Design Patterns

The Factory Design Patterns

Factory
Design Pattern

The Issue:

Object Creation

Since most object-oriented languages provide object instantiation (i.e., “new”) and initialization (i.e., constructors) mechanisms, there may be a tendency to simply use these facilities without forethought to future consequences.

The overuse of this functionality often introduces a great deal of the inflexibility in the system -- direct object instantiation creates an explicit association between the creator and created classes.

While associations are a necessary type of relationship in an object-oriented system, the coupling introduced between classes is extremely difficult to overcome should requirements change (as they always do).

The Issue (cont.):

Object Creation

Assume the following:

- Aircraft is a java interface that contains behaviors common to all aircraft (i.e., “load()”, “takeoff()”, “taxi()”, etc.)
- PassengerCraft is a concrete class (a non-abstract class) that implements the Aircraft interface.

Now assume that we have build an application that uses the classes described above. A key section of this application is shown below:

```
[…]
Aircraft acft = new PassengerCraft();

acft.load();
acft.taxi();
acft.takeoff();
[…]
```

This piece of code above would work fine if we only had one type of Aircraft and we never planned to add any more or change the one we have. The problem is – the moment that you decide that you will never add any more aircraft or change the one that you have is the same moment you find you have to change it!

The Issue:

Object Creation and Change

Change. Unexpectedly, the government awards you a big, fat contract to extend your application to support the wide variety of aircraft for the military. Assuming that were to happen, you would definitely be updating that object creation code.

Rather than the simple object creation code from the previous example where we could only ever create one type of aircraft, you'd need a more complicated setup:

```
Aircraft acft = null;

if (type.equals(“Passenger”))
     acft = new PassengerCraft();
else if (type.equals(“Cargo”))
     acft = new CargoCraft();
else if (type.equals(“Fighter”))
     acft = new FighterCraft();

acft.load();
acft.taxi();
acft.takeoff();
[…]
```

The Issue (cont):

Object Creation

This more complex setup works just fine – it does what we need.

But – there are several design issues that can lead to problems with the maintenance of this code. The class that contains this code (i.e., "AircraftProcessor") is now coupled to more classes.

Initially, we coupled AircraftProcessor to PassengerCraft by referencing PassengerCraft directly within the AircraftProcessor (remember – coupling classes is not a good thing – your code becomes more fragile and less flexible). Now we've coupled CargoCraft and FighterCraft to AircraftProcessor as well.

Now - once it was completed, assume that the government loved the software you created for them and as a result they give you an even bigger contract to enhance your application to support even more types of aircraft. Additionally, you are awarded a long-term software maintenance contract guaranteeing your future for years to come.

You soon realize that you don't really need to support what your software was originally designed for (Passenger Aircraft) so you need to dump that aspect and concentrate on your new contract work.

To enhance the code to meet the needs of our contract, we find that we're once again editing the same old section of code, adding the support for more types of aircraft:

Factory

Design Pattern

The Issue (cont):

Object Creation

```
Aircraft acft = null;

~~if (type.equals(“Passenger”))~~
      ~~acft = new PassengerCraft();~~
~~else if (type.equals(“Cargo”))~~
      ~~acft = new CargoCraft();~~

if (type.equals(“LightTransport”)
      acft = new LightTransportCraft();
else if (type.equals(“HeavyTransport”)
      acft = new HeavyTransportCraft();
else if (type.equals(“Troop”))
      acft = new TroopTransportCraft();
else if (type.equals(“Fighter”))
      acft = new FighterCraft();
else if (type.equals(“Bomber”))
      acft = new BomberCraft();
else if (type.equals(“Gunship”))
      acft = new GunshipCraft();
else if (type.equals(“Medic”))
      acft = new MedicHelicopterCraft();

acft.load();
acft.taxi();
acft.takeoff();
[…]
```

PassengerCraft is no longer supported so it needs to be removed.

CargoCraft has been expanded to Light & Heavy Transport so it needs to be removed.

These are new classes related to our new contract requirements so they have been added.

FighterCraft is the only class we have not had to change from the previous version.

The Issue (cont):

As you can see, this technique of changing, adding & removing object creation code does work, but it involves a lot of maintenance effort. The AircraftProcessor class is certainly not closed for modification (Open-Closed Principle).

Further problems arise when one day you find that object creation occurs in multiple places within your application – first 2, then 3 – then 4 places! Additionally, different developers have coded the object creation code in each of those 4 sections using their own style and interpretation of the logic.

Now you *really* have a maintenance problem. Your AircraftProcessor is now coupled to 7 classes (with more to come), and there are other classes in your application that are performing object creation that are also coupled to those 7 classes.

Additionally, when you want to change the object creation code (to add new classes, remove obsolete classes, etc.), you will have to change 4 different sections of code (which as you recall have been written by a different developer using their own style and interpretation of the logic).

This situation results in brittle, tightly coupled code that is difficult to work with and prone to errors.
If you do not want to lose your lucrative government contracts – something will need to be done about this!

How can your remove this class-coupling and code duplication, and leave your domain classes (i.e., AircraftProcessor) closed for modification?

The Solution:

The (Simple) Factory Pattern

The Simple Factory Pattern defines a technique to encapsulate the details of object creation in a single class known as a “Factory” class whose job it is to create concrete instances of various objects that implement a given interface. The other objects in the application that need new objects created by the factory class are known as “clients” of the factory.

By doing this, we ensure the following:

- Object creation code appears in one place only (rather than multiple times as in our previous example)
- Changes to object creation code are limited to the factory class and do not affect the factory’s clients – changes are made in one place at one time.
- Clients of the factory class need not (and will not) know the actual concrete class type of the objects that are created for it – therefore they are not coupled to any concrete classes.
- The client classes are closed for modification

Factory
Design Pattern

The Solution (cont.):

(Simple) Factory Details

The following example shows how our factory class might look:

```
public class USAircraftFactory
{
     private USAircraftFactory(){} // Static methods, no reason to “new”

     public static Aircraft createAircraft(String type)
     {

          if (type.equals(“LightTransport”)
               returnnew LightTransportCraft();
          else if (type.equals(“HeavyTransport”)
               return new HeavyTransportCraft();
          else if (type.equals(“Troop”))
               return new TroopTransportCraft();
          else if (type.equals(“Fighter”))
               return new FighterCraft();
          else if (type.equals(“Bomber”))
               return new BomberCraft();
          else if (type.equals(“Gunship”))
               return new GunshipCraft();
          else if (type.equals(“Medic”))
               return new MedicHelicopterCraft();

          else return null; // Don’t know what this is…
     }

}
```

The Solution (cont.):

(Simple) Factory Details

Our AircraftProcessor class can now make use of our USAircraftFactory class:

```
public class AircraftProcessor
{
     …

     public void process(String type)
     {
          […]
          Aircraft acft = USAircraftFactory.createAircraft(type);

          acft.load();
          acft.taxi();
          acft.takeoff();
          […]
     }
     […]
}
```

The USAircraftFactory object creates and returns the appropriate Aircraft object without the AircraftProcessor needing to know what was created or how it was created.

As you can see, by referring to concrete objects via generic *Aircraft* references, the AircraftProcessor needs to know nothing about what *type* of aircraft it is actually working with. Any number of new types of concrete Aircraft classes can be created by the factory, and the AircraftProcessor will need no changes to properly handle them.

Factory
Design Pattern

The Solution (cont.):

(Simple) Factory Details

The following illustrates how changes are made to the USAircraftFactory – the changes to not affect any clients.

```
public class USAircraftFactory
{

     private USAircraftFactory() {}

     public static Aircraft createAircraft(String type)
     {
          if (type.equals("LightTransport")
               returnnew LightTransportCraft();
          else if (type.equals("HeavyTransport")
               return new HeavyTransportCraft();
          else if (type.equals("Troop"))
               return new TroopTransportCraft();
          else if (type.equals("Fighter"))
               return new FighterCraft();
          else if (type.equals("Bomber"))
               return new BomberCraft();
          else if (type.equals("Gunship"))
               return new GunshipCraft();
          else if (type.equals("Medic"))
               return new MedicHelicopterCraft();
          else if (type.equals("Stealth"))
               return new StealthFighterCraft();

          else return null; // Don't know what this is…

     }
}
```

The new StealthFighterCraft can now be returned by the AircraftFactory and the AircraftProcessor will not need to know anything about it.

Factory
Design Pattern

Further Issues:

Increasing Factory Complexity

Lets continue our example by assuming that as a result of the success of our application (especially once we implemented our USAircraftFactory), we managed to win an international contract to extend our application to support aircraft from a variety of nations.

Each of these nations has a set of aircraft types that are similar to the aircraft types that our application already works with, but we soon find that they are of different manufacturers and configurations.

For example, our “LightTransportCraft” class consists a set of components that distinguish it from the other aircraft types. Unfortunately, many of our new client nations also have aircraft that they consider “LightTransportCraft” – but those consist of a different set of components than our original class does. As a result, the “LightTransportCraft” objects we are currently creating are not very useful in other contexts.

We want to keep our initial set of classes as they are, because we have a working (and valuable) application that is actively making use of them. Yet, we need a similar set of classes (or more likely, several sets) to represent the various aircraft of our new international clients.

Factory

Design Pattern

Further Issues (cont.):

Increasing Factory Complexity

If I try to add all the new aircraft types from all the various nations to our existing USAircraftFactory, the factory class will quickly get very complicated, large, and ugly. We would be adding a great deal of maintenance overhead (and coupling) to our initial factory that our current users don't even care about.

Any one client is only interested in one set of aircraft, and yet the factory class they are working with would need to maintain creation details of all aircraft types – those of all nations.

What is needed is a *set* of factories – each set up to work with one nation's specific set of aircraft.

However – although a set of factories sounds good, we've already hard-coded (tightly-coupled) our current factory object (i.e., USAircraftFactory) into our AircraftProcessor. We would need our AircraftProcessor class to be flexible enough to work with *any* of those new factories as needed.

What we really need is a generic factory that the AircraftProcessor can work with that can change it's "inner workings" based upon the client nation's set of aircraft.

How can we modify our factory setup to operate in this fashion?

The Solution:

The Abstract Factory Pattern

The Abstract Factory pattern defines a technique wherein we create a set of individual factories that have a common theme (i.e., aircraft creation) that all implement a common *interface*. The client software would refer to an instance of one of these concrete factories via the common interface type.

To make use of the Abstract Factory pattern, we'll create a new interface called "AircraftFactory" that defines the required factory behaviors (i.e., "createAircraft", and later maybe "repairAircraft", "decommisionAircraft", etc.).

Our "USAircraftFactory" class will implement that interface and will therefore implement those factory methods. This factory will create and return our original set of concrete aircraft objects.

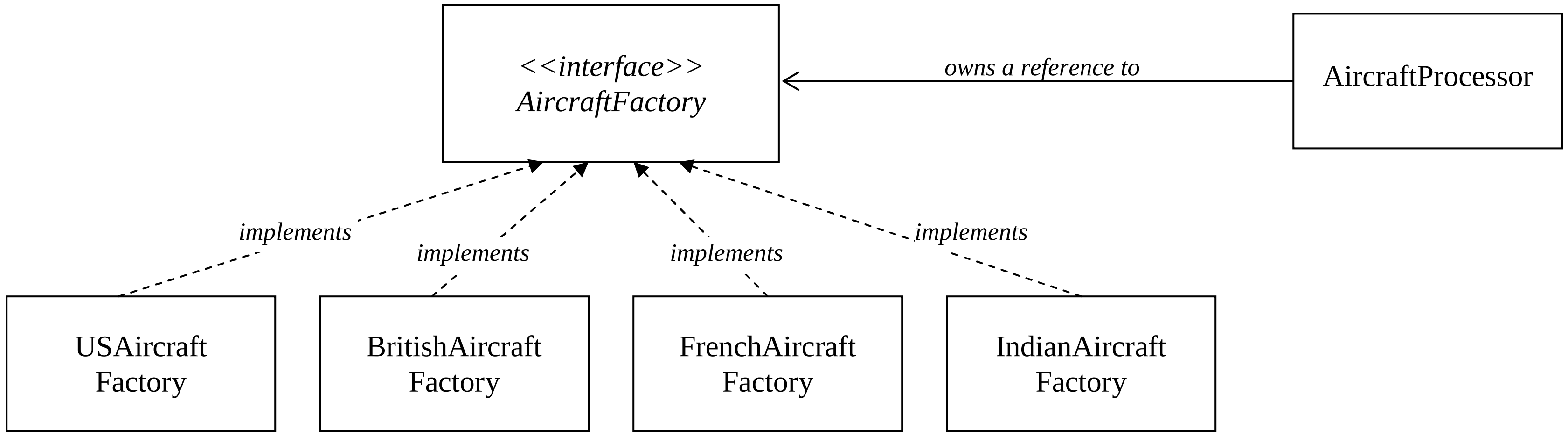
We can then create other factories (i.e., "BritishAircraftFactory", "FrenchAircraftFactory", "IndianAircraftFactory", etc.) for each of our client nations that will return a different set of concrete aircraft objects, specific to that nation's needs.

Our original AircraftProcessor class will have an "AircraftFactory" data member (the interface type) that will be used to refer to a concrete instance of one of our factory classes. That factory will then create concrete aircraft instances.

The Solution (cont.):

The Abstract Factory Pattern

This setup is illustrated below - the AircraftProcessor class has an “AircraftFactory” data member that will be used to refer to a concrete instance of one of our factory classes.



The “AircraftFactory” interface

```
public interface AircraftFactory
{
   public Aircraft createAircraft(String type);
   public Aircraft repairAircraft(String type);
   public Aircraft decommisionAircraft(String type);
}
```

Factory
Design Pattern

The Solution (cont.):

The Abstract Factory Pattern

The concrete “USAircraftFactory” class

```
public class USAircraftFactory implements AircraftFactory
{

      public Aircraft createAircraft(String type)
      {

            if (type.equals(“LightTransport”)
                  return new LightTransportCraft();
            else if (type.equals(“HeavyTransport”)
                  return new HeavyTransportCraft();
            else if (type.equals(“Troop”))
                  return new TroopTransportCraft();
            else if (type.equals(“Fighter”))
                  return new FighterCraft();
            else if (type.equals(“Bomber”))
                  return new BomberCraft();
            else if (type.equals(“Gunship”))
                  return new GunshipCraft();
            else if (type.equals(“Medic”))
                  return new MedicHelicopterCraft();
            else if (type.equals(“Stealth”))
                  return new StealthFighterCraft();

            else return null; // Don’t know what this is…
      }

      […] // other interface methods…
}
```

The Solution (cont.):

The Abstract Factory Pattern

The concrete “BritishAircraftFactory” class

```
public class BritishAircraftFactory implements AircraftFactory
{

      public Aircraft createAircraft(String type)
      {

            if (type.equals(“LightTransport”) || type.equals(“HeavyTransport”))
                  return new BritishTransportCraft();
            else if (type.equals(“Troop”))
                  return new BritishTroopTransportCraft();
            else if (type.equals(“Fighter”) || type.equals(“Bomber”))
                  return new BritishFighterBomberCraft();
            else if (type.equals(“Gunship”))
                  return new BritishGunshipCraft();
            else if (type.equals(“Medic”))
                  return new BritishEvacHelicopterCraft();

            else return null; // Don’t know what this is…
      }

      […] // other interface methods…
}
```

The Solution (cont.):

The Abstract Factory Pattern

The original "AircraftProcessor" class needs no changes.

The "AircraftFactory" data attribute is set to whatever concrete factory instance that is passed into the "AircraftProcessor" constructor.

```
public class AircraftProcessor
{

      private AircraftFactory aircraftFactory = null;

      public AircraftProcessor(AircraftFactory af)
      {
            aircraftFactory = af;
      }

      public void process(String type)
      {
            […]
            Aircraft acft = aircraftFactory.createAircraft(type);

            acft.load();
            acft.taxi();
            acft.takeoff();
            […]
      }
      […]
}
```

This can be an instance of "USAircraftFactory", "BritishAircraftFactory", "FrenchAircraftFactory", "IndianAircraftFactory", etc. - because they all implement the AircraftFactory interface.

This will return any concrete Aircraft instance created by the associated factory object.
All can be treated homogeneously regardless of type since they all must implement the Aircraft interface as described earlier.

An Alternate Solution

A Different Approach

Using our example application, let’s go back in time to *before* we created any factory classes – back when our AircraftProcessor code looked like this:

```
Aircraft acft = null;

if (type.equals(“LightTransport”)
     acft = new LightTransportCraft();
else if (type.equals(“HeavyTransport”)
     acft = new HeavyTransportCraft();
else if (type.equals(“Troop”))
     acft = new TroopTransportCraft();
else if (type.equals(“Fighter”))
     acft = new FighterCraft();
else if (type.equals(“Bomber”))
     acft = new BomberCraft();
else if (type.equals(“Gunship”))
     acft = new GunshipCraft();
else if (type.equals(“Medic”))
     acft = new MedicHelicopterCraft();

acft.load();
acft.taxi();
acft.takeoff();
[…]
```

An Alternate Solution (cont.)

A Different Approach

Remember the problems we encountered at that time:

- This technique of changing, adding & removing object creation code does work, but it involves a lot of maintenance effort. The AircraftProcessor class is certainly not closed for modification (Open-Closed Principle).
- Further problems arise when one day you find that object creation occurs in multiple places within your application –first 2, then 3 – then 4 places! Additionally, different developers have coded the object creation code in each of those 4 sections using their own style and interpretation of the logic.
- Your AircraftProcessor is now coupled to 7 classes (with more to come), and there are other classes in your application that are performing object creation that are also coupled to those 7 classes.
- When you want to change the object creation code (to add new classes, remove obsolete classes, etc.), you will have to change 4 different sections of code (which as you recall have been written by a different developer using their own style and interpretation of the logic).

An Alternate Solution (cont.)

A Different Approach

To solve that dilemma we created a simple factory class to encapsulate aircraft object creation (originally called USAircraftFactory). Let’s do that again, but this time we’ll call that factory class simply “AircraftFactory”.

```
public class AircraftFactory
{
     private AircraftFactory() {}

     public static Aircraft createAircraft(String type)
     {
          if (type.equals(“LightTransport”)
               return new LightTransportCraft();
          else if (type.equals(“HeavyTransport”)
               return new HeavyTransportCraft();
          else if (type.equals(“Troop”))
               return new TroopTransportCraft();
          else if (type.equals(“Fighter”))
               return new FighterCraft();
          else if (type.equals(“Bomber”))
               return new BomberCraft();
          else if (type.equals(“Gunship”))
               return new GunshipCraft();
          else if (type.equals(“Medic”))
               return new MedicHelicopterCraft();

          else return null; // Don’t know what this is…
     }

}
```

An Alternate Solution (cont.)

A Different Approach

As we continued to develop this solution, we realized that the addition of other nation's aircraft to our factory, made the factory very complicated, large, and ugly. We would be adding a great deal of maintenance overhead (and coupling) to our initial factory that our current users don't even care about. Any one client is only interested in one set of aircraft, and yet the factory class they are working with must maintain creation details of all aircraft types – those of all nations. What we needed was a generic factory that the AircraftProcessor can work with that can change its "inner workings" based upon the client nation's set of aircraft.

From there we moved towards the Abstract Factory pattern wherein we created a set of individual factories (i.e., "BritishAircraftFactory", "FrenchAircraftFactory", "IndianAircraftFactory", etc.) that have a common theme (i.e., aircraft creation) that all implement a common interface. The client software would refer to a concrete instance of one of these concrete factories via the common interface type.

That technique worked well - our original AircraftProcessor class used an "AircraftFactory" data member (the interface) that was used to refer to a concrete instance of one of our factory classes. That factory will then create concrete aircraft instances.

Now, rather than use the Abstract Factory pattern, let's look at an alternate approach to solving our need for a generic factory that the AircraftProcessor can work with that can change it's "inner workings" based upon the client nation's set of aircraft. This alternate approach is defined by the *Factory Method Pattern.*

The Solution:

The Factory Method Pattern

The Factory Method pattern defines a technique wherein we define an *abstract method* for creating the objects in our simple factory class, and then create *subclasses* that override that method to specify the actual types of objects that will be created. The client software would refer to a concrete instance of one of these subclasses via a parent factory type reference.

To make use of the FactoryMethod pattern, we’ll first make a change to our simple factory class “AircraftFactory” (remember a few pages back, we changed the name from “USAircraftFactory” to simply “AircraftFactory”).

We’ll remove the implementation of the “createAircraft” method, and instead make it abstract. We’ll also go ahead and make the “AircraftFactory” class itself abstract.

Our simple factory now looks like:

```
public abstract class AircraftFactory
{
     public abstract Aircraft createAircraft(String type);
}
```

The Solution (cont.):

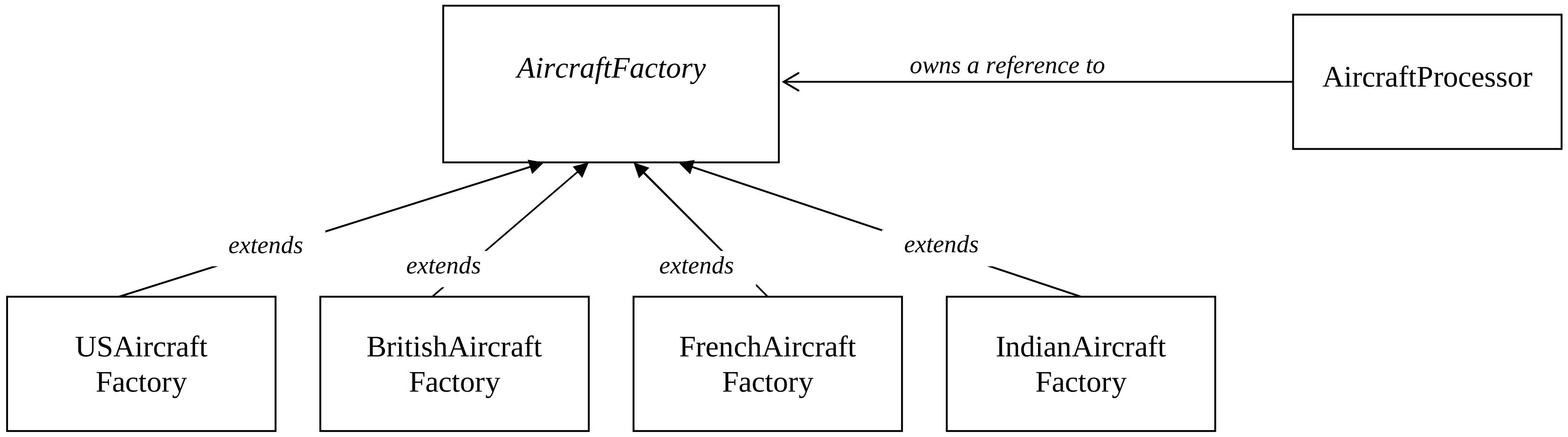
The Factory Method Pattern

Then, rather than create a set of factories that implement a common interface as we did with the Abstract Factory pattern, we'll instead create a set of concrete subclasses that extend the parent "factory class". (i.e., "USAircraftFactory", "BritishAircraftFactory", "FrenchAircraftFactory", "IndianAircraftFactory", etc.)

Each of these subclasses then provide an implementation of the "createAircraft" method for each of our client nations that will return a different set of concrete aircraft objects, specific to that nation's needs.

Our original AircraftProcessor class will have an "AircraftFactory" data member (the abstract parent class type) that will be used to refer to an instance of one of our factory subclasses. That factory will create concrete Aircraft instances.

This is illustrated below - the AircraftProcessor class has an "AircraftFactory" data member that will be used to refer to a concrete instance of one of the "AircraftFactory" subclasses.



Factory

Design Pattern

The Solution (cont.):

The Factory Method Pattern

Subclasses of the abstract “AircraftFactory” class would look like this:

Here we provide a concrete implementation for the abstract “createAircraft” parent method.

We now extend the abstract “AircraftFactory” class.

```
public class USAircraftFactory extends AircraftFactory
{
      public Aircraft createAircraft(String type)
      {
            if (type.equals(“LightTransport”)
                  return new LightTransportCraft();
            else if (type.equals(“HeavyTransport”)
                  return new HeavyTransportCraft();
            else if (type.equals(“Troop”))
                  return new TroopTransportCraft();
            else if (type.equals(“Fighter”))
                  return new FighterCraft();
            else if (type.equals(“Bomber”))
                  return new BomberCraft();
            else if (type.equals(“Gunship”))
                  return new GunshipCraft();
            else if (type.equals(“Medic”))
                  return new MedicHelicopterCraft();

            else return null; // Don’t know what this is…
      }
}
```

Factory
Design Pattern

The Solution (cont.):

The Factory Method Pattern

The original “AircraftProcessor” class needs no changes.

The “AircraftFactory” data attribute is set to whatever concrete factory subclass instance that is passed into the “AircraftProcessor” constructor.

```
public class AircraftProcessor
{

      private AircraftFactory aircraftFactory = null;

      public AircraftProcessor(AircraftFactory af)
      {
            aircraftFactory = af;
      }

      public void process(String type)
      {
            […]
            Aircraft acft = aircraftFactory.createAircraft(type);

            acft.load();
            acft.taxi();
            acft.takeoff();
            […]
      }
      […]
}
```

This can be an instance of “USAircraftFactory”, “BritishAircraftFactory”, “FrenchAircraftFactory”, “IndianAircraftFactory”, etc. – because they all extend the AircraftFactory parent class.

This will return any concrete Aircraft instance created by the associated factory object.
All can be treated homogeneously regardless of type since they all must implement the Aircraft interface as described earlier.

Factory
Design Pattern

Summary:

- In software development, a Factory is the location in the code at which objects are constructed. The intent in employing the pattern is to insulate the creation of objects from their usage. This allows for new derived types to be introduced with no change to the code that uses the base object.
- The factory determines the actual concrete type of object to be created, and it is here that the object is actually created. However, the factory only returns an abstract reference to the created concrete object. This insulates client code from object creation by having clients ask a factory object to create an object of the desired abstract type and to return an abstract reference to the object.
- As the factory only returns an abstract reference, the client code (which requested the object from the factory) does not know - and is not burdened by - the actual concrete type of object that was just created
- The client code has no knowledge whatsoever of the concrete type. The client code deals only with the abstract type. Objects of a concrete type are indeed created by the factory, but the client code accesses such objects only through their abstract interface.
- The Simple Factory pattern defines one or more static methods in a “factory” class that encapsulate the details of object creation whose job it is to create concrete instances of various objects that implement a given interface. The other objects in the application that need new objects created by the factory class are known as “clients” of the factory.
- The Factory Method pattern is an object-oriented design pattern that deals with the problem of creating objects (products) without specifying the exact class of object that will be created. Factory Method handles this problem by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.
- The Abstract Factory Pattern provides a way to encapsulate a group of individual factories that have a common theme. In normal usage, the client software would create a concrete implementation of the abstract factory and then use the generic interfaces to create the concrete objects that are part of the theme. The client does not know (nor care) about which concrete objects it gets from each of these internal factories since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from its general usage.