UNIVERSITY OF WESTMINSTER

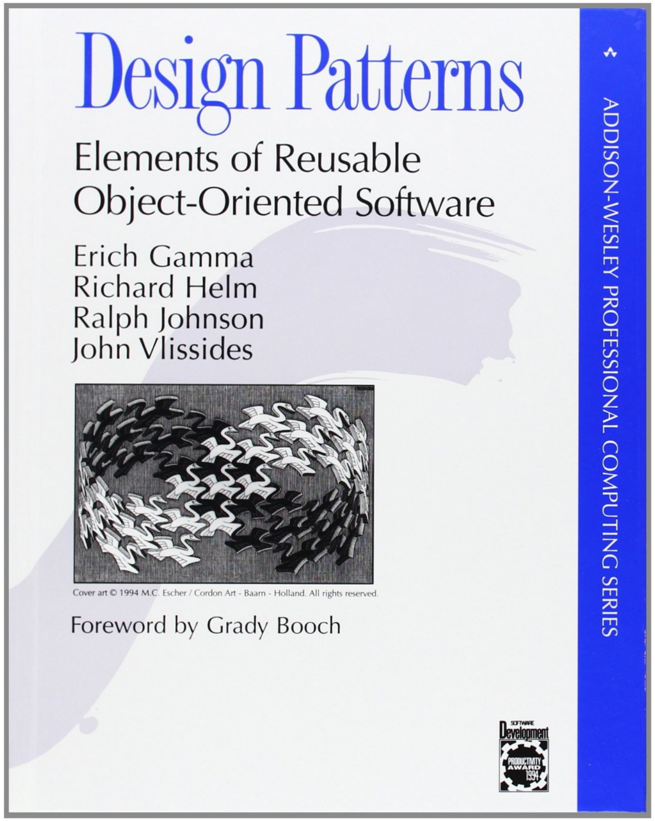
5COSC019W – Object Oriented Programming Week 12

Dr. Barbara Villarini

b.villarini@westminster.ac.uk

Summary

- Introduction: Why a design pattern?
- What is a design pattern
- How we describe a design pattern
- Classification of Design Pattern
- Examples:
  - Composite
  - Singleton
  - Observer
  - Factory


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES
SOFTWARE Development PRODUCTIVITY AWARD 1994

Why a design pattern

- Reusability: one of Wasserman's rules(1996) for an efficient and actual software development
- It helps new designer to have a more flexible and reusable design
- Improves the documentation and maintenance of existing system by furnishing an explicit specification of class end object interactions and their intent

What is Design Pattern

- Christopher Alexander says “ *Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice*”

- A design pattern is a descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

- A pattern is made by four elements:
  - Name
  - Problem
  - Solution
  - Consequences

Design Patter - Name

- Describe a design problems and its solutions in a word or two
- Used to talk about design pattern with our colleagues
- Used in the documentation
- Increase our design vocabulary
- Have to be coherent and evocative

Design Pattern - Problem

- Describes when to apply the patterns

- Explains the problem and its context

- Sometimes include a list of conditions that must be met before it makes sense to apply the pattern

- Have to occurs over and over again in our environment

Design Pattern - Solution

- Describes the elements that make up the design, their relationships, responsibilities and collaborations
- Does not describe a concrete design or implementation
- Has to be well proven in some projects

Design Pattern - Consequences

- Results and trade-offs of applying the pattern
- Helpful for describe design decisions, for evaluating design alternatives
- Benefits of applying a pattern
- Impacts on a system’s flexibility, extensibility or portability

Classification of Design Pattern

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Scope: domain over which a pattern applies
Purpose: reflects what a pattern does

Creational Patterns

- Abstract the instantiation process
- Make a system independent to its realization
- Class Creational use inheritance to vary the instantiated classes
- Object Creational delegate instantiation to an another object

Structural Patterns

- Class Structural patterns concern the aggregation of classes to form largest structures

- Object Structural pattern concern the aggregation of objects to form largest structures

Behavioural Patterns

- Concern with algorithms and assignment of responsibilities between objects
- Describe the patterns of communication between classes or objects
- Behavioral class pattern use inheritance to distribute behavior between classes
- Behavioral object pattern use object composition to distribute behavior between classes

SOME PATTERNS

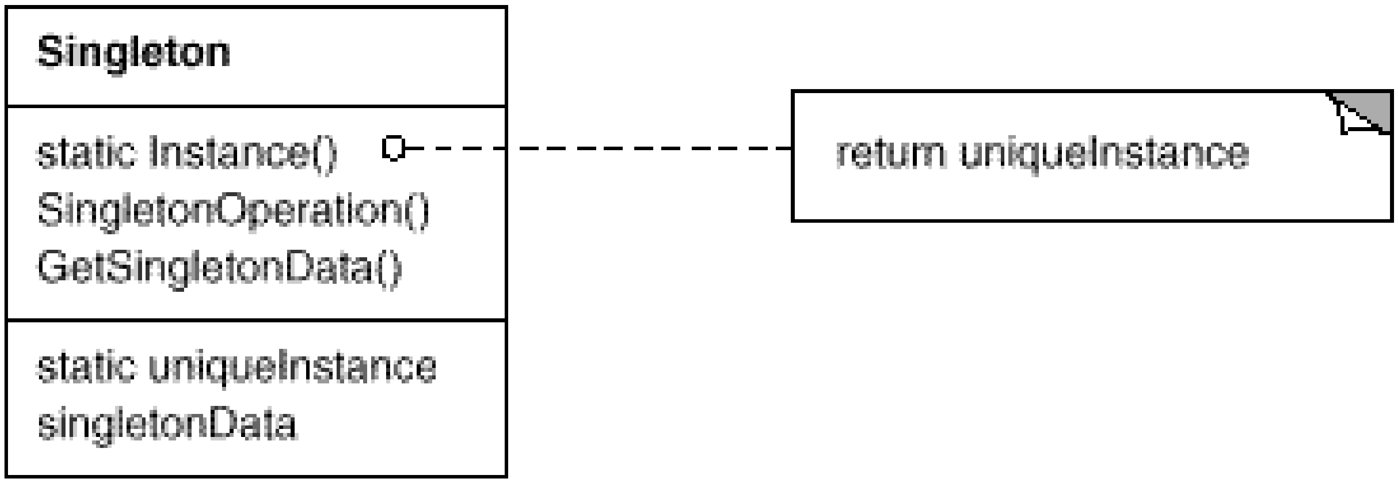
Singleton

- Some classes have conceptually one instance
  - Many printers, but only one print spooler
  - One file system
  - One window manager
- Ensure a class only has one instance
- Provide a global point of access to it

Singleton Motivation

- Class is responsible for tracking its sole instance
  - Make constructor private
  - Provide static method/field to allow access to the only instance of the class
- Benefit:
  - Reuse implies better performance
  - Class encapsulates code to ensure reuse of the object; no need to burden client

Singleton Pattern



Implementing the Singleton method - Java

- In java:

```
public class Singleton {

  private Singleton()  {…}

  final private static Singleton instance = new Singleton();

  public static Singleton getInstance() {
              return instance; }

  protected void demoMethod( ) {
      System.out.println("demoMethod for singleton");
   }
}
```

Constructor

Class Variable

Method to return the instance

Other methods

Singleton Demo

```
public class SingletonDemo {

   public static void main(String[] args) {
      Singleton tmp = Singleton.getInstance( );
      tmp.demoMethod( );
   }
}
```

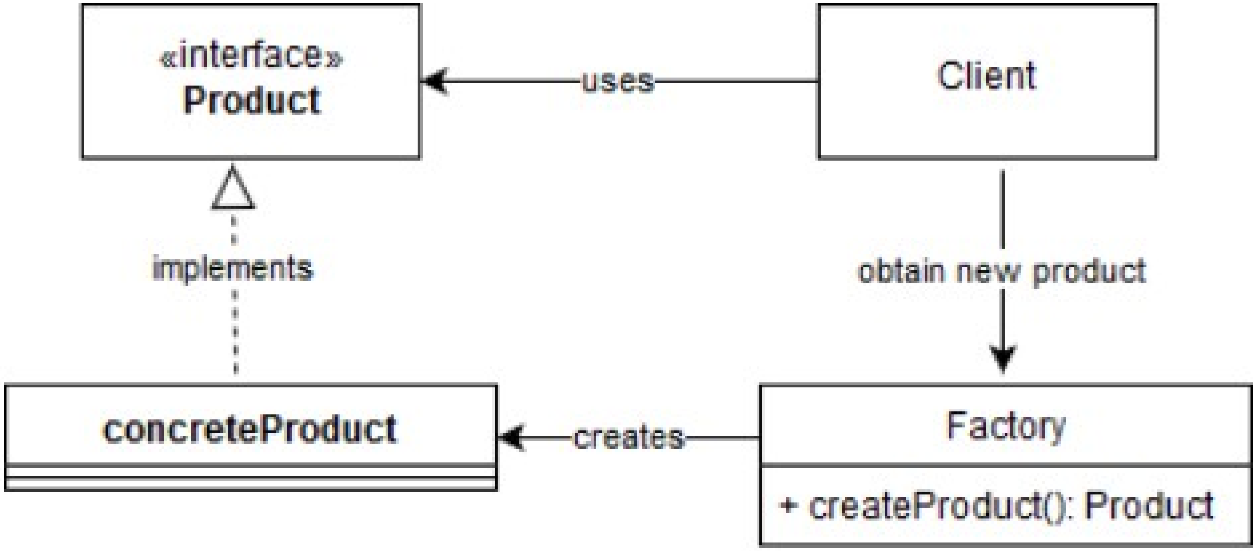
This will produce the following Output:

```
demoMethod for singleton
```

Factory Pattern

- It is a creational pattern that can be used to create objects without specifying the exact classes of the object that will be created
- It is the most used design pattern
- It creates objects by calling a factory method, either in an interface and implemented by child class, or implemented in a base class and optionally overridden by derived classes.
- **Objectives**:
- Create an object in such a manner that subclasses can redefine which class to instantiate.
- Defer instantiation to sub classes.

Factory Pattern

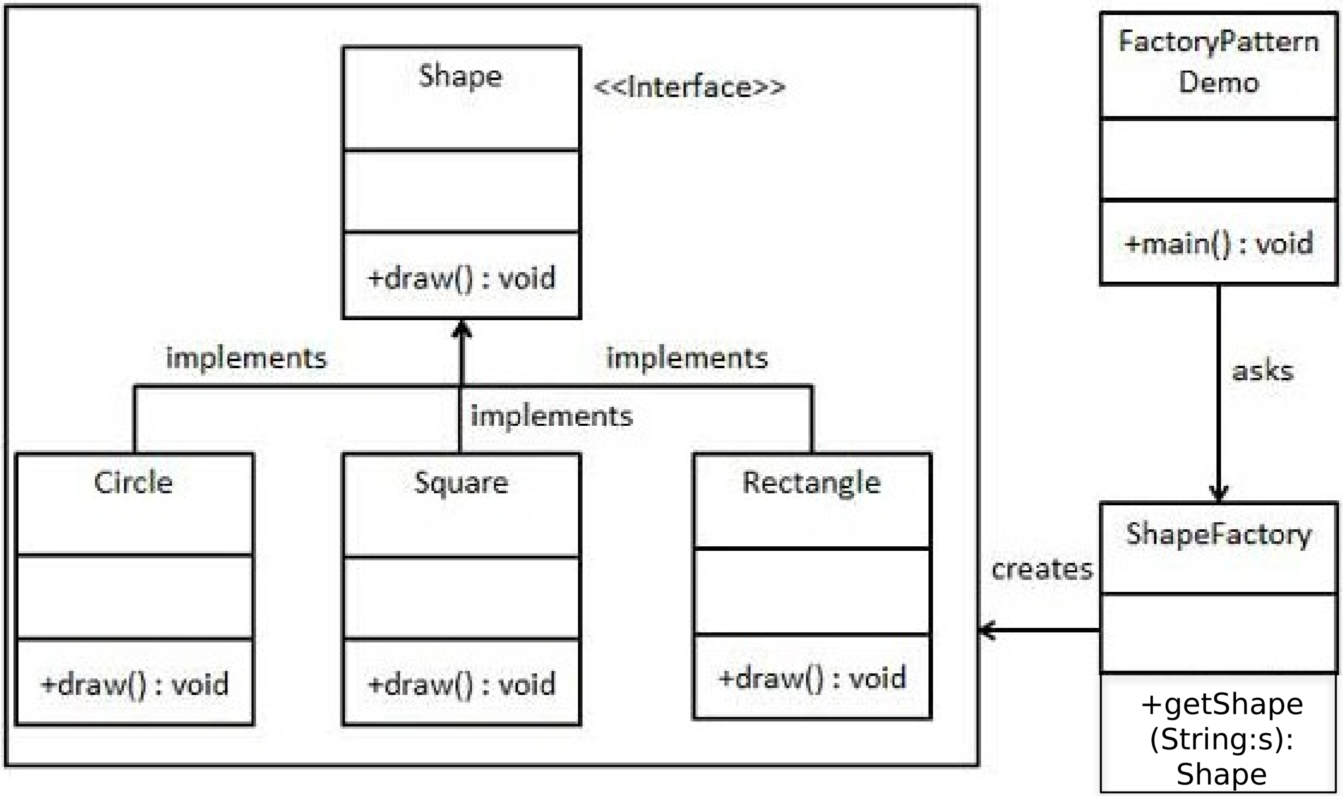


Factory – Implements method to create concrete product objects.
Product – Interface for a product
ConcreteProduct – Implements the Product interface and defines a concrete Product object which is created by the Factory.
Client – Uses Factory to create product which is accessed by the interface.

<u>Benefits</u>:
Isolation of concrete classes
Consistency among products

Example

- We're going to create a Shape interface and concrete classes implementing the Shape interface. A factory class ShapeFactory is defined as a next step.
- FactoryPatternDemo, our demo class will use ShapeFactory to get a Shape object. It will pass information (CIRCLE / RECTANGLE / SQUARE) to ShapeFactory to get the type of object it needs.



Step 1

- Create an interface (Shape.java).

```
public interface Shape {
   void draw();
}
```

Step 2

- Create concrete classes implementing the same interface.

```
public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw()
method.");
    }}
public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }}
public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Inside Circle::draw() method.");
    }}
```

Step 3

- Create a Factory to generate object of concrete class based on given information.

```
public class ShapeFactory {
  //use getShape method to get object of type shape
   public Shape getShape(String shapeType){
      if(shapeType == null){
         return null;
      }
      if(shapeType.equalsIgnoreCase("CIRCLE")){
         return new Circle();

      } else if(shapeType.equalsIgnoreCase("RECTANGLE")){
         return new Rectangle();

      } else if(shapeType.equalsIgnoreCase("SQUARE")){
         return new Square();
      }
      return null;}}
```

Step 4

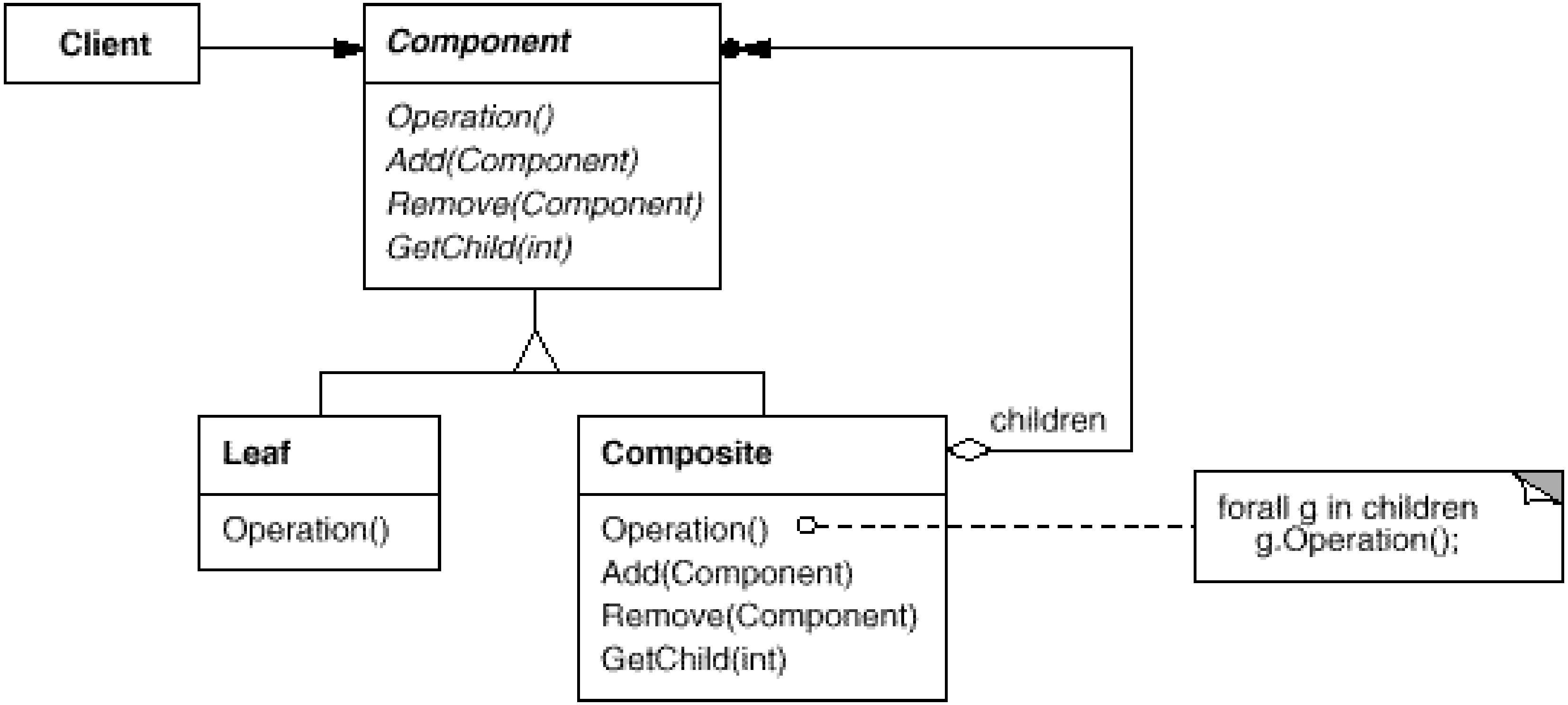
- Use the Factory to get object of concrete class by passing an information such as type.

```
public class FactoryPatternDemo {
      public static void main(String[] args) {
      ShapeFactory shapeFactory = new ShapeFactory();

      //get an object of Circle and call its draw method.
      Shape shape1 = shapeFactory.getShape("CIRCLE");
      //call draw method of Circle
      shape1.draw();

      //get an object of Rectangle and call its draw method.
      Shape shape2 = shapeFactory.getShape("RECTANGLE");
      //call draw method of Rectangle
      shape2.draw();

      //get an object of Square and call its draw method.
      Shape shape3 = shapeFactory.getShape("SQUARE");
      //call draw method of square
      shape3.draw();}}
```

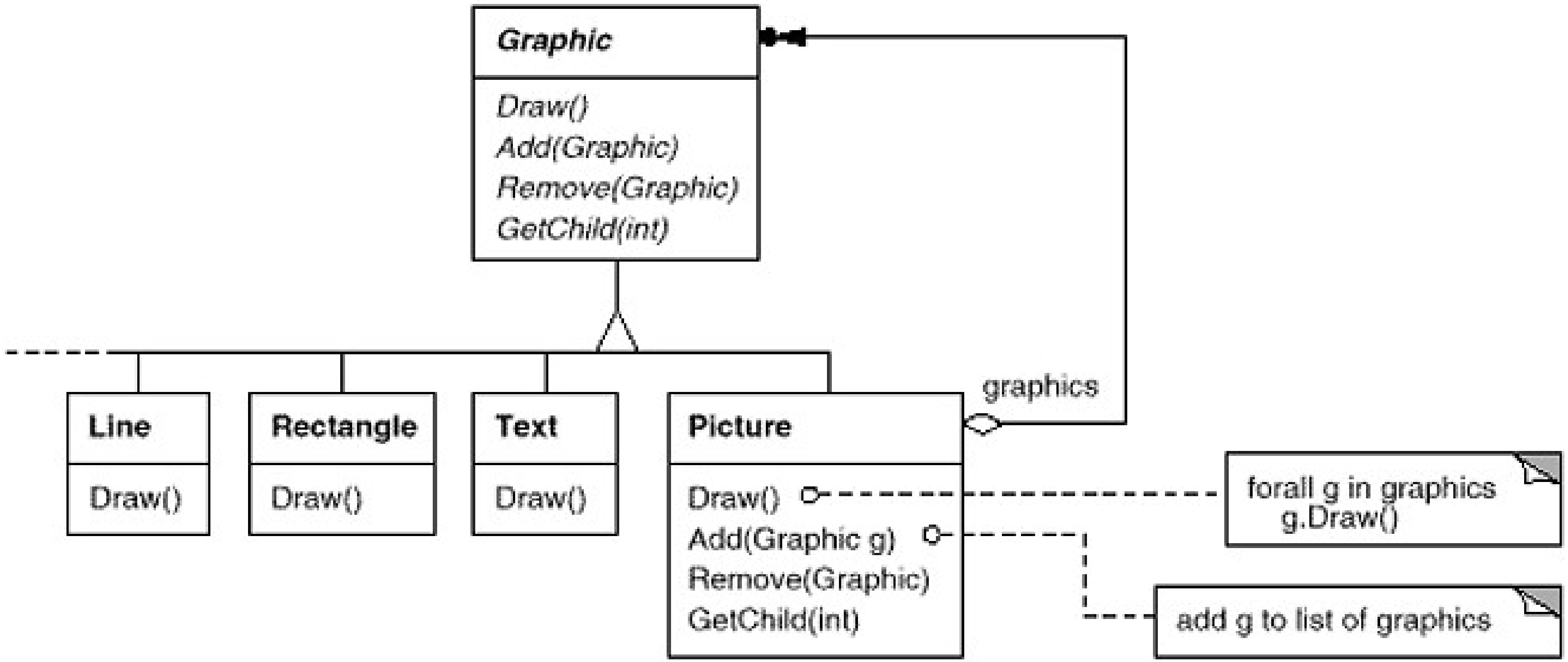
Composite Pattern

- Let’s clients treat individual objects and compositions of objects uniformly
- Compose objects into tree structures to represent part-whole hierarchies
- Motivation:
  - support recursive composition in such a way that a client need to not know the difference between a single and a composite object
- Applicability:
  - when dealing with hierarchically-organized objects (e.g., columns containing rows containing words …)

Example Composite



Composite - example

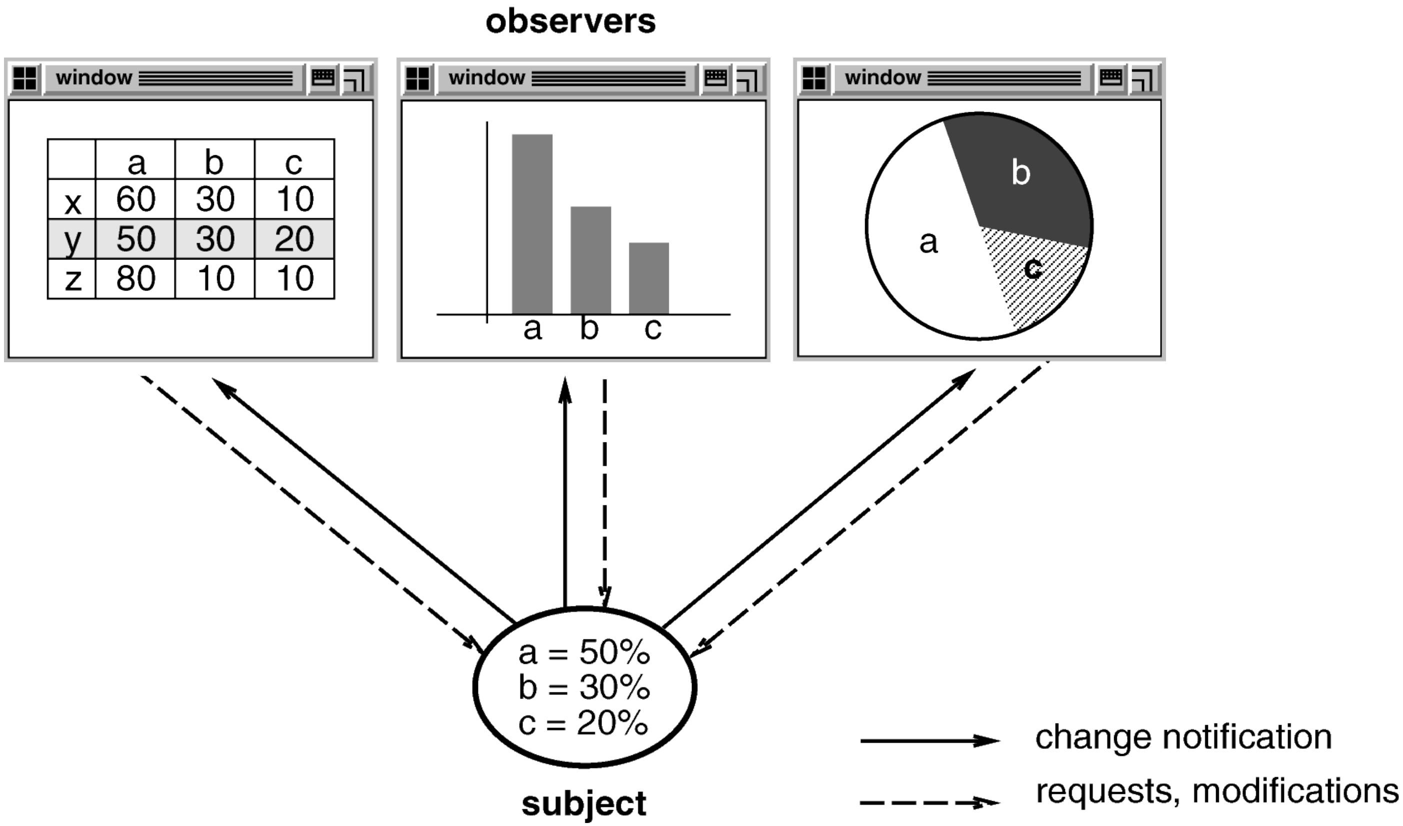


Consequences Composite

- Consequences:
  - class hierarchy has both simple and composite objects
  - simplifies clients
  - aids extensibility
    - clients do not have to be modified
  - too general pattern?
    - difficult to restrict the components of a composite

Observer Pattern

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Motivation:

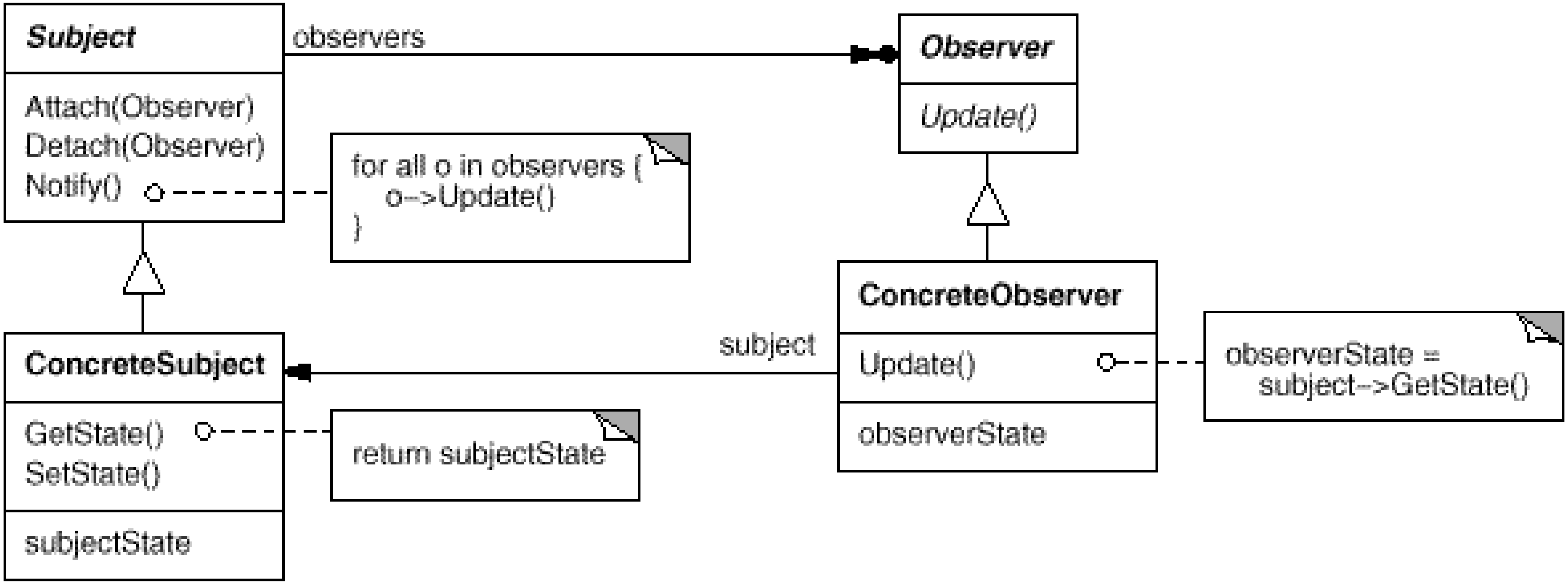


	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

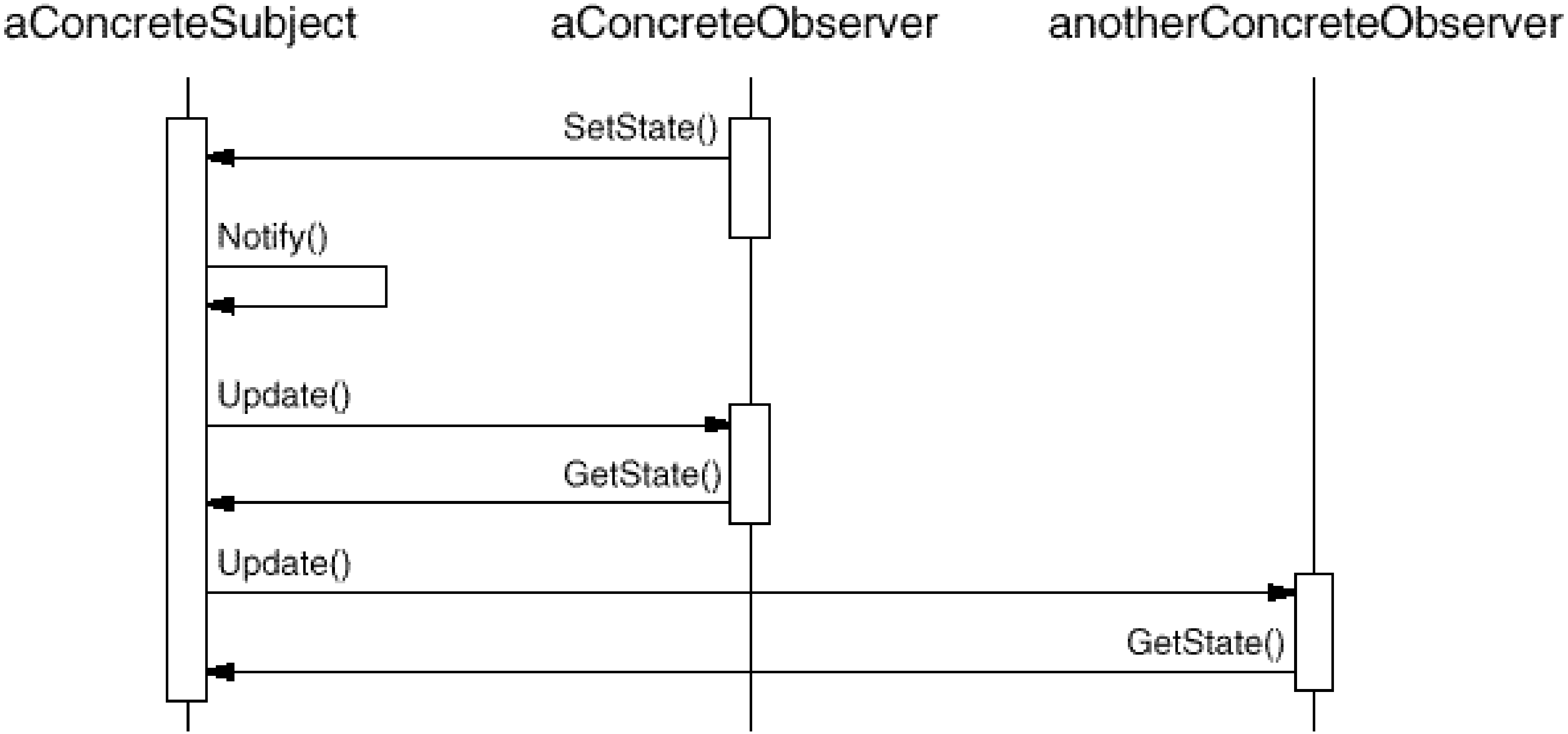
Observer Pattern

- Problem
  - dependent's state must be consistent with master's state
- Solution structure
  - define four kinds of objects:
    - Subject
      - maintain list of dependents; notifies them when master changes
    - Observer
      - define protocol for updating dependents
    - Concrete subject
      - Store state of interest to Concrete Observer object; send notification to its observer when its state change
    - Concrete observers
      - Maintain a reference to Concrete Subject object
      - get new subject state upon receiving update message
      - Implements the Observer updating interface to keep its state consistent with the subject's

Observer structure



Use of Observer



Observer Consequences

- Low coupling between subject and observers
  - subject unaware of dependents
- Support for broadcasting
  - dynamic addition and removal of observers
- Unexpected updates
  - no control by the subject on computations by observers

Example

- The standard Java event model is an example of an observer pattern

SOMETHING MORE…

Prepare to reply to the following questions:

- Why do you want to work for *this company*?
  - Show that you know very well what the company does
  - Show interest about their work
  - Show that you’re passionate about technology
- What do you like about the company work? What would you improve (in terms of their products)?
  - Offering specific recommendations can show your passion for the job.
- How much of your day do you spend coding?
- Can you tell me about a challenging interaction with a teammate?

Your Resume

- Resume screeners look for the same things that interviewers do:
  - Are you smart?
  - Can you code?

- **Relevant Jobs:** Your resume does not - and should not - include a full history of every role you've ever had. Include only the relevant things
- **Writing Strong Bullets**: For each role, try to discuss your accomplishments with the following approach: "Accomplished X by implementing Y which led to Z"
  - Here's an example: "Reduced object rendering time by 75% by applying Floyd's algorithm, leading to a 10% reduction in system boot time "

Projects

- Almost every candidate has some projects, even if they're just academic projects
- List them on your resume! I recommend putting a section called "Projects" on your resume and list your 2 - 4 most significant projects
- State:
  - what the project was
  - which languages or technologies it employed
  - whether it was an individual or a team project

  If your project was not for a course, that's even better! It shows passion, initiative, and work ethic.

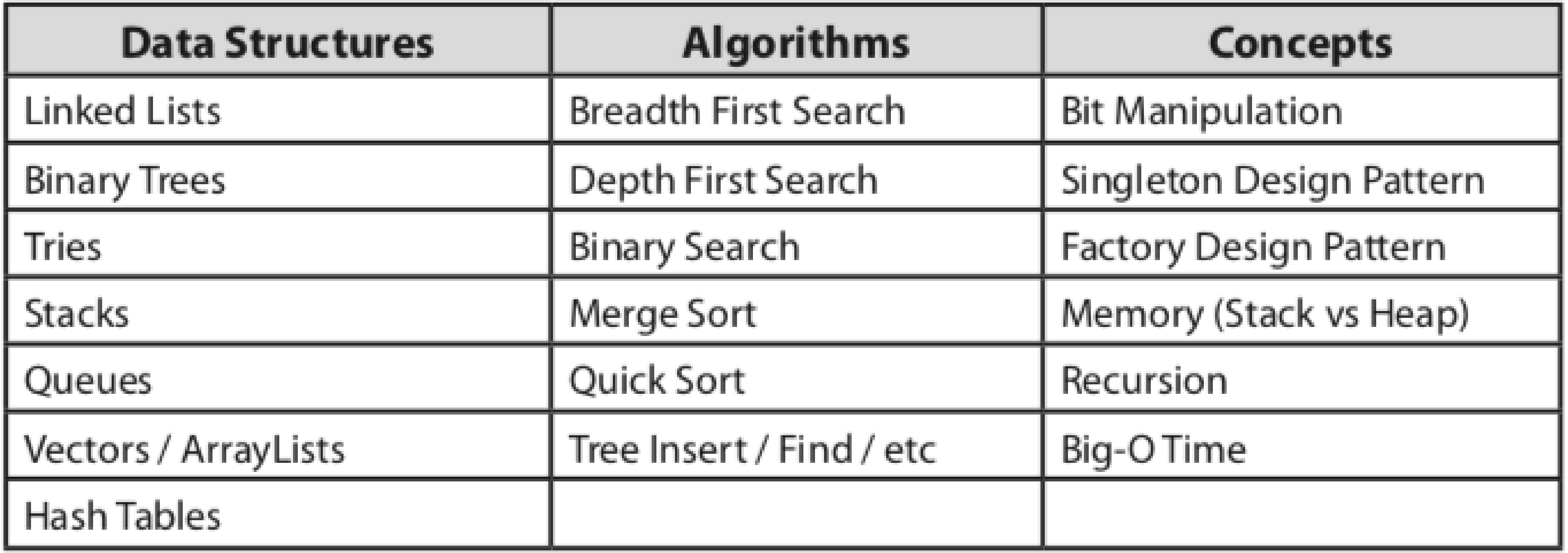
Programming languages and software

- **Software:** It is not good to write that you're familiar with Microsoft Office. Familiarity with developer-specific or highly technical software (e g , Visual Studio, Eclipse, Linux) can be useful.
- **Languages:** Do you list everything you've ever worked with? Or only the ones that you're more comfortable with (even though that might only be one or two languages)? I recommend the following compromise: list most languages you've used, but add your experience level. This approach is shown below:

"Languages: Java (expert), C++ (proficient), JavaScript (prior experience), C (prior experience)"

Technical questions

Get prepared on the following topics:



Data Structures	Algorithms	Concepts
Linked Lists	Breadth First Search	Bit Manipulation
Binary Trees	Depth First Search	Singleton Design Pattern
Tries	Binary Search	Factory Design Pattern
Stacks	Merge Sort	Memory (Stack vs Heap)
Queues	Quick Sort	Recursion
Vectors / ArrayLists	Tree Insert / Find / etc	Big-O Time
Hash Tables		

How you can be prepared for an interview…

Link

- https://www.hackerrank.com
- http://www.indiabix.com/engineering/

Book

- **Cracking the Coding Interview: 150 Programming Questions and Solutions**