Design Patterns

MSc in Computer Science

Produced by

Eamonn de Leastar (edeleastar@wit.ie)

Department of Computing, Maths & Physics
Waterford Institute of Technology

http://www.wit.ie

http://elearning.wit.ie

Waterford Institute *of* Technology
INSTITIÚID TEICNEOLAÍOCHTA PHORT LÁIRGE

eLearning support unit

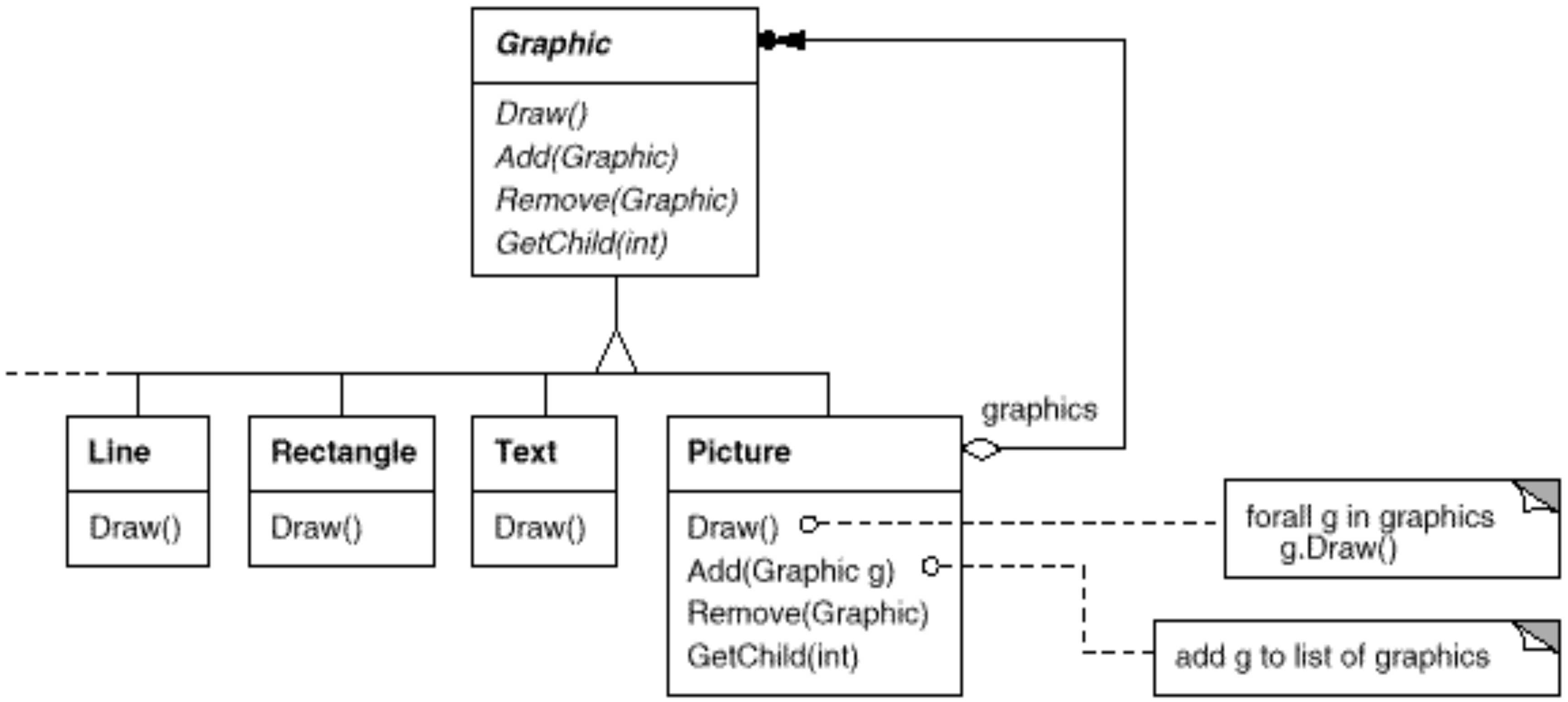
Intent

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.

Motivation (1)

- Graphics drawing editors let users build complex diagrams out of simple components.
    - Components can be grouped to form larger components, which can in turn be further grouped.
- A simple implementation could define classes for graphical primitives such as Text and Lines plus other classes that act as containers for these primitives.
    - Code that uses these classes must treat primitive and container objects differently, even if most of the time the user treats them identically.
- Having to distinguish these objects makes the application more complex.
- The Composite pattern describes how to use recursive composition so that clients don't have to make this distinction.

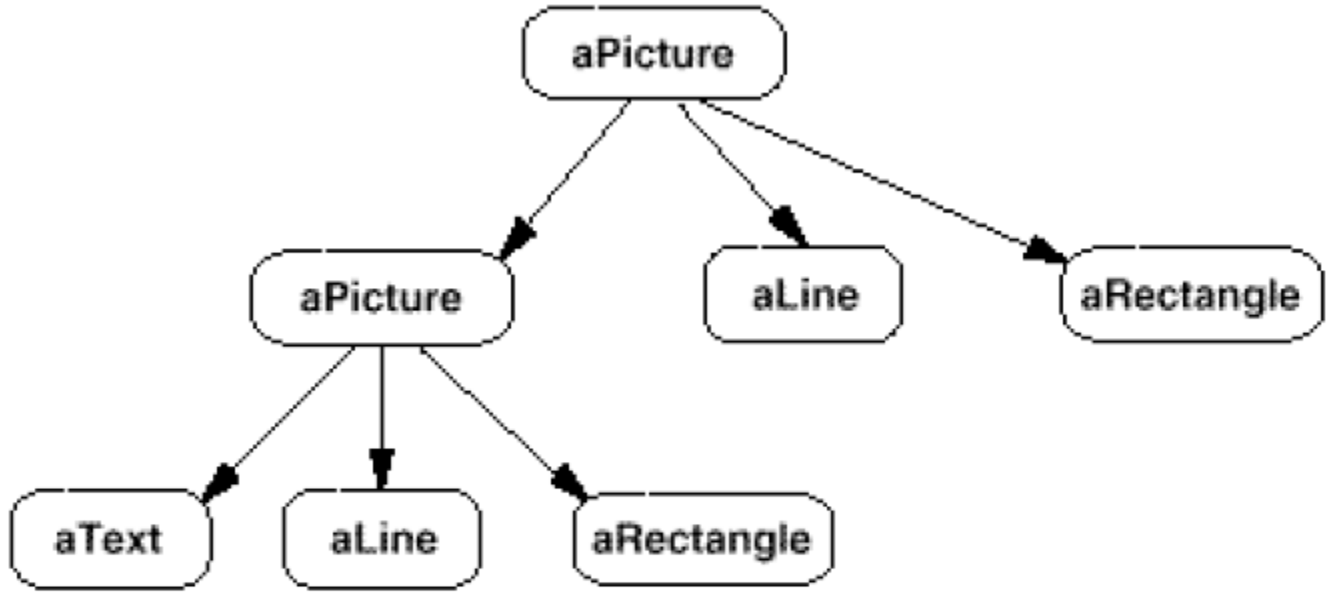
Motivation (2)



Motivation (2)

- The key to the Composite pattern is an abstract class that represents both primitives and their containers.
    - For the graphics system, this class is Graphic. Graphic declares operations like draw() that are specific to graphical objects. It may declare operations that all composite objects share (perhaps for accessing and managing its children).
- The subclasses Line, Rectangle, and Text define primitive graphical objects.
    - These classes implement Draw to draw lines, rectangles, and text, respectively. Since primitive graphics have no child graphics, none of these subclasses implements child-related operations.
- The Picture class defines an aggregate of Graphic objects.
    - Picture implements Draw to call Draw on its children, and it implements child-related operations accordingly.
    - Because the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively

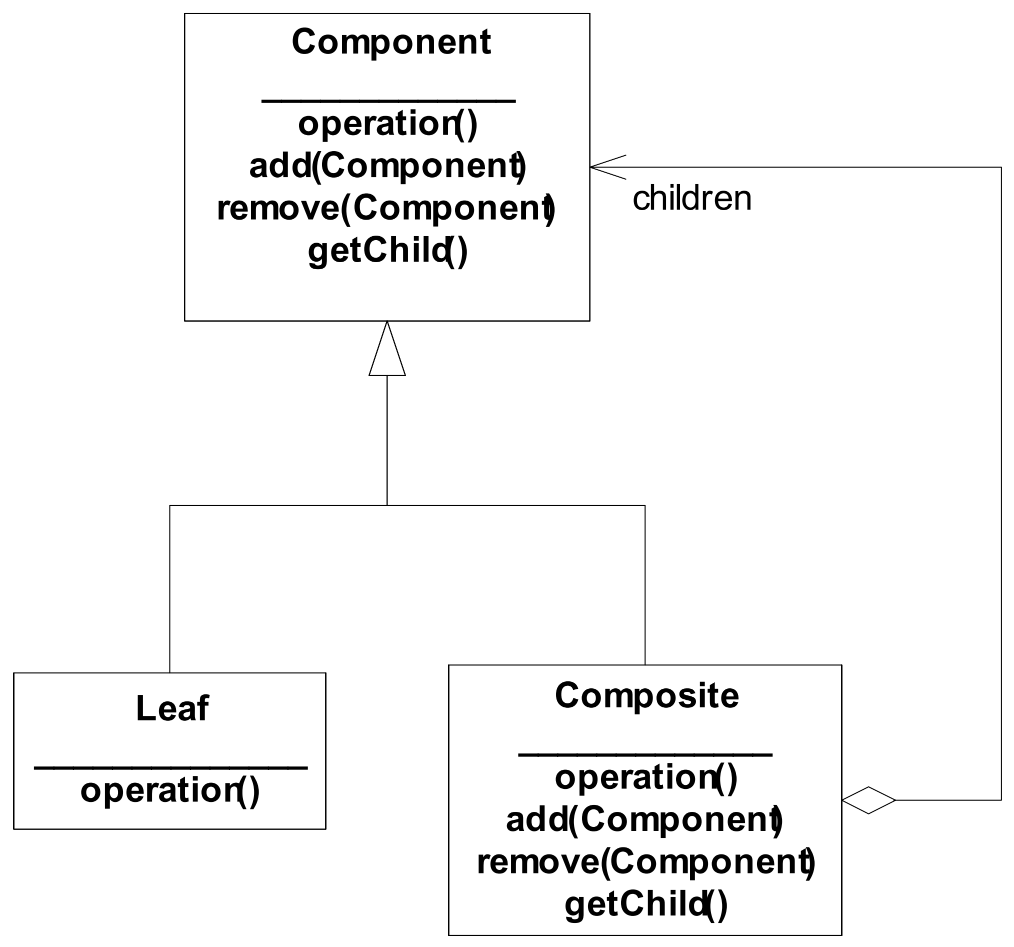
Motivation (3)



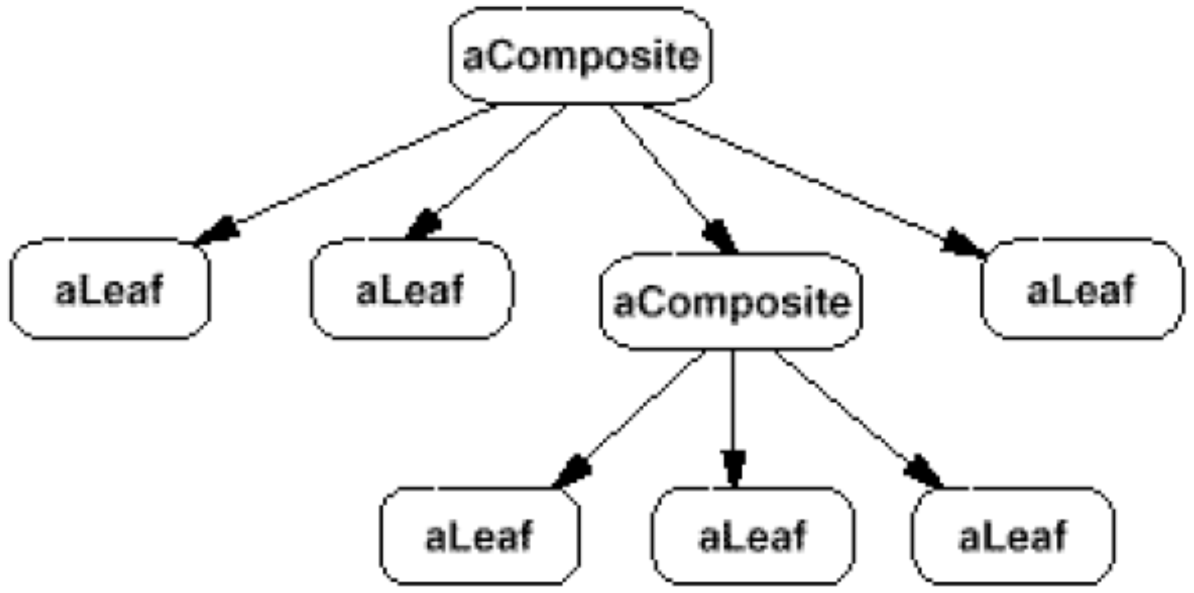
Applicability

- To represent part-whole hierarchies of objects.
- To enable clients to ignore the difference between compositions of objects and individual objects.
- Clients will treat all objects in the composite structure uniformly.

Structure (1)


Component
operation()
add(Component)
remove(Component)
getChild()
children
Leaf
operation()
Composite
operation()
add(Component)
remove(Component)
getChild()

Structure (2)


aComposite
aLeaf
aLeaf
aComposite
aLeaf
aLeaf
aLeaf
aLeaf

Participants

- Component (Graphic)
  - declares the interface for objects in the composition.
  - implements default behaviour for the interface common to all classes, as appropriate.
  - (optional) declares an interface for accessing and managing its child components.
  - (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate.
- Leaf (Rectangle, Line, Text, etc.)
  - represents leaf objects in the composition. A leaf has no children.
  - defines behaviour for primitive objects in the composition.
- Composite (Picture)
  - defines behaviour for components having children.
  - stores child components.
  - implements child-related operations in the Component interface.
- Client
  - manipulates objects in the composition through the Component interface.

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding

Consequences (1)

- Defines class hierarchies consisting of primitive objects and composite objects.
  - Primitive objects can be composed into more complex objects, which in turn can be composed, and so on recursively.
  - Wherever client code expects a primitive object, it can also take a composite object.
  - Makes the client simple.
- Clients can treat composite structures and individual objects uniformly.
  - Clients normally don't know (and shouldn't care) whether they're dealing with a leaf or a composite component.
  - This simplifies client code, because it avoids having to write tag-and-case-statement-style functions over the classes that define the composition.

Consequences (2)

- Makes it easier to add new kinds of components.
  - Newly defined Composite or Leaf subclasses work automatically with existing structures and client code.
  - Clients don't have to be changed for new Component classes.
- Can make your design overly general.
  - The disadvantage of making it easy to add new components is that it makes it harder to restrict the components of a composite.
  - Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. You'll have to use run-time checks instead.

Graphic

```
public class Graphic
{
  protected int x;
  protected int y;

  Graphic(int x, int y)
  {
    this.x = x;
    this.y = y;
  }

  public void draw()
  {
    System.out.print("Origin: X = " + x + ", Y = " + y + ": ");
  }

  public void move(int newX, int newY)
  {
    x = newX;
    y = newY;
  }
}
```

Line

```
public class Line extends Graphic
{
  private int    endX;
  private int    endY;

  public Line(int startX, int startY, int endX, int endY)
  {
    super(startX, startY);
    this.endX = endX;
    this.endY = endY;
  }

  public void draw()
  {
    super.draw();
    System.out.println("line to X = " + endX + ", Y = " + endY);
  }
}
```

Rectangle

```
public class Rectangle extends Graphic
{
  private int width;
  private int height;

  public Rectangle(int X, int Y, int width, int height)
  {
    super(X, Y);
    this.width = width;
    this.height = height;
  }

  public void draw()
  {
    super.draw();
    System.out.println("Recangle: width = " + width + ", height = " + height);
  }
}
```

Picture

```
public class Picture extends Graphic
{
  private Collection<Graphic> graphics;

  public Picture(int x, int y)
  {
    super(x, y);
    graphics = new ArrayList<Graphic>();
  }

  void add(Graphic graphic)
  {
    graphics.add(graphic);
  }

  void remove(Graphic graphic)
  {
    graphics.remove(graphic);
  }

  ...
```

```
  public void draw()
  {
    System.out.print("Picture ");
    super.draw();
    System.out.println();

    for (Graphic graphic : graphics)
    {
      graphic.draw();
    }
  }

  public void move(int newX, int newY)
  {
    super.move(newX, newY);

    for (Graphic graphic : graphics)
    {
      graphic.move(newX, newY);
    }
  }
}
```

TestGraphic(1)

```
public class TestGraphic extends TestCase
{
  private Picture   mainGraphic;
  private Picture   subGraphic1;
  private Picture   subGraphic2;
  private Line      line1;
  private Line      line2;
  private Text      text1;
  private Text      text2;
  private Rectangle rect1;
  private Rectangle rect2;

  public void setUp()
  {
    mainGraphic = new Picture(10, 10);

    subGraphic1 = new Picture(11, 11);
    subGraphic2 = new Picture(12, 12);

    line1 = new Line(13, 13, 14, 14);
    line2 = new Line(15, 15, 16, 16);

    text1 = new Text(17, 17, "this is text 1");
    text2 = new Text(18, 18, "this is text 2");

    rect1 = new Rectangle(19, 19, 20, 20);
    rect2 = new Rectangle(21, 21, 22, 22);
  }
```

```
  public void testDraw()
  {
    System.out.println("Draw Graphics directly:");
    line1.draw();
    line2.draw();
    text1.draw();
    text2.draw();
    rect1.draw();
    rect2.draw();
    System.out.println("done.");
  }

  public void testMainComposite()
  {
    mainGraphic.add(line1);
    mainGraphic.add(line2);
    mainGraphic.add(text1);
    mainGraphic.add(text2);
    mainGraphic.add(rect1);
    mainGraphic.add(rect2);

    mainGraphic.draw();
    System.out.println("done.");
  }
```

TestGraphic(2)

```
  public void testSubComposites()
  {
    subGraphic1.add(line1);
    subGraphic1.add(line2);

    subGraphic2.add(text1);
    subGraphic2.add(text2);

    mainGraphic.add(subGraphic1);
    mainGraphic.add(subGraphic2);
    mainGraphic.add(rect1);
    mainGraphic.add(rect2);

    mainGraphic.draw();
    System.out.println("done.");
  }
```

```
  public void testMove()
  {
    subGraphic1.add(line1);
    subGraphic1.add(line2);
    subGraphic2.add(text1);
    subGraphic2.add(text2);
    mainGraphic.add(subGraphic1);
    mainGraphic.add(subGraphic2);
    mainGraphic.add(rect1);
    mainGraphic.add(rect2);

    mainGraphic.draw();
    System.out.println("done.");

    mainGraphic.move(0, 0);
    mainGraphic.draw();
    System.out.println("done.");
  }
}
```

Implementation - Explicit parent

- Maintaining references from child components to their parent can simplify the traversal and management of a composite structure:
  - The parent reference simplifies moving up the structure and deleting a component
  - The usual place to define the parent reference is in the Component class. Leaf and Composite classes can inherit the reference and the operations that manage it.
- With parent references, it's essential to maintain the invariant that all children of a composite have as their parent the composite that in turn has them as children.
  - The easiest way to ensure this is to change a component's parent only when it's being added or removed from a composite.
  - If this can be implemented once in the Add and Remove operations of the Composite class, then it can be inherited by all the subclasses, and the invariant will be maintained automatically.

Implementation – Component

- A goal Composite is to make clients unaware of the specific Leaf or Composite they're using.
    - Therefore Component should define as many common operations for Composite and Leaf classes as possible.
- Component can provide default implementations, and Leaf and Composite subclasses will override them.
    - However, this goal may conflict with the principle of class hierarchy design that says a class should only define operations that are meaningful to its subclasses.
- Some operations that Component supports that don't seem to make sense for Leaf classes.
    - How can Component provide a default implementation for them? …

Implementation – Component

- The interface for accessing children is a fundamental part of a Composite class but not necessarily Leaf classes.
  - View a Leaf as a Component that never has children, then define a default operation for child access in the Component class that never returns any children.
  - Leaf classes can use the default implementation, but Composite classes will reimplement it to return their children.

Implementation – Child

- Although Composite implements add() and remove() an important issue is where are these declared:
  - In the Component and make them meaningful for Leaf classes
  - In Composite and its subclasses
- Trade-off between safety and transparency:
  - In Component gives transparency, as all components can be treated uniformly. It costs in safety, however, because clients may try to do meaningless things like add and remove objects from leaves.
  - In Composite gives safety, because any attempt to add or remove objects from leaves will be caught at compile-time. But some transparency is lost, as leaves and composites have different interfaces.

Implementation – Child

- One approach is to declare an operation Composite getComposite() in the Component class.
- Component provides a default operation that returns null.
- The Composite class redefines this operation to return itself through the this reference

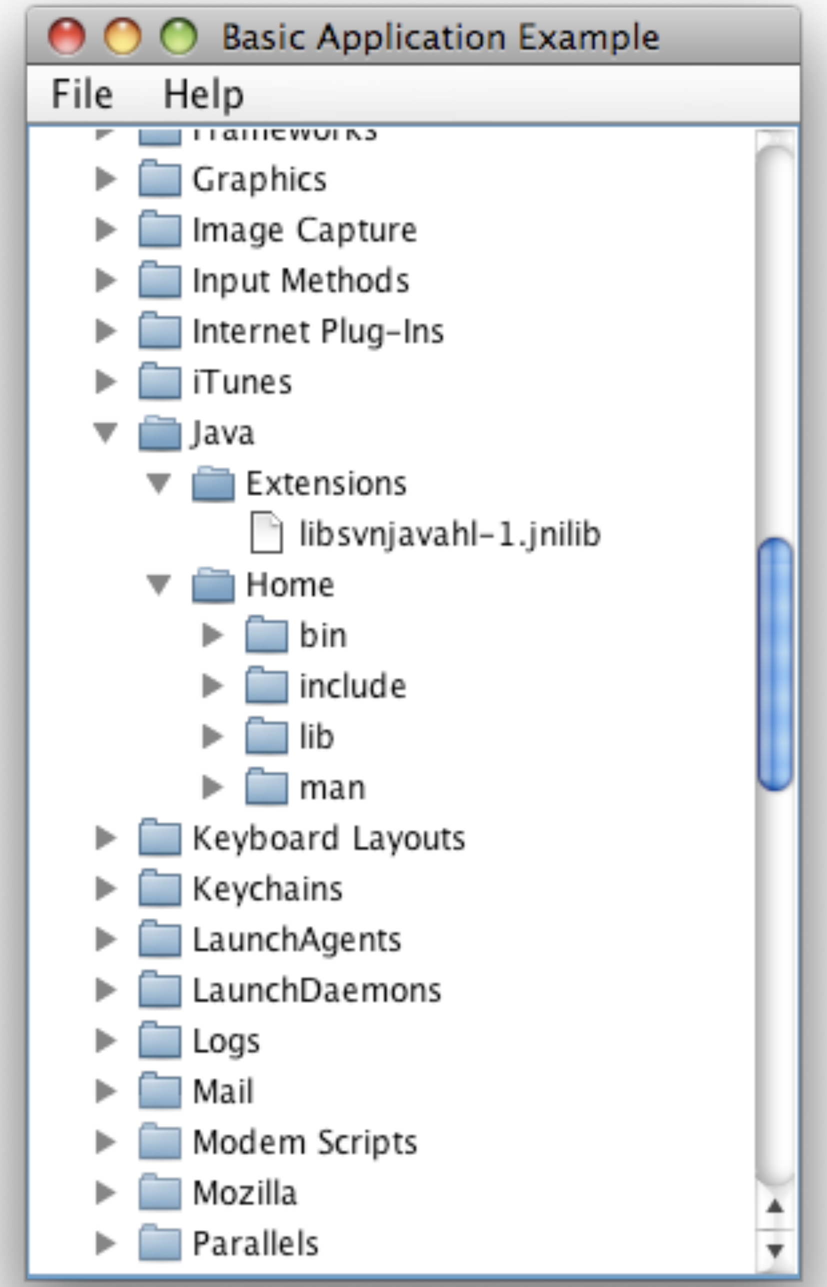
```
class Component
{
  //...
  Composite getComposite()
  {
    return null;
  }
  //...
}

class Composite extends Component
{
 Composite getComposite()
  {
    return this;
  }
}

class Leaf : public Component
{
  //...
}
```

Composite File System Example

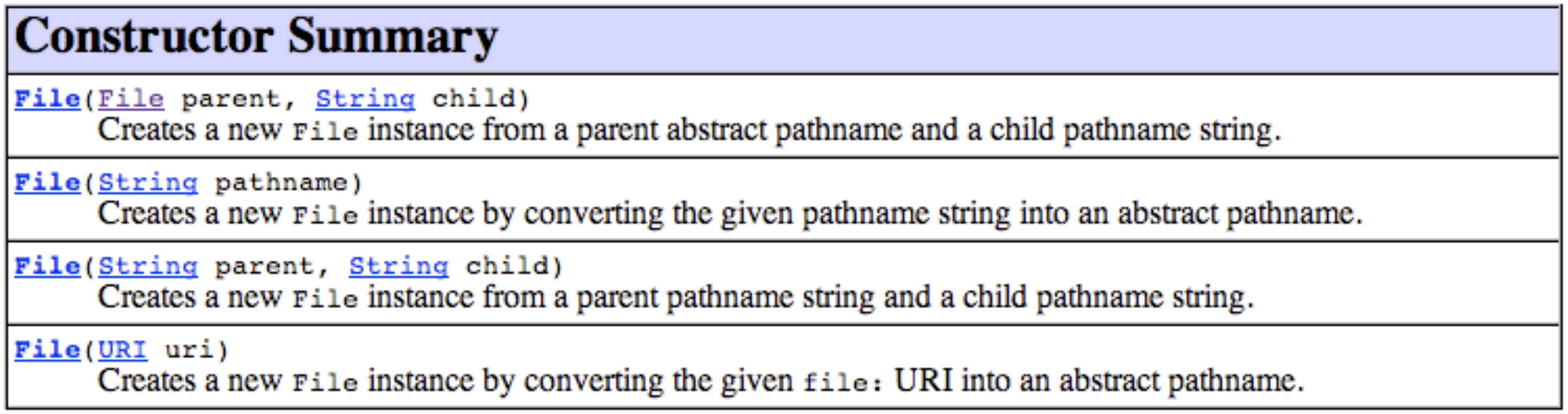
- Filesystem & Composite Pattern
- java.io.File
- Composite Pattern
- Eager & Lazy versions

File System

- A file system is a domain that can be elegantly modeled using the Composite Pattern.
- Each Node in the structure is either a file or a directory.
- A File is a Leaf node
- A Directory is an “internal” node i.e. a node that has children - a Composite
- These children may be files, or directories.



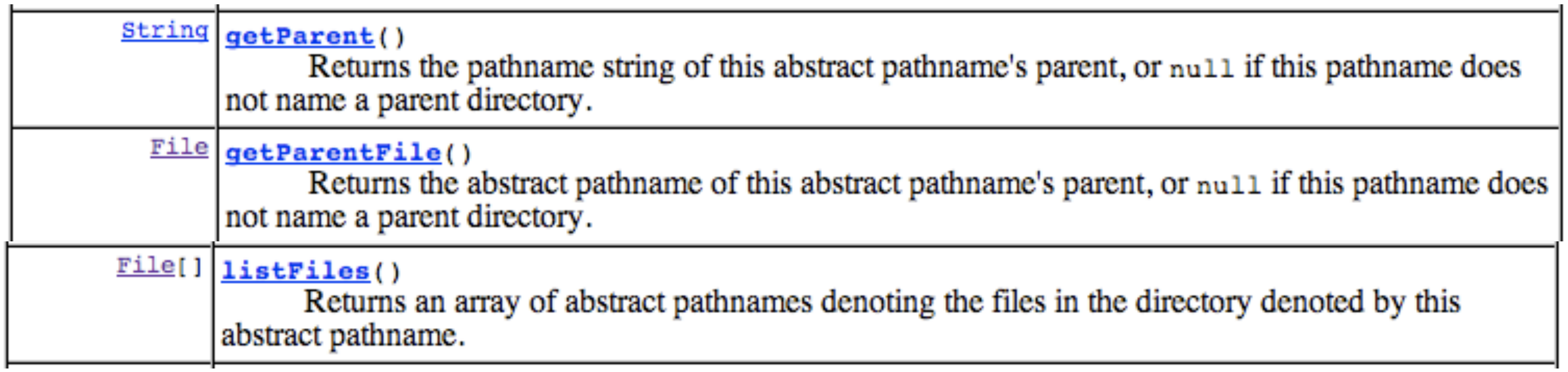
java.file.io



Constructor Summary
`File(File parent, String child)`   Creates a new `File` instance from a parent abstract pathname and a child pathname string.
`File(String pathname)`   Creates a new `File` instance by converting the given pathname string into an abstract pathname.
`File(String parent, String child)`   Creates a new `File` instance from a parent pathname string and a child pathname string.
`File(URI uri)`   Creates a new `File` instance by converting the given `file:` URI into an abstract pathname.

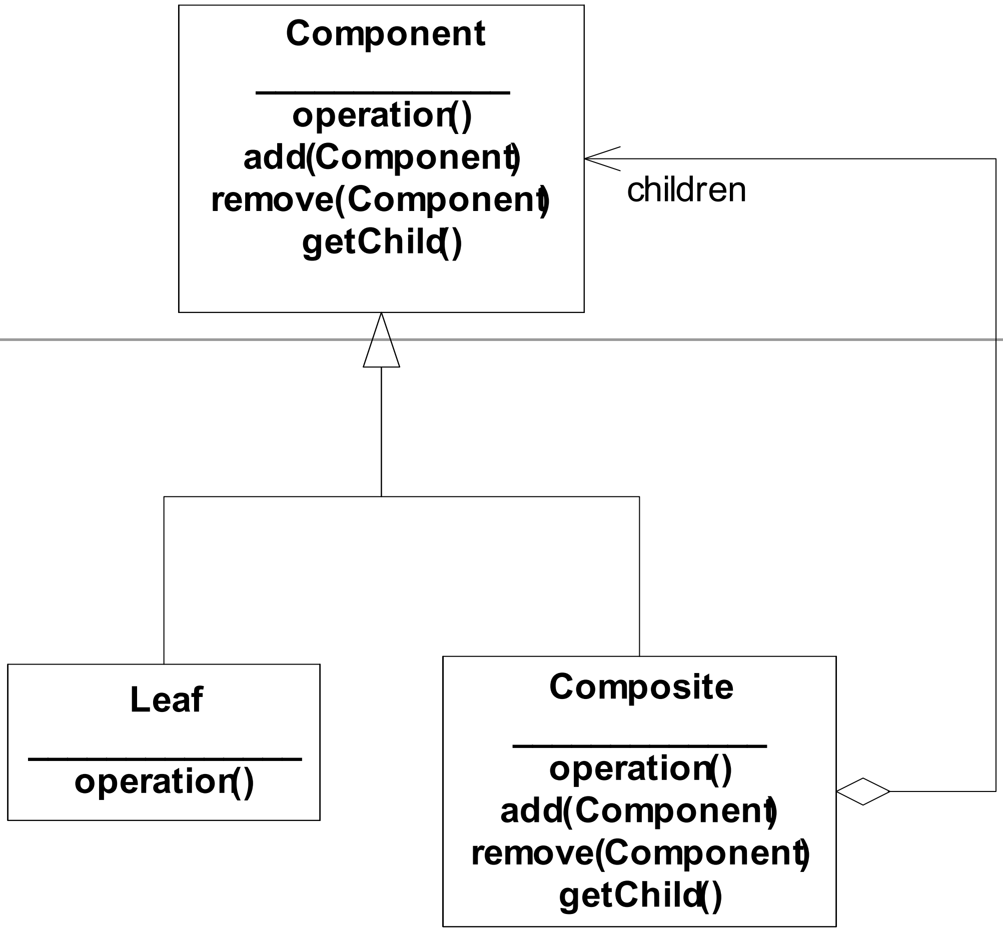
- File class can be viewed as modeling a node in a file system tree

Composite Methods



String	getParent() Returns the pathname string of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
File	getParentFile() Returns the abstract pathname of this abstract pathname's parent, or `null` if this pathname does not name a parent directory.
File[]	listFiles() Returns an array of abstract pathnames denoting the files in the directory denoted by this abstract pathname.

Composite Pattern



```
public class Component
{
  protected File file;

  public Component (File file)
  {
    this.file = file;
  }

  public boolean hasChildren()
  {
    return false;
  }

  public int getNumberChildren()
  {
    return 0;
  }

  public Component getChild (int index)
  {
    return null;
  }

  public String toString()
  {
    return file.getName();
  }
}
```

```
public class Leaf extends Component
{
  public Leaf(File file)
  {
    super (file);
  }
}
```

Composite Class
- Lazy Version

- Constructor merely stores the file reference.
- File system is only traversed if the getChild() method is called.

```
public class Composite extends Component
{
  public Composite(File file)
  {
    super(file);
  }

  public boolean hasChildren()
  {
    return file.isDirectory();
  }

  public int getNumberChildren()
  {
    return file.listFiles().length;
  }

  public Component getChild(int index)
  {
    File childFile = file.listFiles()[index];
    if (childFile.isDirectory())
    {
      return new Composite(childFile);
    }
    else
    {
      return new Leaf(childFile);
    }
  }
}
```

Command Line Explorer

- Explorer constructor takes the path to the root of the file system to be explored.
- A single root object is created.
- visit() traverses this tree, printing each file name, suitably indented.

```
public class Explorer
{
  private Composite root;

  public Explorer(String path)
  {
    File file = new File(path);
    root = new Composite(file);
  }

  public void print(int level)
  {
    visit(root, level);
  }

  private void visit(Component node, int level)
  {
    String format = "%" + level*2 + "s%s";
    System.out.println(String.format(format, " ", node.toString()));
    if (node.hasChildren())
    {
      level++;
      for (int i = 0; i < node.getNumberChildren(); i++)
      {
        Component child = node.getChild(i);
        visit(child, level);
      }
    }
  }

  public static void main(String[] args)
  {
    Explorer explorer = new Explorer("/Test");
    explorer.print(1);
  }
}
```

Composite Class - Eager Version

- Constructor takes a single “root” from a file system.
- Fills its components collection with Leafs (for files) and Composites (for directories).
- Once constructor has completed, the full filesystem (from root) will have been traversed.

```java
public class EagerComposite extends Component
{
  private List<Component> components;

  public EagerComposite(File file)
  {
    super(file);
    components = new ArrayList<Component>();
    for (File eachFile : file.listFiles())
    {
      if (eachFile.isDirectory())
      {
        components.add(new EagerComposite(eachFile));
      }
      else
      {
        components.add(new Leaf(eachFile));
      }
    }
  }

  public boolean hasChildren()
  {
    return !components.isEmpty();
  }

  public int getNumberChildren()
  {
    return components.size();
  }

  public Component getChild(int index)
  {
    return components.get(index);
  }
}
```