Advanced OO & Design Patterns

IS-A and HAS-A

```
class BeachUmbrella extends Umbrella implements SunProtector {
    Stand stans;
}
class Umbrella{}
interface SunProtector{}
class Stand{}
```

IS-A and HAS-A

```
class BeachUmbrella extends Umbrella implements SunProtector {
    Stand stans;
}
class Umbrella{}
interface SunProtector{}
class Stand{}
```

- BeachUmbrella **IS-A** Umbrella
- BeachUmbrella **IS-A** SunProtector
- BeachUmbrella **HAS-A** Stand
- BeachUmbrella **IS-A** Object

IS-A and HAS-A

```
class A extends B {
        C tail;
}
```

Which is true?

A. A **HAS-A** B and A **HAS-A** C
B. A **HAS-A** B and A **IS-A** C
C. A **IS-A** B and A **HAS-A** C
D. A **IS-A** B and A **IS-A** C
E. B **IS-A** A and A **HAS-A** C
F. B **IS-A** A and A **IS-A** C

Coupling

Coupling is the manner and degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules.

Loose Coupling means reducing dependencies of a class that use a different class directly.
In **tight coupling**, classes and objects are dependent on one another. In general, tight coupling is usually bad because it reduces flexibility and re-usability of code and it makes changes much more difficult and impedes testability etc.

Tight Coupling

```
public class Car {
        public void move { }
}
public class Traveller {
        Car car = new Car();
        public void startJourney(){
        car.move();
    }
}
```

Loose Coupling

```
public class Car implements Vehicle {
        public void move { }
}
public class Traveller {
        Vehicle v;
        public void setVehicle(Vehicle v){
                this.v = v;
    }
        public void startJourney(){
        v.move();
    }
}
```

Cohesion

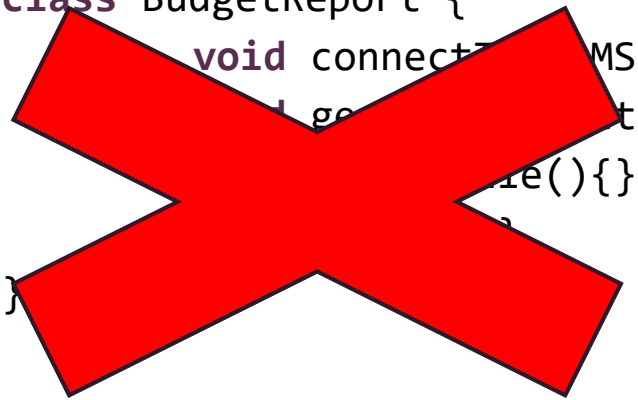
Cohesion refers to the degree to which the elements of a module belong together. Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is.

```
class BudgetReport {
        void connectToRDBMS() {}
        void generateBudgetReport(){}
        void saveToFile(){}
        void print(){}
}
```

Cohesion

Cohesion refers to the degree to which the elements of a module belong together. Thus, it is a measure of how strongly related each piece of functionality expressed by the source code of a software module is.

```
class BudgetReport {
        void connectToRDBMS() {}
        void generateBudgetReport(){}
        void printReport(){}
}
```



```
class BudgetReport {
        Options getReportingOptions(){}
        void generateBudgetReport(){}
}
class RDBMSManager {}
class PrintStuff {}
class FileSaver {}
```

Coupling and Cohesion

1. Given the code:

```
class Window { // it represents graphical window control
  public void show (String path) throws IOException{
    FileInputStream fis = new FileInputStream(path);
    byte[] buf = new byte[1024];
    fis.read(buf);
    fis.close(); // other operations on the buf
  }
}
```

Which of below is true:

1. High cohesion
2. Low cohesion
3. Loose coupling
4. Tight coupling

Object Composition Principles

Object composition refers to one object having another as an instance variable (HAS-A). Sometimes, that instance variable might be the same type as the object we are writing.

```
public interface Box {
        void pack();
        void seal();
}
```

Object Composition Principles

Object composition refers to one object having another as an instance variable (HAS-A). Sometimes, that instance variable might be the same type as the object we are writing.

```
public interface Box {
        void pack();
        void seal();
}
public class GiftBox implement Box {
        public void pack() {System.out.println("pack box");}
        public void seal() {System.out.println("seal box");}
}
```

Object Composition Principles

Now that we've figured out Box, it's time to build a MailerBox:

```
public class MailerBox implements Box {
public void pack() {
System.out.println("pack box");  // problem?
}
public void seal() {
System.out.println("seal box"); // problem?
}
public void ship() {
System.out.println("put in mailbox");
}
}
```

Object Composition Principles

Object composition refers to one object having another as an instance variable (HAS-A). Sometimes, that instance variable might be the same type as the object we are writing.

```
public interface Box {
        void pack();
        void seal();
}
public class MailerBox implement Box {
        public void pack() {...}
        public void seal() {...}
        public void ship() {...}
}
```

Object Composition Principles

```
public interface Mailer {
        void ship();
}
public class MailerBox implements Box, Mailer {
        private Box box;
public MailerBox(Box box) {
        this.box = box;
}
public void pack(){
        box.pack();
}
public void seal(){
        box.seal();
}
public void ship(){...}
}
```

The first thing to notice is that the logic to pack and seal a box is only in one place—in the Box hierarchy where it belongs. In fact, the MailerBox doesn't even know what kind of Box it has. This allows us to be very flexible.

Object Composition Principles

```
public interface Mailer {
        void ship();
}
public class MailerBox implements Box, Mailer {
        private Box box;
public MailerBox(Box box) {
        this.box = box;
}
public void pack(){
        box.pack();
}
public void seal(){
        box.seal();
}
public void ship(){...}
}
```

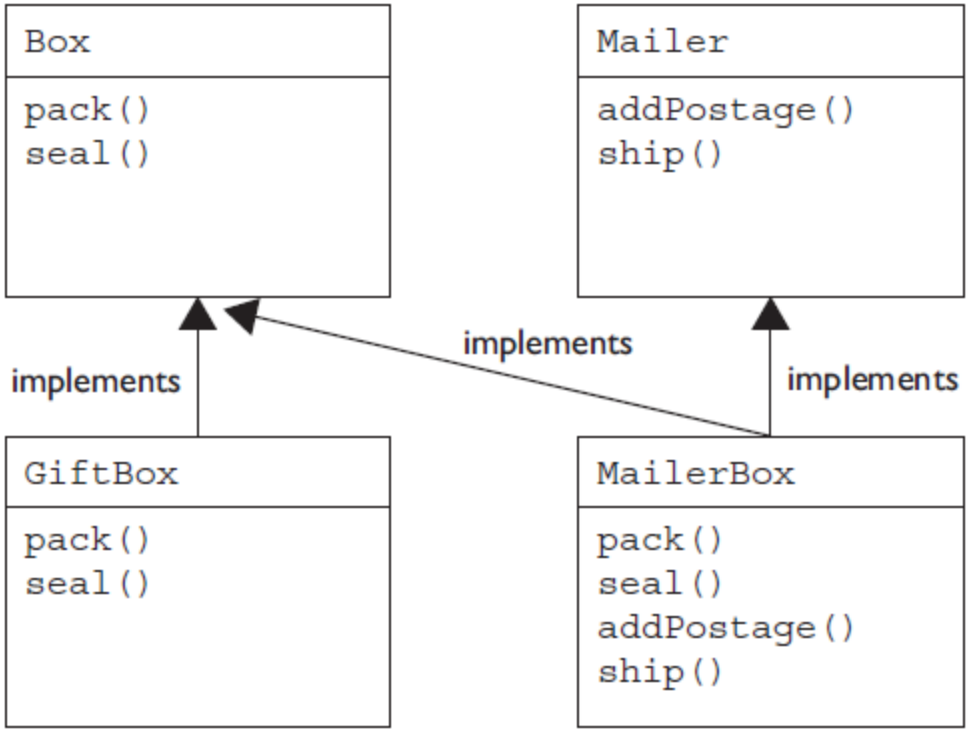
Next, notice the implementation of pack() and seal(). That's right—each is one line. We delegate to Box to actually do the work. This is called method forwarding or method delegation. These two terms mean the same thing.

Object Composition Principles

```
public interface Mailer {
        void ship();
}
public class MailerBox implements Box, Mailer {
        private Box box;
public MailerBox(Box box) {
        this.box = box;
}
public void pack(){
        box.pack();
}
public void seal(){
        box.seal();
}
public void ship(){...}
}
```

Finally, notice that MailerBox is both a Box and a Mailer. This allows us to pass it to any method that needs a Box or a Mailer.

Polymorphism



You can see the composition part. MailerBox both IS-A Box and HAS-A Box. MailerBox is composed of a Box and delegates to Box for logic. That's the terminology for object composition.

Benefits of composition

Benefits of composition include:

Reuse An object can delegate to another object rather than repeating the same code.

Preventing a proliferation of subclasses We can have one class per functionality rather than needing one for every combination of functionalities.

What is a design pattern?

Wikipedia currently defines a design pattern as “a general reusable solution to a commonly occurring problem within a given context.” What does that mean? As programmers, we frequently need to solve the same problem repeatedly. Such as how to only have one of a class of an object in the application. Rather than have everyone come up with their own solution, we use a “best practice” type solution that has been documented and proven to work. The word “general” is important. We can’t just copy and paste a design pattern into our code. It’s just an idea. We can write an implementation for it and put that in our code.

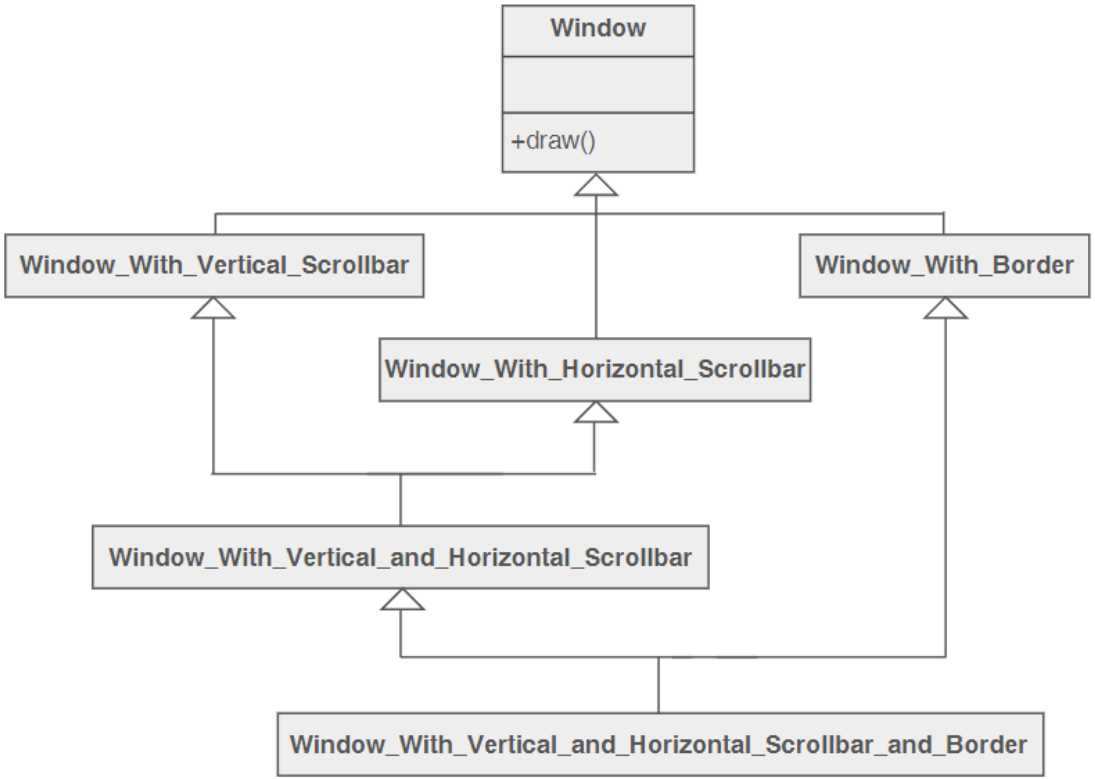
Design Patterns

Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation (eg. Singleton).

Structural Design Patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities (eg. Decorator)

Behavioral Design Patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication (eg. Iterator)

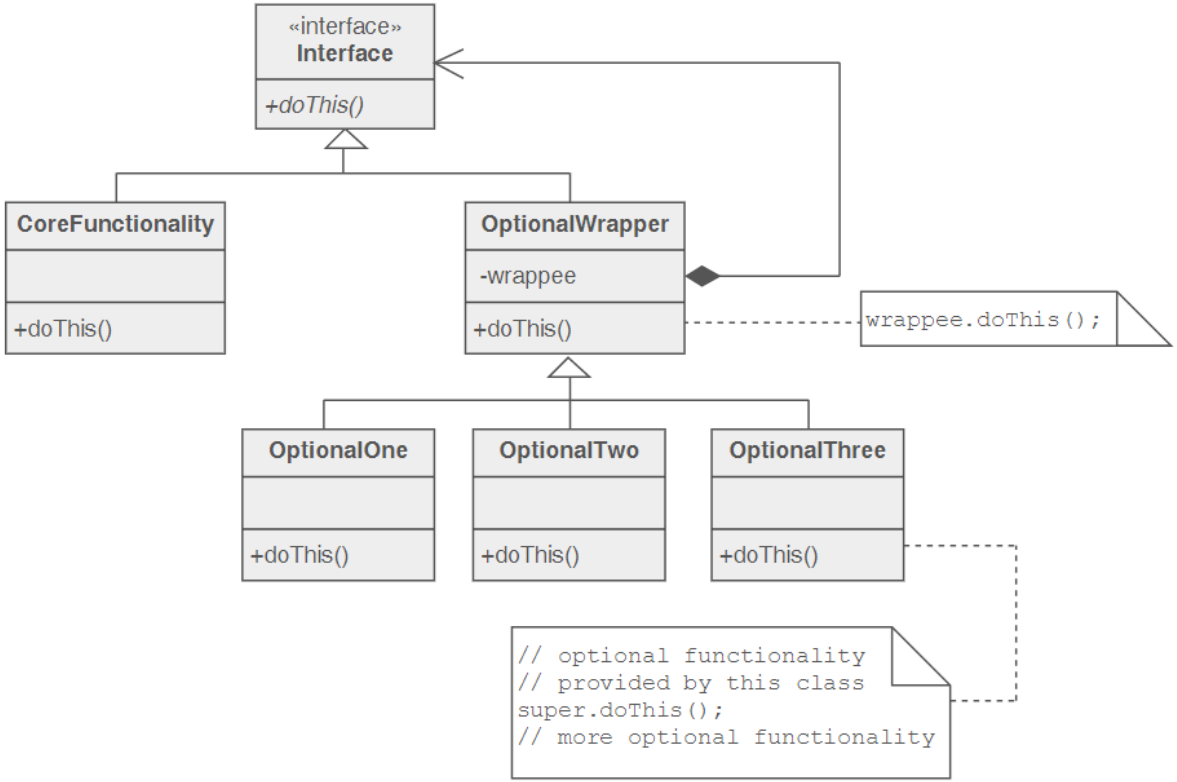
Problem


Window
+draw()
Window_With_Vertical_Scrollbar
Window_With_Horizontal_Scrollbar
Window_With_Border
Window_With_Vertical_and_Horizontal_Scrollbar
Window_With_Vertical_and_Horizontal_Scrollbar_and_Border

Problem

```
class A {
        public void doIt() { System.out.print('A'); } }
class AwithX extends A {
        public void doIt() { super.doIt(); doX(); }
        private void doX() { System.out.print('X'); } }
class AWithY extends A {
        public void doIt() { super.doIt(); doY(); }
        public void doY() { System.out.print('Y'); } }
class AWithZ extends A {
        public void doIt() { super.doIt(); doZ(); }
        public void doZ() { System.out.print('Z'); } }
class AwithXY extends AwithX {
        private aWithY obj = new aWithY();
        public void doIt() { super.doIt(); obj.doY(); } }
class AwithXYZ extends AwithX {
        private aWithY obj1 = new aWithY();
        private aWithZ obj2 = new aWithZ();
        public void doIt() { super.doIt(); obj1.doY(); obj2.doZ(); } }
```

Decorator Design Pattern



Decorator Design Pattern

```
interface I {
        void doIt(); }

class A implements I {
        public void doIt() { System.out.print('A'); } }

abstract class D implements I {
        private I core; public D(I inner) { core = inner; }
        public void doIt() { core.doIt(); } }
class X extends D {
        public X(I inner) { super(inner); }
        public void doIt() { super.doIt(); doX(); }
        private void doX() { System.out.print('X'); } }
class Y extends D {
        public Y(I inner) { super(inner); }
        public void doIt() { super.doIt(); doY(); }
        private void doY() { System.out.print('Y'); } }
class Z extends D {
        public Z(I inner) { super(inner); }
        public void doIt() { super.doIt(); doZ(); }
        private void doZ() { System.out.print('Z'); } }
```

Singleton Design Pattern

```
import java.util.*;
public class Show {
  private static final Show INSTANCE  = new Show();   // (this is the singleton)
  private Set<String> availableSeats;
  public static Show getInstance() {                  // callers can get to
    return INSTANCE;                                  // the instance
  }
  private Show(){        // callers can't create directly anymore. Must use getInstance()
    availableSeats = new HashSet<String>();
    availableSeats.add("1A");
  }
  public static void main(String[] args) {
    ticketAgentBooks("1A");
    ticketAgentBooks("1A");
  }
  private static void ticketAgentBooks(String seat) {
    Show show = Show.getInstance();
}
```

Singleton Design Pattern

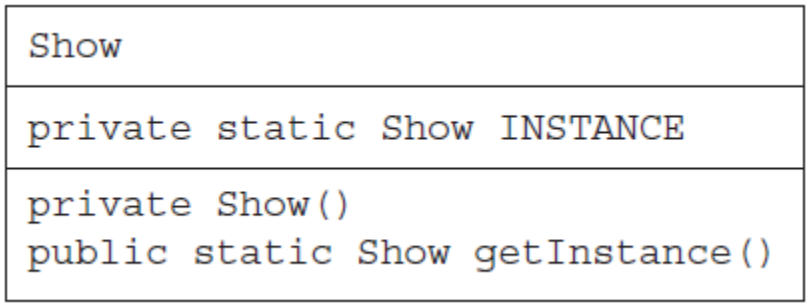
The key parts of the singleton pattern are:

1. A private static variable to store the single instance called the singleton. This variable is usually final to keep us from accidentally changing it.
2. A public static method for callers to get a reference to the instance.
3. A private constructor so no callers can instantiate the object directly.

Singleton Design Pattern

Remember, the code doesn't create a new Show each time, but merely returns the singleton instance of Show each time getInstance() is called.

If the constructor weren't private, we wouldn't have a singleton. Callers would be free to ignore getInstance() and instantiate their own instances. Which would leave us with multiple instances in the program and defeat the purpose entirely.



Singleton Design Pattern

If getInstance() weren't public, we would still have a singleton. However, it wouldn't be as useful because only static methods of the class Show would be able to use the singleton.

If getInstance() weren't static, we'd have a bigger problem. Callers couldn't instantiate the class directly, which means they wouldn't be able to call getInstance() at all.

If INSTANCE weren't static and final, we could have multiple instances at different points in time. These keywords signal that we assign the field once and it stays that way for the life of the program.

Singleton Design Pattern

One "feature" of the above implementation is that it creates the Show object before we need it. This is called eager initialization, which is good if the object isn't expensive to create or we know it will be needed for every run of the program. Sometimes, however, we want to create the object only on the first use. This is called lazy initialization.

Singleton Design Pattern

```
private static Show INSTANCE;
private Set<String> availableSeats;
public static Show getInstance() {
  if (INSTANCE == null) {
    INSTANCE = new Show();
  }
  return INSTANCE;
}

// can you see any problem with this implementation?
```

Singleton Design Pattern

```
class Foo {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    // other code...
}
```

Singleton Design Pattern

```
// double checked locked pattern
class Foo {
    private volatile Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    // other code...
}
```

Singleton Design Pattern

Benefits of the singleton pattern include the following:

The primary benefit is that there is only one instance of the object in the program. When an object's instance variables are keeping track of information that is used across the program, this becomes useful. For example, consider a web site visitor counter. You only want one count that is shared.

Another benefit is performance. Some objects are expensive to create. For example, maybe we need to make a database call to look up the state for the object.

Problem

DAO stands for "Data Access Object." A DAO is only responsible for storing data. Nothing else. Why can't we do this in the object with everything else, you ask?
Suppose we have few objects in our program.
Already there is a problem. We want each class to have a single purpose. Storing and searching objects in the database is NOT that purpose. Having that database code all over makes it hard to focus on the classes' core purpose for existing, which is clearly for our entertainment. Since dealing with a database is very common, separating out that responsibility is a pattern—the DAO.

DAO Design Pattern

```
import java.util.*;
public class Book {
  private static Map<String, Book> bookstore // storage: extra
     = new HashMap<String, Book>();          // responsibility

  private String isbn;                       // core responsibility:
  private String title;                      // book instance
  private String author;                     // variables

  public Collection<Book> findAllBooks() {   // more storage
    return bookstore.values();               // extra responsibility
  }
  public Book findBookByIsbn(String isbn) {  // more storage
    return bookstore.get(isbn);
  }
  public void create() {
    bookstore.put(isbn, this);
  }
  public void delete() {                     // still more storage
    bookstore.remove(isbn);
  }
  public void update() {                     // yet still more storage
    // no operation - for an in-memory database,
    // we update automatically in real time
  }
  // omitted getters and setters
}
```

DAO Design Pattern

```
import java.util.*;
public class Book {
  private static Map<String, Book> bookstore // storage: extra
     = new HashMap<String, Book>();          // responsibility

  private String isbn;                       // core responsibility:
  private String title;                      // book instance
  private String author;                     // variables

  public Collection<Book> findAllBooks() {   // more storage
    return bookstore.values();               // extra responsibility
  }
  public Book findBookByIsbn(String isbn) {
    return bookstore.get(isb
  }
  public void create() {
    bookstore.put(isbn, this)
  }
  public void delete() {
    bookstore.remove(isbn);
  }
  public void update() {                     // yet still more storage
    // no operation - for an in-memory database,
    // we update automatically in real time
  }
  // omitted getters and setters
}
```

Object	Responsibilities	Still More Responsibilities
Book	Store book information, be read	Store and search in database
CD	Store CD information, be listened to	Store and search in database
DVD	Store DVD information, be watched	Store and search in database

DAO Design Pattern

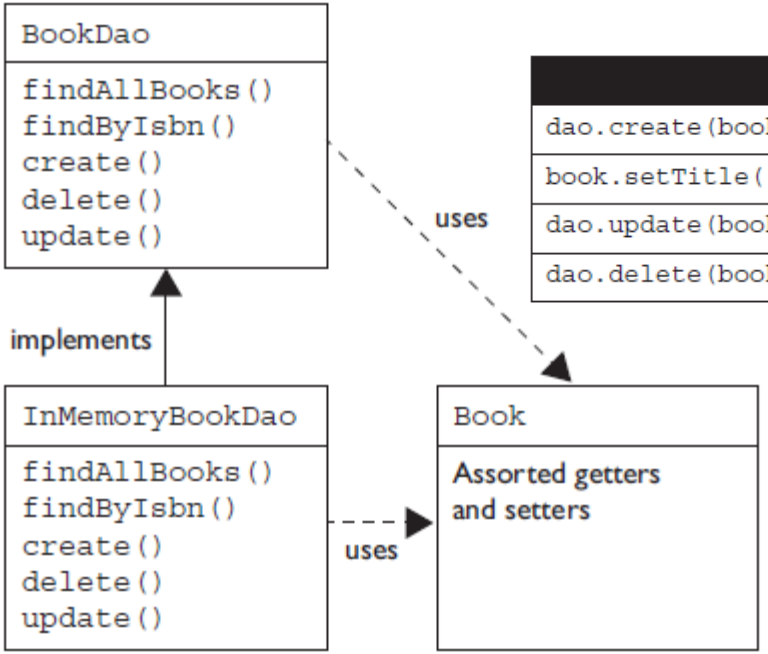
```
public class Book {
  private String isbn;              // core responsibility:
  private String title;             // book instance
  private String author;            // variables

  // omitted getters and setters
}
```

DAO Design Pattern

```
import java.util.*;

public class InMemoryBookDao {

  private static Map<String, Book> bookstore  // storage:
          = new HashMap<String, Book>();     // core responsibility

  public Collection<Book> findAllBooks() {
    return bookstore.values();
  }
  public Book findBookByIsbn(Book book) {
    return bookstore.get(book.getIsbn());
  }
  public void create(Book book) {
    bookstore.put(book.getIsbn(), book);
  }
  public void delete(Book book) {
    bookstore.remove(book.getIsbn());
  }
  public void update(Book book) {
                                    // no operation - for an in-memory
                                    // database,
                                    // we update automatically in real time
  }  }
```

```
public class Book {
  private String isbn;              // core responsibility:
  private String title;             // book instance
  private String author;            // variables

  // omitted getters and setters
}
```

DAO Design Pattern



	Book	DAO
`dao.create(book)`		Deals with datastore
`book.setTitle("updated")`	Changes a Book instance variable	
`dao.update(book)`		Deals with datastore
`dao.delete(book)`		Deals with datastore

DAO Design Pattern

To review, the benefits of the DAO pattern are as follows:

The main object (Book in this case) is cohesive and doesn't have database code cluttering it up.

All the database code is in one part of the program, making it easy to find.

We can change the database implementation without changing the business object.

Reuse is easier. As the database code grows, we can create helper classes and even helper superclasses.

Factory Design Pattern

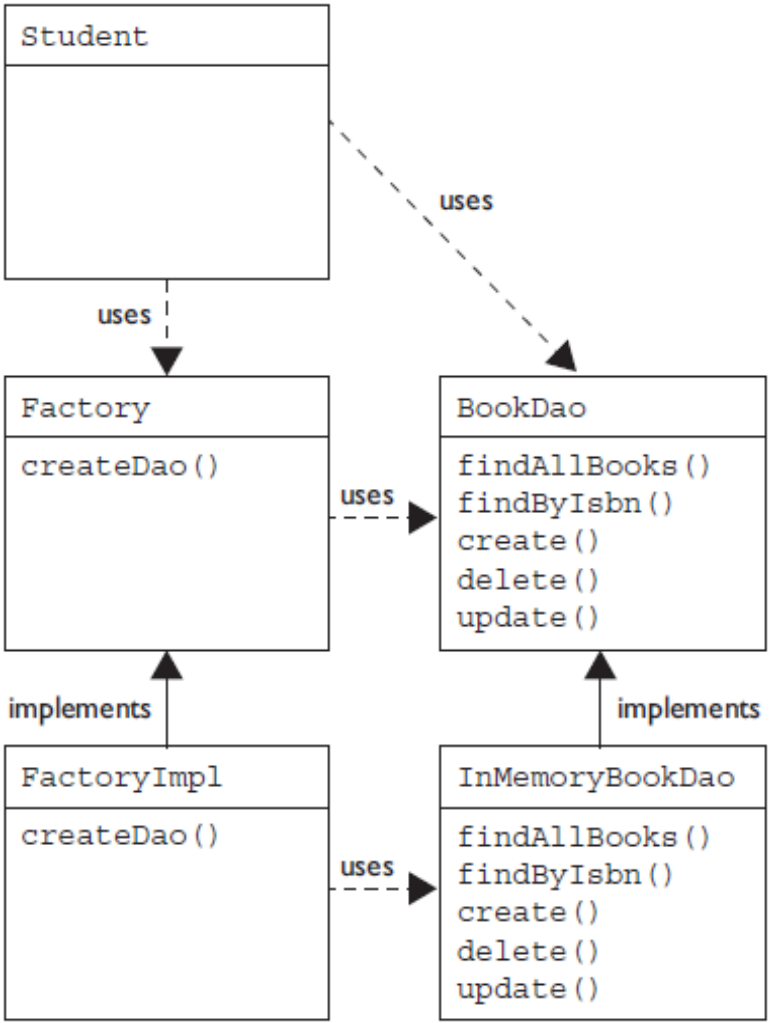
Like the singleton design pattern, the factory design pattern is a creational design pattern. Unlike the singleton, it doesn't limit you to only having one copy of an object. The factory design pattern creates new objects of whatever implementation it chooses.

Factory Design Pattern

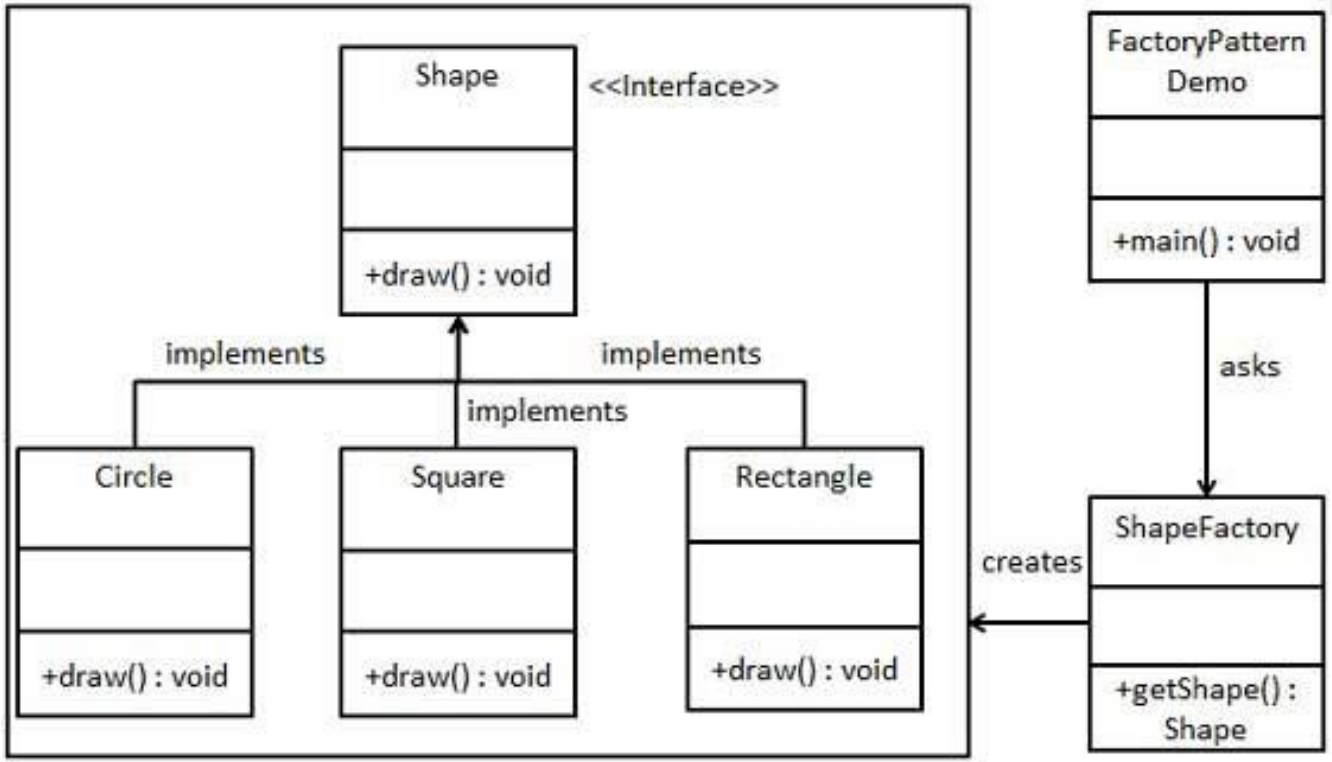
```
public abstract class Factory {
  public abstract BookDao createDao();
}
public class FactoryImpl extends Factory {
  public BookDao createDao() {           // right now, we only
    return new InMemoryBookDao();        // have one DAO
  }
}
public class Student {
  public static void main(String[] args) {
    Factory factory = new FactoryImpl();
    BookDao dao = factory.createDao();   // create the DAO
    // work with dao
  }
}
```

Factory Design Pattern

```
public class FactoryImpl extends Factory {
  public BookDao createDao() {
    if (Util.isTestMode()) {
      return new InMemoryBookDao();
    } else {
      return new OracleBookDao();
    }
  }
}
```



Factory Design Pattern



Factory Design Pattern

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

```
public class Rectangle implements Shape {
        @Override public void draw() { System.out.println("Inside Rectangle::draw() method."); } }
```

```
public class Square implements Shape {
        @Override public void draw() { System.out.println("Inside Square::draw() method."); } }
```

```
public class Circle implements Shape {
        @Override public void draw() { System.out.println("Inside Circle::draw() method."); } }
```

Factory Design Pattern

```
public class ShapeFactory { //use getShape method to get object of type shape
                public Shape getShape(String shapeType){
                                if(shapeType == null){ return null; }
                                if(shapeType.equalsIgnoreCase("CIRCLE")){ return new Circle(); }
                                else if(shapeType.equalsIgnoreCase("RECTANGLE")){ return new Rectangle(); }
                                else if(shapeType.equalsIgnoreCase("SQUARE")){ return new Square(); }
                                return null; } }
```

```
public class FactoryPatternDemo {
public static void main(String[] args) {
                                ShapeFactory shapeFactory = new ShapeFactory();
                                //get an object of Circle and call its draw method.
                                Shape shape1 = shapeFactory.getShape("CIRCLE"); //call draw method of Circle shape1.draw();
                                //get an object of Rectangle and call its draw method.
                                Shape shape2 = shapeFactory.getShape("RECTANGLE"); //call draw method of Rectangle shape2.draw();
                                 //get an object of Square and call its draw method.
                                Shape shape3 = shapeFactory.getShape("SQUARE"); //call draw method of circle shape3.draw(); } }
```

Factory Design Pattern

Benefits of the factory design pattern include the following:

The caller doesn't change when the factory returns different subclasses. This is useful when the final implementation isn't ready yet. For example, maybe the database isn't yet available. It's also useful when we want to use different implementations for unit testing and production code. For example, you want to write code that behaves the same way, regardless of what happens to be in the database.

Centralizes creation logic outside the calling class. This prevents duplication and makes the code more cohesive.

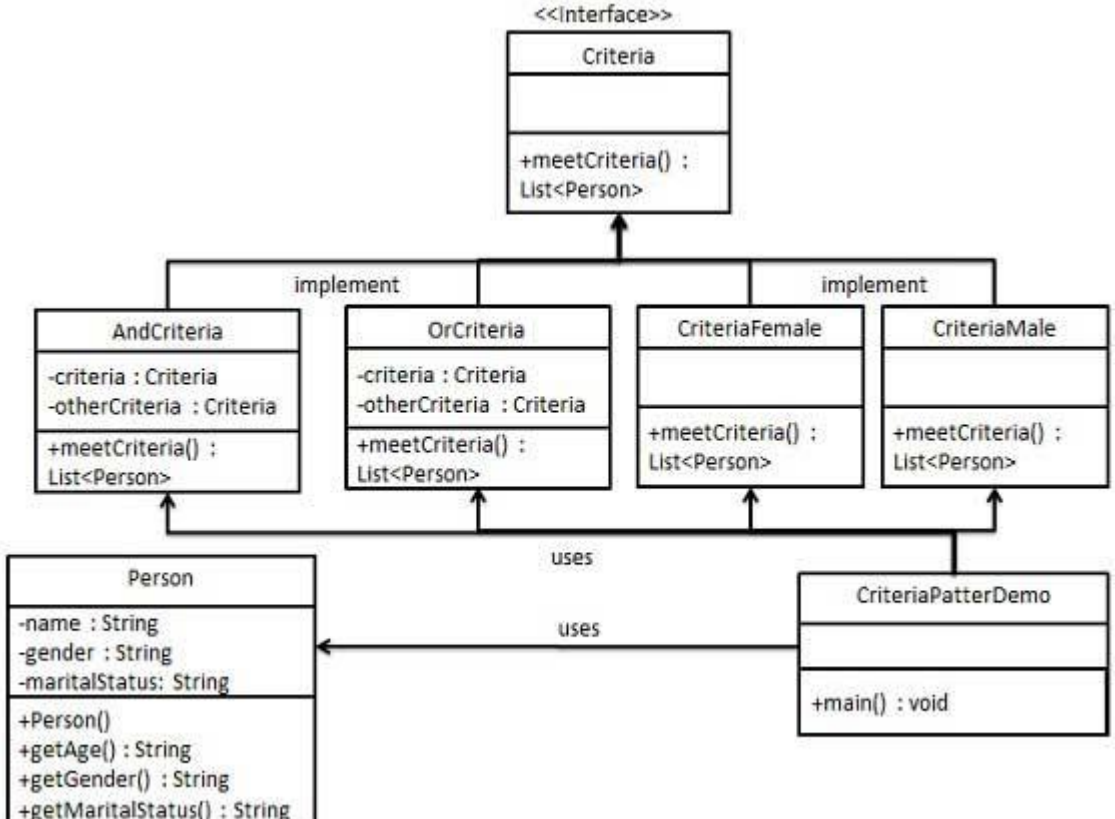
Allows for extra logic in the object creation process. For example, an object is time-consuming to create, and you want to reuse the same one each time.

Filter Pattern

Filter pattern or Criteria pattern is a design pattern that enables developers to filter a set of objects using different criteria and chaining them in a decoupled way through logical operations. This type of design pattern comes under structural pattern as this pattern combines multiple criteria to obtain single criteria.

We're going to create a Person object, Criteria interface and concrete classes implementing this interface to filter list of Person objects. CriteriaPatternDemo, our demo class uses Criteria objects to filter List of Person objects based on various criteria and their combinations.

Filter Pattern



Filter Pattern

```
// Person.java
public class Person {
   private String name;
   private String gender;
   private String maritalStatus;

   public Person(String name, String gender, String maritalStatus){
      this.name = name;
      this.gender = gender;
      this.maritalStatus = maritalStatus;
   }

//setters and getters ommited
}
```

Filter Pattern

```
// Criteria.javaI
import java.util.List;

public interface Criteria {
   public List<Person> meetCriteria(List<Person> persons);
}
```

Filter Pattern

```
//CriteriaMale.java
import java.util.ArrayList; import java.util.List;
public class CriteriaMale implements Criteria {
public List<Person> meetCriteria(List<Person> persons) {
    List<Person> malePersons = new ArrayList<Person>();
    for (Person person : persons) {
      if(person.getGender().equalsIgnoreCase("MALE")){
        malePersons.add(person);
      }
    }
    return malePersons;
  }
}
```

Filter Pattern

```
//CriteriaFemale.java
import java.util.ArrayList; import java.util.List;
public class CriteriaFemale implements Criteria {
public List<Person> meetCriteria(List<Person> persons) {
    List<Person> femalePersons = new ArrayList<Person>();
    for (Person person : persons) {
      if(person.getGender().equalsIgnoreCase("FEMALE")){
        femalePersons.add(person);
      }
    }
    return femalePersons;
  }
}
```

Filter Pattern

```
//CriteriaSingle.java
import java.util.ArrayList; import java.util.List;
public class CriteriaSingle implements Criteria {
public List<Person> meetCriteria(List<Person> persons) {
    List<Person> singlePersons = new ArrayList<Person>();
    for (Person person : persons) {
      if(person.getMaritalStatus().equalsIgnoreCase("SINGLE")){
        singlePersons.add(person);
      }
    }
    return singlePersons;
  }
}
```

Filter Pattern

```
import java.util.List; // AndCriteria.java
public class AndCriteria implements Criteria {
  private Criteria criteria;
  private Criteria otherCriteria;

  public AndCriteria(Criteria criteria, Criteria otherCriteria) {
    this.criteria = criteria;
    this.otherCriteria = otherCriteria;
  }
public List<Person> meetCriteria(List<Person> persons) {
    List<Person> firstCriteriaPersons = criteria.meetCriteria(persons);
    return otherCriteria.meetCriteria(firstCriteriaPersons);
  }
}
```

Filter Pattern

```
import java.util.List; // OrCriteria.java
public class OrCriteria implements Criteria {
   private Criteria criteria;
   private Criteria otherCriteria;
   public OrCriteria(Criteria criteria, Criteria otherCriteria) {
      this.criteria = criteria;
      this.otherCriteria = otherCriteria;
   }
public List<Person> meetCriteria(List<Person> persons) {
      List<Person> firstCriteriaItems = criteria.meetCriteria(persons);
      List<Person> otherCriteriaItems = otherCriteria.meetCriteria(persons);

      for (Person person : otherCriteriaItems) {
         if(!firstCriteriaItems.contains(person)){
            firstCriteriaItems.add(person);
         }
      }
      return firstCriteriaItems;    } }
```

Filter Pattern

```
public class CriteriaPatternDemo {
   public static void main(String[] args) {
      List<Person> persons = new ArrayList<Person>();

      persons.add(new Person("Robert","Male", "Single"));
      persons.add(new Person("John", "Male", "Married"));
      persons.add(new Person("Laura", "Female", "Married"));

      Criteria male = new CriteriaMale();
      Criteria female = new CriteriaFemale();
      Criteria single = new CriteriaSingle();
      Criteria singleMale = new AndCriteria(single, male);
      Criteria singleOrFemale = new OrCriteria(single, female);

      System.out.println("Males: ");
      printPersons(male.meetCriteria(persons));
      System.out.println("Females: ");
      printPersons(female.meetCriteria(persons));
      System.out.println("Single Males: ");
      printPersons(singleMale.meetCriteria(persons));
   }

   public static void printPersons(List<Person>
persons){

      for (Person person : persons) {
         System.out.println("Person : [ Name : " +
person.getName() + ", Gender : " +
person.getGender() + ", Marital Status : " +
person.getMaritalStatus() + " ]");
      }
   }   }
```

Questions

Which statements indicate the need to use the factory pattern? (Choose all that apply.)

A. You don’t want the caller to depend on a specific implementation
B. You have two classes that do the same thing
C. You only want one instance of the object to exist
D. You want one class to be responsible for database operations
E. You want to build a chain of objects

Questions

Which is a benefit of the DAO pattern? (Choose all that apply.)
A. Reuse is easier
B. The database code is automatically generated
C. We can change the database implementation independently
D. Your business object extends the DAO pattern to reduce coding
E. You are limited to one DAO object

Questions

Given:
1) ClassA has a ClassD
2) Methods in ClassA use public methods in ClassB
3) Methods in ClassC use public methods in ClassA
4) Methods in ClassA use public variables in ClassB

Which is most likely true? (Choose only one.)
A. ClassD has low cohesion.
B. ClassA has weak encapsulation.
C. ClassB has weak encapsulation.
D. ClassB has strong encapsulation.
E. ClassC is tightly coupled to ClassA.