Certainly! Let's explore each of these structural design patterns in detail, including their concepts, real-time examples, and code snippets.

1. Adapter Pattern
Concept: The Adapter Pattern allows incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, enabling them to communicate.

Real-Time Example: A power adapter that allows a two-prong plug to fit into a three-prong socket.

Code Example:

Java

Collapse

```
// Existing class
class RoundPeg {
    private double radius;

    public RoundPeg(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
         return radius;
     }
}

// New class with incompatible interface
class SquarePeg {
    private double width;

    public SquarePeg(double width) {
        this.width = width;
    }

    public double getWidth() {
        return width;
    }
}

// Adapter class
class SquarePegAdapter extends RoundPeg {
    private SquarePeg squarePeg;

    public SquarePegAdapter(SquarePeg squarePeg) {
        super(squarePeg.getWidth() * Math.sqrt(2) / 2);
        this.squarePeg = squarePeg;
    }
}

// Client code
public class AdapterPatternExample {
    public static void main(String[] args) {
        RoundPeg roundPeg = new RoundPeg(5);
        SquarePeg squarePeg = new SquarePeg(5);
        SquarePegAdapter adapter = new SquarePegAdapter(squarePeg);

        System.out.println("Round peg radius: " + roundPeg.getRadius());
        System.out.println("Square peg adapted radius: " + adapter.getRadius());
    }
}
```

2. Bridge Pattern
Concept: The Bridge Pattern decouples an abstraction from its implementation, allowing them to vary independently. It is used to separate the abstraction from the implementation.

Real-Time Example: A remote control (abstraction) that can work with different types of devices (implementations) like TV, DVD player, etc.

Code Example:

Java

```
 Collapse
1// Implementor
2interface Device {
3    void turnOn();
4    void turnOff();
5}
6
7// Concrete Implementor
8class TV implements Device {
9    @Override
10    public void turnOn() {
11        System.out.println("TV is on");
12    }
13
14    @Override
15    public void turnOff() {
16        System.out.println("TV is off");
17    }
18}
19
20// Abstraction
21abstract class RemoteControl {
22    protected Device device;
23
24    public RemoteControl(Device device) {
25        this.device = device;
26    }
27
28    public abstract void togglePower();
29}
30
31// Refined Abstraction
32class BasicRemoteControl extends RemoteControl {
33    public BasicRemoteControl(Device device) {
34        super(device);
35    }
36
37    @Override
38    public void togglePower() {
39        System.out.println("Toggling power");
40        device.turnOn();
41        device.turnOff();
42    }
43}
44
45// Client code
46public class BridgePatternExample {
47    public static void main(String[] args) {
48        Device tv = new TV();
49        RemoteControl remote = new BasicRemoteControl(tv);
50        remote.togglePower();
51    }
52}
```

3. Composite Pattern

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

Real-Time Example: A file system where files and directories are treated uniformly.

Code Example:

Java

```
 Collapse
1import java.util.ArrayList;
```

```
2import java.util.List;
3
4// Component
5interface FileSystemComponent {
6    void showDetails();
7}
8
9// Leaf
10class File implements FileSystemComponent {
11    private String name;
12
13    public File(String name) {
14        this.name = name;
15    }
16
17    @Override
18    public void showDetails() {
19        System.out.println("File: " + name);
20    }
21}
22
23// Composite
24class Directory implements FileSystemComponent {
25    private String name;
26    private List<FileSystemComponent> components = new ArrayList<>();
27
28    public Directory(String name) {
29        this.name = name;
30    }
31
32    public void addComponent(FileSystemComponent component) {
33        components.add(component);
34    }
35
36    @Override
37    public void showDetails() {
38        System.out.println("Directory: " + name);
39        for (FileSystemComponent component : components) {
40            component.showDetails();
41        }
42    }
43}
44
45// Client code
46public class CompositePatternExample {
47    public static void main(String[] args) {
48        File file1 = new File("file1.txt");
49        File file2 = new File("file2.txt");
50
51        Directory directory = new Directory("Documents");
52        directory.addComponent(file1);
53        directory.addComponent(file2);
54
55        directory.showDetails();
56    }
57}
```

4. Decorator Pattern

Concept: The Decorator Pattern allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

Real-Time Example: Adding features to a basic coffee order, like milk, sugar, or whipped cream.

Code Example:

Java

Collapse

```
1// Component
```

```
2interface Coffee {
3    String getDescription();
4    double getCost();
5}
6
7// Concrete Component
8class BasicCoffee implements Coffee {
9    @Override
10    public String getDescription() {
11        return "Basic Coffee";
12    }
13
14    @Override
15    public double getCost() {
16        return 2.0;
17    }
18}
19
20// Decorator
21abstract class CoffeeDecorator implements Coffee {
22    protected Coffee coffee;
23
24    public CoffeeDecorator(Coffee coffee) {
25        this.coffee = coffee;
26    }
27
28    @Override
29    public String getDescription() {
30        return coffee.getDescription();
31    }
32
33    @Override
34    public double getCost() {
35        return coffee.getCost();
36    }
37}
38
39// Concrete Decorator
40class MilkDecorator extends CoffeeDecorator {
41    public MilkDecorator(Coffee coffee) {
42        super(coffee);
43    }
44
45    @Override
46    public String getDescription() {
47        return coffee.getDescription() + ", Milk";
48    }
49
50    @Override
51    public double getCost() {
52        return coffee.getCost() + 0.5;
53    }
54}
55
56// Client code
57public class DecoratorPatternExample {
58    public static void main(String[] args) {
59        Coffee coffee = new BasicCoffee();
60        System.out.println(coffee.getDescription() + " $" + coffee.getCost());
61
62        Coffee milkCoffee = new MilkDecorator(coffee);
63        System.out.println(milkCoffee.getDescription() + " $" + milkCoffee.getCost());
64    }
65}
```

5. Facade Pattern

Concept: The Facade Pattern provides a simplified interface to a complex subsystem. It hides the complexities of the system and provides an interface to the client from where the client can access the system.

Real-Time Example: A home theater system where a single remote control (facade) controls

multiple devices like TV, sound system, and DVD player.

Code Example:

Java

Collapse

```
// Subsystem classes
class TV {
    public void on() {
        System.out.println("TV is on");
    }

    public void off() {
        System.out.println("TV is off");
    }
}

class SoundSystem {
    public void on() {
        System.out.println("Sound system is on");
    }

    public void off() {
        System.out.println("Sound system is off");
    }
}

class DVDPlayer {
    public void on() {
        System.out.println("DVD player is on");
    }

    public void off() {
        System.out.println("DVD player is off");
    }
}

// Facade
class HomeTheaterFacade {
    private TV tv;
    private SoundSystem soundSystem;
    private DVDPlayer dvdPlayer;

    public HomeTheaterFacade(TV tv, SoundSystem soundSystem, DVDPlayer dvdPlayer) {
        this.tv = tv;
        this.soundSystem = soundSystem;
        this.dvdPlayer = dvdPlayer;
    }

    public void watchMovie() {
        tv.on();
        soundSystem.on();
        dvdPlayer.on();
        System.out.println("Movie is playing");
    }

    public void endMovie() {
        tv.off();
        soundSystem.off();
        dvdPlayer.off();
        System.out.println("Movie has ended");
    }
}

// Client code
public class FacadePatternExample {
    public static void main(String[] args) {
        TV tv = new TV();
        SoundSystem soundSystem = new SoundSystem();
```

```
        DVDPlayer dvdPlayer = new DVDPlayer();

        HomeTheaterFacade homeTheater = new HomeTheaterFacade(tv, soundSystem, dvdPlayer);
        homeTheater.watchMovie();
        homeTheater.endMovie();
    }
}
```

6. Flyweight Pattern

Concept: The Flyweight Pattern is used to minimize memory usage or computational expenses by sharing as much as possible with similar objects. It is used to reduce the number of objects created and to decrease memory footprint and increase performance.

Real-Time Example: A text editor where each character is a flyweight object shared among different documents.

Code Example:

Java

Collapse

```
import java.util.HashMap;
import java.util.Map;

// Flyweight
class Character {
    private char symbol;

    public Character(char symbol) {
        this.symbol = symbol;
    }

    public void display() {
        System.out.println("Character: " + symbol);
    }
}

// Flyweight Factory
class CharacterFactory {
    private Map<Character, Character> characters = new HashMap<>();

    public Character getCharacter(char symbol) {
        Character character = characters.get(symbol);
        if (character == null) {
            character = new Character(symbol);
            characters.put(symbol, character);
        }
        return character;
    }
}

// Client code
public class FlyweightPatternExample {
    public static void main(String[] args) {
        CharacterFactory factory = new CharacterFactory();

        Character a1 = factory.getCharacter('a');
        Character a2 = factory.getCharacter('a');
        Character b = factory.getCharacter('b');

        a1.display();
        a2.display();
        b.display();

        System.out.println("a1 and a2 are the same instance: " + (a1 == a2)); // Output: true
    }
}
```

7. Proxy Pattern

Concept: The Proxy Pattern provides a surrogate or placeholder for another object to control access to it. It is used to create a representative object that controls access to another

object, which may be remote, expensive to create, or in need of security.

Real-Time Example: A bank account proxy that controls access to the actual bank account object.

Code Example:

Java

Collapse

```
// Subject
interface BankAccount {
    void deposit(double amount);
    void withdraw(double amount);
    double getBalance();
}

// Real Subject
class RealBankAccount implements BankAccount {
    private double balance;

    @Override
    public void deposit(double amount) {
        balance += amount;
    }

    @Override
    public void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
        }
    }

    @Override
    public double getBalance() {
        return balance;
    }
}

// Proxy
class BankAccountProxy implements BankAccount {
    private RealBankAccount realBankAccount;

    public BankAccountProxy() {
        this.realBankAccount = new RealBankAccount();
    }

    @Override
    public void deposit(double amount) {
        realBankAccount.deposit(amount);
    }

    @Override
    public void withdraw(double amount) {
        realBankAccount.withdraw(amount);
    }

    @Override
    public double getBalance() {
        return realBankAccount.getBalance();
    }
}

// Client code
public class ProxyPatternExample {
    public static void main(String[] args) {
        BankAccount account = new BankAccountProxy();
        account.deposit(100);
        account.withdraw(50);
        System.out.println("Balance: " + account.getBalance()); // Output: Balance: 50.0
```

```
61    }
62}
```

Summary

Adapter Pattern: Allows incompatible interfaces to work together by acting as a bridge.
Bridge Pattern: Decouples an abstraction from its implementation, allowing them to vary independently.
Composite Pattern: Treats individual objects and compositions of objects uniformly, useful for part-whole hierarchies.
Decorator Pattern: Adds behavior to individual objects dynamically without affecting other objects.
Facade Pattern: Provides a simplified interface to a complex subsystem, hiding its complexities.
Flyweight Pattern: Minimizes memory usage by sharing objects, reducing the number of objects created.
Proxy Pattern: Provides a surrogate or placeholder for another object to control access to it.
These structural design patterns help in organizing code, making it more modular, and improving the maintainability and scalability of software systems. Understanding and applying these patterns can significantly enhance the design of your applications.