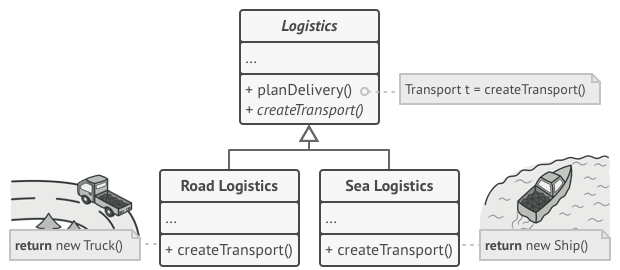
1. **Link:** [**https://refactoring.guru/design-patterns/factory-method**](https://refactoring.guru/design-patterns/factory-method)

**Solution**

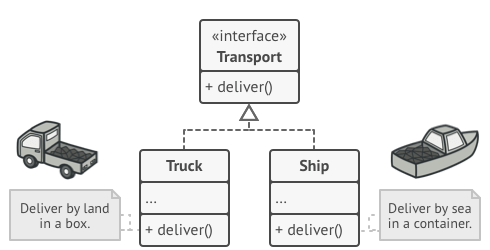
The Factory Method pattern suggests that you replace direct object construction calls (using the new operator) with calls to a special *factory* method. Don’t worry: the objects are still created via the new operator, but it’s being called from within the factory method. Objects returned by a factory method are often referred to as “products.”



*Subclasses can alter the class of objects being returned by the factory method.*

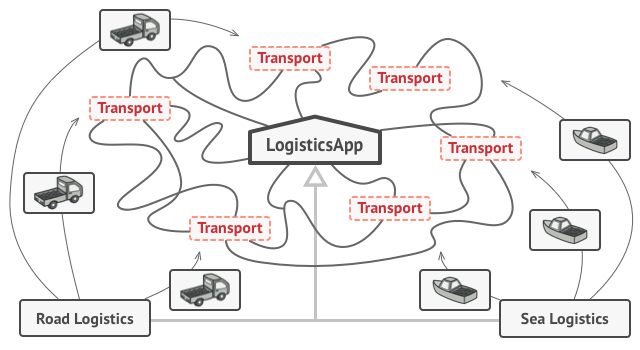
At first glance, this change may look pointless: we just moved the constructor call from one part of the program to another. However, consider this: now you can override the factory method in a subclass and change the class of products being created by the method.

There’s a slight limitation though: subclasses may return different types of products only if these products have a common base class or interface. Also, the factory method in the base class should have its return type declared as this interface.



*All products must follow the same interface.*

For example, both Truck and Ship classes should implement the Transport interface, which declares a method called deliver. Each class implements this method differently: trucks deliver cargo by land, ships deliver cargo by sea. The factory method in the RoadLogistics class returns truck objects, whereas the factory method in the SeaLogistics class returns ships.



*As long as all product classes implement a common interface, you can pass their objects to the client code without breaking it.*

The code that uses the factory method (often called the *client* code) doesn’t see a difference between the actual products returned by various subclasses. The client treats all the products as abstract Transport. The client knows that all transport objects are supposed to have the deliver method, but exactly how it works isn’t important to the client.

1. **Link:** [**https://sourcemaking.com/design\_patterns/factory\_method/cpp/1**](https://sourcemaking.com/design_patterns/factory_method/cpp/1)

**Factory Method in *C++: Before and after***

**Before**

The architect has done an admirable job of decoupling the client from Stooge concrete derived classes, and, exercising polymorphism. But there remains coupling where instances are actually created. If we design an "extra level of indirection" (a "factory method") and have clients use it (instead of "new"), then the last bit of coupling goes away. The "factory method" (aka "virtual constructor") can be defined in the Stooge base class, or, in a separate "factory" class. Note that main() is no longer dependent on Stooge derived classes.

**class** **Stooge**

{

**public**:

**virtual** **void** slap\_stick() = 0;

};

**class** **Larry**: **public** Stooge

{

**public**:

**void** slap\_stick()

    {

        cout << "Larry: poke eyes\n";

    }

};

**class** **Moe**: **public** Stooge

{

**public**:

**void** slap\_stick()

    {

        cout << "Moe: slap head\n";

    }

};

**class** **Curly**: **public** Stooge

{

**public**:

**void** slap\_stick()

    {

        cout << "Curly: suffer abuse\n";

    }

};

**int** **main**()

{

  vector<Stooge\*> roles;

**int** choice;

**while** (**true**)

  {

    cout << "Larry(1) Moe(2) Curly(3) Go(0): ";

    cin >> choice;

**if** (choice == 0)

**break**;

**else** **if** (choice == 1)

      roles.push\_back(**new** Larry);

**else** **if** (choice == 2)

      roles.push\_back(**new** Moe);

**else**

      roles.push\_back(**new** Curly);

  }

**for** (**int** i = 0; i < roles.size(); i++)

    roles[i]->slap\_stick();

**for** (**int** i = 0; i < roles.size(); i++)

**delete** roles[i];

}

**Output**

Larry(1) Moe(2) Curly(3) Go(0): 2

Larry(1) Moe(2) Curly(3) Go(0): 1

Larry(1) Moe(2) Curly(3) Go(0): 3

Larry(1) Moe(2) Curly(3) Go(0): 0

Moe: slap head

Larry: poke eyes

Curly: suffer abuse

**After**

A factory method is a static method of a class that returns an object of that class' type. But unlike a constructor, the actual object it returns might be an instance of a subclass. Another advantage of a factory method is that it can return existing instances multiple times.

**class** **Stooge**

{

**public**:

*// Factory Method*

**static** Stooge \*make\_stooge(**int** choice);

**virtual** **void** slap\_stick() = 0;

};

**int** **main**()

{

  vector<Stooge\*> roles;

**int** choice;

**while** (**true**)

  {

    cout << "Larry(1) Moe(2) Curly(3) Go(0): ";

    cin >> choice;

**if** (choice == 0)

**break**;

    roles.push\_back(Stooge::make\_stooge(choice));

  }

**for** (**int** i = 0; i < roles.size(); i++)

    roles[i]->slap\_stick();

**for** (**int** i = 0; i < roles.size(); i++)

**delete** roles[i];

}

**class** **Larry**: **public** Stooge

{

**public**:

**void** slap\_stick()

    {

        cout << "Larry: poke eyes\n";

    }

};

**class** **Moe**: **public** Stooge

{

**public**:

**void** slap\_stick()

    {

        cout << "Moe: slap head\n";

    }

};

**class** **Curly**: **public** Stooge

{

**public**:

**void** slap\_stick()

    {

        cout << "Curly: suffer abuse\n";

    }

};

Stooge \***Stooge::make\_stooge**(**int** choice)

{

**if** (choice == 1)

**return** **new** Larry;

**else** **if** (choice == 2)

**return** **new** Moe;

**else**

**return** **new** Curly;

}

1. **Link:**

[**https://sourcemaking.com/design\_patterns/factory\_method**](https://sourcemaking.com/design_patterns/factory_method)

**Discussion**

Factory Method is to creating objects as Template Method is to implementing an algorithm. A superclass specifies all standard and generic behavior (using pure virtual "placeholders" for creation steps), and then delegates the creation details to subclasses that are supplied by the client.

Factory Method makes a design more customizable and only a little more complicated. Other design patterns require new classes, whereas Factory Method only requires a new operation.

People often use Factory Method as the standard way to create objects; but it isn't necessary if: the class that's instantiated never changes, or instantiation takes place in an operation that subclasses can easily override (such as an initialization operation).

Factory Method is similar to Abstract Factory but without the emphasis on families.

Factory Methods are routinely specified by an architectural framework, and then implemented by the user of the framework