1. Identify the aspect of your application that vary separate them from what stays the same

- Take parts that vary and encapsulate them, so that later you can alter or extend the parts that vary without affecting those that don't.

2. Program to an interface, not an implementation

- We can program to an interface without having to actually use a Java interface

Programming to an implementation would be:

```
Dog d = new Dog();
d.bark();
```

declaring the variable `d` forces to code to concrete implementation

programming to an interface/supertype would be:

```
Animal animal = new Dog();
animal.make_sound();
```

it is given that the animal is Dog, but now Animal reference can be used polymorphically. Assign concrete implementation object at runtime:

```
Animal animal = getAnimal();
animal.make_sound();
```

3. Favor composition over inheritance

Creating systems using composition gives a flexibility. It encapsulates a family of algorithms into their own set of subclasses, also let us ***change behavior at run time***

4. Strive for loosely coupled designs between objects that interact

Loosely coupled designs allow us to build flexible OO system that can handle changes because they minimize the interdependency between objects

5. The Open-Closed Principle: Classes should be open for extension but closed for modification

6. The Dependency Inversion Principle: Depend upon abstractions. Do not depend upon concrete classes

- High-level components should not depend on low-level components
- High-level components is class with behavior defined in terms of other, Low-level components

7. Principle of Least Knowledge: talk only to your immediate friends.

11. Hollywood Principle: Don't call us, we'll call you

- Low-level components can participate in the computation But the high-level component controls when and how. A low-level component never calls high-level component directly.

12. Single Responsibility Principle: A class should have only one reason to change.