name
- Intent
  - What problem does it solve?
- Participants
  - What classes participate
    - These classes usually have very general names, the pattern is meant to be used in many situations!
- Structure
  - How are the classes organized?
  - How do they collaborate?

A Menagerie of Patterns!



- Fundamental patterns
  - Delegation pattern: an object outwardly expresses
  - Functional design: assures that each modular part
  - Interface pattern: method for structuring programs
  - Proxy pattern: an object functions as an interface t
  - Facade pattern: provides a simplified interface to a
  - Composite pattern: defines Composite object (e.g. were a simple object.
- Creational patterns which deal with the creation of ol
  - Abstract factory pattern: centralize decision of wha
  - Factory method pattern: centralize creation of an o
  - Builder pattern: separate the construction of a com
  - Lazy initialization pattern: tactic of delaying the cre
  - Object pool: avoid expensive acquisition and relea
  - Prototype pattern: used when the inherent cost of
  - Singleton pattern: restrict instantiation of a class to
- Structural patterns that ease the design by identifyin
  - Adapter pattern: 'adapts' one interface for a class i
  - Aggregate pattern: a version of the Composite pat
  - Bridge pattern: decouple an abstraction from its im
  - Composite pattern: a tree structure of objects whe
  - Decorator pattern: add additional functionality to a
  - Extensibility pattern: aka. Framework - hide compl



  - Facade pattern: create a simplified interface of an existir
  - Flyweight pattern: a high quantity of objects share a com
  - Proxy pattern: a class functioning as an interface to anot
  - Pipes and filters: a chain of processes where the output
  - Private class data pattern: restrict accessor/mutator acce
- Behavioral patterns that identify common communication p
  - Chain of responsibility pattern: Command objects are ha
  - Command pattern: Command objects encapsulate an a
  - Interpreter pattern: Implement a specialized computer la
  - Iterator pattern: Iterators are used to access the element
  - Mediator pattern: Provides a unified interface to a set of i
  - Memento pattern: Provides the ability to restore an objec
  - Null Object pattern: Designed to act as a default value o
  - Observer pattern: aka Publish/Subscribe or Event Listen
  - State pattern: A clean way for an object to partially chan
  - Strategy pattern: Algorithms can be selected on the fly
  - Specification pattern: Recombinable Business logic in a
  - Template method pattern: Describes the program skelet
  - Visitor pattern: A way to separate an algorithm from an o
  - Single-serving visitor pattern: Optimise the implementati
  - Hierarchical visitor pattern: Provide a way to visit every r

Pattern Classifications

Creational Patterns

- deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
- Useful as system evolve: the classes that will be used in the future may not be known now

Structural Patterns

- ease the design by identifying a simple way to realize relationships between entities
- Techniques to compose objects to form larger structures

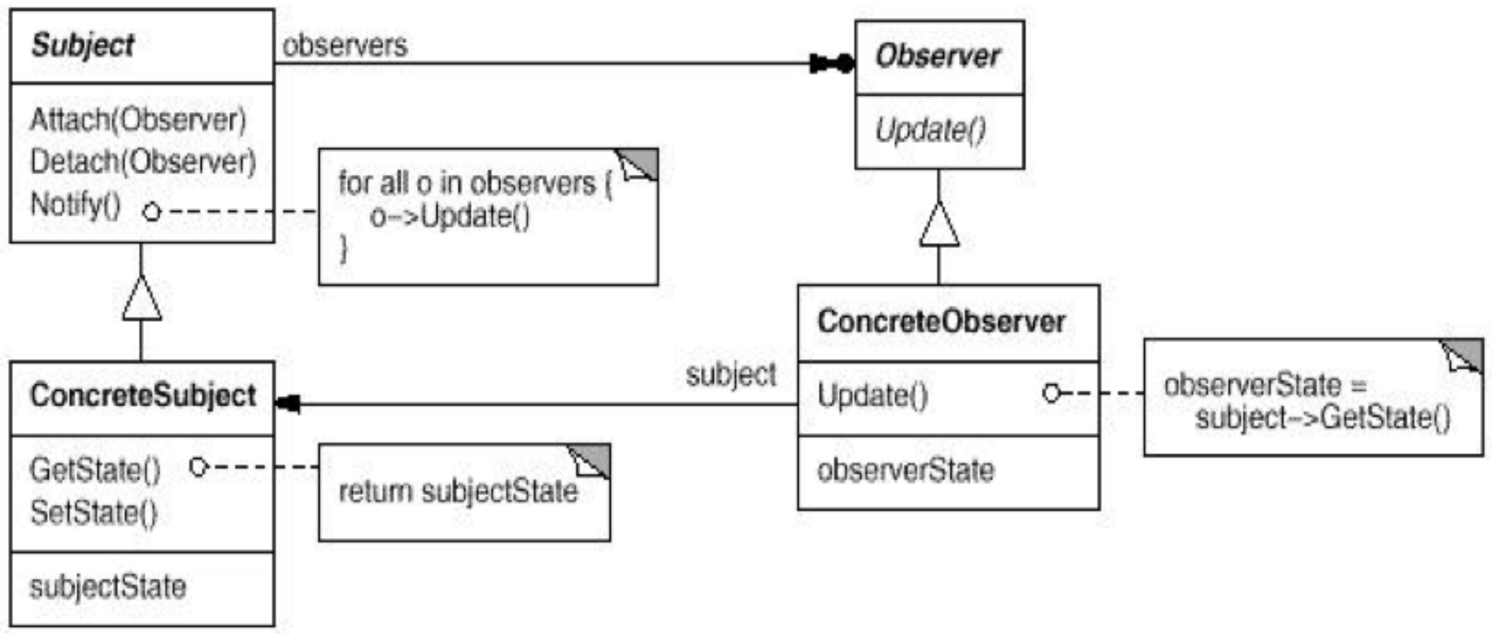
Behavioral Patterns

- Concerned with communication between objects (common communication patterns)
- Describe complex control flow

Discussion Question

Which class does the *Observer* pattern belong to?

- Creational, Structural, Behavioural?



Behavioral Patterns

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

Communication hub for multiple objects

Let's an object watch other objects

Iterate over a hierarchy…

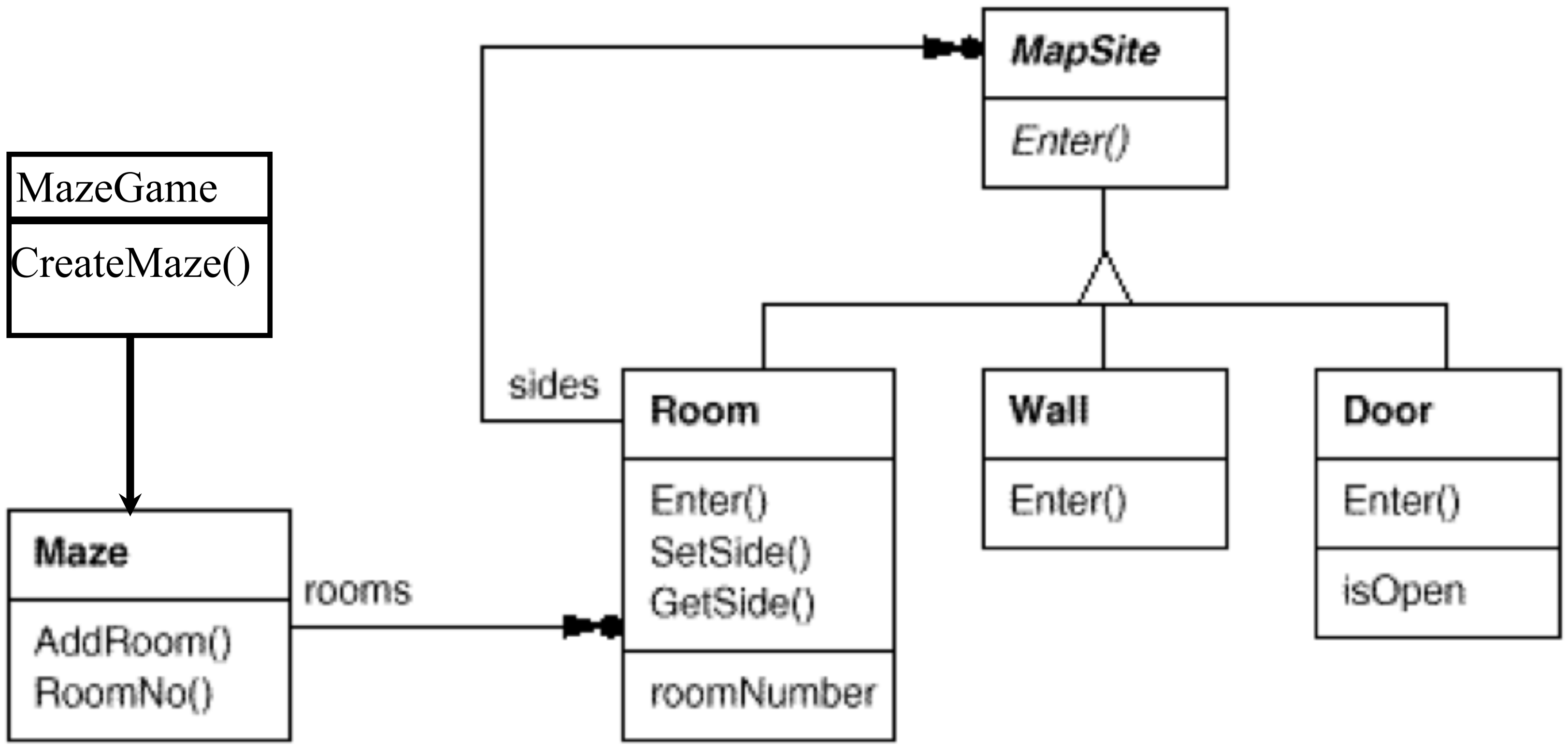
…

Creational Patterns

Singleton	*make one thing*
Factory Method	*make something*
Abstract Factory	*make a family of somethings*
Builder	*make something slowly*
Prototype	*clone something*
…	

Design problem

- Build a maze for a computer game
- A maze is a set of rooms
- A room knows its neighbours: room, door, wall
- Ignore players, movement, etc.


MapSite
Enter()
MazeGame
CreateMaze()
sides
Room
Enter()
SetSide()
GetSide()
roomNumber
Wall
Enter()
Door
Enter()
isOpen
Maze
AddRoom()
RoomNo()
rooms

Exercise

1. Implement the function MazeGame:CreateMaze() to design a maze with 2 rooms and a connecting door.

2. Update that function to make a Maze containing a Room with a bomb in it.

Example

```
// in the class MazeGame
public Maze createMaze() {
  Maze maze = new Maze();
  Room room = new Room();
  Room room2 = new Room();
  Door door = new Door();
  maze.addRoom(room);
  maze.addRoom(room2);
  maze.addDoor(door);
  return maze;
}
```

What's wrong with this?

We can only use this method to create a maze that uses a Room and a Door. What if we want to create a different type of maze?

example from Design Patterns by Gamma et al. 33

Example cont'd

```
// in the class MazeGame
public Maze createBombMaze() {
  Maze maze = new BombMaze();
  Room room = new RoomWithABomb();
  Room room2 = new RoomWithABomb();
  Door door = new Door();
  maze.addRoom(room);
  maze.addRoom(room2);
  maze.addDoor(door);
  return maze;
}
```

Example cont'd

```
// in the class MazeGame
public Maze createEnchantedMaze() {
  Maze maze = new Maze();
  Room room = new EnchantedRoom();
  Room room2 = new EnchantedRoom();
  Door door = new DoorNeedingSpell();
  maze.addRoom(room);
  maze.addRoom(room2);
  maze.addDoor(door);
  return maze;
}
```

Abstract Factory

Sample Problem:
Your game needs to create *rooms*, but you are not quite sure yet how these *rooms* will be implemented and you think they will be extended in the future.

Solution 1:

```
// TODO: Change next line when we know what is a
// room
Room r = new TempRoom();
// Note: TempRoom is a subclass of Room
```

Problem? (any design principle violated?)

Abstract Factory

Solution 2:

```
// myRoomFactory is an abstract factory!
Room r = myRoomFactory.createRoom();
```

Advantage:

Just set *myRoomFactory* once, then the good room will be created!

Remark:

Setting *myRoomFactory* is referred to as *Dependency Injection*: the class who is dependent on *myRoomFactory* doesn’t retrieve it, but waits until someone else *injects* it.

Solution!

```
// in the class MazeGame
public Maze createMaze(MazeFactory factory) {
  Maze maze = factory.createMaze();
  Room room = factory.createRoom();
  Room room2 = factory.createRoom();
  Door door = factory.createDoor();
  maze.addRoom(room);
  maze.addRoom(room2);
  maze.addDoor(door);
  return maze;
}
```

Now, we can use the same `createMaze` method in all three situations, as long as we pass in a different `MazeFactory` each time

Solution cont'd

In this situation, MazeFactory is a concrete class. Then, the EnchantedMazeFactory and BombedMazeFactory can just override the particular methods that they need.

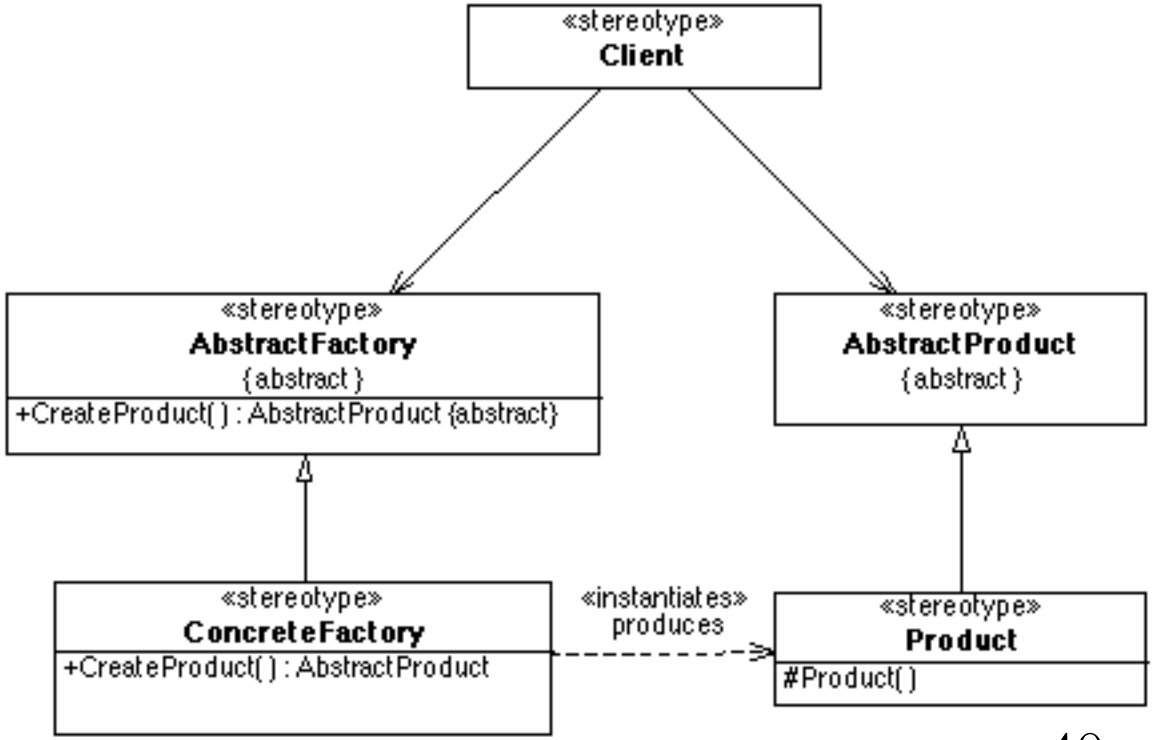
Abstract Factory

Name: Abstract Factory

Intent: Interface for creating families of related objects

Participants

& Structure:



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?

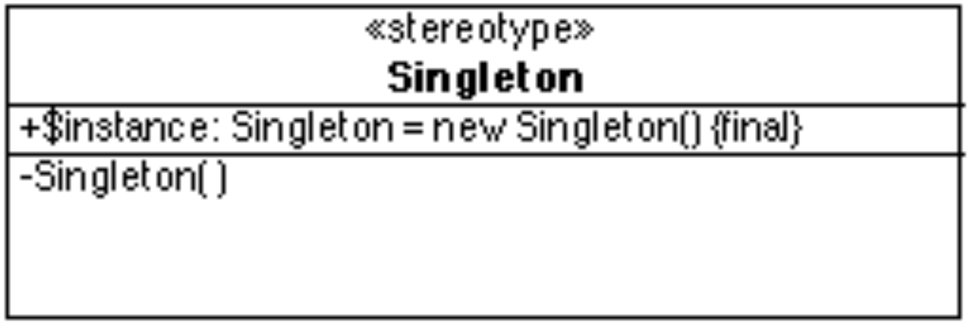
(Question: How could your preferences become inconsistent if your class was instantiated more than once?)

Singleton

Name: Singleton

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

Participants & Structure:



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() {…}
```

Singleton

Is this the only way to solve the problem of a class that should only ever be instantiated once?

- No, of course not! But, like all design patterns, it is a well-tested and well-understood solution.

Structural Patterns

Façade	*Simple interface to a class*
Composite	*Tree structure, uniform access*
Decorator	*Adds to an object's behaviour*
Adapter	*Link between two* hierarchies
Bridge	**…**
Flyweight	
Proxy	

Sample problem

You have created an awesome, but complicated, home theatre system. In order to watch a movie, you have to

- Dim the lights
- Pull down the screen
- Turn the projector on
- Set the projector input to DVD
- Put the projector on widescreen mode
- Turn the sound amplifier on
- Set the sound amplifier input to DVD
- Set the volume
- Turn the DVD player on
- Start the DVD player

example from Head First Design Patterns

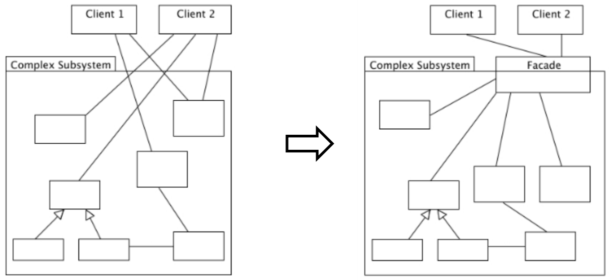
Sample problem cont'd

That sounds complicated!

Wouldn't it be better if you could use a simpler interface to your home theatre system?

The simple interface could allow you to perform common tasks easily. But, you still have full access to your home theatre system if you need to make any changes.

Façade

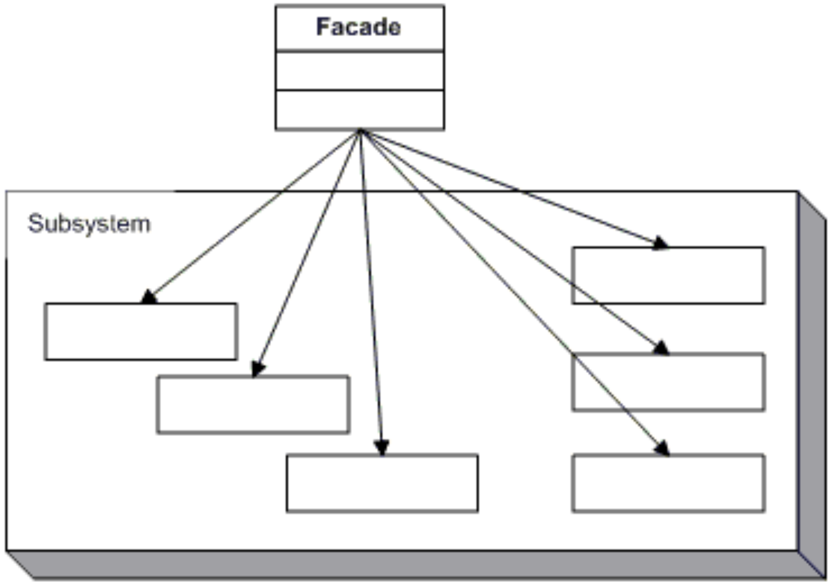


Name: Façade

Intent: Provide a unified interface to a set of interfaces in a subsystem. Defines a higher-level interface. (wrap a complicated interface with a simpler one)

Participants &

Structure:



Software Example

Consider a programming environment that gives applications access to its compiler subsystem.

The subsystem contains classes that implement the compiler (such as Scanner, Parser, Program Node, BytecodeStream and ProgramNodeBuilder)

Some applications may need to access these classes directly, but most applications just want the compiler to compile some code and don't want to have to understand how all the classes work together. The low-level interfaces are powerful, but unnecessarily complex for these applications.

example from Design Patterns by Gamma et al.

Software Example cont'd

In this situation, a Façade can provide a simple interface to the complex subsystem, eg. a class Compiler, with the method compile()

The Façade (Compiler) knows which subsystem classes are responsible for a request and delegates the request to the appropriate subsystem objects

The subsystem classes (Scanner, Parser, etc.) implement the subsystem functionality, handle work assigned by the Façade object and have no knowledge of the Façade object (ie, keep no reference to it)

Sample Problem

You are implementing a menu that has a recursive structure for a restaurant. Their menu contains (sub)menus and/or menu items. Each (sub)menu has (sub)menus and/or menu items.

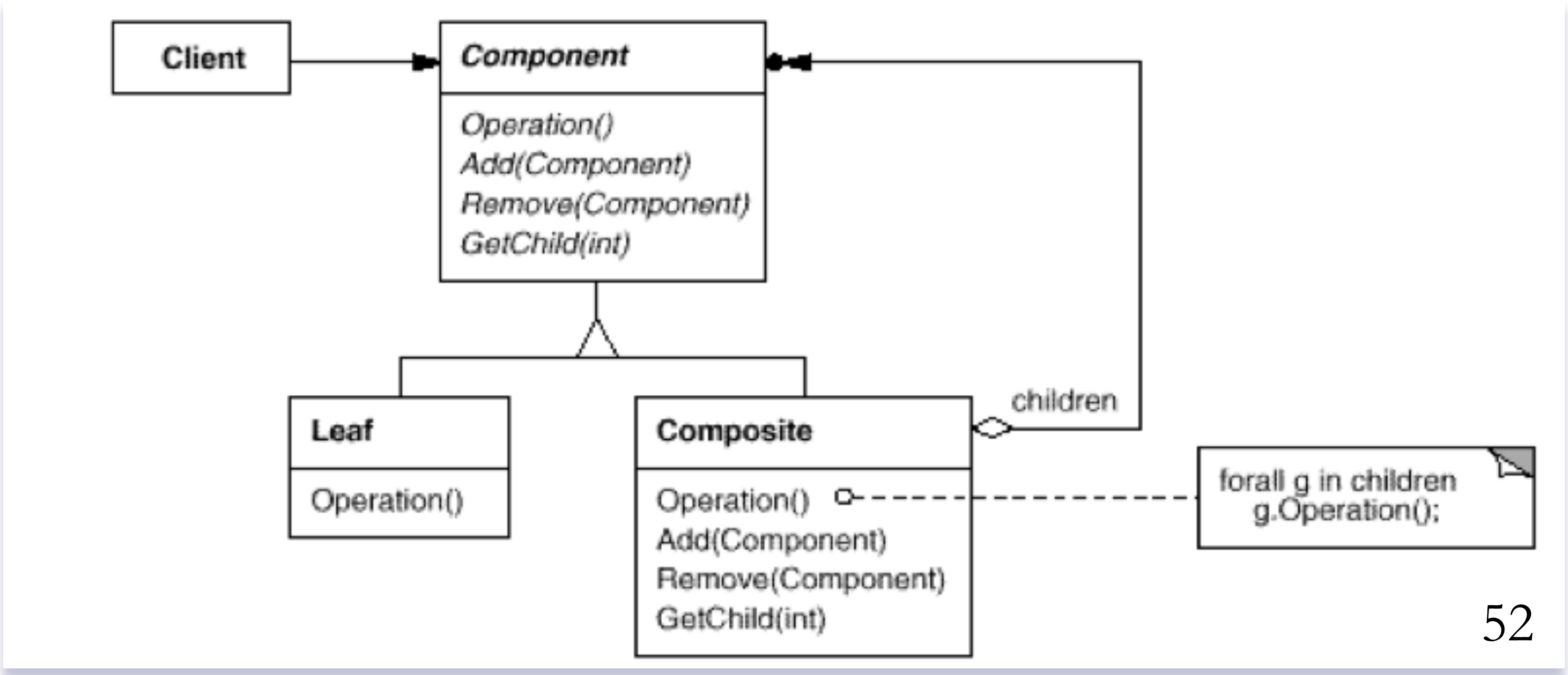
You want to be able to represent this hierarchy, and you want to be able to easily perform operations on the whole menu, or any of its parts.

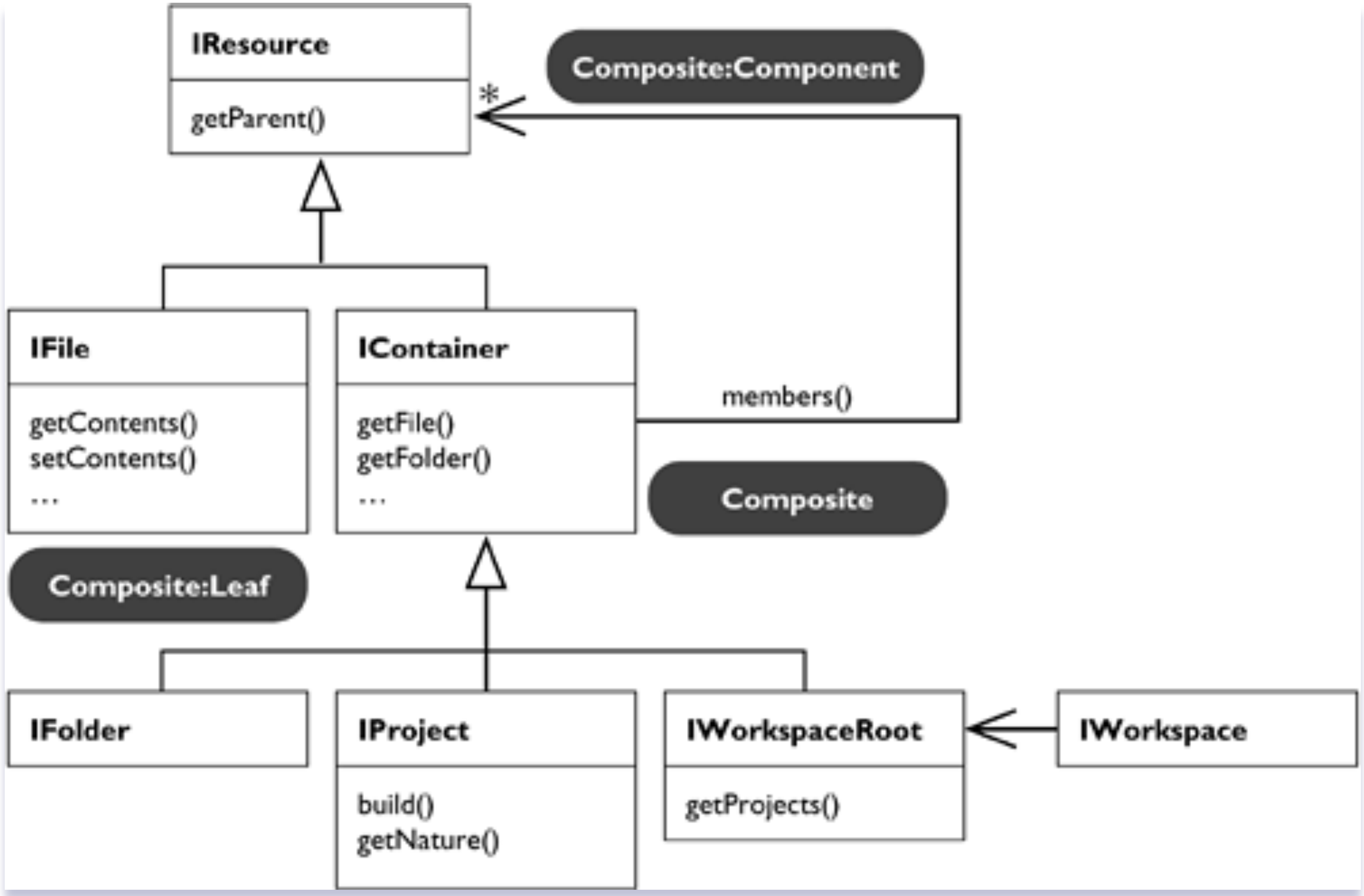
Composite

Name: Composite

Intent: Compose objects into tree structures. Lets clients treat individual objects and compositions uniformly.

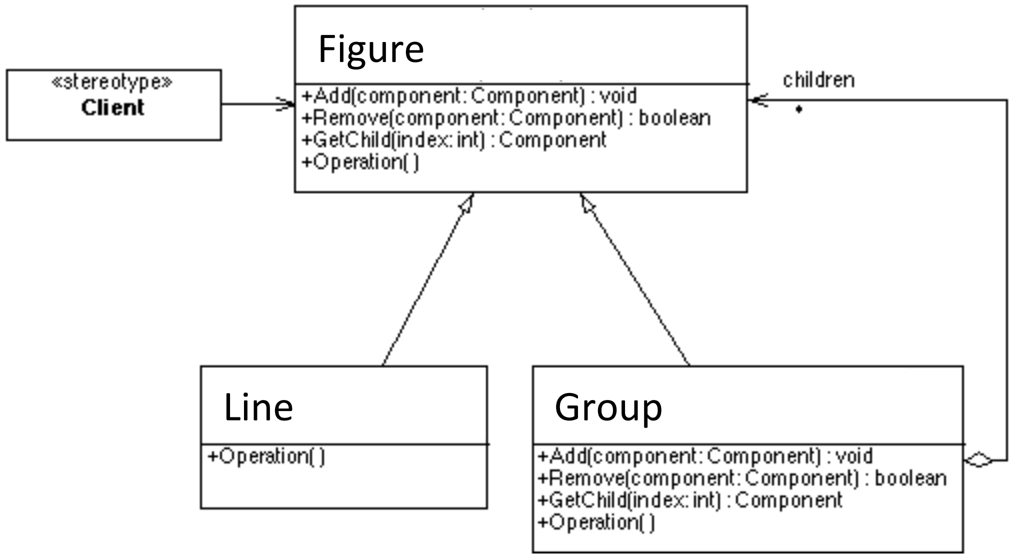
Participants & Structure:




IResource
getParent()
*
Composite:Component
IFile
getContents()
setContents()
...
IContainer
getFile()
getFolder()
...
members()
Composite
Composite:Leaf
IFolder
IProject
build()
getNature()
IWorkspaceRoot
getProjects()
IWorkspace

Software Example

- Drawing application often has figures such as *lines, rectangles, circles…*
- But they also have *groups* of such figures


«stereotype»
Client
Figure
+Add(component: Component) : void
+Remove(component: Component) : boolean
+GetChild(index: int) : Component
+Operation( )
children
*
Line
+Operation( )
Group
+Add(component: Component) : void
+Remove(component: Component) : boolean
+GetChild(index: int) : Component
+Operation( )

Component (Figure)

- declares the interface for objects in the composition and implements any common behaviour
- declares an interface for accessing and managing its child components

Leaf (Line)

- represents leaf objects in the composition (a leaf has no children)
- defines behaviour for figure objects in the composition

Composite (Group)

- defines behaviour for components having children
- stores child components
- implements child related options in the Component interface

Client

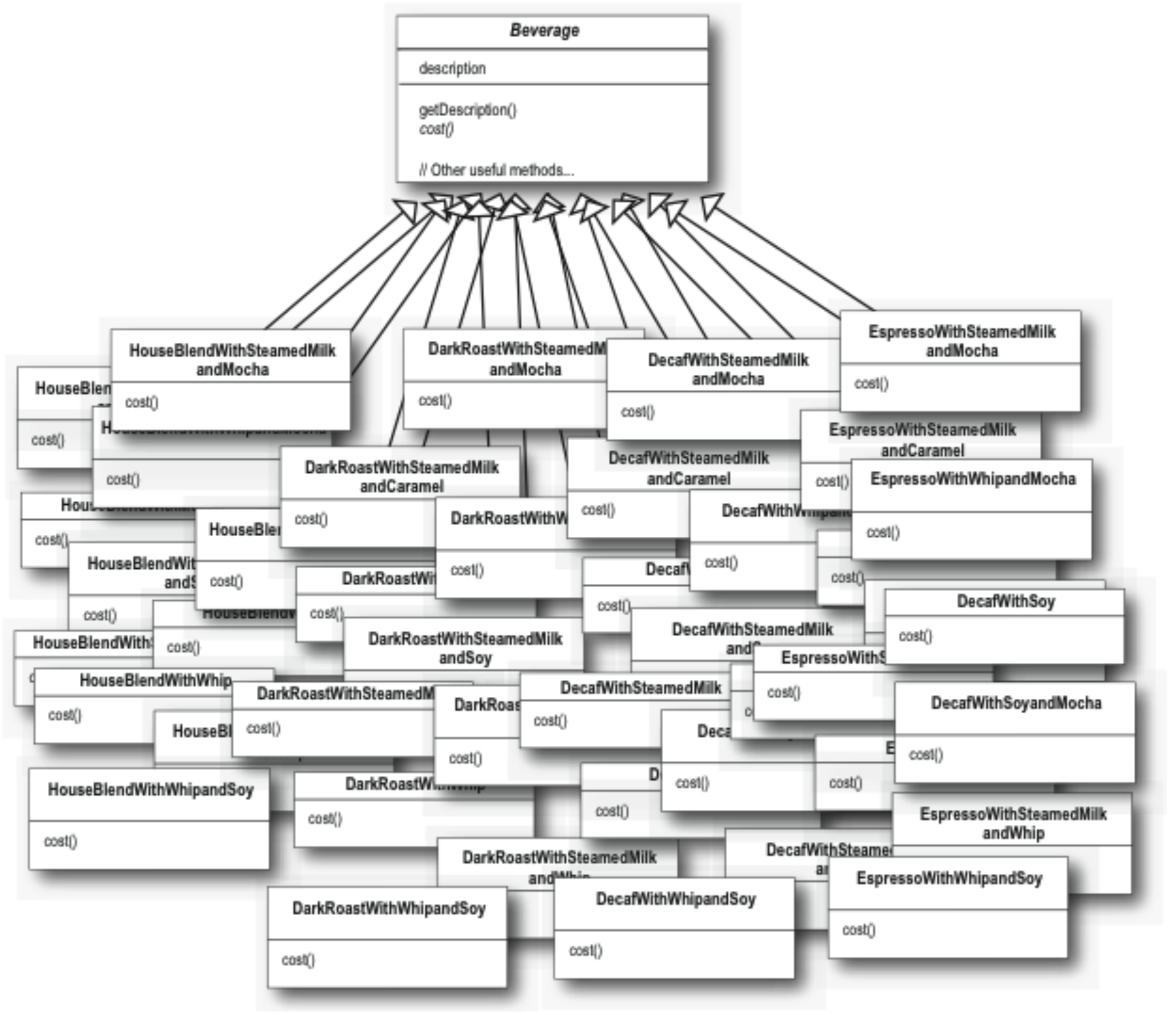
- manipulates objects in the composition through the Component interface

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.

Solving this problem with inheritance



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

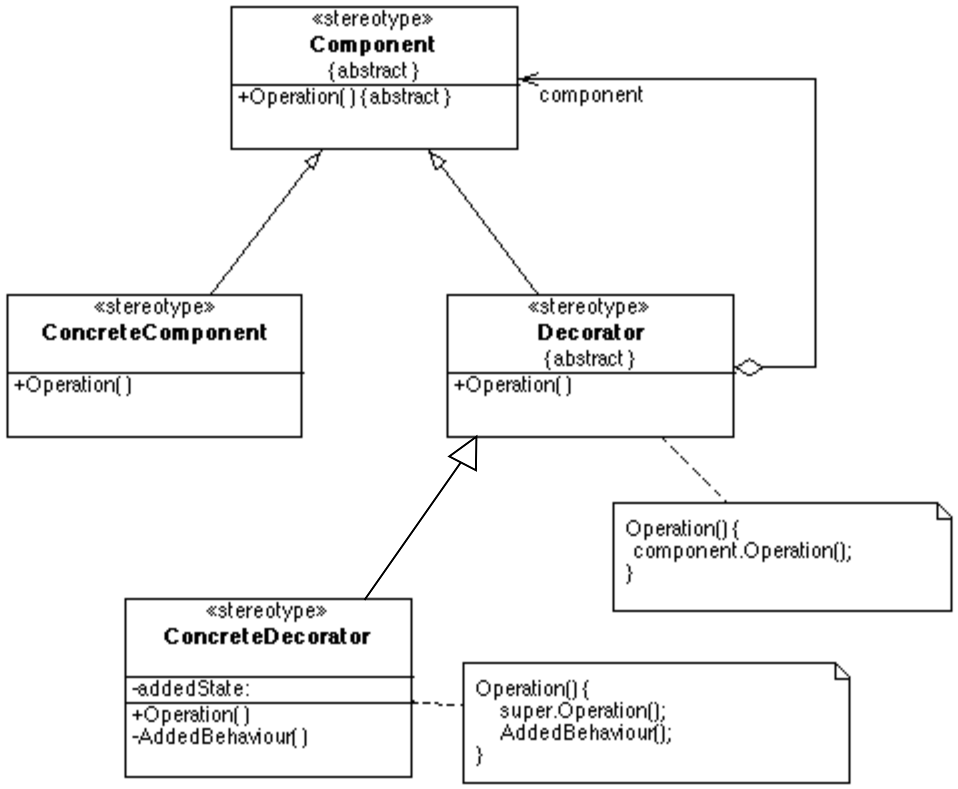
Freeman, et al. *Design Patterns, Head First*

Decorator

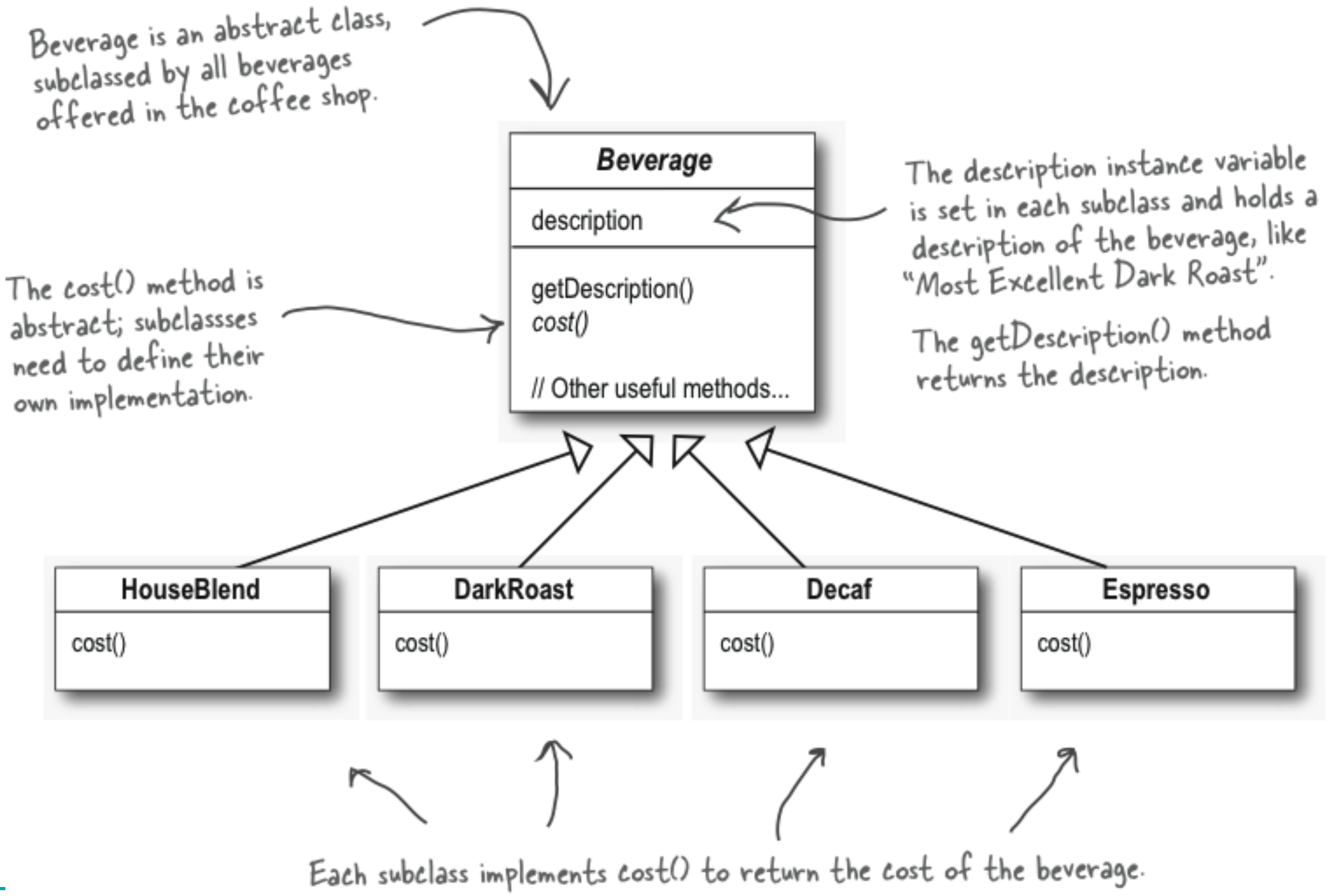
Name: Decorator

Intent: Attach additional responsibilities to an object dynamically

Participants & Structure:

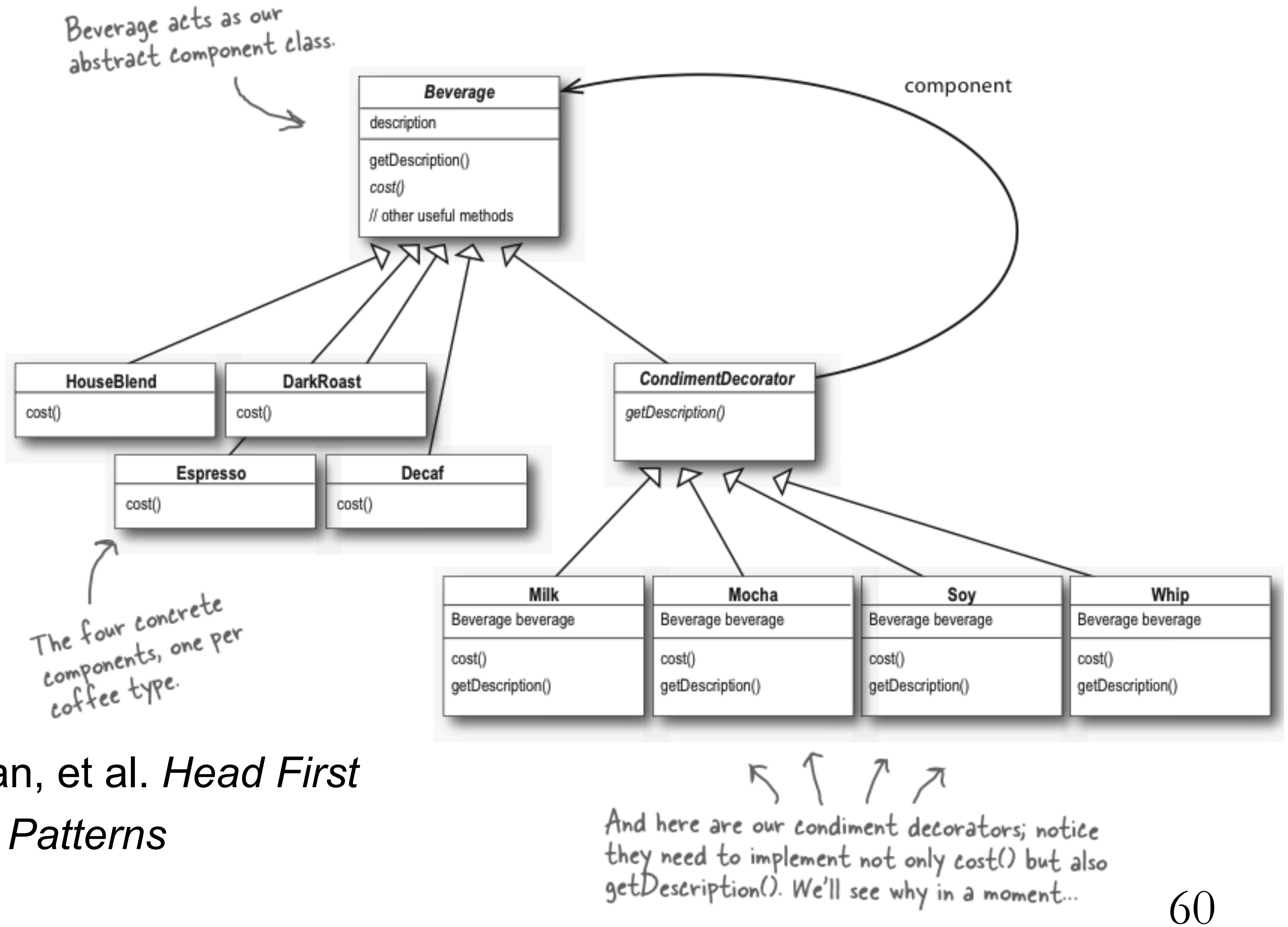


Solving this problem with Decorators


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



Freeman, et al. *Head First Design Patterns*

Class Activity

- How do you create a soy mocha with whip?

How to use Design Patterns

1. Know the **problems** common Design Patterns solve
2. During design, **identify problems** that Design Patterns can solve
3. **Look up** the Design Pattern
4. **Integrate** into design

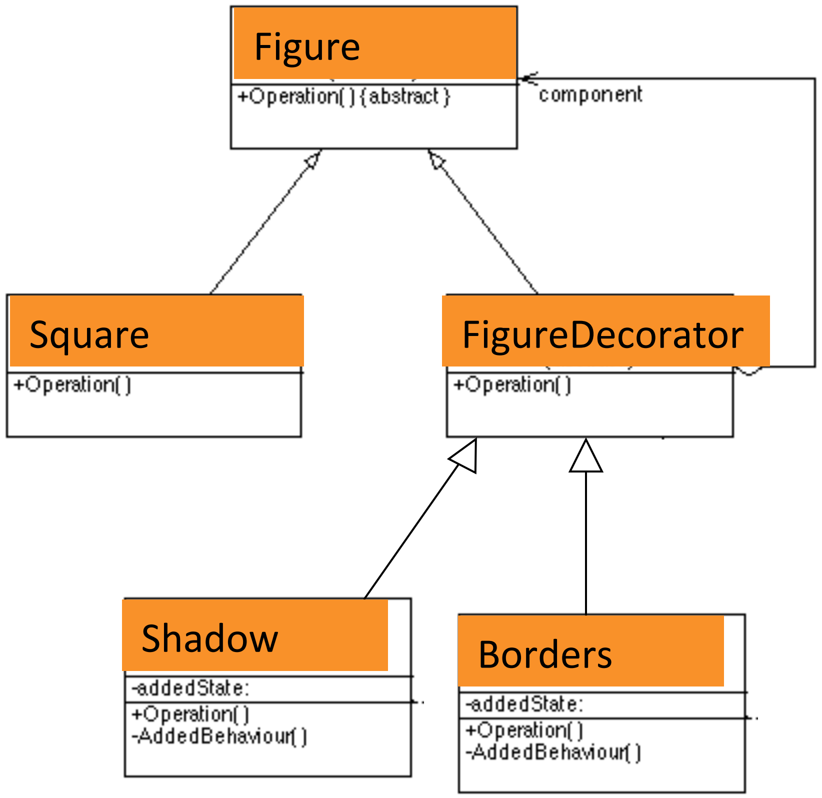
    Find which of your classes should replace the "stereotypes" provided by the pattern

Integrating Patterns: Example 1

- You want to add *borders*, *drop shadows*, *glowing effects*, *outline*… to all the figures in your drawing program
- Which pattern do you use?
- How do you use it?

Integrating Patterns: Example 1

- Look it up, apply it!



Discussion Question

Look-and-Feel:
a GUI framework should support several look and feel standards, such as Motif and Windows look, for its widgets. The widgets are the interaction elements of a user interface such as scroll bars, windows, boxes, buttons. Each style defines different looks and behaviors for each type of widget.

Which pattern is most applicable:

A. Observer

B. Decorator

C. Composite

D. Abstract Factory

Class Activity

The designer of an adventure game wants a player to be able take (and drop) various items found in the rooms of the game. Two of the items found in the game are bags and boxes. Both bags and boxes can contain individual items as well as other bags and boxes. Bags and boxes can be opened and closed and items can be added to or taken from a bag or box.

Choose a pattern and adapt it to this situation

Always this easy?

- No!
- Sometime a pattern won’t work directly
  - Adapt to a situation
  - Use multiple patterns to solve the problem
- First step in mastering patterns?
  - Recognizing them!
  - Take the test (hard!)
    - www.vincehuston.org/dp/patterns_quiz.html

Design Patterns Summary

- Patterns are reusable, abstract “blocks”
- Embody good design principles
- Types of patterns
  - Creational, Structural, Behavioral
- Know your patterns
  - Their name, intent, and structure
  - Master the basic patterns mentioned here
- How to integrate patterns in your designs

Resources

- Gamma, Helm, Johnson, Vlissides. *Design Patterns.* Addison-Wesley.
- Freeman et. Al. Head First Design Patterns.
- Wikipedia (don’t trust it blindly!)
- Bob Tarr’s course
  - http://userpages.umbc.edu/~tarr/dp/spr03/cs491.html
- Quick design patterns reference cards
  - www.mcdonaldland.info/2007/11/28/40/