

Composite / Flyweight / Object Pool



Advanced Java Programming 2012-13

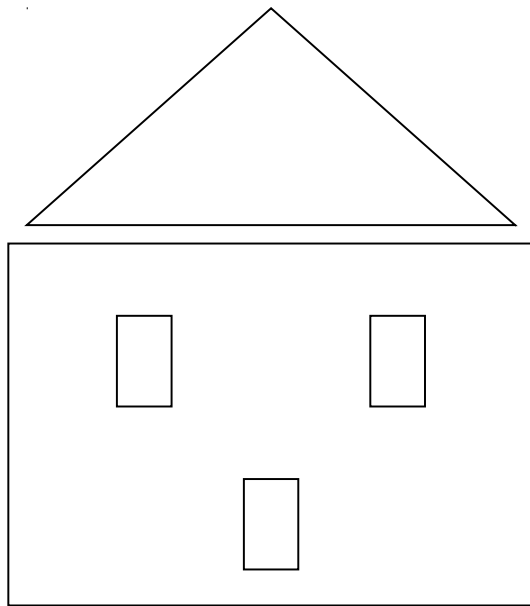
OO Design Patterns and Principles

Lecture Outline

- Composite pattern
- Flyweight pattern
- Object pool pattern
- Independent projects

The Composite pattern

- A composite object is an object made up from other primitive (indivisible) objects
 - For example, a *drawing* is an object that composed of several primitives (*lines*)



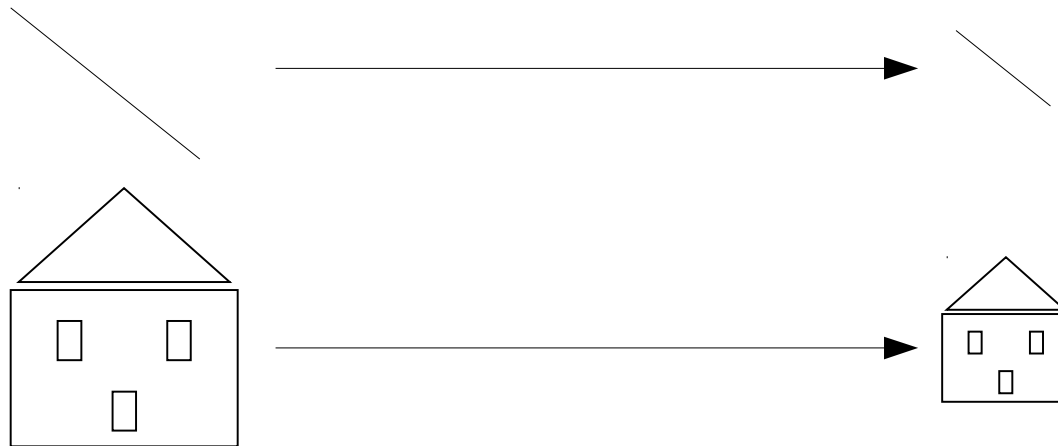
**DRAWING – A
COMPOSITE OBJECT**



**A LINE – A PRIMITIVE
OBJECT**

The Composite pattern

- Sometimes we want to manipulate composites in exactly the same way we manipulate primitive objects
 - For example, in a graphics program lines can be drawn, moved, and resized
 - And we can usually perform exactly the same operations on drawings



RESIZING COMPOSITE AND PRIMITIVE OBJECTS

The Composite pattern

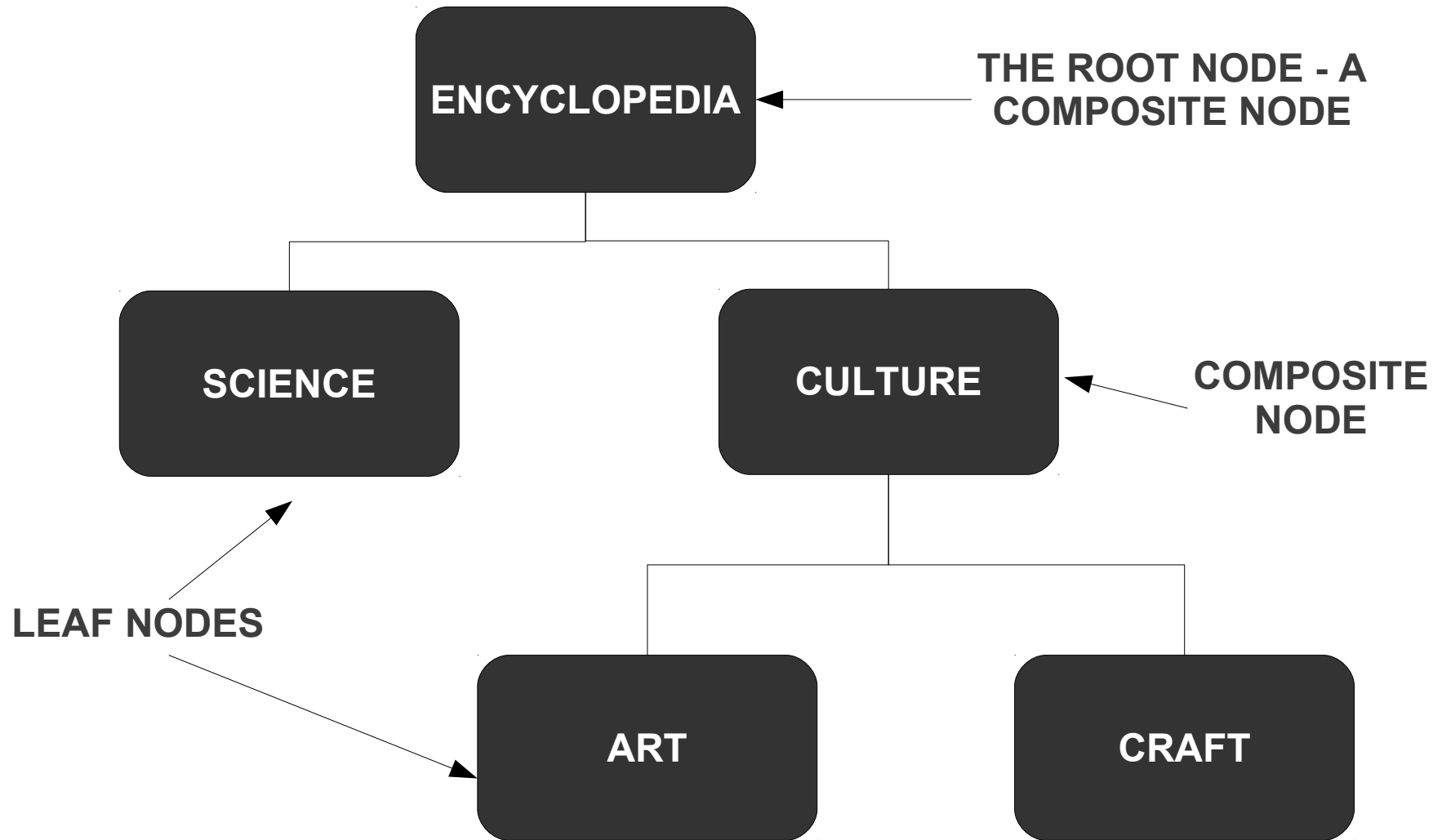
- Ideally, we want to perform operations on primitive objects and composites *using the same code*
- If we have to distinguish between primitive objects and composites, our code would become more complex and difficult to maintain
- This brings us to the *composite* pattern
- Whenever you are dealing with a problem that involves identical operations on *things* and objects made of those *things*, think composite pattern

The Composite pattern

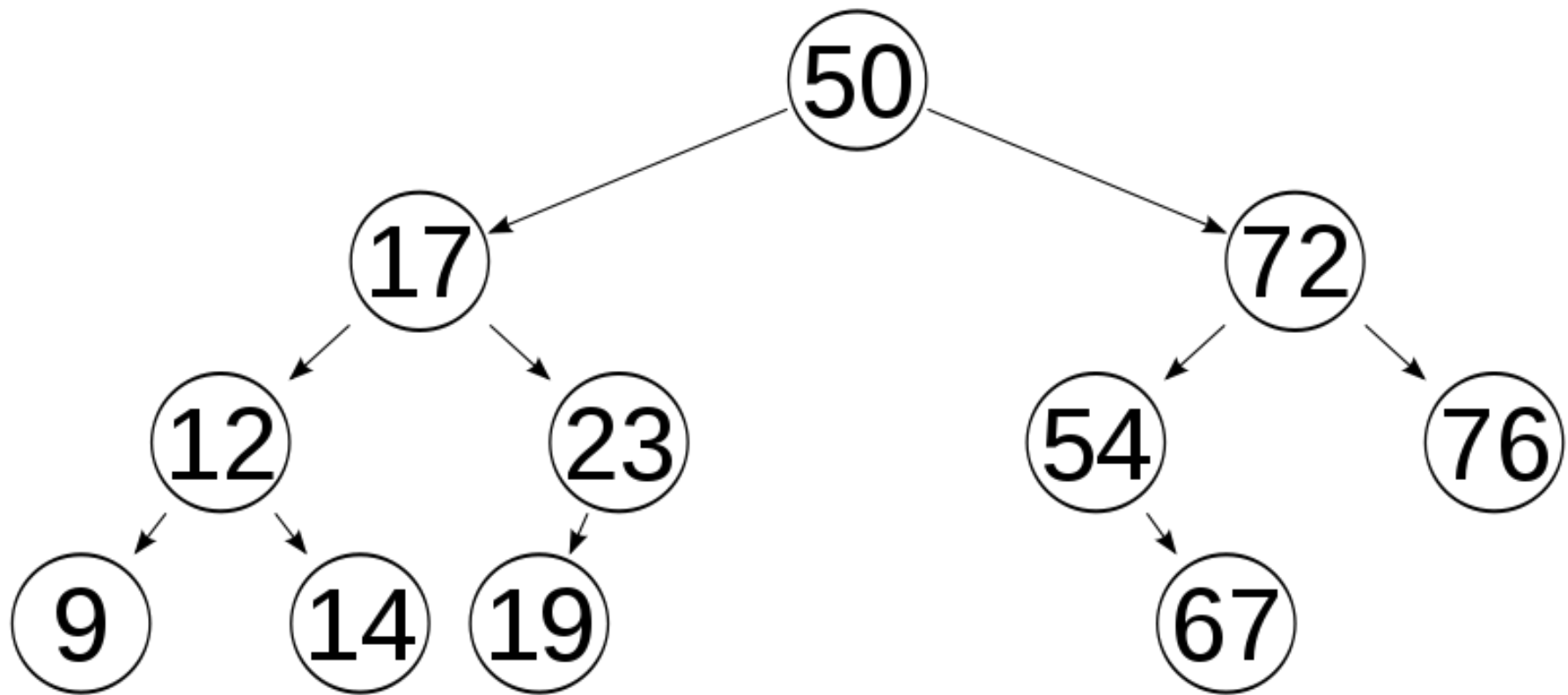
- Before we meet the composite pattern, we need to discuss *tree structures*
 - A tree structure is a way of representing a hierarchical structure in a graphical form
 - It is named a 'tree structure' because the classic representation resembles a tree
 - Actually, the chart is generally upside down compared to an actual tree, having the 'root' at the top and the 'leaves' at the bottom



The Composite pattern

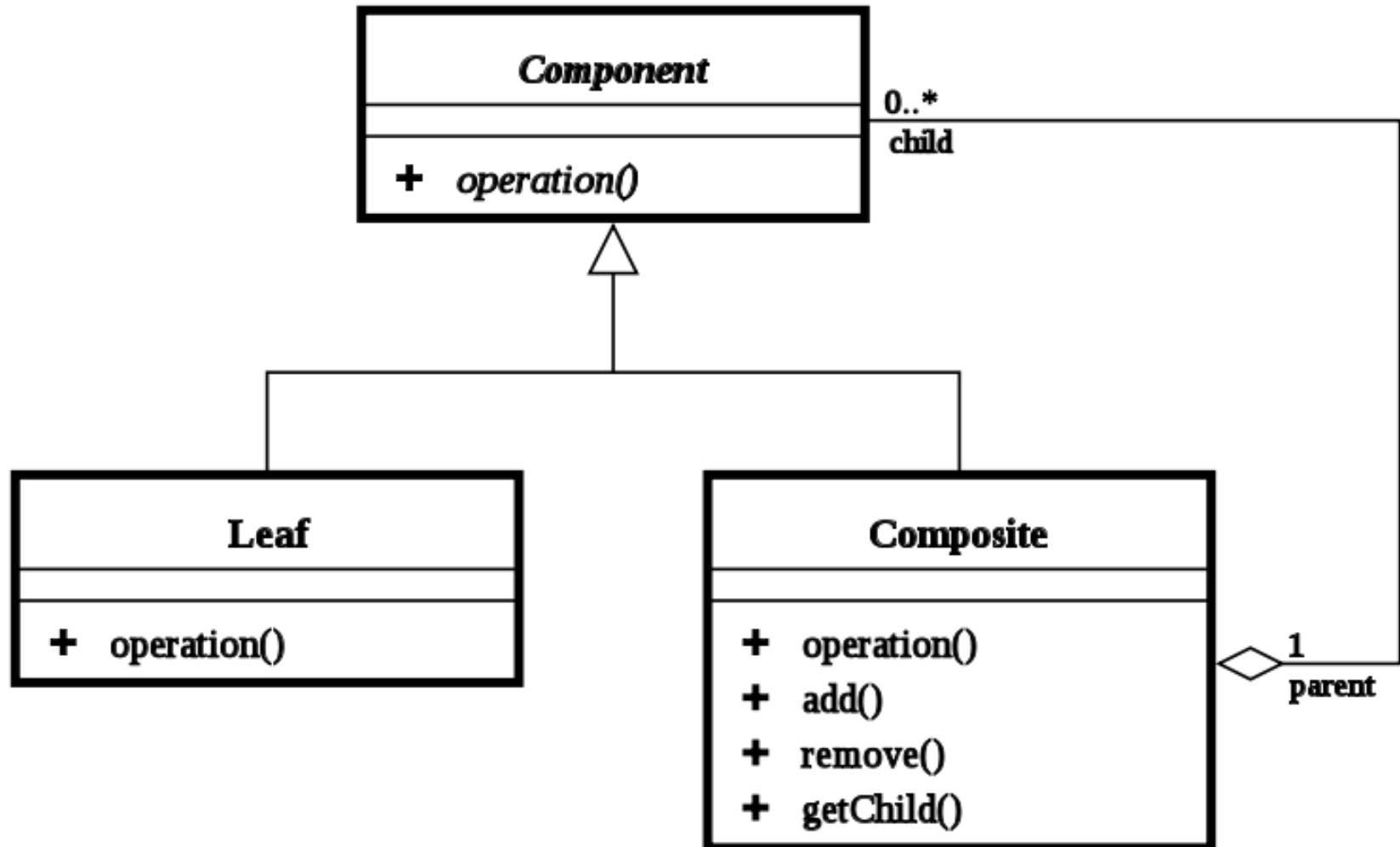


The Composite pattern



Nodes 9, 14, 19 and 67 are leaf nodes. All other nodes are internal composite nodes. Note how 17 is a composition of 12 and 23, which are themselves compositions of {9,14} and {19} respectively.

The Composite pattern



UML diagram of the *Composite* pattern

The Composite pattern

- **The *Leaf* class**

- Represents a primitive object
- In our drawing application example, a *Leaf* object would be a single line

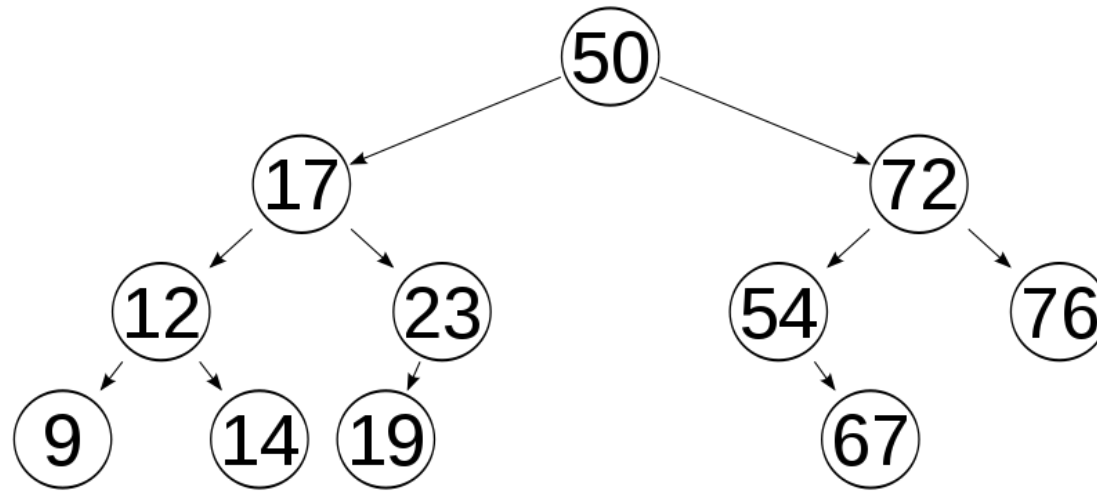
- **The *Composite* class**

- Represents a composite object
- In our drawing application example, a *Composite* object is a drawing made up from primitive objects

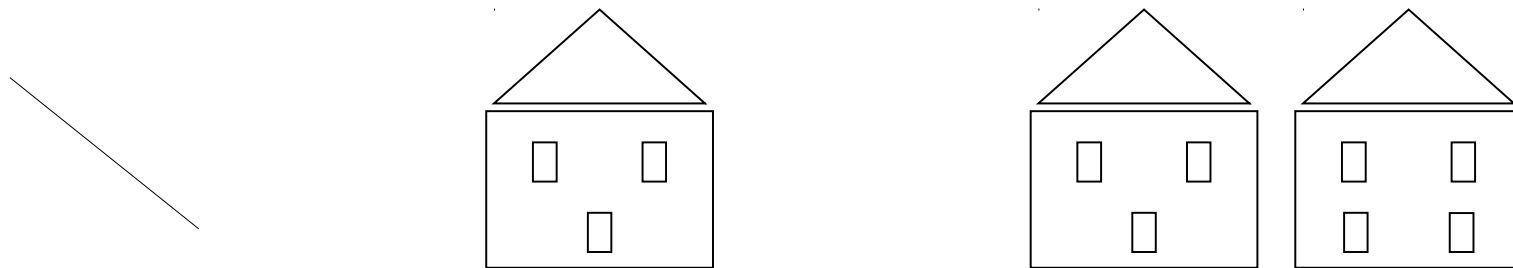
- ***Component* interface**

- Defines method(s) you wish to perform on both *Leaf* objects and *Composite* objects e.g. `move()`, `scale()`

The Composite pattern



Important - Note that the *Composite* class can represent a composition of leaf objects or a composite of other composites



LEAF
OBJECT

A COMPOSITE OF
LEAF OBJECTS

A COMPOSITE OF
COMPOSITES!

The Composite pattern

● How do we implement this pattern?

- Assume that we are making an application that allows a user to move a single line using the mouse
- We also want the user to be able to move a composite drawing of 2+ lines using the mouse
- We start by defining an interface that is implemented by both the leaf class (*Line*) and the composite class (*Drawing*)
- We will call this interface *Graphic*, and the method `move()`

```
public interface Graphic
{
    public void move();
}
```

The Composite pattern

```
public class Drawing implements Graphic
{
    private ArrayList<Graphic> graphics = new ArrayList<Graphic>();
    public void move()
    {
        for (Graphic g : graphics)
        { g.move(); }
    }
    public void add(Graphic g)
    {
        graphics.add(g);
    }
    public void remove(Graphic g)
    {
        graphics.remove(g);
    }
}
```

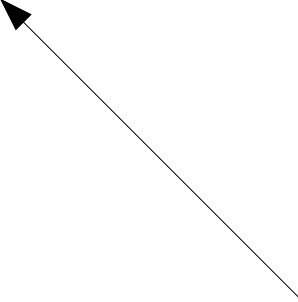
A *Drawing* class holds a collection of *Graphic* objects.

When the *move()* method is called, we iterate through the collection of *Graphic* objects, calling *move()* on each one. This is delegation

Note methods for adding and removing *Graphic* objects from the list. Similar to *Observer*

The Composite pattern

```
public class Line implements Graphic
{
    public void move()
    {
        System.out.println("Moving a single line");
    }
}
```



The *Leaf* implementation. When the *move()* is called, we print out a line of text. This simple bit of code represents the more sophisticated code need to delete and re-draw lines on a Java 2D canvas.

The Composite pattern

```
//Initialize leaf node objects
```

```
Line l1 = new Line();
```

```
Line l2 = new Line();
```

```
//Initialize composite Drawing object
```

```
Drawing d1 = new Drawing();
```

```
//Compose the composite Drawing object
```

```
d1.add(l1);
```

```
d1.add(l2);
```

```
// Exercise the move() of the composite Drawing object
```

```
d1.move();
```

OUTPUT:

Moving a single line

Moving a single line

The Composite pattern

```
//Initialize leaf node objects
```

```
Line l1 = new Line();
```

```
Line l2 = new Line();
```

```
//Initialize three Drawing objects
```

```
Drawing d1 = new Drawing();
```

```
Drawing d2 = new Drawing();
```

```
Drawing d3 = new Drawing();
```

```
//Compose the composite objects
```

```
d1.add(l1);
```

```
d2.add(l2);
```

```
d3.add(d1); d3.add(d2);
```

A composition of compositions



```
// Exercise the move() of the composite Drawing object
```

```
d3.move();
```




The Flyweight Pattern

Flyweight pattern

- Flyweight is a structural design pattern
 - It 'facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient' (GOF definition)
- This pattern is useful in scenarios where there are lot of objects which share some common intrinsic information
 - Instead of storing the common intrinsic information n times for n objects, it is stored only once in one object and is referenced by all the objects which want to use it
 - The object which contains all the common intrinsic information is called the *flyweight* object

Flyweight pattern

- Consider, for example, a WW2 game which features a large number of *Soldier* objects on a virtual battlefield
 - Each *Soldier* object has a *draw()* method which draws a soldier graphic at a certain point on the battlefield
 - The actual location of the *Soldier* depends on the value of X and Y coordinates (instance variables)
- The battlefield is created by creating a large number of *Soldier* objects, then drawing them on the battlefield
 - This seems unavoidable - battles involve lots of soldiers
- However, the system is being swamped by 1000's of objects
- The answer to this problem is the *Flyweight pattern*

Flyweight pattern

```
class Soldier
{
    private int x;
    private int y;

    // getters and setters ...

    public void draw()
    {
        // draws a soldier at (x,y)
    }
}
```

With 1000s of *Soldier* objects on the map, performance of the application is very sluggish. Can the flyweight pattern help us? Yes. Obviously.

Flyweight pattern

- What if, instead of having thousands of *Soldier* objects, you could redesign your application so that you only have one!
- Although the graphical representation of a *Soldier* is the same for all instances, their location can vary greatly
- So we have some common intrinsic information i.e. `draw()` method
 - This has to stay in the *Soldier* class
- And we have some 'stateful' information (i.e. the X,Y coordinates of the *Soldier*) which is different for each instance
 - We move this out of the class to create our *Flyweight*

Flyweight pattern

```
class Soldier
{
    public void draw(int x, int y)
    {
        // draws a soldier at (x,y)
    }
}
```

Our *Flyweight* class. The common information (how to draw a *Soldier*) has been retained and the 'stateful' information (X,Y coords) has been removed to another class

Flyweight pattern

The position of all the (virtual) soldier objects is stored in an array list of *Point* type.

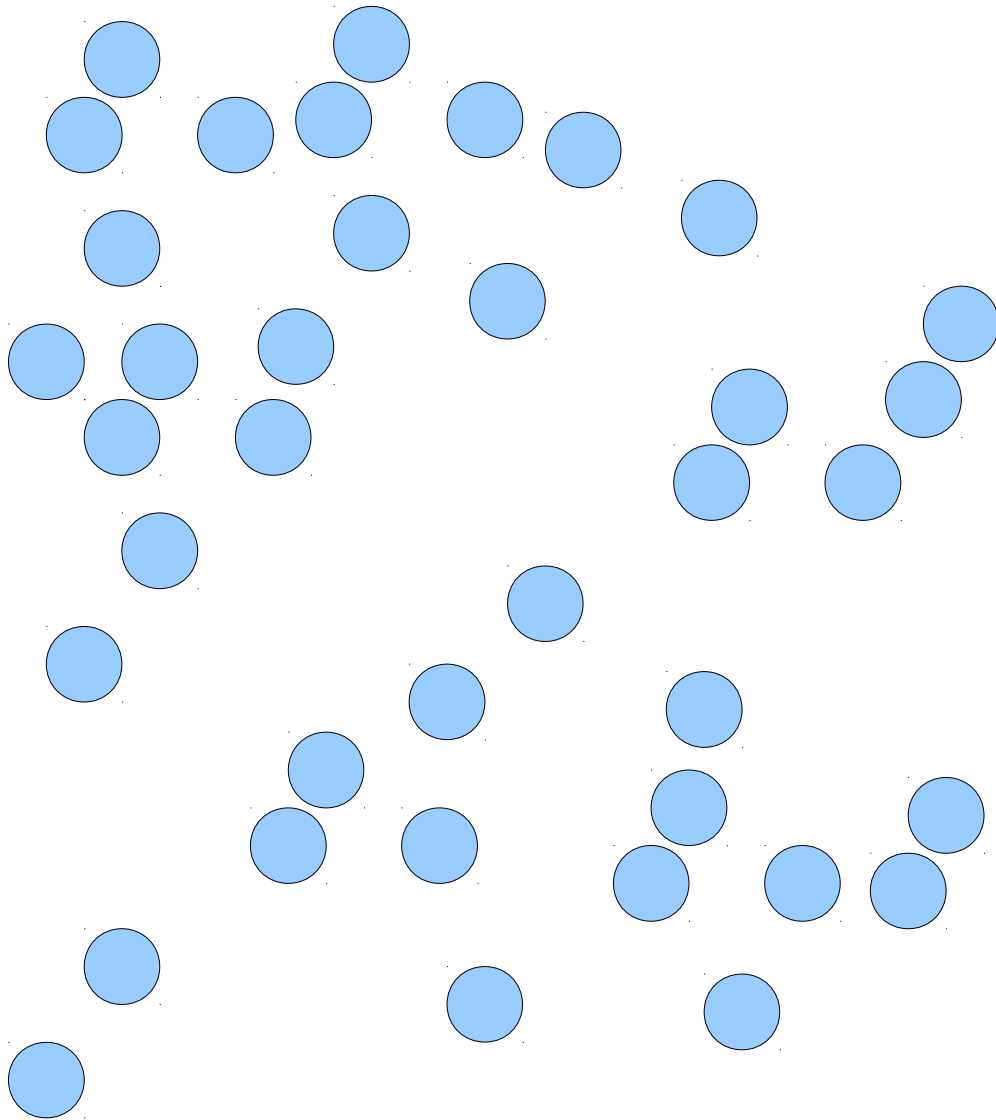
```
class SoldierManager
{
    private ArrayList<Point> positions;
    private Soldier s = new Soldier();

    public void displaySoldiers()
    {
        for(Point p : positions)
        {
            s.draw(p.getX(), p.getY());
        }
    }
}
```

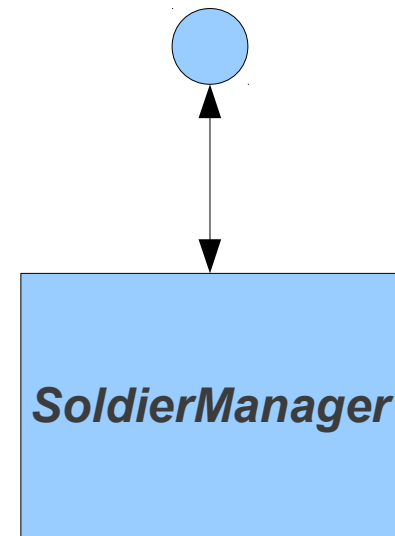
We only need one Soldier object. All the rest are virtual

When the display Soldiers method is called, we iterate through the ArrayList and call the *draw()* method of the *Soldier* object, passing coordinates to the draw() method. We have 'centralised' the state of many virtual *Soldier* objects into a single location.

Flyweight pattern



BEFORE – 1000s of *Soldier* objects



AFTER – 1 *Soldier* flyweight and a class to encapsulate state



Object Pool Pattern

Object pool pattern

- Object pool is a *creational* design pattern
 - An object pool is a set of initialised objects that are kept ready to use, rather than allocated and destroyed on demand
- A client of the pool will *request* an object from the pool and perform operations on the returned object
- When the client has finished with an object, it *returns* it to the pool, rather than destroying it
- Object pooling can offer a significant performance boost in situations where the cost of initializing a class instance is high
 - Classic example – database connections

Object pool pattern

- Generally the pool will be a growing pool, i.e. the pool itself will create new objects if the pool is empty
 - We can restrict the number of objects created by setting a marker known as a *high water mark*
- It is desirable to keep all reusable objects in the same object pool so that they can be managed by one coherent policy
 - To achieve this, object pools are often singletons
- When writing an object pool, the programmer has to be careful to make sure the state of the objects returned to the pool is *reset*
 - Object pools full of objects with dangerously stale state are sometimes called object *cesspools* and regarded as an anti-pattern

Object pool pattern

- Example of pattern in real life
 - When you go bowling, you change your street shoes for a special pair of bowling shoes
 - The shoe shelf at the bowling alley is the object pool
 - The shoes are the (finite) objects in the pool
 - When you want to play, you check out (acquire) a pair of shoes
 - After the game, you return (release) your shoes
 - When the staff spray lysol into the shoes, they are resetting their state

1. `myShoes = shelf.acquireShoes();`

2. `client.wear(myShoes);`



SHELF (OBJECT POOL)




4. `shelf.releaseShoes(myShoes);`

3. `client.play();`

Object pool pattern

- When creating object pools
 - Create an *ObjectPool* class with private array of *Objects* inside
 - Create *acquire* and *release* methods in the *ObjectPool* class
 - Make sure that your *ObjectPool* is Singleton
 - Devise a mean to reset object state
 - Inadequate resetting of objects may cause an information leak
 - If an object contains confidential data (e.g. a user's credit card numbers) that isn't cleared before the object is passed to a new client, a malicious or buggy client may disclose the data to an unauthorized party



Independent projects

Independent projects

- Many of you are already thinking about jobs / placements
- Things that will help
 - Solid academic performance (2.1 or above to avoid being shuffled into the bin)
 - Excellent references (try to excel in a module, get to know the lecturer, use him/her as reference)
 - Excel means a 1st class mark
- Things I would look for in an interviewee for an SE position
 - Technical skill, in one or more languages
 - Familiarity with SE tools (Subversion, Maven, GitHub, IDEs etc.)
 - Interpersonal skills (enthusiastic, friendly, team player)
 - **Evidence of independent, collaborative work ← what have you made? can I download it? does it work?**

The last point is very important. It is time to start working on an independent project NOW, so you can have a viable project in play by the time you start applying for jobs.

Independent projects

- If you are interested in starting an independent project
 - Find 1-3 like minded students (not just friends)
 - Appoint a project lead (you can take turns)
 - Agree on the commitment level (e.g. 3 hours per week)
 - Decide the type of project
 - There is a good list [here](#) if you are stuck for ideas
 - Write and agree a formal specification describing the proposed application listing all functional and non-functional requirements
 - Write and agree and agree a timetable including milestones that describes when you expect to finish the application
 - Short haul projects are a good way to get some momentum
 - Be realistic and set achievable targets

Independent projects

- Decide the technology (Java, Android, .Net, Python)
 - You can use the project as an excuse to learn a new language or develop your skills in a language you already know
- Agree on a style and documentation quality level
- Decide a software methodology (TDD, Agile, Spiral etc.)
- Agree on a software license
- Agree on a CVS system (Mercurial, Subversion, GitHub)
 - Google Code gives you Subversion + web site + team tools for free
- Get used to reviewing each others code
- Get used to criticism
- Give talks to each other that force you to develop
 - For example, research profiling and brief your team mate

Summary

- **The Composite pattern**

- The composite pattern describes that a group of objects are to be treated in the same way as a single instance of an object
- Implementing the composite pattern lets clients treat individual objects and compositions uniformly

- **The Flyweight pattern**

- Flyweight is a structural design pattern that makes that minimizes memory use by sharing as much data as possible with other similar objects

- **Object Pool**

- An object pool is a set of initialised objects that are kept ready to use, rather than allocated and destroyed on demand

- **Pre-reading**

- SOLID - read relevant principles from web article [here](#)