

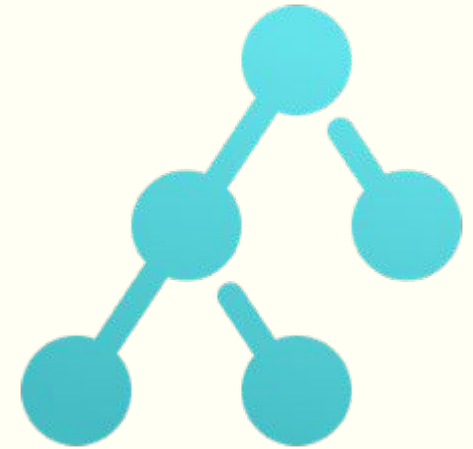
Composite Design Pattern

Presenters:
Yunong Li
Harika Bandaru
Shashank Singh



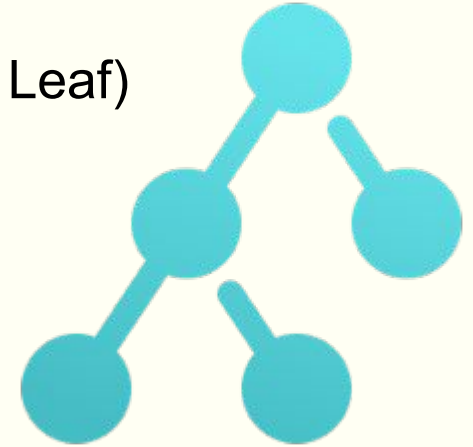
Intent

- Compose objects into Tree Structure
- Allow the Tree Structure to represent part-whole hierarchies
- Allow Client to treat individual objects and composite object uniformly



Element

- Component (Graphic)
Provide an interface to be implemented for subclass (Composite, Leaf)
- Composite (CompositeGraphic)
Could hold individual objects and other composite objects
Inherited from Component class
- Leaf (Circle, Eclipse, Rectangle)
Do not hold other individuals objects or composite objects
Inherited from Component class

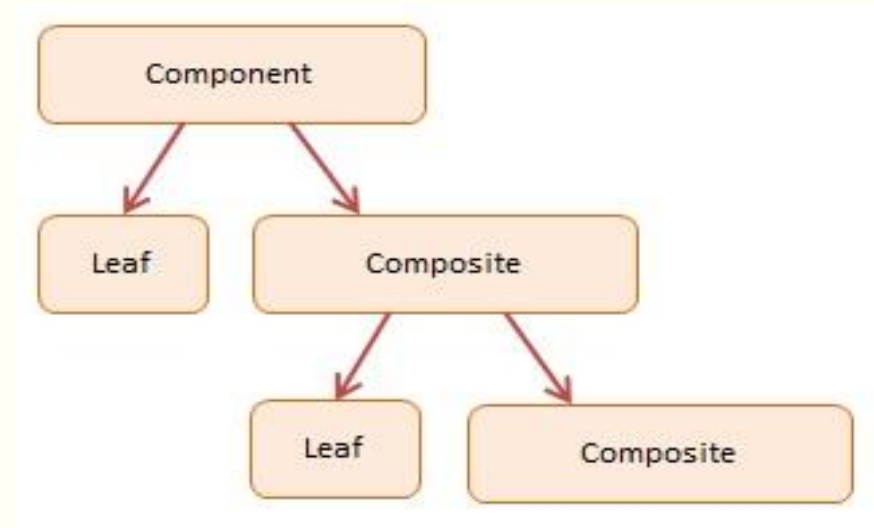
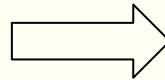
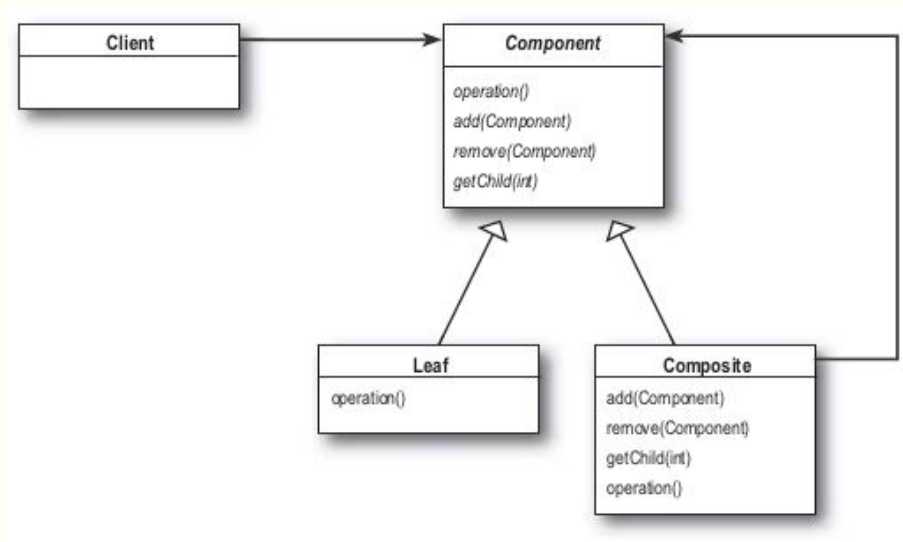


Motivation

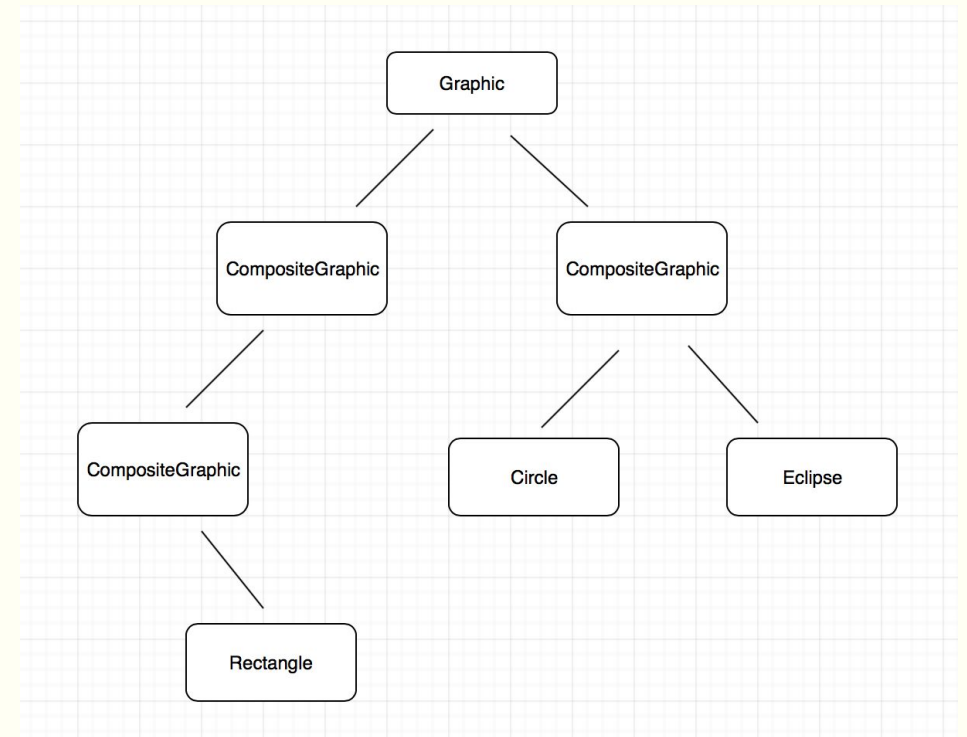
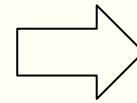
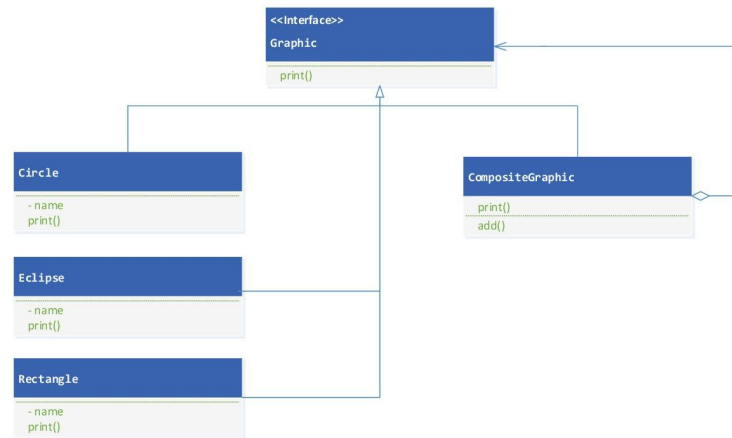
- Differentiate Component object and Leaf object ?
- How to Solve?

Motivation

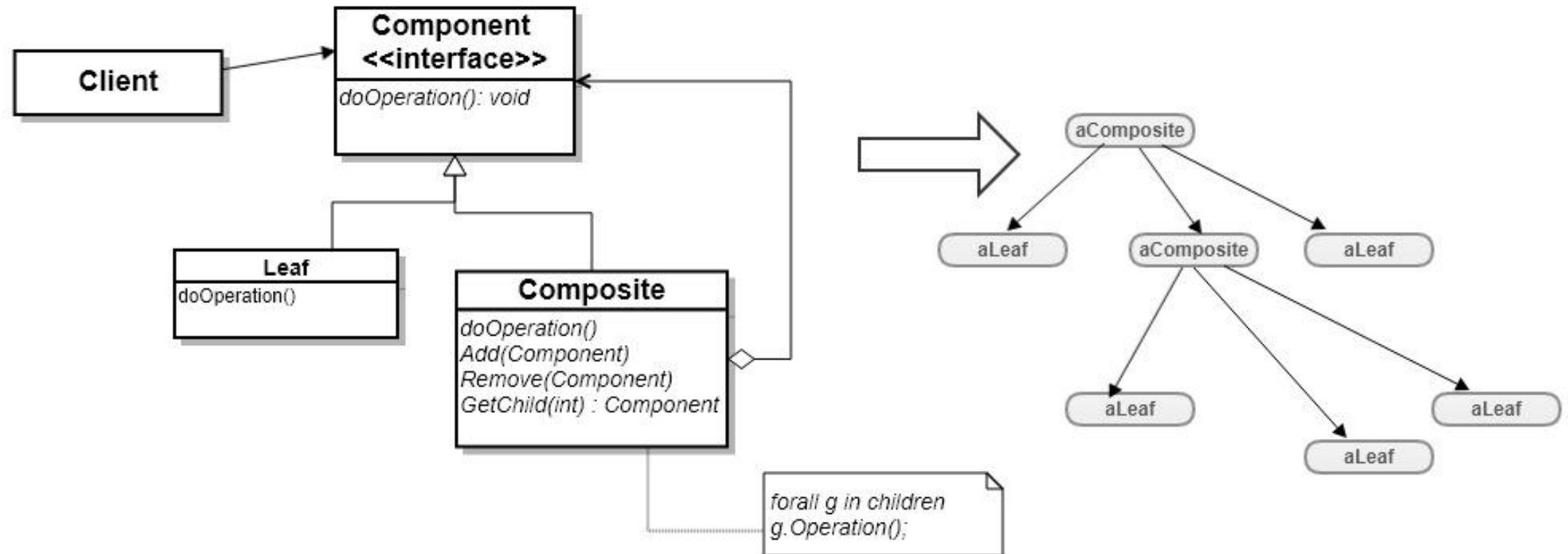
- Transfer the Structure into Tree Structure



Application



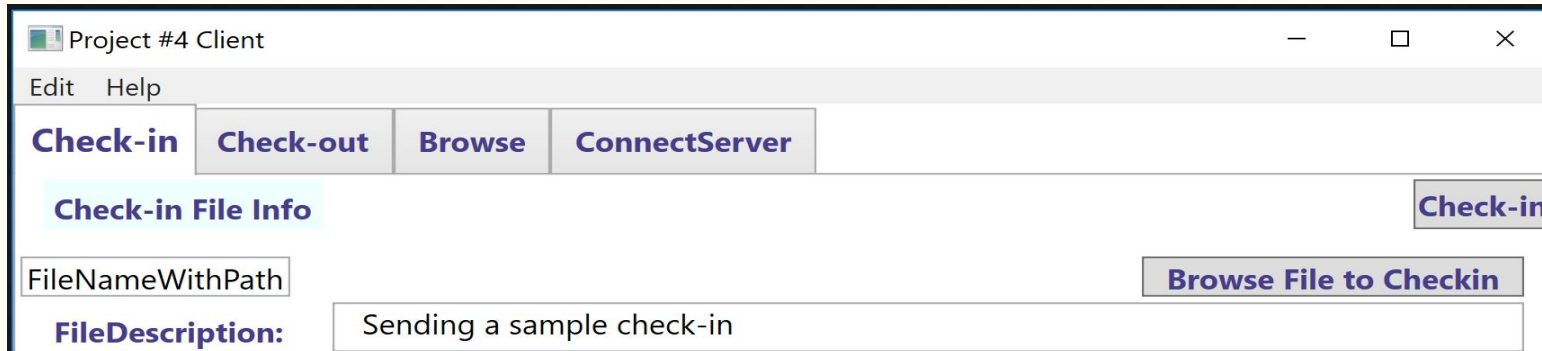
Structure



Examples

- Grids in WPF
- JPanel in java

```
<Grid Grid.Row="0" Grid.ColumnSpan="3">
  <Grid.RowDefinitions>
    <RowDefinition Height="25"></RowDefinition>
    <RowDefinition Height="*"></RowDefinition>
  </Grid.RowDefinitions>
  <Grid Grid.Row="0">
    <Button Width="Auto" Height="25" DockPanel.Dock="Right" Click="chckinClick" HorizontalAlignment="Right">Check-in</Button>
    <Label DockPanel.Dock="Left" Margin="10 0" HorizontalAlignment="Left" FontSize="14" FontWeight="Bold" BackgroundColor="LightBlue">Check-in File Info</Label>
  </Grid>
  <Grid Grid.Row="1">
    <TextBlock Text="File Name With Path" HorizontalAlignment="Left" FontSize="14" FontWeight="Bold" BackgroundColor="LightBlue" />
    <TextBlock Text="File Description: Sending a sample check-in" HorizontalAlignment="Left" FontSize="14" FontWeight="Bold" BackgroundColor="LightBlue" />
  </Grid>
</Grid>
```



Project #4 Client

Edit Help

Check-in Check-out Browse ConnectServer

Check-in File Info Check-in

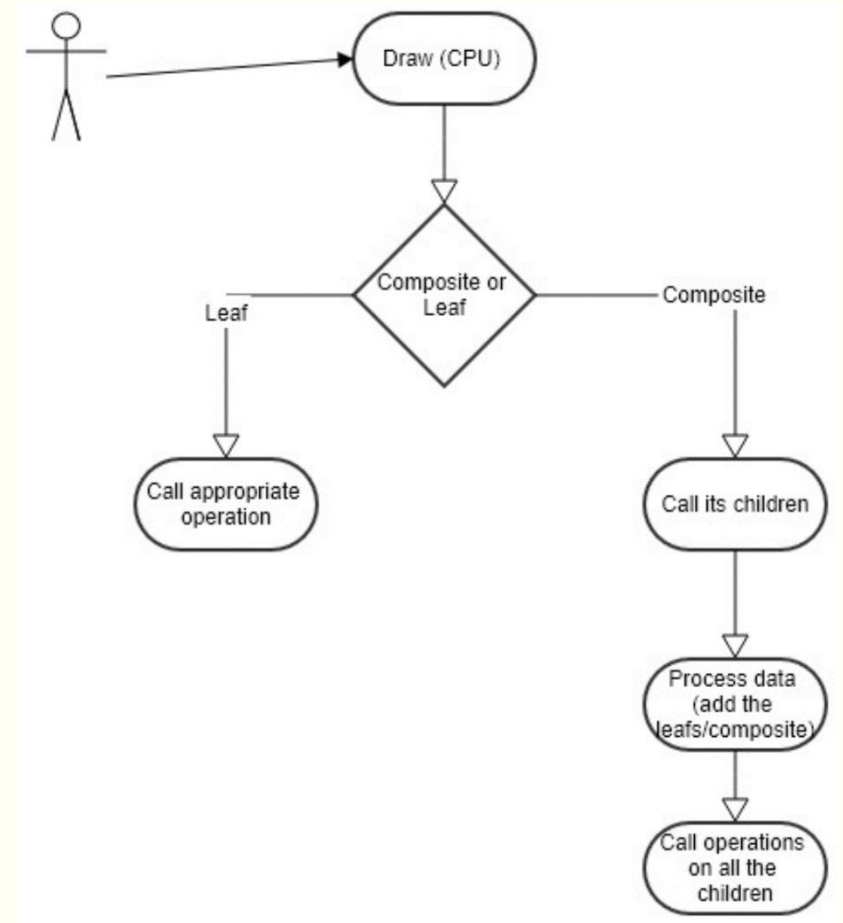
FileNameWithPath Browse File to Checkin

FileDescription: Sending a sample check-in

Participants

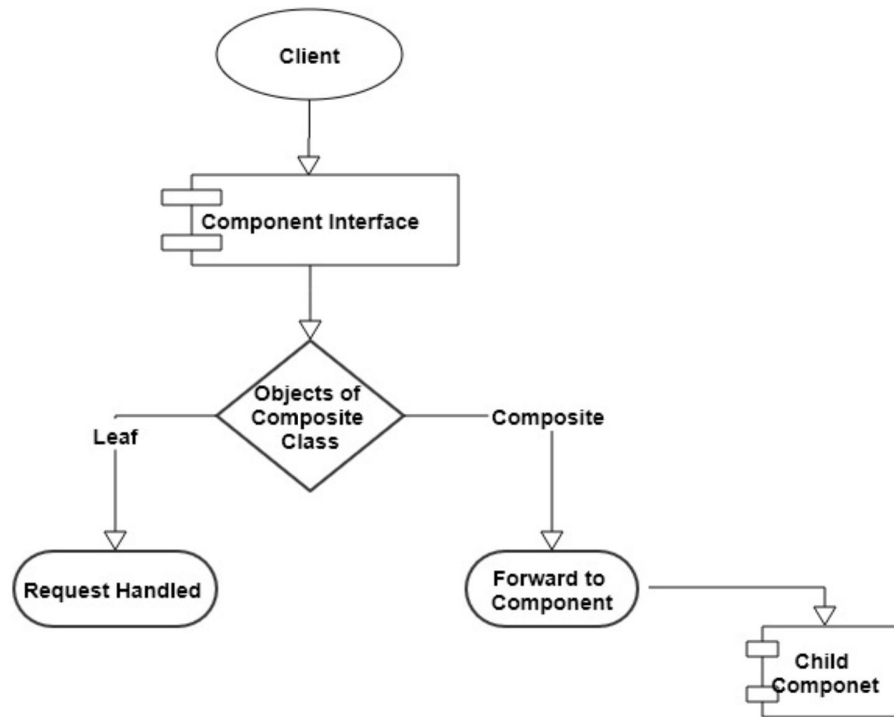
Four participants that should be focused

- Component
- Leaf
- Composite
- Client

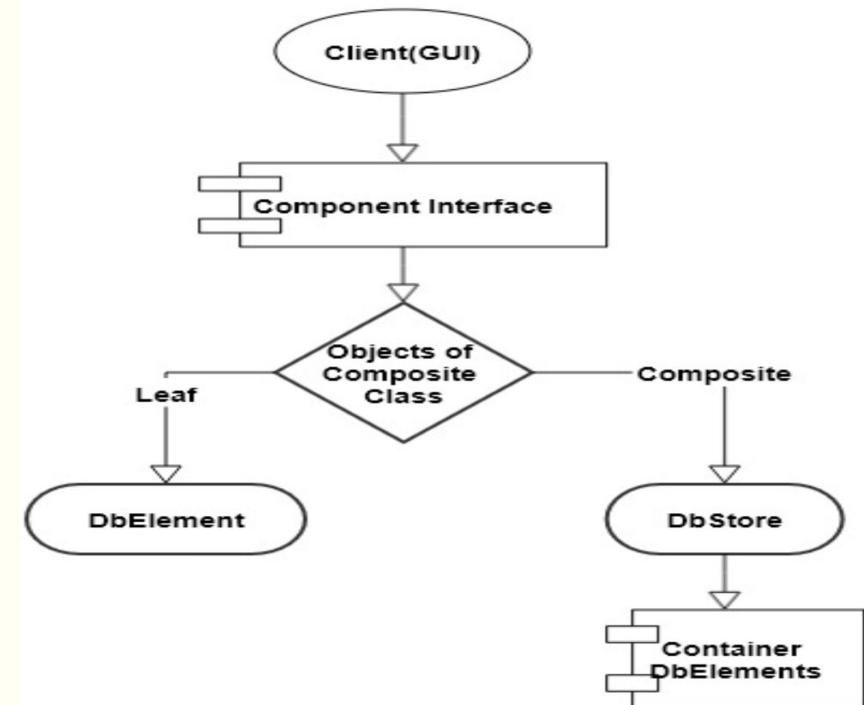


Collaborations

General Usage



Example



Consequences

- Makes the client simple.
- Makes it easier to add new kinds of components.
- Transparency Vs Safety
- Make your design overly general, there might be cases to keep some restrictions.
Ex: Document vs paragraphs.

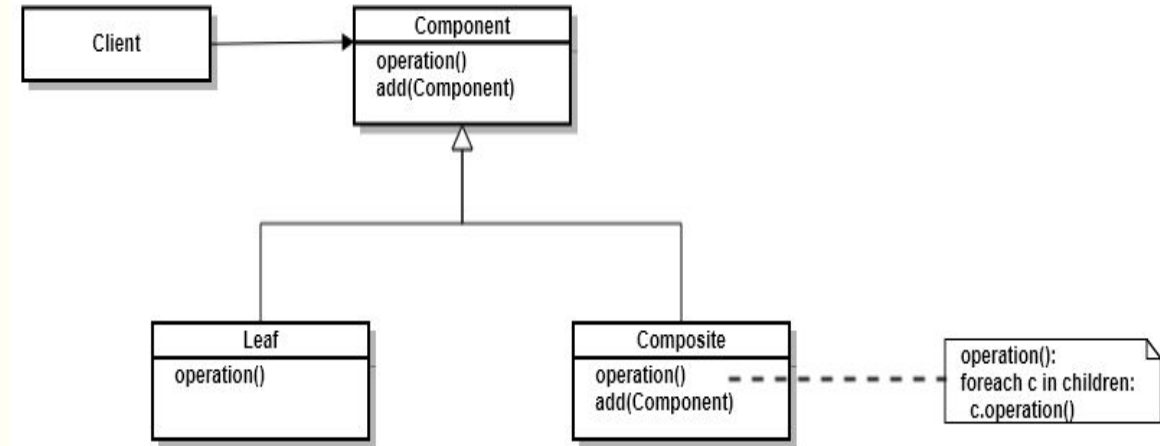
Implementation

Component interface

```
class Graphic {  
  
public:  
    virtual void print() = 0;  
    virtual void add(Graphic* graphic) {};  
};
```

Composite class

```
class CompositeGraphic : public Graphic {  
    std::vector<Graphic *> childGraphics;  
public:  
    void print();  
    void add(Graphic* graphic);  
};
```



```
void CompositeGraphic::print() {  
    for (Graphic* graphic : childGraphics) {  
        graphic->print();  
    }  
}  
  
void CompositeGraphic::add(Graphic* graphic) {  
    childGraphics.push_back(graphic);  
}
```

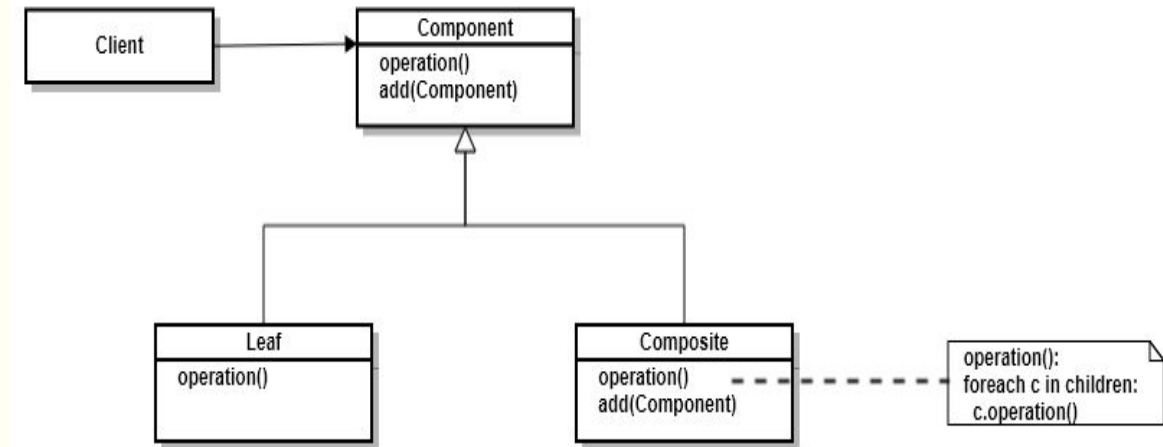
Implementation

Leaf classes

```
class Circle : public Graphic {  
public:  
    void print() {  
        std::cout << "Circle" << std::endl;  
    }  
};
```

```
class Eclipse : public Graphic {  
public:  
    void print() {  
        std::cout << "Eclipse" << std::endl;  
    }  
};
```

```
class Rectangle : public Graphic {  
public:  
    void print() {  
        std::cout << "Rectangle" << std::endl;  
    }  
};
```

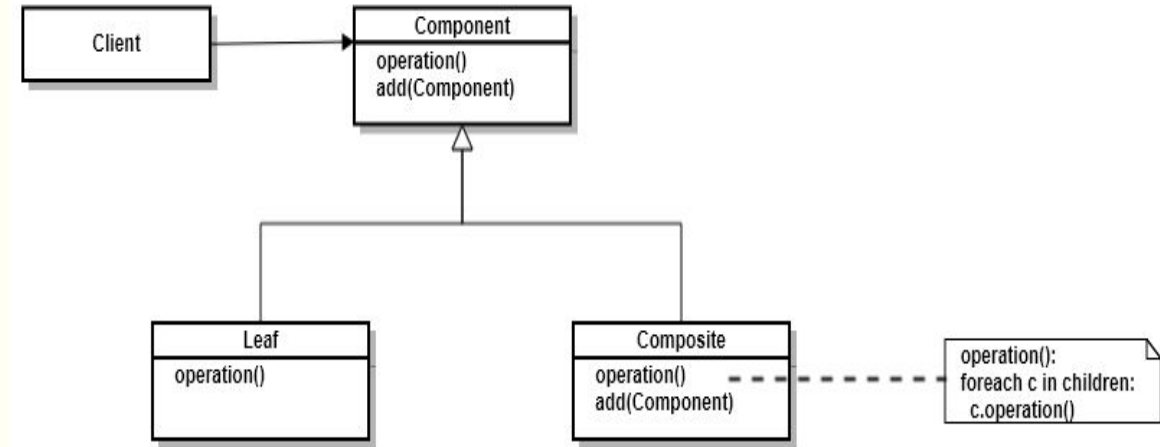


```
class Triangle : public Graphic {  
public:  
    void print() {  
        std::cout << "Triangle" << std::endl;  
    }  
};
```

Implementation

Client Operation

```
int main() {  
    Graphic* circle = new Circle();  
    Graphic* eclipse = new Eclipse();  
    Graphic* rectangle = new Rectangle();  
    Graphic* triangle = new Triangle();  
  
    Graphic* graph = new CompositeGraphic();  
    CompositeGraphic* subgraph1 = new CompositeGraphic();  
    CompositeGraphic* subgraph2 = new CompositeGraphic();  
  
    graph->add(subgraph1);  
    graph->add(subgraph2);  
  
    subgraph1->add(triangle);  
    subgraph2->add(eclipse);  
    subgraph2->add(rectangle);  
    subgraph2->add(circle);  
  
    graph->print();  
  
    return 0;  
}
```



Output

```
Triangle
Eclipse
Rectangle
Circle
Press any key to continue . . .
```

When to use??

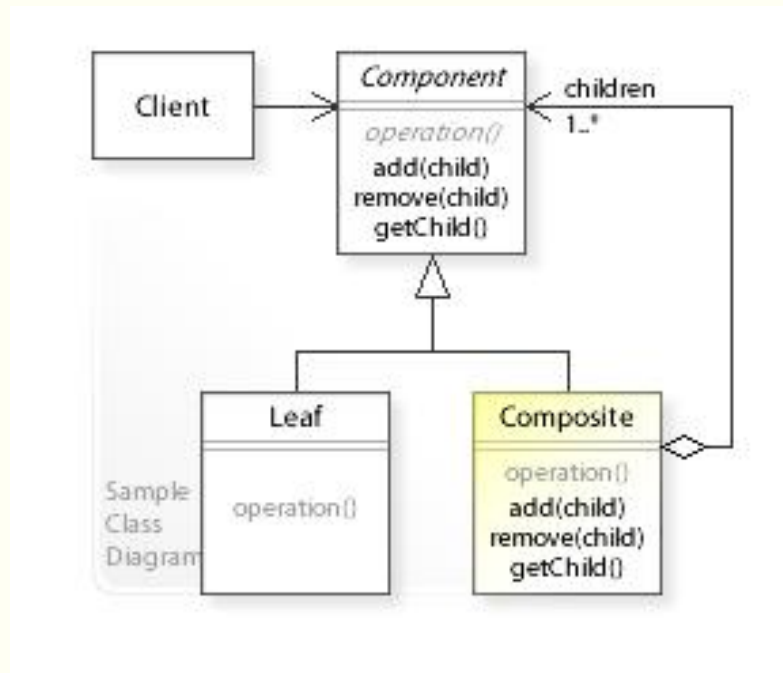
- When clients ignore the difference between compositions of objects and individual objects.
- When the code size increased exponentially due to re-using the same code again and again.
- Design for uniformity
- Design for type safety

Examples

- Destruction of objects in C++
- Graphical Editors

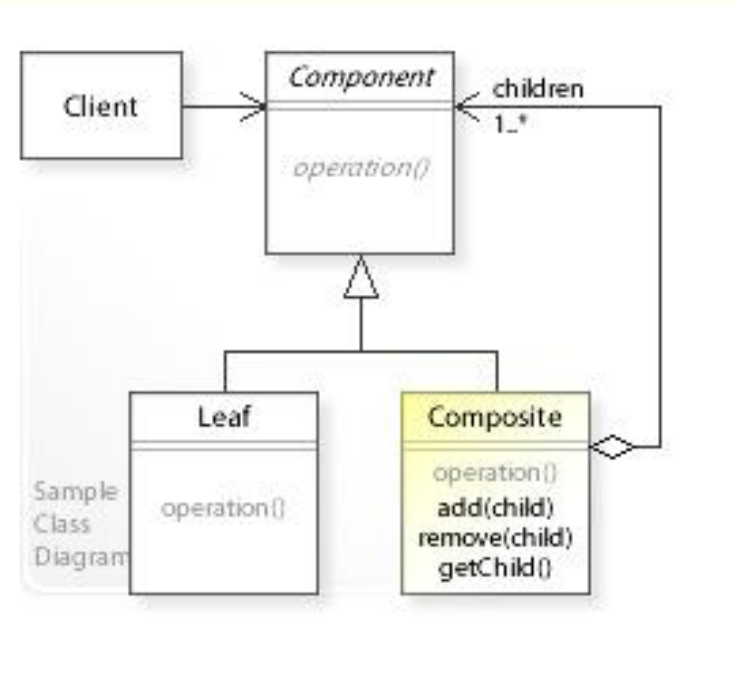
Implementation Issues

Design for Uniformity



- To provide **uniformity** is to define child-related operations in the **Component** interface.
- This enables **uniformity** because clients can treat **Leaf** and **Composite** objects uniformly.

Design for Type Safety



- To provide **type safety** is to define child-related operations solely in the **Composite** class.
- This enables **type safety** because we can rely on the type system to enforce type constraints.

Related Patterns

- Flyweight

Leaf objects can be implemented as shared flyweight objects.

- Chain of Responsibility

Existing composite object structures can be used to define the successor chain.

- Iterator

Iterator provides a way to traverse the elements of an object structure.

Reference

https://sourcemaking.com/design_patterns/composite

<http://w3sdesign.com/?gr=s03&ugr=implem#gf>

Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, et. al., Addison-Wesley, 1994, ISBN 0-201-63361-2