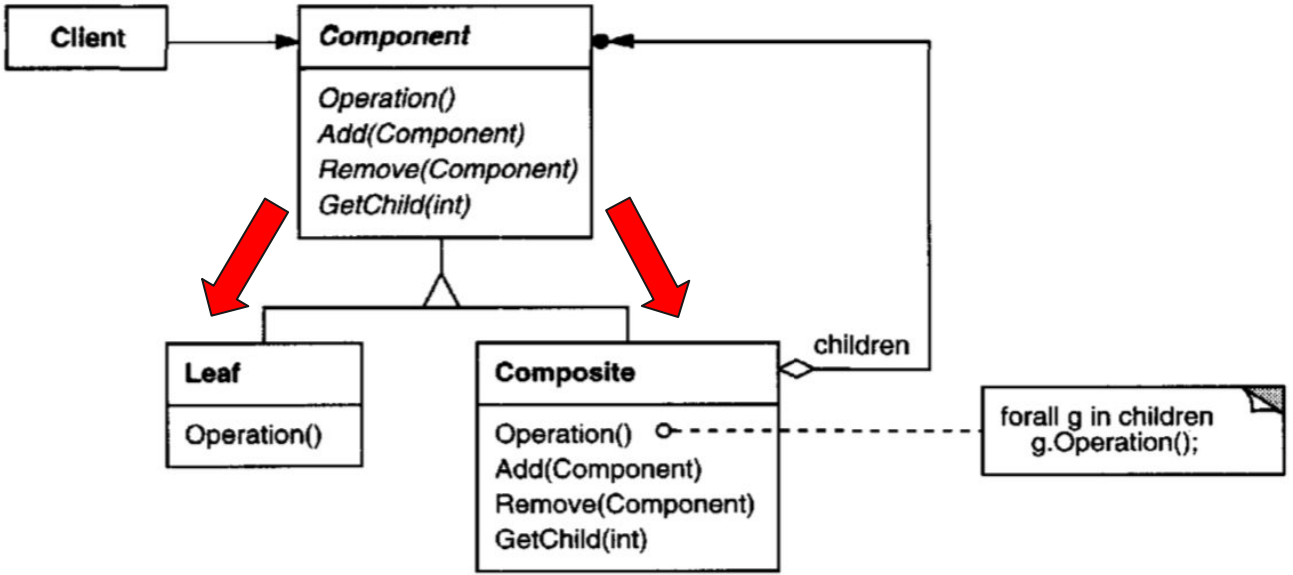
Composite Pattern

Felicia Jiang
Winter 2019 - OOP

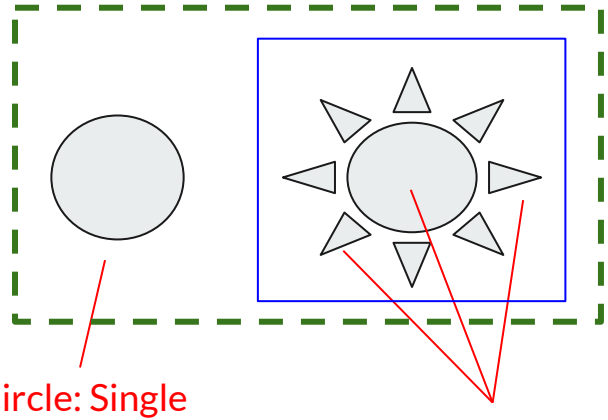
Why composite?

- Objective: represent part-whole hierarchies
- Objective: conceal differences between **compositions** of objects and **individual** objects from clients

Component class contains all the methods present in leaf and composite



A graphic example

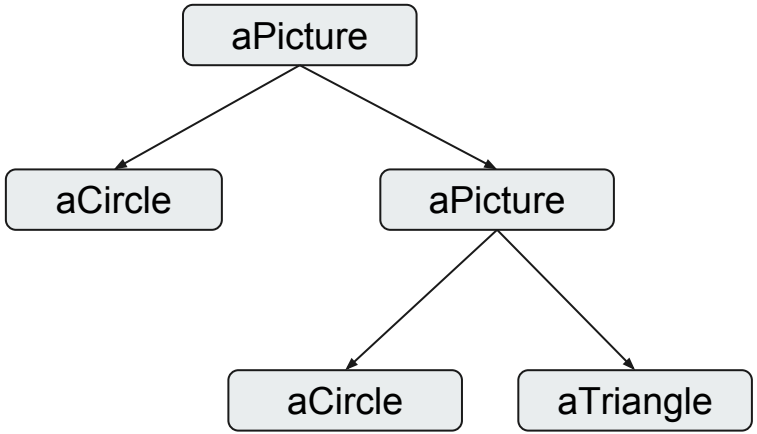


Picture:
grouped objects

Also a picture:
single object + picture

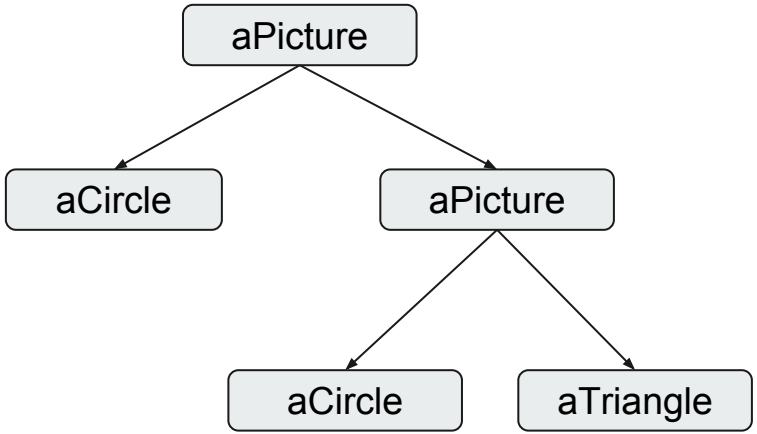
Circle: Single object

Triangle, circle: Single object(s)



A graphic example

- **Component** → Graphic
  - This is the base class both composite and leaf inherit from
- **Composite** → Picture
  - Has children
- **Leaf** → Circle, Triangle, etc
  - Has no children



```python
# Component class
class Graphic:

    def __init__(self):
        self._identifier = None
        self._parent = None
        self._specifications = None
        self._children = None

    def getComposite(self):
        '''
        Default to None for leaves, override in Composite class
        '''
        return None

    def addParent(self, parent):
        self._parent = parent

    def removeParent(self):
        self._parent = None

    def addChild(self, child):
        '''
        Check that self is not a leaf.
        '''
        if self.getComposite() or self.getComposite() == set():
            self._children.add(child)
            child.addParent(self)

    def removeChild(self, child):
        '''
        Check that self is not a leaf.
        '''
        if self.getComposite():
            self._children.discard(child)
            child.removeParent()

    def printSpecifications(self, indent):
        '''
        Default for leaf
        '''
        print(indent*'\t' + self._identifier + ': ' + str(self._specifications))
```

```python
# composite class
class Picture(Graphic):

    def __init__(self):
        self._identifier = 'Picture'
        self._children = set()

    def getComposite(self):
        '''
        Override base class method which returns None
        '''
        return self._children

    def printSpecifications(self, indent):
        '''
        Override base class
        '''
        print(indent*'\t' + 'In picture:')
        for child in self._children:
            child.printSpecifications(indent+1)
```

override

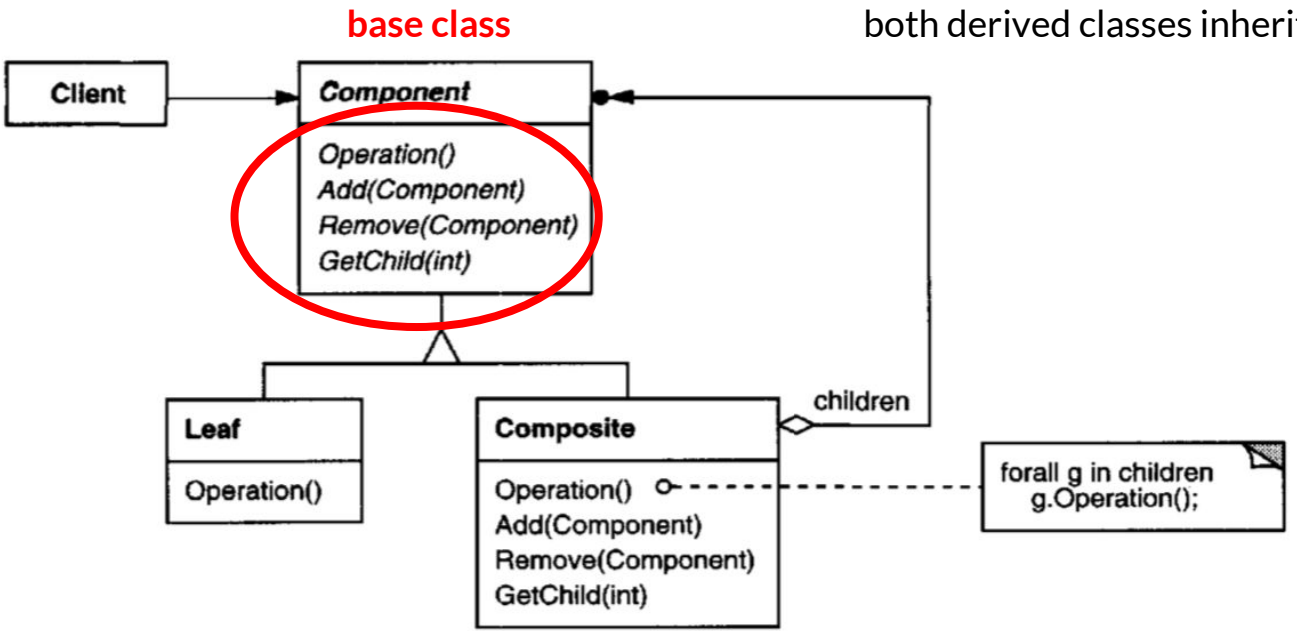
override

```python
# leaf classes
class Rectangle(Graphic):
    def __init__(self, given_width, given_length):
        self._identifier = 'Rectangle'
        self._specifications = {'width': given_width, 'length': given_length}


class Circle(Graphic):
    def __init__(self, given_radius):
        self._identifier = 'Circle'
        self._specifications = {'radius': given_radius}


class Text(Graphic):
    def __init__(self, given_text, given_size):
        self._identifier = 'Text'
        self._specifications = {'text': given_text, 'size': given_size}
```

Component class specifies child-management operations that both derived classes inherit from



Summary of implementation concerns

1. **Explicit parent references**:
    a. parent knows children, child knows parent
2. **Maximize common interface**:
    a. component should define as many common operations for Leaf and Composite as possible
3. **Child management**:
    a. addChild, removeChild methods can be defined in Component to fail by default, override in Composite class
4. **Data structure for tracking children**:
    a. I chose set, can use tree, linked list, hash map, etc

Component (Graphic)

- Define as many methods common to leaf and composite as possible
- Parent-child references

```
# Component class
class Graphic:

    def __init__(self):
        self._identifier = None
        self._parent = None
        self._specifications = None
        self._children = None

    def getComposite(self):
        '''
        Default to None for leaves, override in Composite class
        '''
        return None

    def addParent(self, parent):
        self._parent = parent

    def removeParent(self):
        self._parent = None

    def addChild(self, child):
        '''
        Check that self is not a leaf.
        '''
        if self.getComposite() or self.getComposite() == set():
            self._children.add(child)
            child.addParent(self)

    def removeChild(self, child):
        '''
        Check that self is not a leaf.
        '''
        if self.getComposite():
            self._children.discard(child)
            child.removeParent()

    def printSpecifications(self, indent):
        '''
        Default for leaf
        '''
        print(indent*'\t' + self._identifier + ': ' + str(self._specifications))
```

Composite (Picture)

- Define as many methods common to leaf and composite as possible
- Parent-child references

```
# composite class
class Picture(Graphic):

    def __init__(self):
        self._identifier = 'Picture'
        self._children = set()


    def getComposite(self):
        '''
        Override base class method which returns None
        '''
        return self._children


    def printSpecifications(self, indent):
        '''
        Override base class
        '''
        print(indent*'\t' + 'In picture:')
        for child in self._children:
            child.printSpecifications(indent+1)
```

Leaf (Circle, etc)

- Cannot have children
- Inherits child-management methods that return nothing

```
# leaf classes
class Rectangle(Graphic):
    def __init__(self, given_width, given_length):
        self._identifier = 'Rectangle'
        self._specifications = {'width': given_width, 'length': given_length}


class Circle(Graphic):
    def __init__(self, given_radius):
        self._identifier = 'Circle'
        self._specifications = {'radius': given_radius}


class Text(Graphic):
    def __init__(self, given_text, given_size):
        self._identifier = 'Text'
        self._specifications = {'text': given_text, 'size': given_size}
```