Here's the complete version with explanations, when to use each type of design pattern, and corresponding code examples for all patterns.

Creational Design Patterns

Purpose: These patterns deal with object creation. They make the creation process more flexible and efficient.
When to use: When you want to control or simplify how objects are created.

1. **Singleton Pattern**
   Ensures only one instance of a class is created.

```java
public class Singleton {
    private static Singleton instance;

    private Singleton() {}  // private constructor

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

2. **Factory Pattern**
   Creates objects without specifying the exact class.

```java
interface Shape {
    void draw();
}

class Circle implements Shape {
    public void draw() { System.out.println("Circle"); }
}

class Square implements Shape {
    public void draw() { System.out.println("Square"); }
}

class ShapeFactory {
    public Shape getShape(String shapeType) {
        if (shapeType.equals("CIRCLE")) {
            return new Circle();
        } else if (shapeType.equals("SQUARE")) {
            return new Square();
        }
        return null;
    }
}
```

3. **Builder Pattern**
   Builds complex objects step by step.

```java
class Computer {
    private String CPU;
    private String RAM;

    public static class Builder {
        private String CPU;
        private String RAM;

        public Builder setCPU(String CPU) {
            this.CPU = CPU;
            return this;
        }

        public Builder setRAM(String RAM) {
            this.RAM = RAM;
            return this;
        }

        public Computer build() {
            Computer computer = new Computer();
            computer.CPU = this.CPU;
            computer.RAM = this.RAM;
            return computer;
        }
    }
}
```

4. **Prototype Pattern**
   Creates new objects by copying an existing object.

```java
class Prototype implements Cloneable {
    public Prototype clone() throws CloneNotSupportedException {
        return (Prototype) super.clone();
    }
}
```

Structural Design Patterns

Purpose: These patterns deal with the structure of objects and classes. They help ensure that different parts of a system work well together.
When to use: When you need to organize or connect objects and classes efficiently.

1. **Adapter Pattern**
   Allows incompatible interfaces to work together.

```java
interface MediaPlayer {
    void play(String audioType);
}

class AudioPlayer implements MediaPlayer {
    public void play(String audioType) {
        System.out.println("Playing " + audioType);
    }
}
```

```
class MediaAdapter implements MediaPlayer {
    private AudioPlayer audioPlayer;

    public MediaAdapter() {
        audioPlayer = new AudioPlayer();
    }

    public void play(String audioType) {
        audioPlayer.play(audioType);
    }
}
```

2. **Decorator Pattern**
   Adds new functionality to an object dynamically.

```
java
```

```
interface Coffee {
    String getDescription();
}

class BasicCoffee implements Coffee {
    public String getDescription() {
        return "Basic Coffee";
    }
}

class MilkDecorator implements Coffee {
    private Coffee coffee;

    public MilkDecorator(Coffee coffee) {
        this.coffee = coffee;
    }

    public String getDescription() {
        return coffee.getDescription() + ", Milk";
    }
}
```

3. **Proxy Pattern**
   Controls access to another object.

```
java
```

```
interface RealSubject {
    void request();
}

class RealObject implements RealSubject {
    public void request() {
        System.out.println("RealObject: Handling request.");
    }
}

class Proxy implements RealSubject {
    private RealObject realObject;

    public void request() {
        if (realObject == null) {
            realObject = new RealObject();
        }
        realObject.request();
```

```
    }
}
```

Behavioral Design Patterns

Purpose: These patterns focus on communication and interaction between objects. They simplify complex interactions between them.
When to use: When you want to manage how objects talk to each other and handle complex logic.

1. **Observer Pattern**
   Notifies multiple objects when the state of one object changes.

```java
import java.util.ArrayList;
import java.util.List;

interface Observer {
    void update();
}

class Subject {
    private List<Observer> observers = new ArrayList<>();

    void addObserver(Observer observer) {
        observers.add(observer);
    }

    void notifyAllObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

class ConcreteObserver implements Observer {
    public void update() {
        System.out.println("Observer updated!");
    }
}
```

2. **Strategy Pattern**
   Allows the behavior of an object to change dynamically.

```java
interface Strategy {
    int doOperation(int num1, int num2);
}

class AddStrategy implements Strategy {
    public int doOperation(int num1, int num2) {
        return num1 + num2;
    }
}

class SubtractStrategy implements Strategy {
    public int doOperation(int num1, int num2) {
        return num1 - num2;
    }
}
```

```
class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int num1, int num2) {
        return strategy.doOperation(num1, num2);
    }
}
```

This cheat sheet provides a brief explanation of each design pattern category, when to use it, and simple code examples for each pattern.