➢ Structural Pattern

1. Behavioral patterns deal with how class and objects are composed and simplify by identifying relationship
2. Behavioral pattern concerned with how classes are inherited each other

➢ Adapter Pattern

1. Adapter pattern used to convert on interface of class to another interface that client wants
2. Adapter pattern is also called as Wrapper
3. In adapter pattern we have 4 components
   a. Target interface
   b. Adoptee interface
   c. Adapter
   d. client

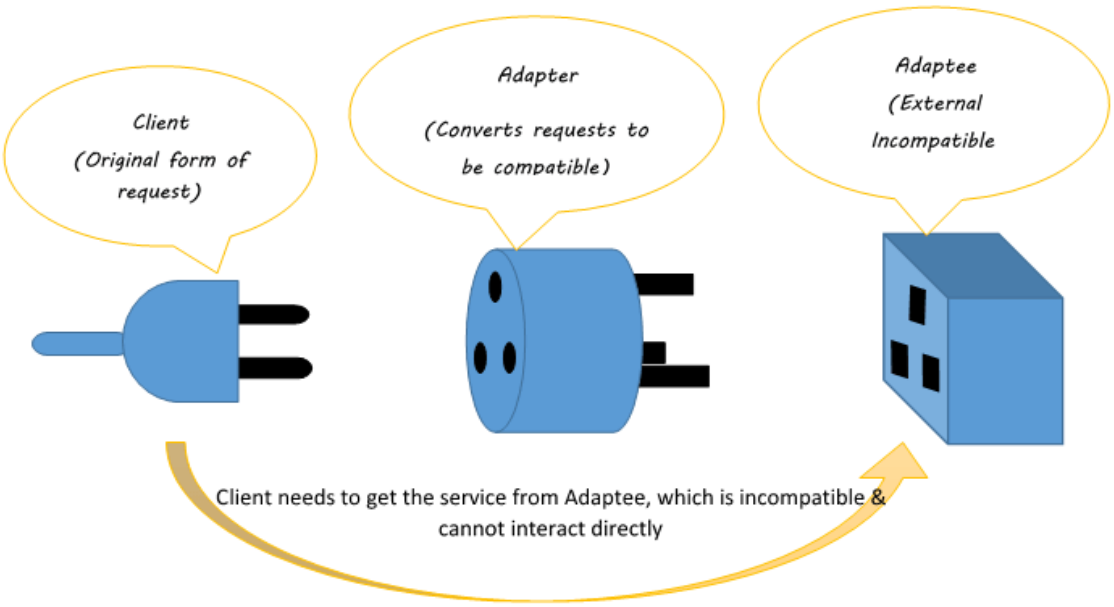


Figure 1-Adapter Pattern Concept

4. **Example**

```
/** target interface */
interface GermanPlug {
    fun provideElectricity() : String
}
```

```
class GermanSockets : GermanPlug{
    override fun provideElectricity() : String{
        return "German Elctricity"
    }
}
```

```kotlin
/** Adaptee interface */
interface UkPlug {
    fun provideElectricity() : String
}

class UKSockets : UkPlug{
    override fun provideElectricity() : String{
        return "UK Electricity"
    }
}
```

```kotlin
/** Adapter converts adaptee to target*/
class UkToGermanPlugConvertorAdapter : UkPlug {
    lateinit var germanPlug: GermanPlug
    constructor(germanPlug: GermanPlug){
        this.germanPlug = germanPlug
    }

    override fun provideElectricity() : String {
        return germanPlug.provideElectricity()
    }
}
```

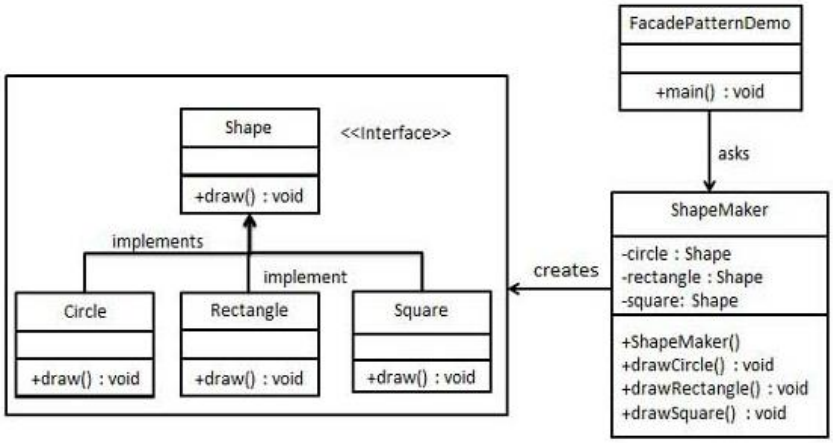
```kotlin
/** Adapter Patterns
 * Client*/
val germanPlug : GermanPlug = GermanSockets()
Log.d(ADAPTER_PATTERN, germanPlug.provideElectricity())

val ukPlug : UkPlug = UKSockets()
Log.d(ADAPTER_PATTERN, ukPlug.provideElectricity())

val adapter : UkPlug = UkToGermanPlugConvertorAdapter(germanPlug)
Log.d(ADAPTER_PATTERN, adapter.provideElectricity())
```

➢ Facade Pattern

1. As the name indicates façade hides how the complex implementation is done
2. In façade pattern client interact with only with façade class which gives required objects to client
3. Façade deals with only interface no implementation
4. Façade pattern used when complex system needs to be hidden to client




```
interface IShape {
    fun draw() : String
}
```

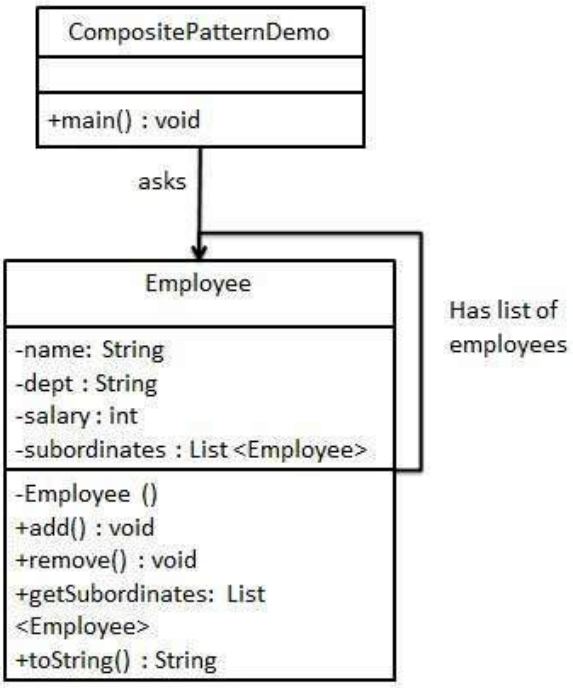
```
class Circle : IShape {
    override fun draw(): String {
        return "Circle"
    }
}
```

```
class Rectangle : IShape {
    override fun draw() : String {
        return "Rectangle"
    }
}
```

```
class ShapeMaker {
    private lateinit var rectangle : IShape
    private lateinit var circle: IShape
    constructor(){
        rectangle = Rectangle()
        circle = Circle()
    }
    fun drawCircle() : String{
        return circle.draw()
    }
    fun drawRectangle() : String{
        return rectangle.draw()
    }
}
/**Facade Patters
* Client*/
val shapeMaker = ShapeMaker()
Log.d(FACADE_PATTERN, shapeMaker.drawRectangle())
Log.d(FACADE_PATTERN, shapeMaker.drawCircle())
```

➢ Composite Pattern

1. In Composite pattern group of objects are treated as similar way as single objects
2. Composite pattern used in hierarchical objects structure using tree representation




```
class Employee(var name : String, var id : Int, var designation : String) {
    private var list = ArrayList<Employee>()
    fun add(employee: Employee){
        list.add(employee)
    }
    fun remove(employee: Employee){
        list.remove(employee)
    }
    fun getEmployees() : ArrayList<Employee>{
        return list
    }

    override fun toString(): String {
        return "EMPLOYEE : [name = '$name', " +
                "id = '$id', " +
                "designation = '$designation']"
    }
}
```

```
 * client*/                                                    ⚠2 ⚠2
val ceo = Employee( name: "Ravi", id: 1, designation: "CEO")
val manager1 = Employee( name: "Rupa",  id: 2,  designation: "Manager")
val manager2 = Employee( name: "CV",  id: 3,  designation: "Manager")
ceo.add(manager1)
ceo.add(manager2)

val supervisor = Employee( name: "Manju",  id: 4,  designation: "Supervisor")
manager1.add(supervisor)

val softwareEngg1 = Employee( name: "Gautem",  id: 5,  designation: "Software Engineer")
val softwareEngg2 = Employee( name: "Priya",  id: 6,  designation: "Software Engineer")
supervisor.add(softwareEngg1)
supervisor.add(softwareEngg2)

Log.d(COMPOSITE_PATTERN, ceo.toString())
for (managers in ceo.getEmployees()){
    Log.d(COMPOSITE_PATTERN, managers.toString())
    for(sup in managers.getEmployees()){
        Log.d(COMPOSITE_PATTERN, sup.toString())
        for (sw in sup.getEmployees()){
            Log.d(COMPOSITE_PATTERN, sw.toString())
        }
    }
}
```

```
EMPLOYEE : [name = 'Ravi', id = '1', designation = 'CEO']
EMPLOYEE : [name = 'Rupa', id = '2', designation = 'Manager']
EMPLOYEE : [name = 'Manju', id = '4', designation = 'Supervisor']
EMPLOYEE : [name = 'Gautem', id = '5', designation = 'Software Engineer']
EMPLOYEE : [name = 'Priya', id = '6', designation = 'Software Engineer']
EMPLOYEE : [name = 'CV', id = '3', designation = 'Manager']
```