Обектно ориентираното програмиране е утвърдена парадигма. Известни са множество често срещани проблеми при проектиране на системи, както и решения за тях. Концептуалните решения за създаване на добра структура на програма се наричат софтуерни шаблони за дизайн или design patterns.

Един от най-често срещаните казуси в ООП е създаване на различен тип обект в зависимост от динамичен параметър. Например при разработка на софтуер за чертане е възможно да имаме няколко вида стандартни геометрични форми (триъгълник, правоъгълник, кръг и т.н.). Потребителят трябва да може да избира каква форма би искал да нарисува по време на изпълнение на програмата. Логично е формите да бъдат описани като класове със съответстващи променливи. Проблемът възниква, когато се налага да създадем обекти от тези класове в зависимост от потребителския вход:

```
switch (type)
    {
       case "Triangle":
          Triangle shape = new Triangle();
       case "Circle":
          Circle shape = new Circle();
       case "Rectangle":
          Rectangle shape = new Rectangle();
    }
```

На пръв поглед няма нищо нередно с този код и той ще работи. Но ако се наложи геометричните форми да бъдат създавани в друг клас или към тях да бъде добавен нов тип, би се наложило фрагментът код да се модифицира в множество файлове. Създаването на отделна променлива за всеки възможен тип и възможното дублирането на код не са добра практика и биха могли да внесат грешки в програмата.

ООП предлага решение за първия проблем – формите да бъдат групирани в една променлива чрез използване на полиморфизъм. Геометричните форми имат общо поведение – те могат да бъдат начертани. Нека тогава ги обединим чрез интерфейс:

```
public interface ShapeBase {
   void draw();
}
public class Circle implements ShapeBase {
@Override
 public void draw() {
    System.out.println("Drew a circle with center coordinates: " + centerX + " " + centerY + " and radius: " + radius + ".");
}
```

Кодът за създаване на обектите става независим от типа и съответно е с по-високо ниво на абстракция:

```
ShapeBase shape = null;
switch (type)
    {
       case "Triangle":
          shape = new Triangle();
       case "Circle":
          shape = new Circle();
       case "Rectangle":
          shape = new Rectangle();
    }
```

Добавянето на нов Shape все още представлява трудност, ако проверката за параметъра присъства в повече от един клас. Добро решение е цялата функционалност за създаване на обектите да бъде делегирана на отделен клас. Такива класове са известни като factory класове и използването им е част от шаблона Factory pattern. Класът ShapeFactory има следния вид:

```
public class ShapeFactory
{
   public static ShapeBase createShape(String type)
   {
      switch (type)
      {
         case "Triangle":
            return new Triangle();
         case "Circle":
            return new Circle();
         default:
            return null;
      }
   }
}
```

Сега кодът се намира в един клас и е лесно той да бъде модифициран. Динамичното селектиране на тип също е улеснено:

```
ShapeBase shape = ShapeFactory.createShape("Triangle");
```

Остава да се добави функционалност, която ще позволи на потребителя да въведе стойности за координатите. Тъй като тя отново ще бъде обща за всички геометрични форми, най-подходящо е просто да се добави като метод на интерфейса - void initialize(). Всяка форма след това дефинира собствена имплементация. Например:

```
public class Circle implements ShapeBase
{
   @Override
   public void initialize()
   {
      Scanner scanner = new Scanner(System.in);
      System.out.println("Enter radius: ");
      radius = scanner.nextFloat();
      …
   }
}
```

В така написания код се наблюдава проблем от гледна точка на разделение на функционалностите. Методът initialize() има за цел единствено да прочете стойностите от конзолата и да ги зададе на съответните променливи. Скенерът е зависимост или dependency на тази функция, но не е добра практика тя да се налага да създава негова инстанция. Създаването на обект и използването му са две отделни операции и следва да бъдат разделени.

Подаването на необходима за изпълнението зависимост на клас или метод е известен софтуерен шаблон – Dependency injection. Целта му е именно да разграничи инициализирането на обекти и тяхната употреба. Имплементацията е тривиална – зависимостта се създава другаде и се подава като параметър на метода, където ще бъде използвана:

```
public void initialize(Scanner scanner)
   {
      System.out.println("Enter radius: ");
      radius = scanner.nextFloat();
      …
   }
```

Чрез използване на шаблоните Factory и Dependency Injection успешно създадохме система от класове за динамично създаване на геометрични форми, която лесно може да бъде модифицирана, тествана и поддържана.

Задача:

Да се реализира функционалност за компютърна игра, която позволява на герой да се бие с различни видове чудовища. Чудовищата имплементират интерфейса Monster, който им позволява да атакуват героя. Методът **float attack()** връща различна стойност в зависимост от типа:

Zombie – 5

Vampire – 10

Dragon – 20

Програмата трябва да може да генерира битки с произволна група чудовища. Да се създаде клас MonsterFactory с метод, който приема като параметър генератор на случайни числа от 0 до 2 и създава съответния тип чудовище. За целта използвайте метода **nextInt(int bound)** на класа **Random**. Генерираните числа са между 0 и bound.

Да се създаде масив с 5 чудовища. Ако общата им атака е под 50, да се изпише, че героят е победил. Ако е 50 или повече – героят е сразен.