CSE327 — Design Patterns Cheatsheet

1. Factory Method

Purpose:
Encapsulates object creation logic inside a method, not a class. Prevents tight coupling and centralizes instantiation logic.

Example:
A logger that decides whether to return a file-based, DB, or console logger.

```
class LoggerFactory {
    public static Logger getLogger(String type) {
        if (type.equals("FILE")) return new FileLogger();
        else if (type.equals("DB")) return new DBLogger();
        else return new ConsoleLogger();
    }
}
```

Key Concept:
You call a method → it decides what subclass to return based on input.

UML Highlights:

- `Product` interface
- `ConcreteProductA`, `ConcreteProductB`
- `Creator` has `factoryMethod()`

2. Abstract Factory

Problem:
If you manually inject parts like:

```
CarAssembler ca = new CarAssembler(new ToyotaChassis(), new WaltonEngine(), ...);
```

It leads to mismatched parts.

Solution:

Use a `PartsFactory` to return compatible parts only.

```
interface PartsFactory {
    Chassis getChassis();
    Engine getEngine();
    Body getBody();
    Logo getLogo();
}


class ToyotaFactory implements PartsFactory {
    public Chassis getChassis() { return new ToyotaChassis(); }
    public Engine getEngine() { return new ToyotaEngine(); }
    ...
}


class CarAssembler {
    PartsFactory factory;
    CarAssembler(PartsFactory pf) { this.factory = pf; }

    Car assemble() {
        Body b = factory.getBody();
        Logo l = factory.getLogo();
        ...
    }
}
```

UML Highlights:

- `AbstractFactory` + multiple `Product` types
- `ConcreteFactoryA`, `ConcreteFactoryB`

3. Composite

Concept:

Let individual objects and groups (composites) be treated the same way. Useful in UIs or file systems.

Example:

A graphics editor where `CompositeShape` holds both simple shapes and groups.

```
class CompositeShape implements Shape {
    ArrayList<Shape> list;

    void addToComposite(Shape s) {
        list.add(s);
    }

    void resize(float factor) {
        for (Shape s : list) {
            s.resize(factor);
        }
    }
}
```

UML Highlights:

- `Component` defines operations (`resize()`)
- `Leaf` is the basic unit
- `Composite` stores children, applies operations recursively

4. Strategy

Concept:

Encapsulates interchangeable behaviors (algorithms). Change behavior without altering the object.

Example:

Switch character movement dynamically in a game.

```
interface MoveStrategy {
    void move();
}

class Run implements MoveStrategy {
    public void move() { System.out.println("Running"); }
}

class Fly implements MoveStrategy {
    public void move() { System.out.println("Flying"); }
}


class Player {
    MoveStrategy strategy;
    void setStrategy(MoveStrategy s) { strategy = s; }
    void performMove() { strategy.move(); }
}
```

UML Highlights:

- `Context` uses `Strategy` interface
- Multiple `ConcreteStrategy` implementations

5. Template Method

Concept:

Define a template (skeleton) of an algorithm in a superclass, but leave the actual steps to subclasses.

Example:

Making a cake:

```
abstract class Cake {
    final void makeCake() {
        makeBatter();
        bake();
        decorate();
    }

    abstract void makeBatter();
    abstract void bake();
    abstract void decorate();
}


class ChocolateCake extends Cake {
    void makeBatter() { ... }
    void bake() { ... }
    void decorate() { ... }
}
```

UML Highlights:

- `AbstractClass` defines template method
- `ConcreteClass` overrides steps

6. Singleton

Concept:

Ensures a class has only one instance and provides global access.

Example:

AppConfig, Logger, DBConnectionManager, etc.

```
class Singleton {
    private static Singleton instance;
    private Singleton() {}

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

UML Highlights:

- Private constructor
- Static instance method
- Class responsible for its own instance

7. Decorator

Concept:
Adds features to an object dynamically by wrapping it. Avoids subclass explosion.

Example:
Build a donut and add toppings:

```
interface Donut {
    String make();
}

class PlainDonut implements Donut {
    public String make() { return "Plain Donut"; }
}

class ChocolateDecorator implements Donut {
    Donut base;
    ChocolateDecorator(Donut d) { base = d; }

    public String make() {
        return base.make() + " + Chocolate";
    }
}


Donut d = new ChocolateDecorator(new PlainDonut());
System.out.println(d.make()); // Plain Donut + Chocolate
```

UML Highlights:

- `Component` interface
- `ConcreteComponent`, `Decorator`
- `ConcreteDecorator` wraps another `Component`

8. Observer

Concept:

One-to-many relationship: when one object changes state, all dependents are notified.

Example:

Bell rings → students react.

```java
interface Observer {
    void update(String msg);
}

class Student implements Observer {
    String name;
    Student(String n) { name = n; }
    public void update(String msg) {
        System.out.println(name + " got: " + msg);
    }
}

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

    void attach(Observer o) { observers.add(o); }
    void notifyAll(String msg) {
        for (Observer o : observers) o.update(msg);
    }
}
```

UML Highlights:

- `Subject` maintains a list of `Observer`
- Calls `update()` on state change