**What problems do design pattern solve?**

**1. Finding Appropriate Objects**

Design patterns help you identify less-obvious abstractions and the objects that can capture them.

**2. Determining Object Granularity**

Facade pattern describes how to represent complete subsystems as objects.

Flyweight pattern describes how to support huge numbers of objects at the finest granularities.

Abstract Factory and Builder yield objects whose only responsibilities are creating other objects.

Visitor and Command yield objects whose only responsibilities are to implement a request on another object or group of objects.

**3. Specifying Object Interfaces**

The set of all signatures defined by an object's operations is called the interface to the object.

A type is a name used to denote a particular interface.

An object may have many types, and widely different objects can share a type.

A type is a subtype of another if its interface contains the interface of its supertype, or a subtype *inheriting* the interface of its supertype.

Objects are known only through their interfaces. An object's interface says nothing about its implementation

When a request is sent to an object, the particular operation that's performed depends on *both* the request *and* the receiving object.

Dynamic binding: the run-time association of a request to an object and one of its operations.

Polymorphism: dynamic binding can substitute objects that have identical interfaces for each other at run-time

Design patterns help you define interfaces by identifying their key elements and the kinds of data that get sent across an interface.

Design patterns also specify relationships between interfaces.

**4. Specifying Object Implementations**

An object's implementation is defined by its class.

An abstract class is one whose main purpose is to define a common interface for its subclasses.

A mixin class is a class that's intended to provide an optional interface or functionality to other classes.

An object's class defines how the object is implemented.

An object's type only refers to its interface.

An object can have many types.

Objects of different classes can have the same type.

Relationship between class and type: class as type (C++) vs. interface as type (Java).

Class inheritance: Sub-typing + Implementation inheritance

Interface inheritance: Sub-typing only (Polymorphism)

Pure abstract classes as interfaces.

Many of the design patterns depend on the distinction between class and interface inheritances

First Principle of reusable object-oriented design: Programming to an Interface, not an Implementation

Class inheritance-based implementation reuse is only half the story. Inheritance's ability to define families of objects with *identical* interfaces is also important, because polymorphism depends on it.

Two benefits to manipulating objects solely in terms of the interface defined by abstract classes:

Clients remain unaware of the specific types of objects they use, as long as the objects adhere to the interface that clients expect.

Clients remain unaware of the classes that implement these objects. Clients only know about the abstract class(es) defining the interface.

Don't declare variables to be instances of particular concrete classes.

Creational patterns ensure that your system is written in terms of interfaces, not implementations.