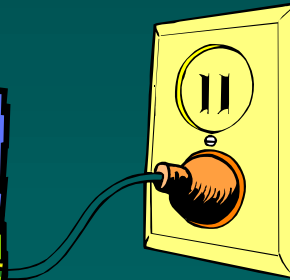
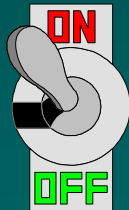


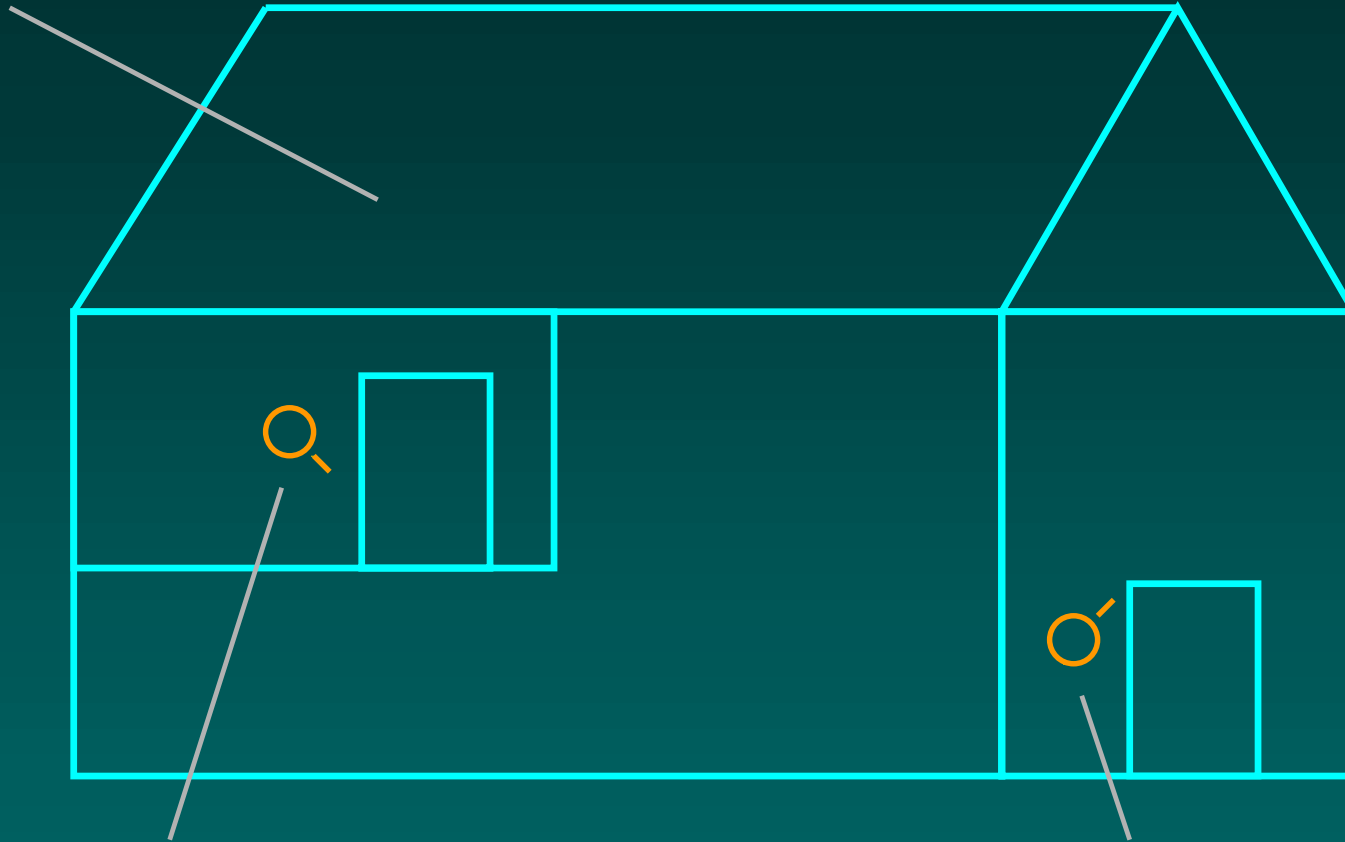
## Quick Introduction to objects and components

- Classes and Objects
- Plugging together objects and Interfaces
- Component Assembly



## Object: identity, state, and behavior

Object “myHouse” is an instance of class House

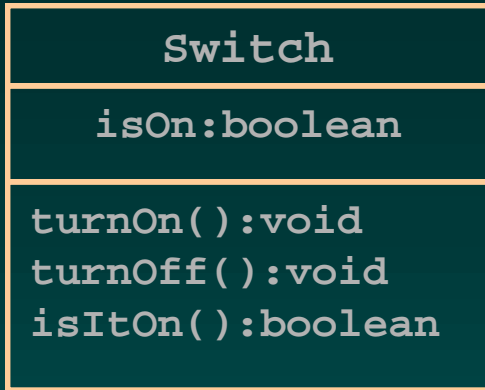


Object “bedroomSwitch” is an **instance** of Switch

Another instance of Switch

**Identity:** bedroomSwitch  
**state:** off  
**behavior:** turnOn, turnOff

**Identity:** hallwaySwitch  
**state:** on  
**behavior:** turnOn, turnOff



Class (UML)

```
public class Switch {  
    private boolean isOn = false;  
    public void turnOn() { isOn = true;}  
    public void turnOff() { isOn = false;}  
    public boolean isItOn() {  
        return isOn;  
    }  
}
```

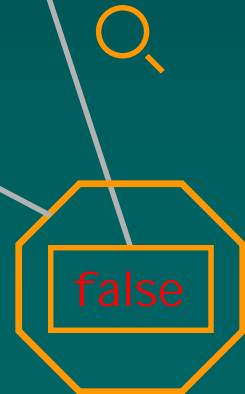
Class (Java)

Object reference

New Object is created

```
...  
Switch bedroomSwitch = new Switch();  
...
```

object creation (Java)



object in memory



Actual Switch



Abstraction of a Switch

**Abstraction**  
Emphasize what is relevant, and de-emphasize what is irrelevant

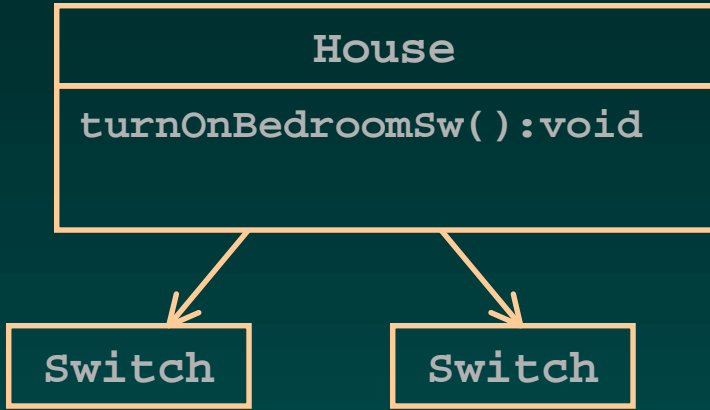
**Why do we abstract?**  
We use abstraction to manage complexity

**Problem Dependent**  
What is a good abstraction for one problem domain, may not be a good abstraction for another problem domain

```
Switch
isOn:boolean

turnOn():void
turnOff():void
isItOn():boolean
```

## UML class diagram



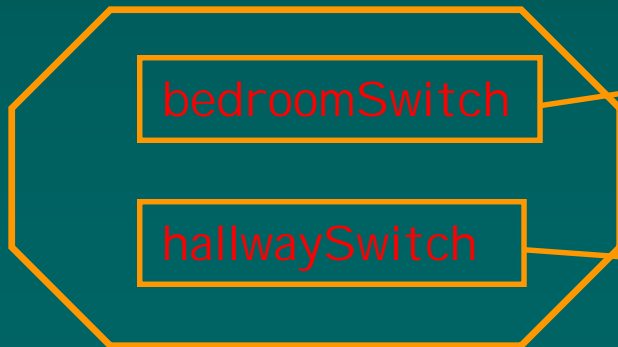
## Class (Java)

```
public class House {  
    private Switch bedroomSwitch = new Switch();  
    private Switch hallWaySwitch = new Switch();  
    public void turnOnBedroomSw() {  
        bedroomSwitch.turnOn();  
    }  
}
```

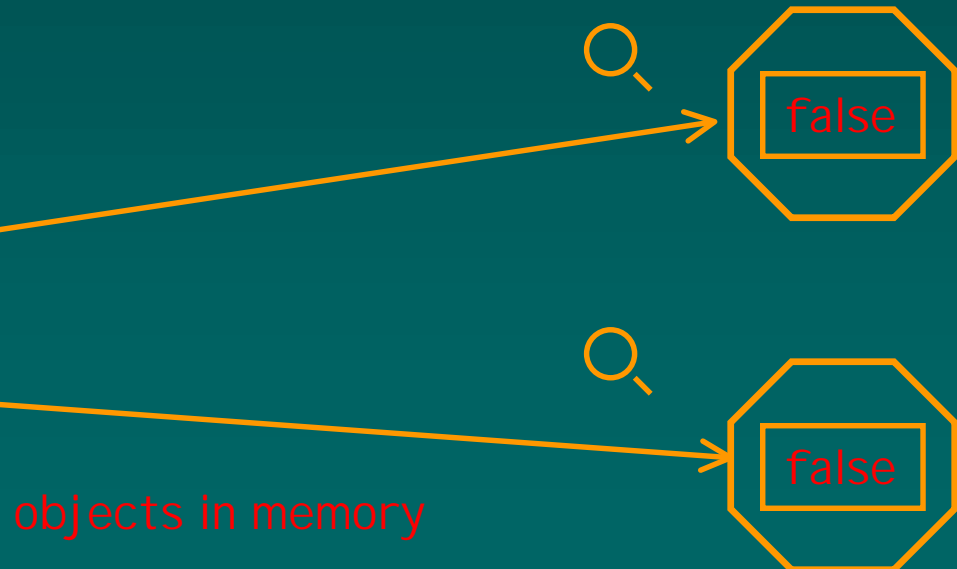
## Object "myHouse" creation

```
House myHouse = new House();
```

### Object of type House



### Objects of type Switch

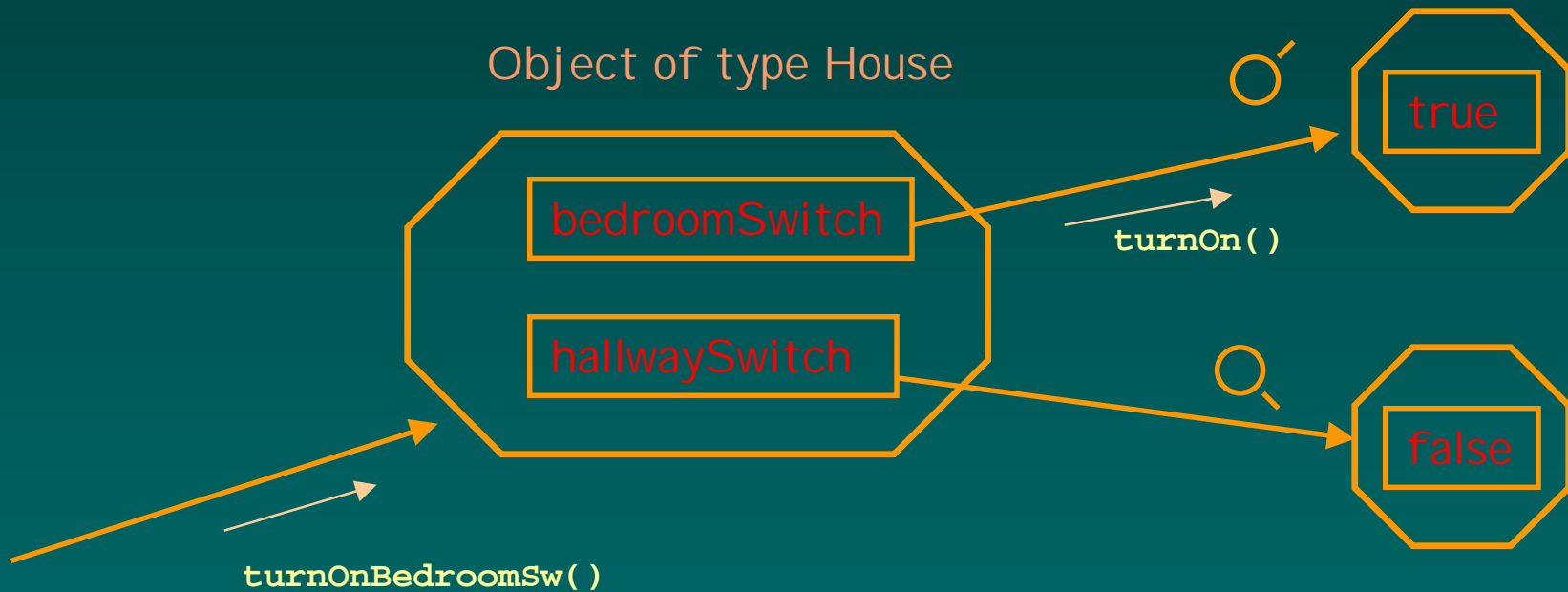


objects in memory

## Message invocation (Java)

```
House myHouse = new House();  
myHouse.turnOnBedroomSw();  
...
```

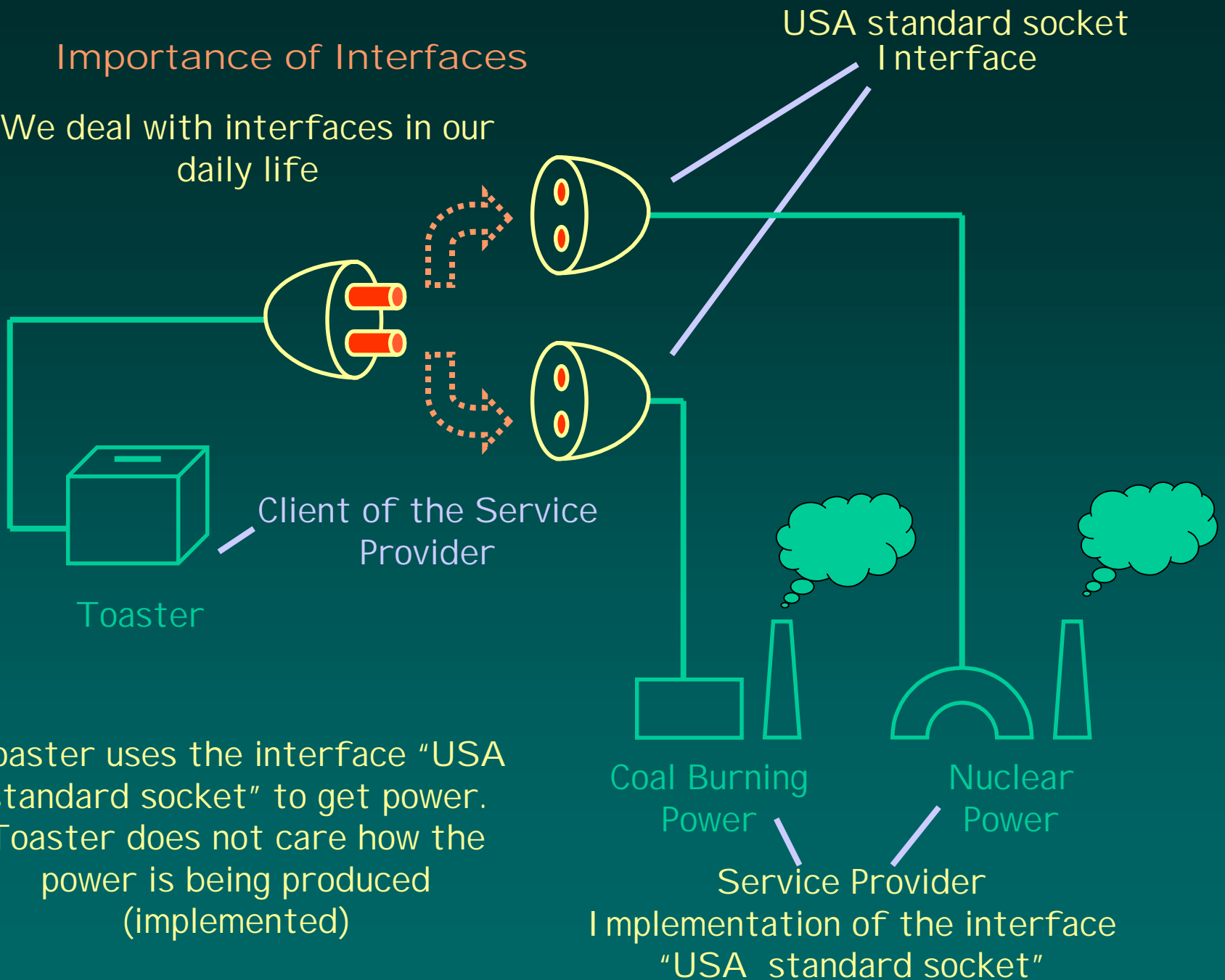
Objects of type Switch



Object invoking messages among each other

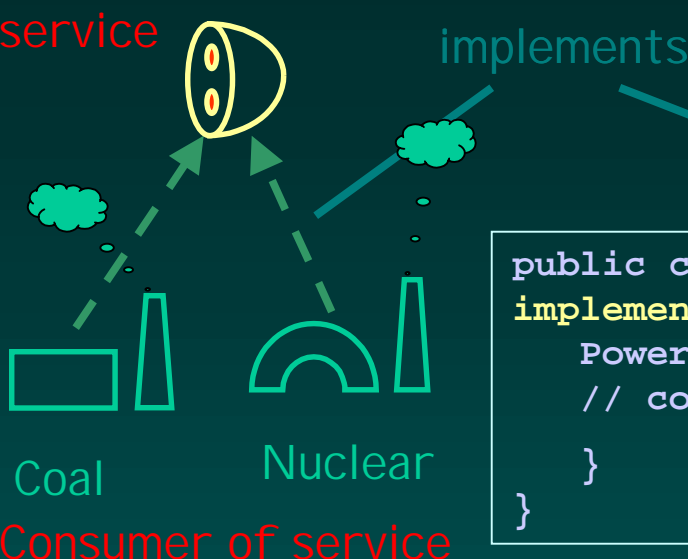
# Importance of Interfaces

We deal with interfaces in our daily life



Toaster uses the interface "USA standard socket" to get power. Toaster does not care how the power is being produced (implemented)

Coal Burning Power  
Nuclear Power  
Service Provider  
Implementation of the interface  
"USA standard socket"



implements

```
public interface USAsocket {
    Power getPower();
}
```

```
public class CoalPower
implements USAsocket{
    Power getPower(){
        // coal implementation
    }
}
```

```
public class NuclearPower
implements USAsocket {
    Power getPower(){
        // nuclear implementation
    }
}
```

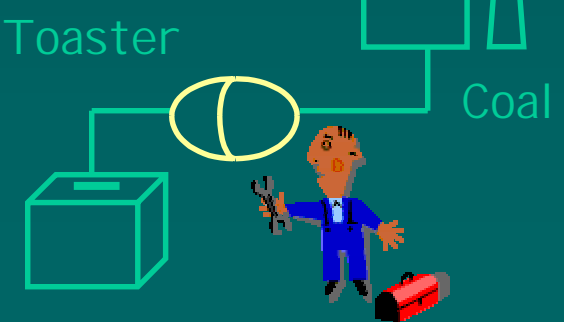
Toaster knows nothing about coal or nuclear power



Toaster

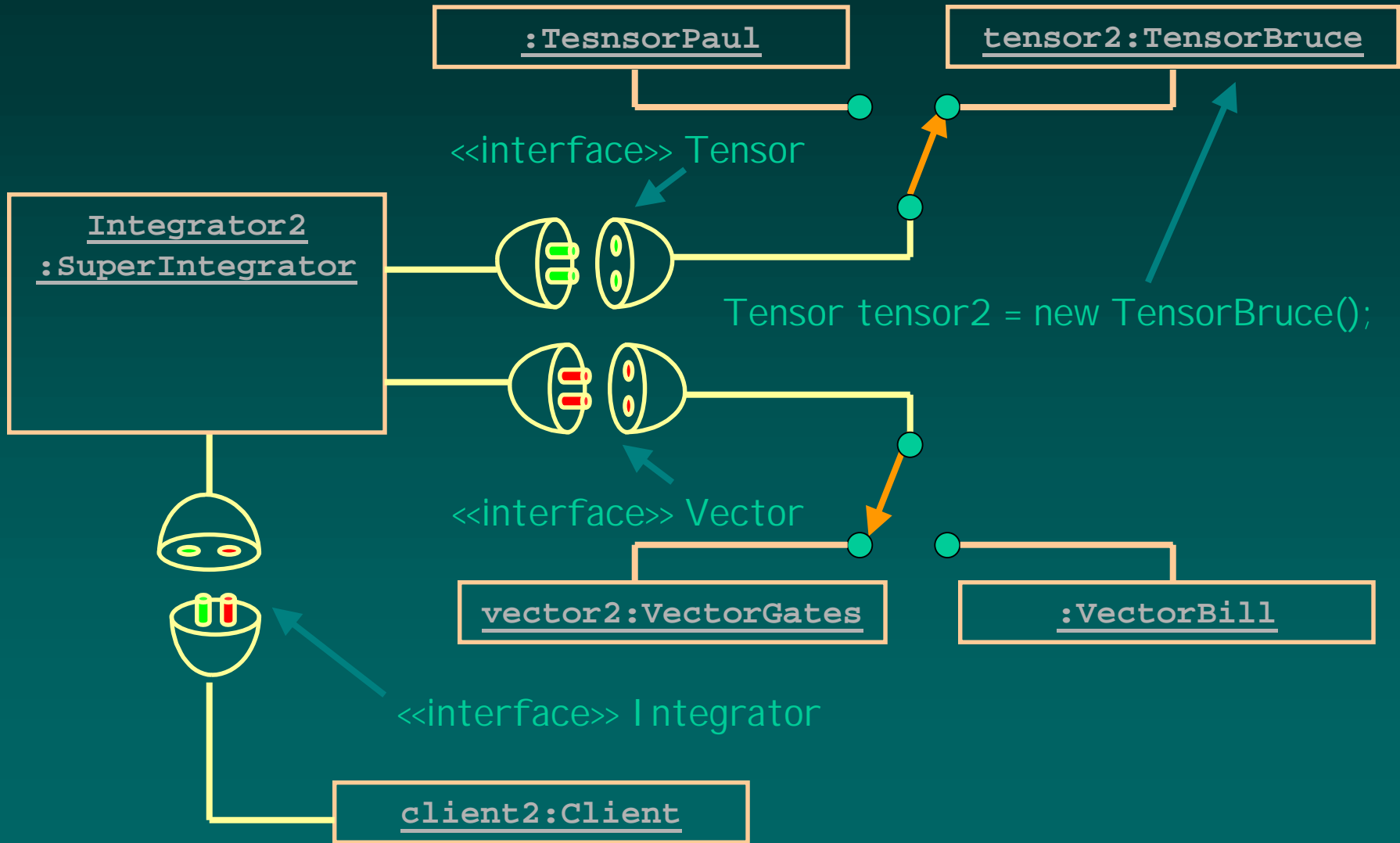
```
public class Toaster {
    private USAsocket mySocket = null;
    public void plugIn(USAsocket soc) {
        mySocket = soc;
    }
}
```

Assembly



```
public class ToasterUser {
    // construct a new Coal Power plant
    USAsocket mySocket = new CoalPower();
    // buys a new Toaster
    Toaster myToaster = new Toaster();
    // Plugs the Toaster into the socket
    myToaster.plugIn(mySocket);
    ...
}
```

# Component Assembly



Components are reusable  
only within the same  
Architectural Vision

