Software Design and Architecture

Iterator and Composite Patterns

Design principles

High-level principles

- **S**ingle Responsibility
- **O**pen/Closed
- **L**iskov Substitution Principle
- **I**nterface Segregation
- **D**ependency Inversion

Low-level principles

- Encapsulate what varies
- Program to interfaces, not implementations
- Favor composition over inheritance
- Strive for loose coupling

Iterator

Behavioral Patterns

- observer
- decorator
- strategy
- command
- template
- null object
- state
- **iterator**

Creational Patterns

- factory method
- abstract factory
- singleton

Structural Patterns

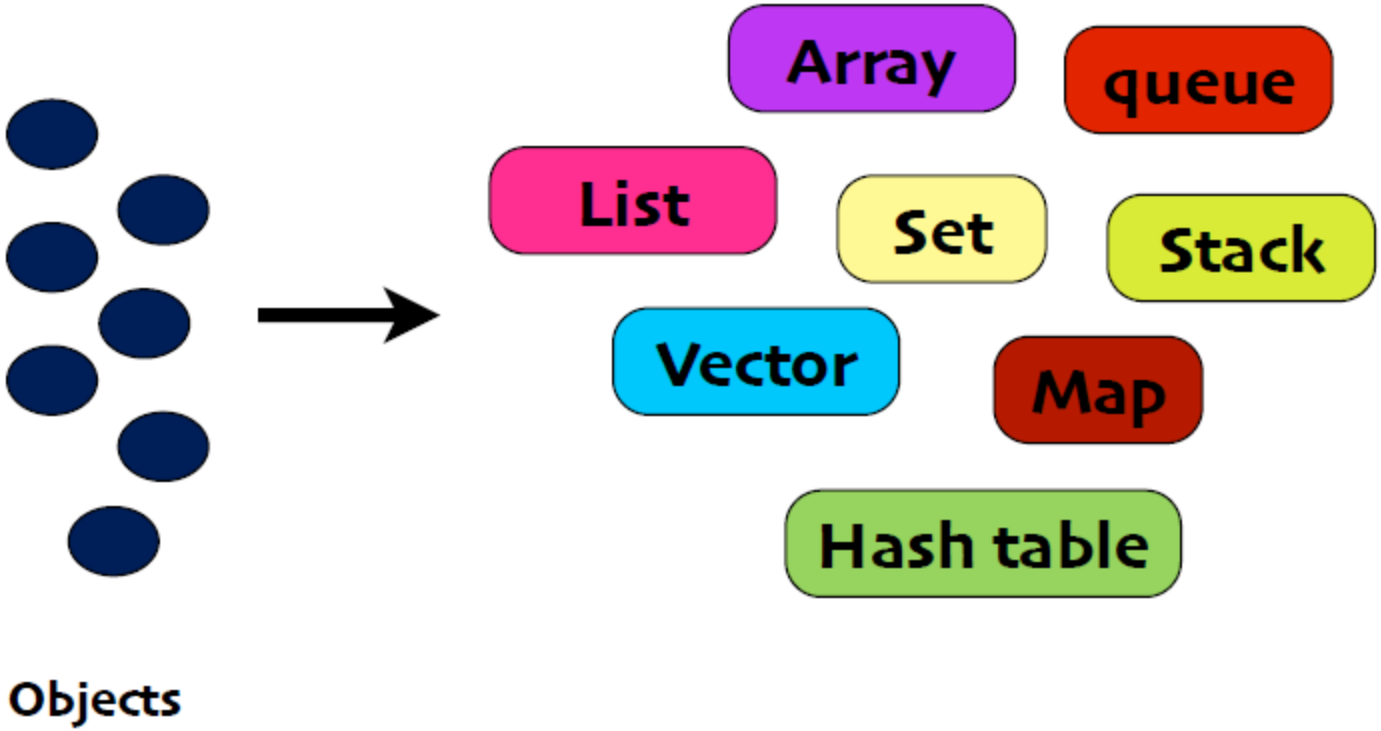
- adapter
- facade

Problem

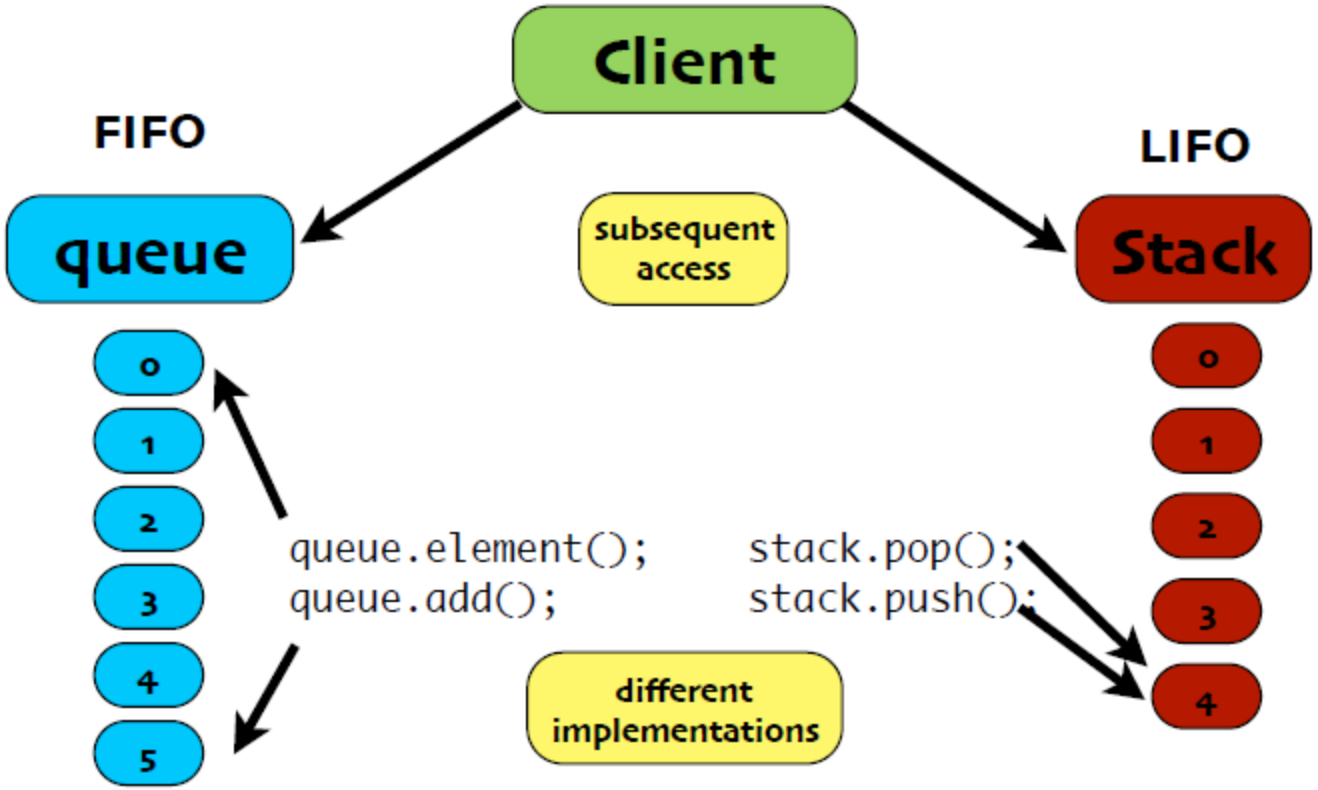
Aggregate objects such as a list should allow a way to traverse its elements without exposing its internal structure

You don't want to know how this is implemented

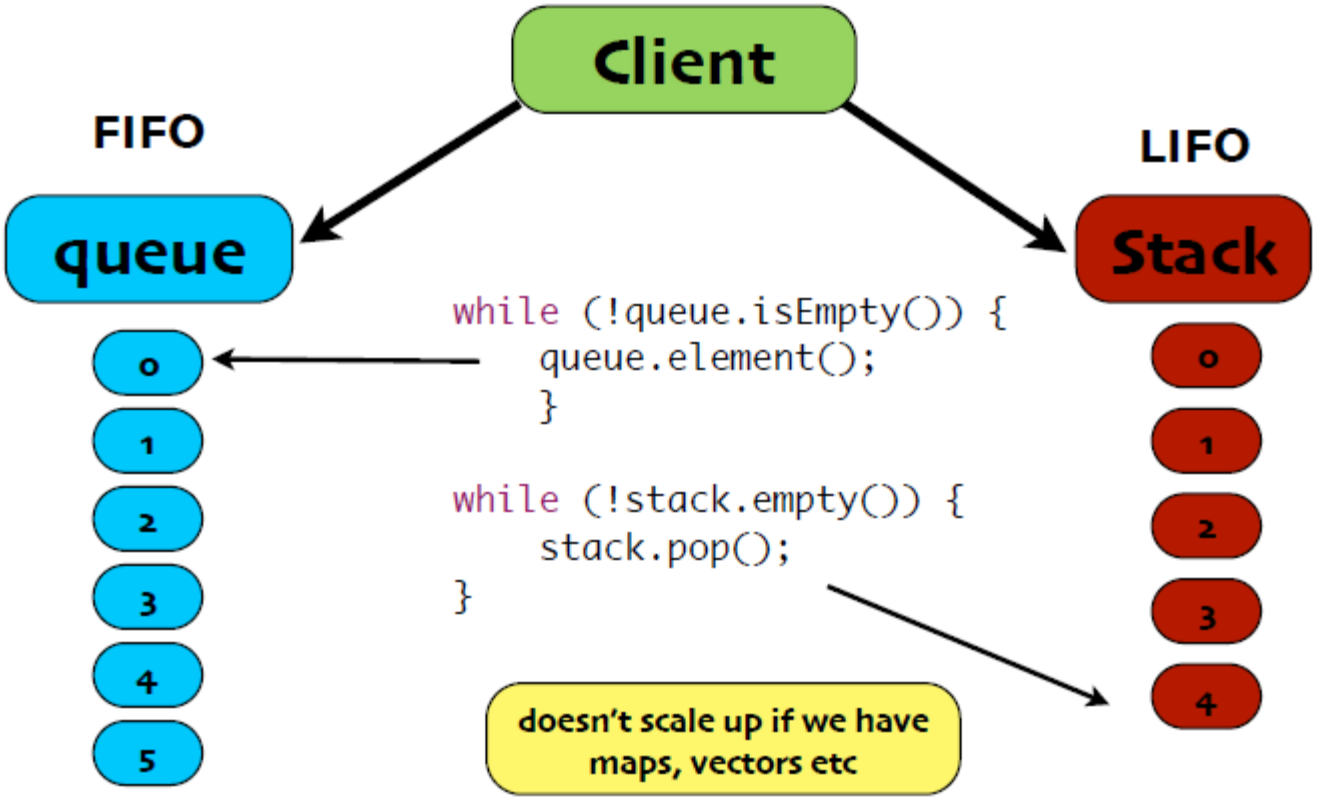
Collection Managers



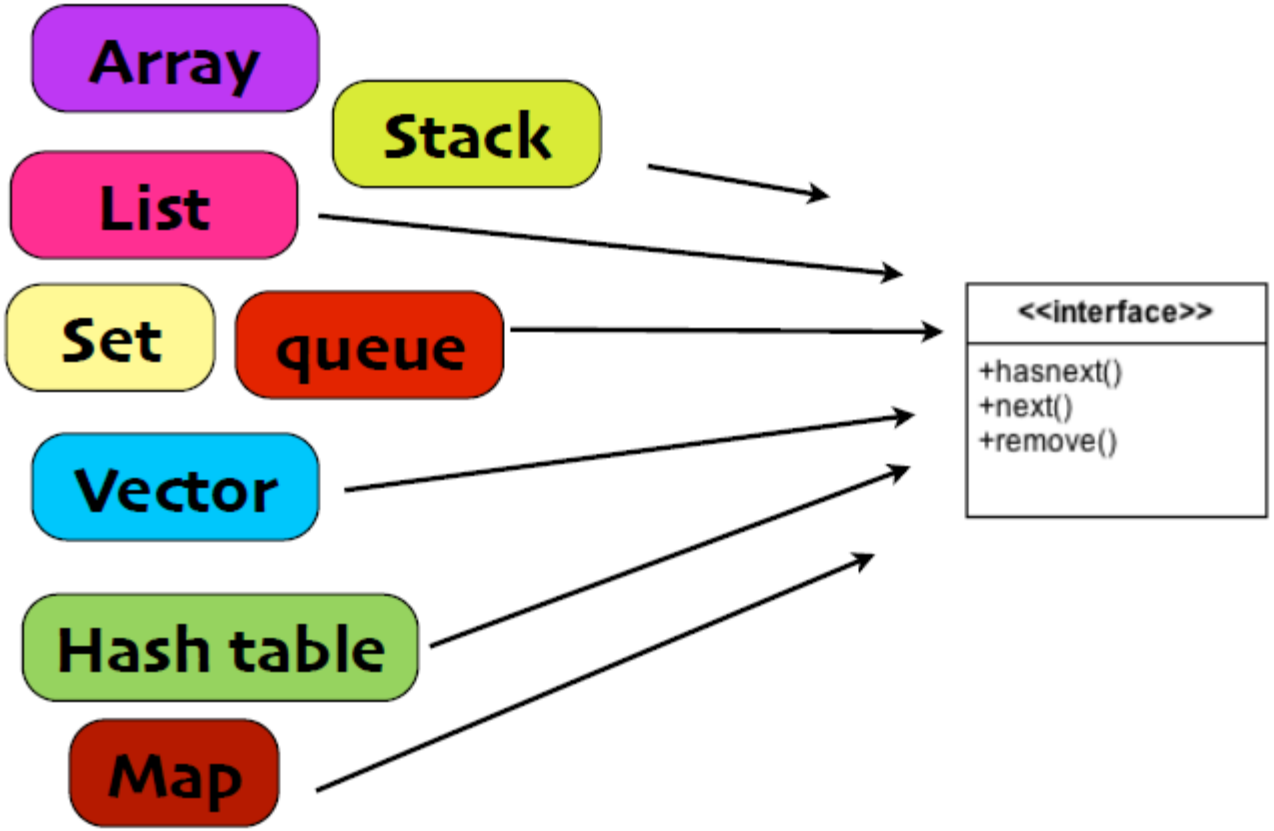
Problem


Client
FIFO
LIFO
queue
subsequent access
Stack
0
1
2
3
4
5
queue.element();
queue.add();
stack.pop();
stack.push();
different implementations
0
1
2
3
4

Ugly Way to Access



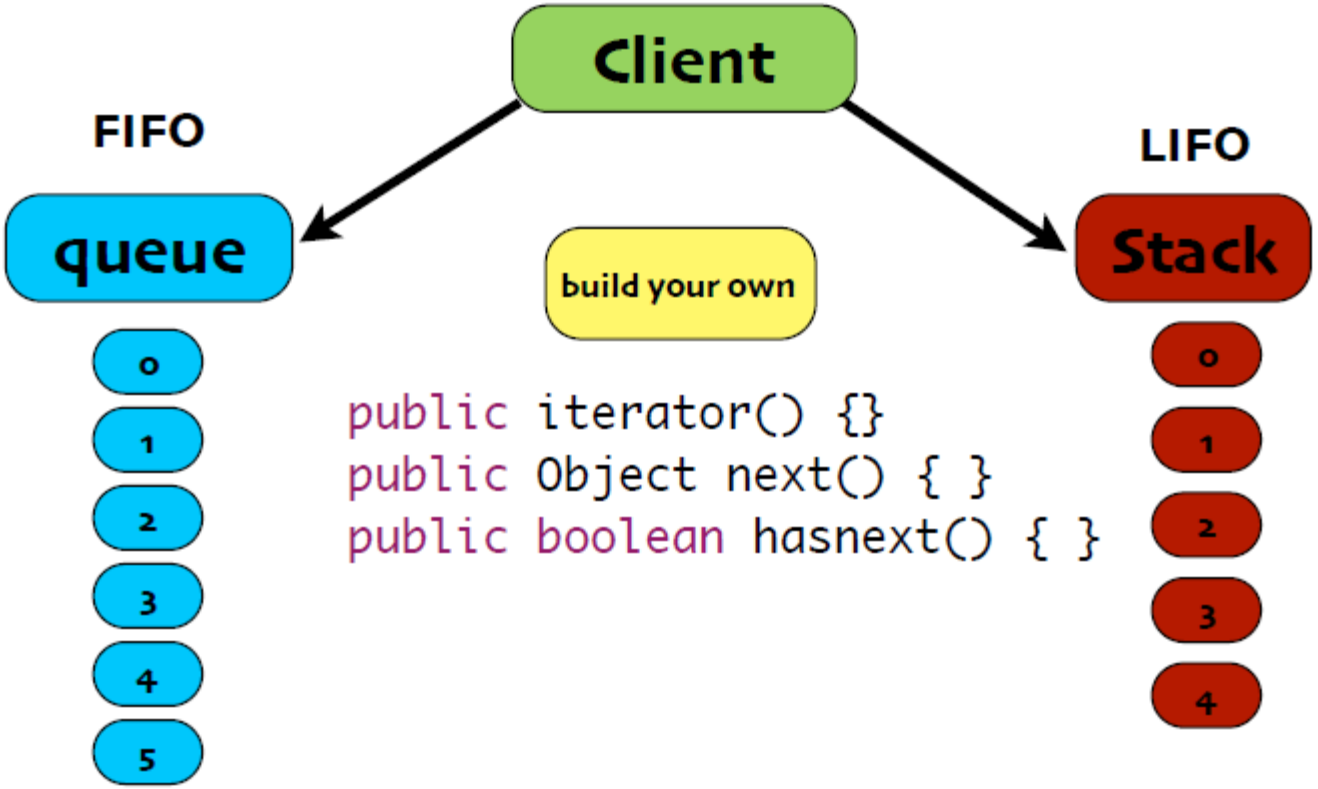
One interface?



Iterator Pattern

The Iterator Pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

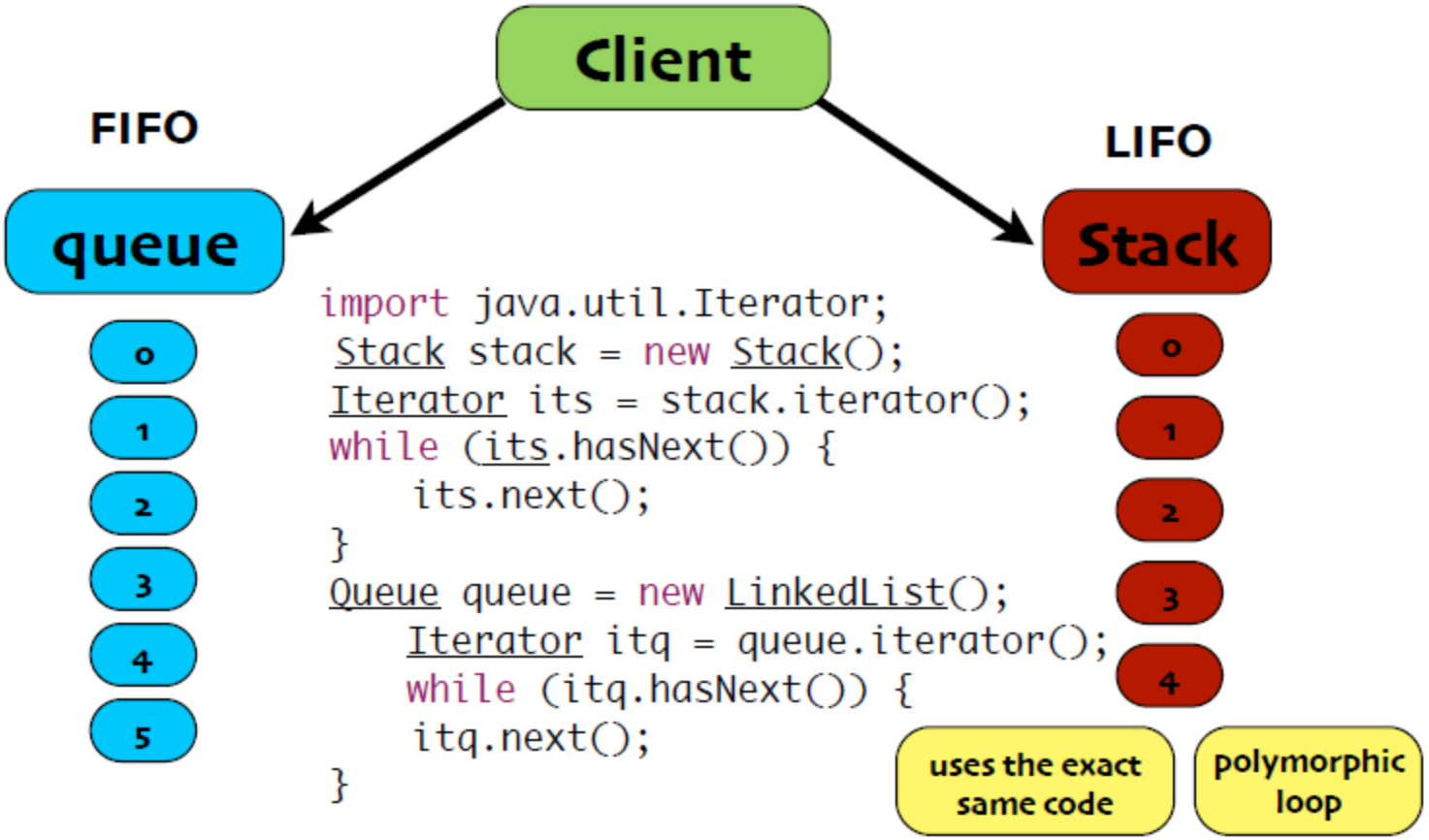
Iterator approach



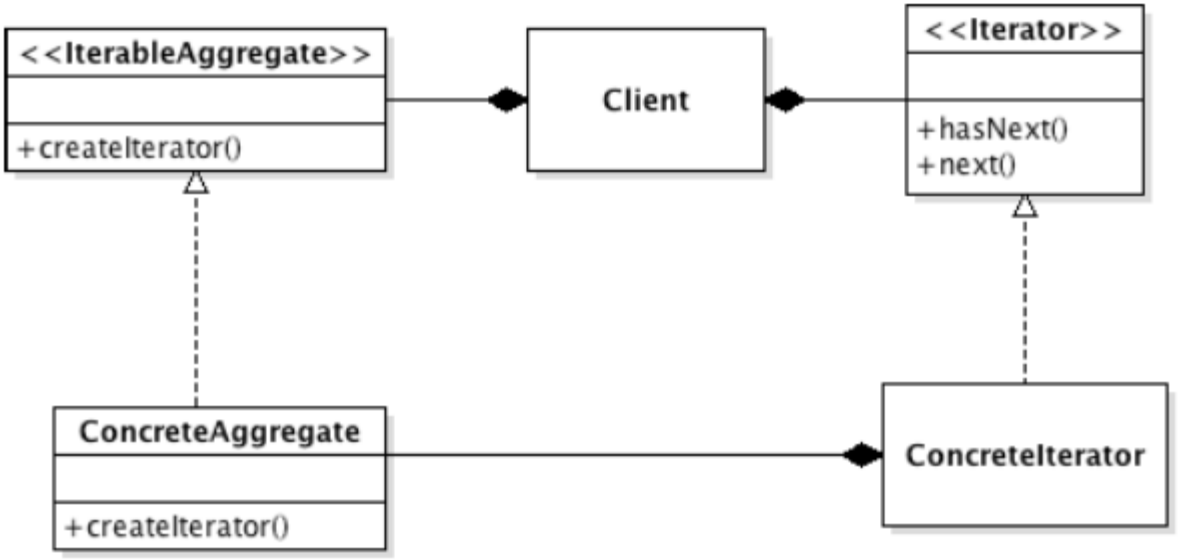
Build Your Own

```
public Interface iterator{
 boolean hasNext();
 Object next();
}
public class QueueIterator implements Iterator {

public QueueIterator(Queue q) { }

public Object next() {  }

public boolean hasNext() { }
}
```

or Use Java


Client
FIFO
queue
0
1
2
3
4
5
LIFO
Stack
0
1
2
3
4
import java.util.Iterator;
Stack stack = new Stack();
Iterator its = stack.iterator();
while (its.hasNext()) {
its.next();
}
Queue queue = new LinkedList();
Iterator itq = queue.iterator();
while (itq.hasNext()) {
itq.next();
}
uses the exact same code
polymorphic loop

Class Diagram

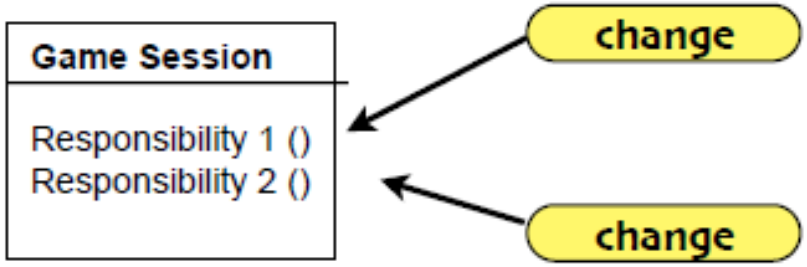


SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Design Principle

A Class should have only one reason to change

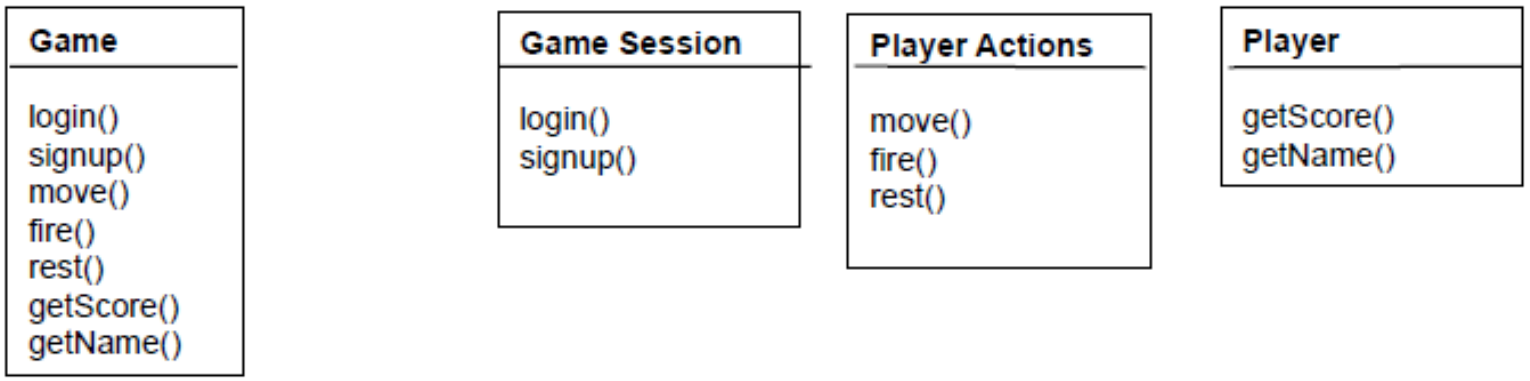


Cohesion

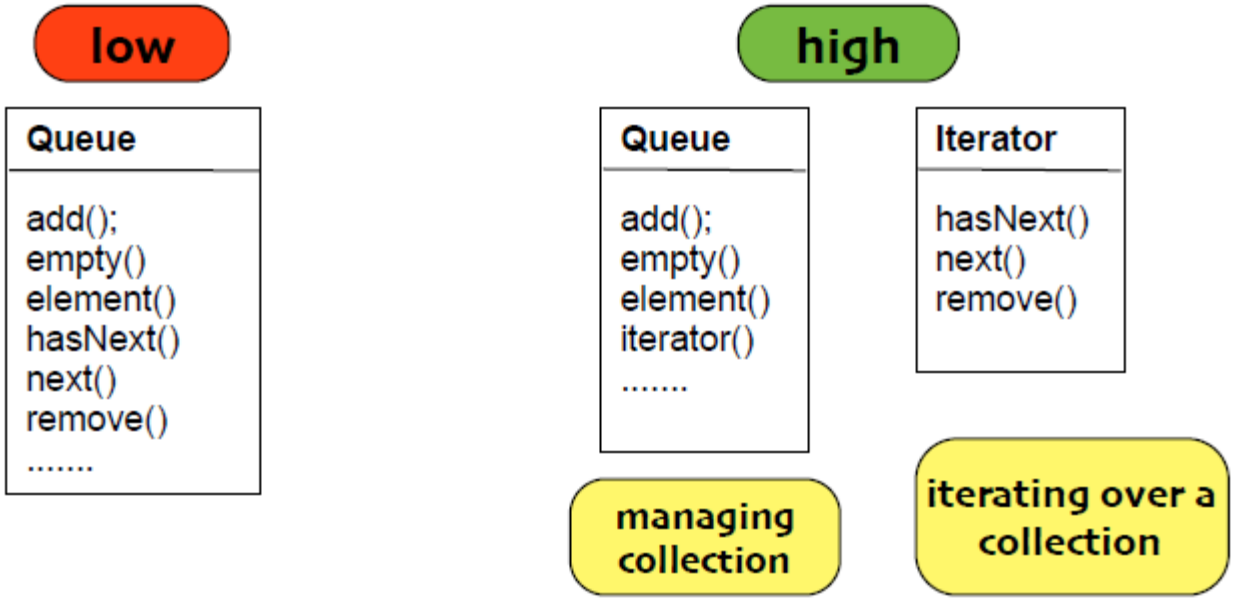
How closely a class supports a single purpose

low

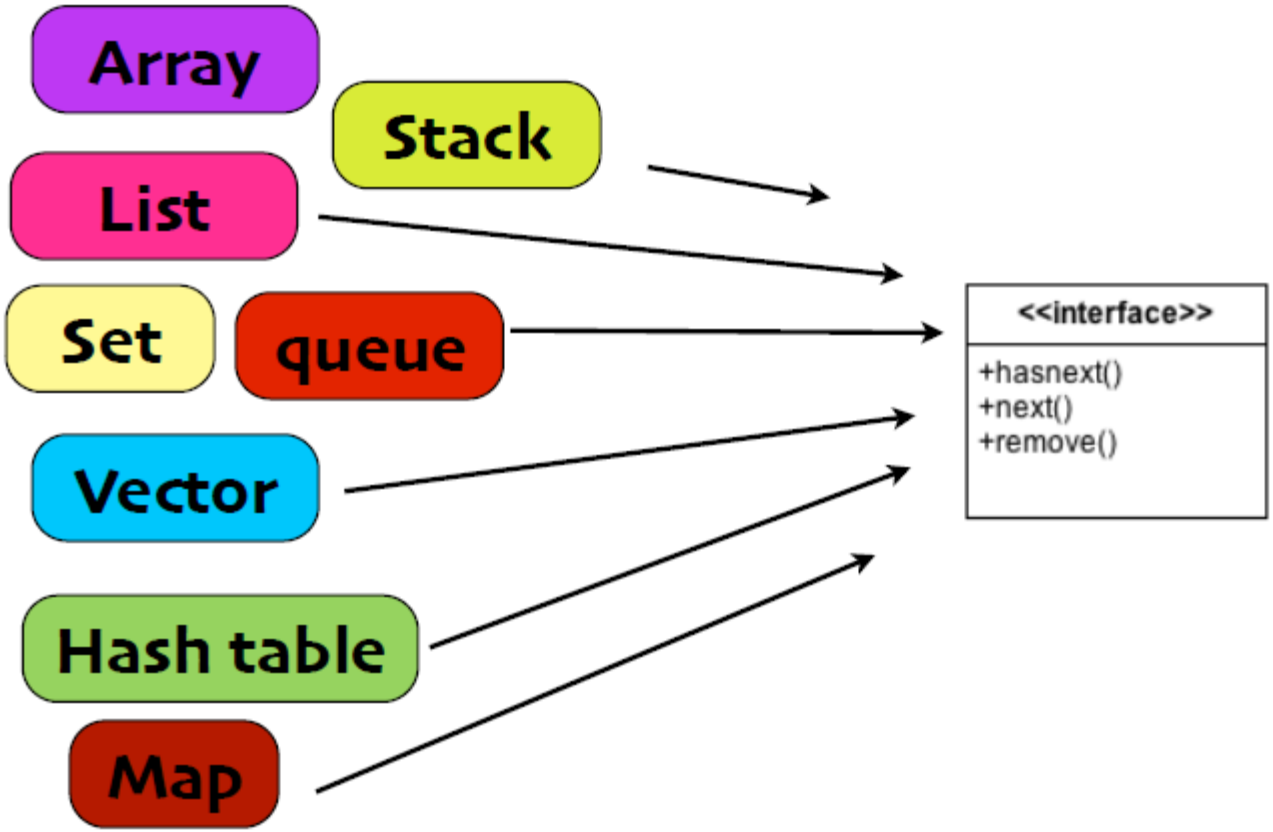
high



Class for Iterator



Collections Interface



Composite

Behavioral Patterns
- observer
- decorator
- strategy
- command
- template
- state
- iterator

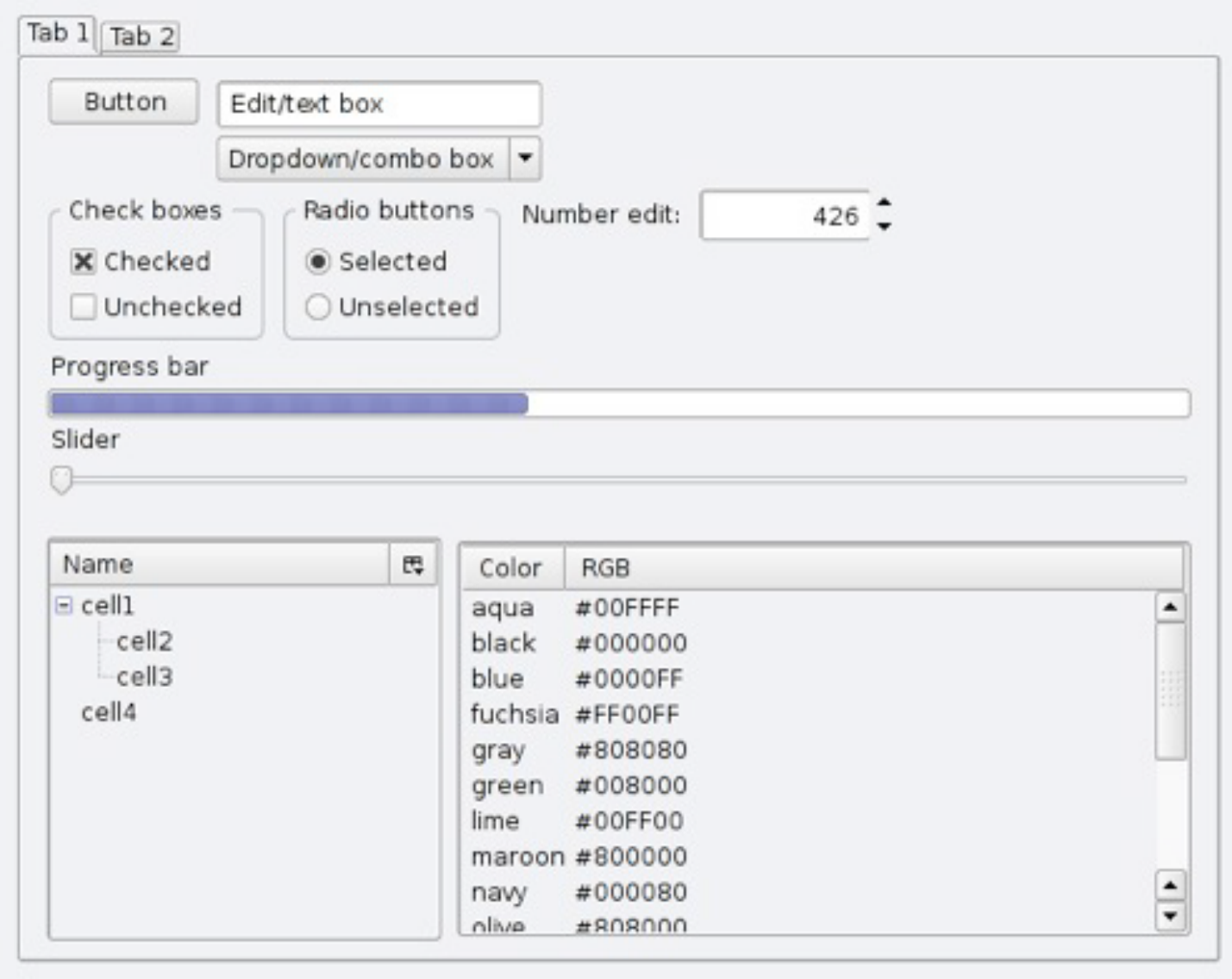
Creational Patterns
- factory method
- abstract factory
- singleton

Structural Patterns
- adapter
- façade
- **composite**

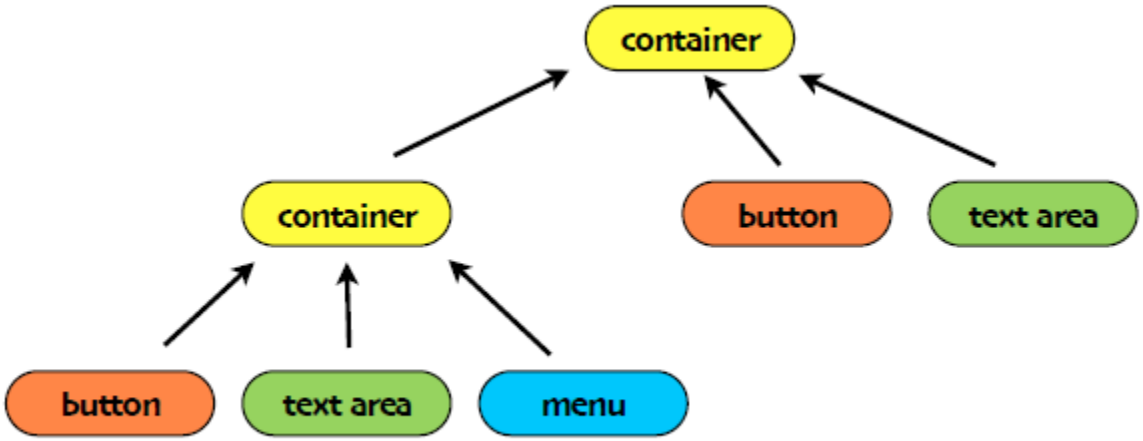
Problem

When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch. This makes code more complex, and therefore, error prone.

Example GUI's



widget


container
container
button
text area
button
text area
menu

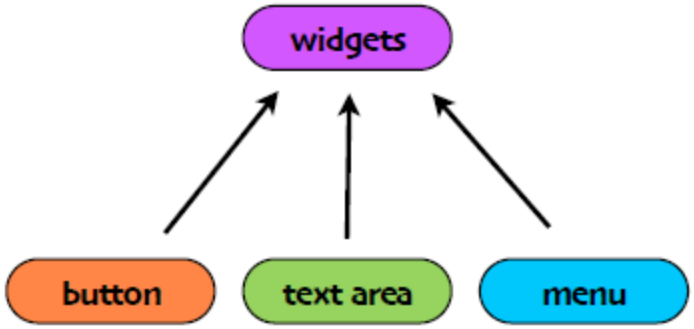
Possible implementation

pretty ugly

```
public class Window {
Button[] buttons;
Menu[] menus;
TextArea[] textAreas;
WidgetContainer[] containers;

public void update() {
if (buttons != null) {
        for (int k = 0; k < buttons.length; k++) buttons[k].draw();
    }
}
if (menus != null) for (int k = 0; k < menus.length; k++) {
        menus[k].refresh();
}
if (containers!=null) {
      for (int k = 0; k < containers.length; k++ ) {
        containers[k].updateWidgets();
}
```

"Classes should be open for extension, but closed for modification"

Abstraction



Refactor

```
public class Window {
Widget[] widgets;
WidgetContainer[] containers;


public void update() {

if (widgets != null) for (int k = 0; k < widgets.length; k++) {
            widgets[k].update();
}

if (containers != null) {
        for (int k = 0; k < containers.length; k++ ) {
            containers[k].updateWidgets();
        }
 }
```

"program to an interface"

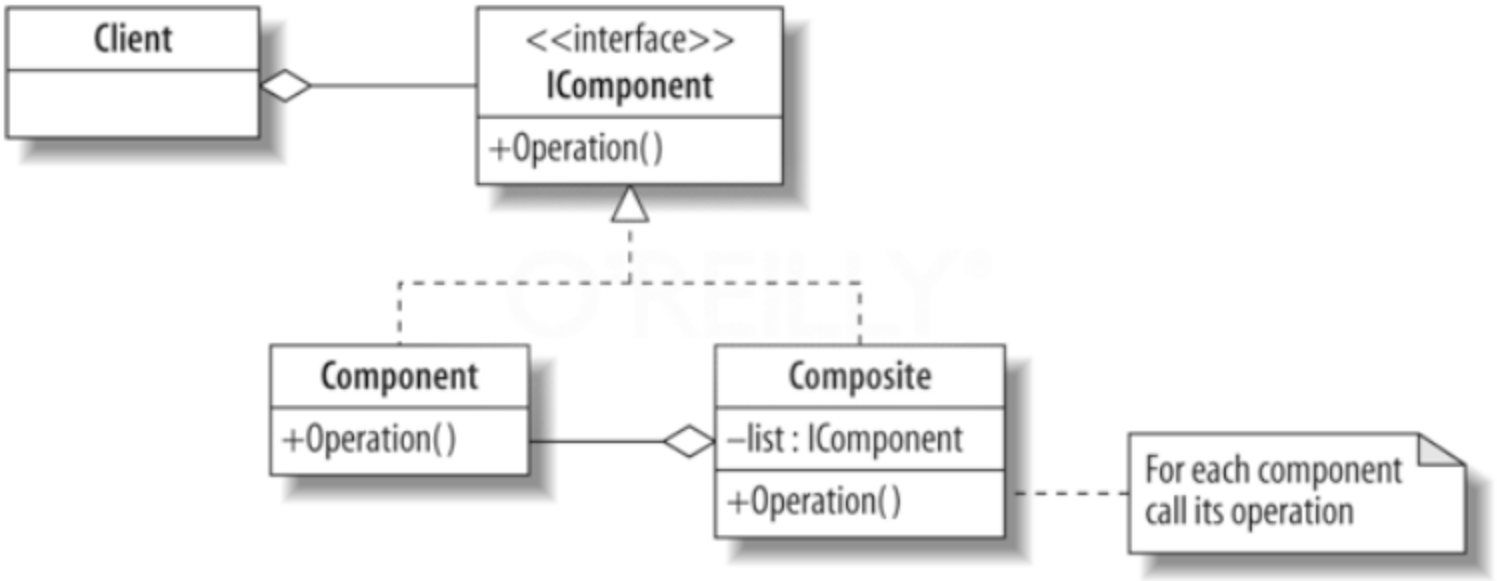
all widgets support update()

we still distinguish between containers and widgets

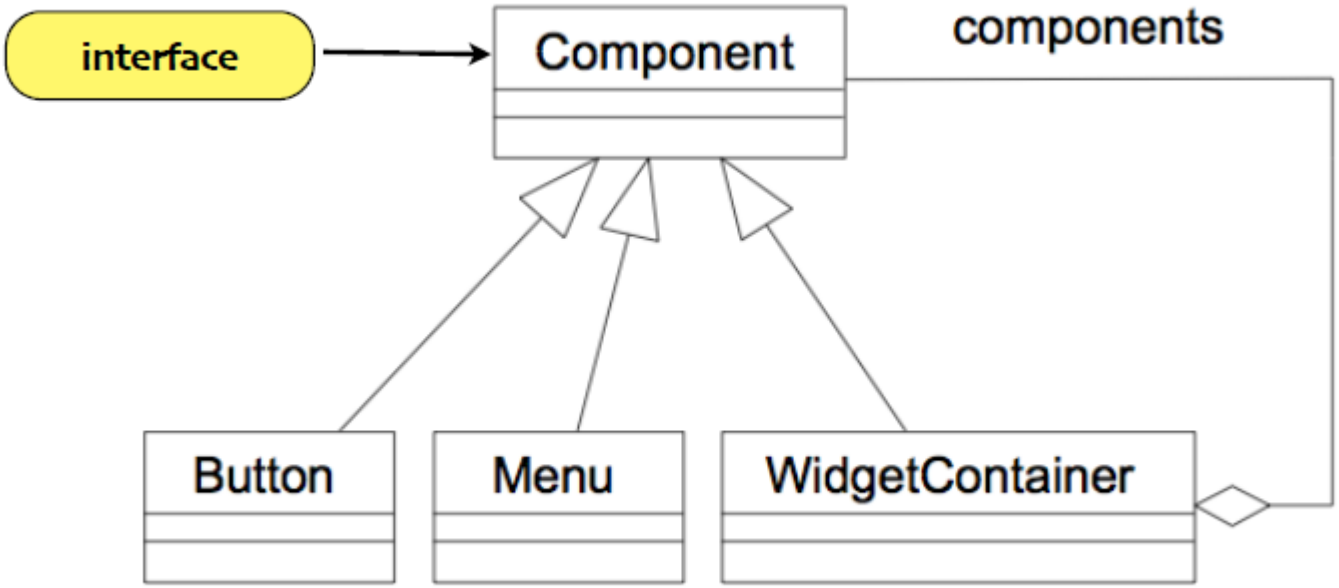
Composite Pattern

The Composite Pattern allows you to compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly

Class Diagram



for our GUI

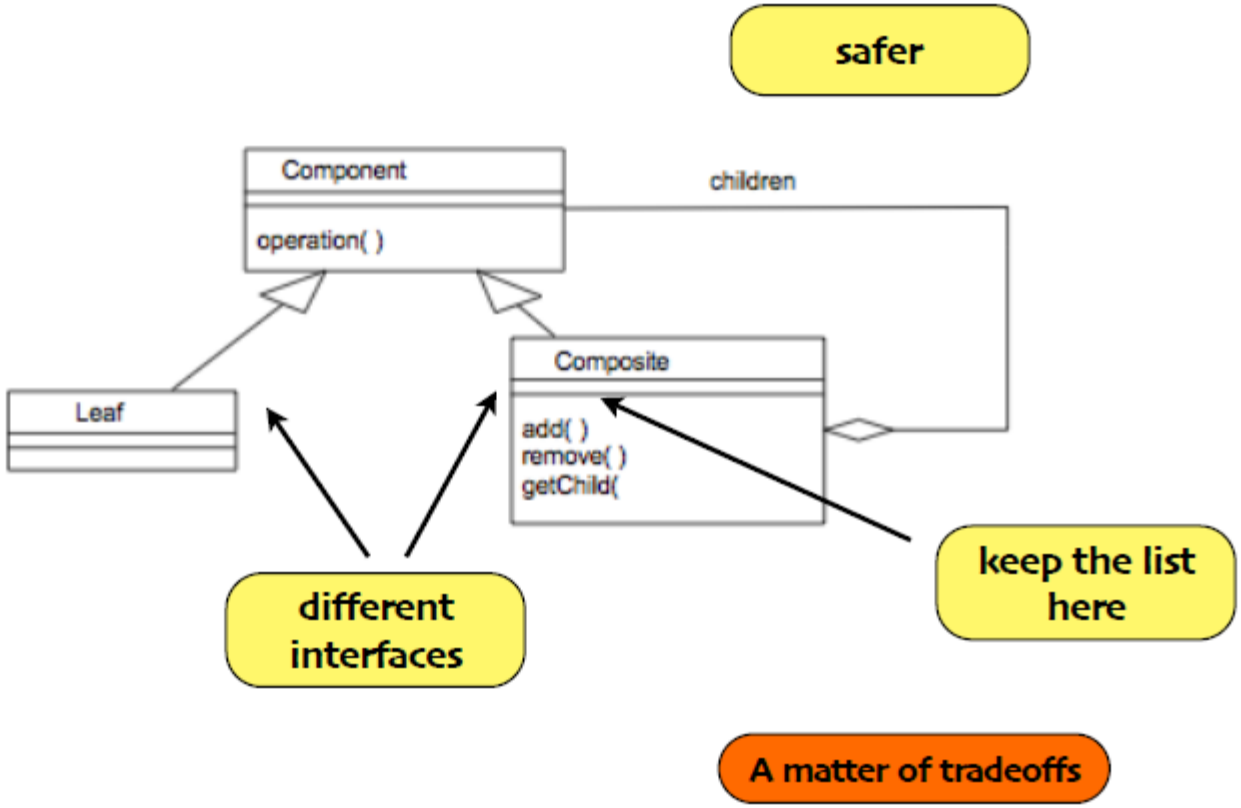



```
public class Window {
    Component[] components;

public void update() {
    if (components != null) {
            for (int k = 0; k < components.length; k++) {
             components[k].update();
        }
    }
}
```

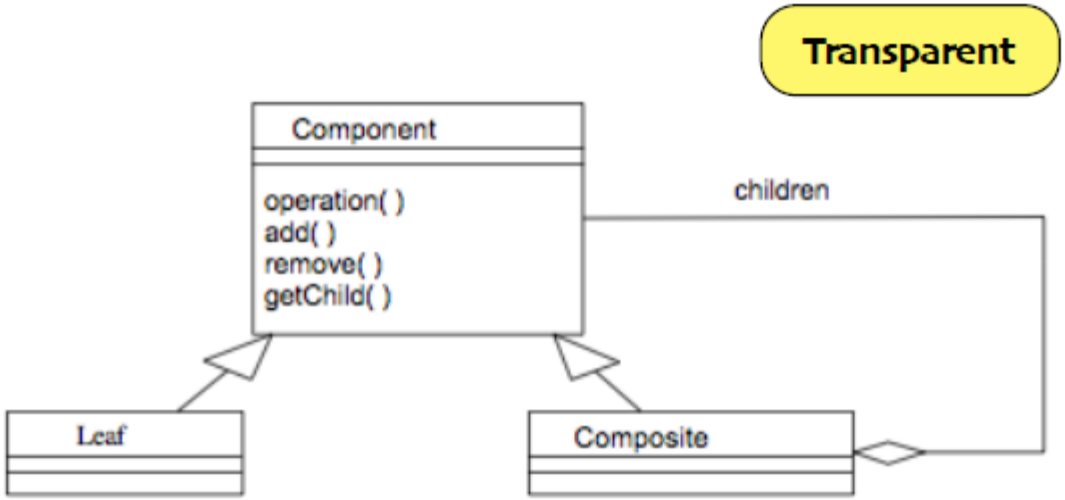
Implementation issues

Where should the child management methods (add(), remove(), getChild()) be declared?

In Composite?



In Component?



Internal Iterator : see Menu Package

```
public void doSomething() {
 throw new UnsupportedOperationException();
}
```

composite

ugly but safe

```
public void doSomething() {
// do something
}
```

leaf

```
public void doSomething() {
// do something
Iterator iterator = menuComponents.iterator();
while (iterator.hasNext()) {
  Component component = (Component) iterator.next();
  component.doSomething();
  }
}
```

Composite

External Iterator: see Menulterator package