Factory Method [GoF]

Intent

Given an interface for creating an object, but let subclasses decide which class to instantiate, then subclasses use factory methods. The *factory method* pattern lets a class defer instantiation to subclasses.

Also Known As

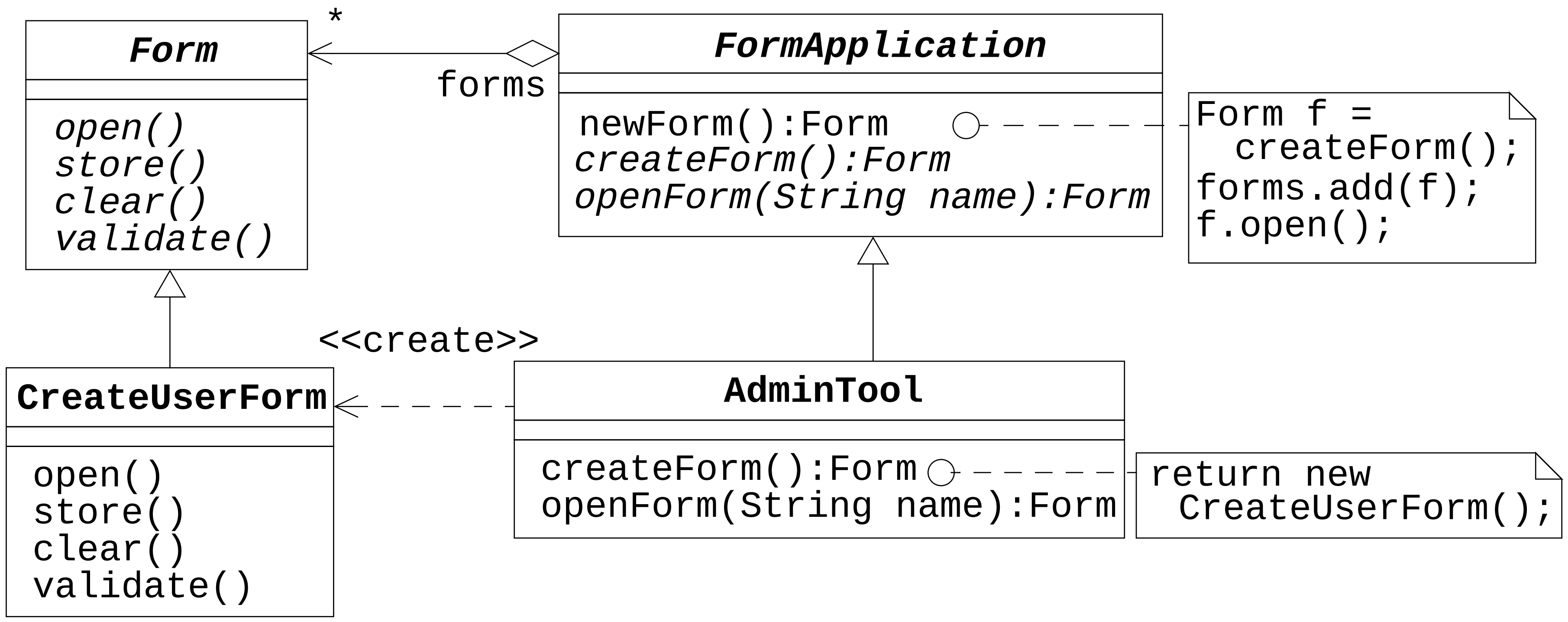
Virtual constructor

Motivation

Frameworks use abstract classes to define and maintain relationships between objects. A framework is often responsible for creating these objects a well.

Abstract base classes define an interface used to return new instantiations of a specific type of an object which is a subclass from another base class. The concrete type of the object returned by the *factory method* is not known by the base class.

Consider a framework for creating and maintaining forms:



Here, key abstractions are: `Form` and `FormApplication`. Both classes are abstract, and framework programmers have to subclass them, for example by providing classes `CreateUserForm` and `AdminTool`. The `FormApplication` class is responsible for managing forms and will create them as required — when the user selects “Open...” or “New...” from a menu, for example.

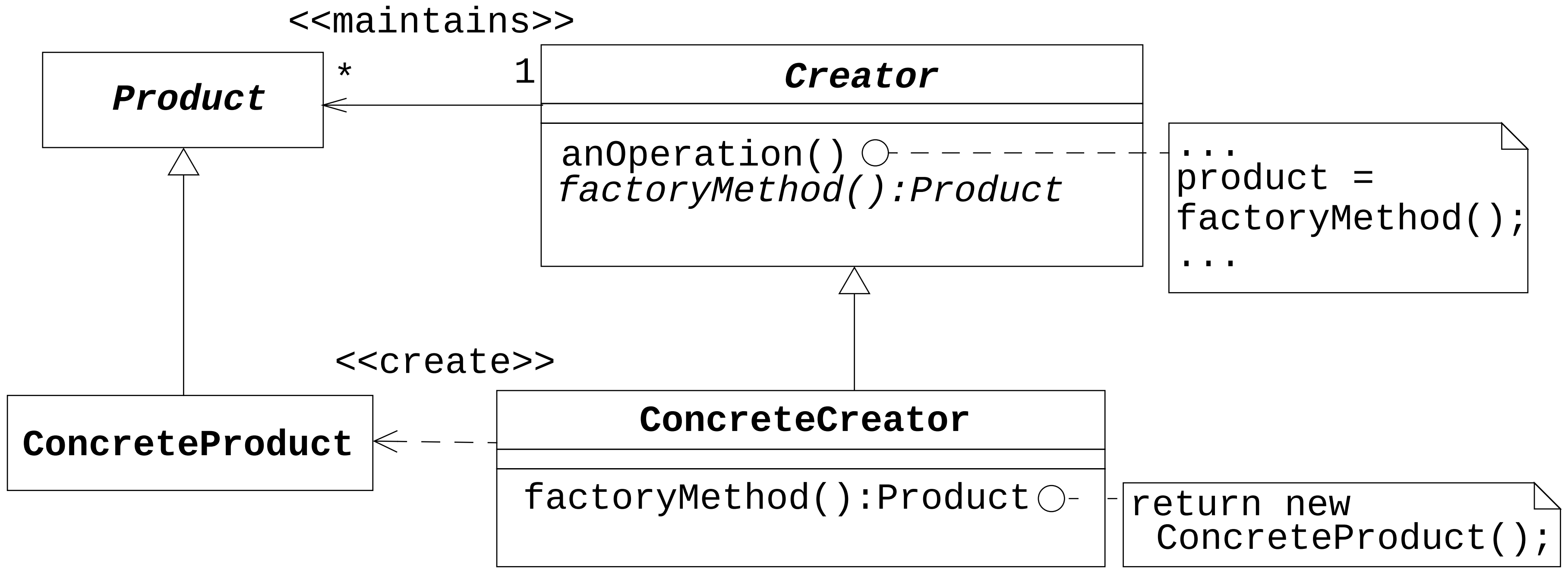
`FormApplication` does not know *what kind* of `Form` has to be created — it only knows *when* a new `Form` should be created. The Factory Method pattern provides a solution to this dilemma.

Applicability

Use the Factory Method pattern

- A class can’t anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.

Structure


<<maintains>>
Product
*
1
Creator
anOperation()
factoryMethod():Product
...
product =
factoryMethod();
...
<<create>>
ConcreteProduct
ConcreteCreator
factoryMethod():Product
return new
ConcreteProduct();

Participants

- **`Product`** (`Form`):
  - defines the interface of objects the factory method creates
- **`ConcreteProduct`** (`CreateUserForm`):
  - implements the `Product` interface
- **`Creator`** (`FormApplication`):
  - declares the factory method, which returns an object of type `Product`. `Creator` may also define a default implementation of the factory method that returns a default `ConcreteProduct` object.
  - may call the factory method to create a product object.
- **`ConcreteCreator`** (`AdminTool`):
  - overrides the factory method to return an instance of `ConcreteProduct`

Collaborations

- `Creator` relies on its subclasses to return instances of the appropriate `ConcreteProduct` object.

Consequences

Factory methods eliminate the need to bind application-specific code into your framework. This mechanism allow you to “plug in” subclasses (at so-called *hooks*) that have knowledge about class implementations of which the framework is unaware.

Through polymorphic mechanisms, the framework classes can still interact with subclass implementations. Interactions occur within the framework between its native base classes, even though the instances themselves are application-specific subclasses.

Implementation

- **Two varieties.** (1) The `Creator` class is abstract and does not an implementation for the factory method; (2) `Creator` is concrete and provides a default implementation.
- **Parameterized factory methods.** The factory method takes a parameter that identifies the kind of object to create. In Java, this parameter could be the name of a class or a `Class` object. All objects the factory method creates will share a common interface.
- **Naming convention.** It’s a good practice to use a naming convention such as `Product createProduct()`.

Sample Code

We use a parameterized factory method in our example. An application is able to present different kinds of forms to the user. The exact type of the form, however, is fixed at run-time. For example, the form’s type specification (in our example, a `Class` object) could be obtained dynamically, e.g., from a server. (In Java, its also possible to *load* the corresponding class dynamically from a remote server.)

First, let’s specify a common abstraction of the application’s forms:

```
public abstract class EntryForm {
    public abstract void open();
    public abstract void close();
    public abstract void validate();
    public abstract void clear();
}
```

A creator class is responsible (with the help of a concrete subclass) to create instances of subclasses of `EntryForm`. The abstraction of the creator class is given in class `FormCreator`:

```
public abstract class FormCreator {
    public abstract EntryForm createEntryForm(Class c)
        throws Exception;
}
```

Notice the parameter of type `Class` in the factory method. It will be used in concrete subclasses of `FormCreator` (in our example there will be only one) to instantiate a concrete `EntryForm` object. Since the instantiation mechanism to be use may raise exceptions we must add a corresponding `throws` clause to the method's signature.

Let’s provide two classes implementing (in part) a form a user can fill out. Imaging that the difference of the forms is the number and kind of fields, etc.

```
public class TaskForm extends EntryForm {
   public void open() { /* Code omitted. */ }
   public void close() { /* Code omitted. */ }
   public void validate() { /* Code omitted. */ }
   public void clear() { /* Code omitted. */ }
}
public class AppointmentForm extends EntryForm {
   public void open() { /* Code omitted. */ }
   public void close() { /* Code omitted. */ }
   public void validate() { /* Code omitted. */ }
   public void clear() { /* Code omitted. */ }
}
```

A concrete subclass able to create any kind of forms of base type `EntryForm` might look like:

```
public class EntryFormCreator extends FormCreator {
   public EntryForm createEntryForm(Class c) throws Exception {
      return (EntryForm) c.newInstance();
}  }
```

Notice the required downcast in the return's expression. If any of the two kinds of `newInstance`'s `InstantiationException` or `IllegalAccessException` are raised, or if the downcast raises a `ClassCastException` since the `Class` object passed as argument does not denote a concrete subclass of `EntryForm`, then the exception is returned to the caller of the method.

A client instantiates a `EntryFormCreator` object. Later, the client will also receive appropriate `Class` objects for instantiation concrete forms. In the code below, a main class simulates this fact by choosing randomly a `Class` object from an array:

```
public class FactoryMethod {
   // array containing class objects:
   private static Class[] formClasses = {
      TaskForm.class,
      AppointmentForm.class
   };

   // to be continued...
```

```
        public static void main(String[] args) {
            int index = (int) (Math.random() * formClasses.length);
            EntryForm currentForm = null;
            try {
                FormCreator fc = new EntryFormCreator();
                currentForm = fc.createEntryForm(formClasses[index]);
                currentForm.open();
                currentForm.validate();
                currentForm.clear();
                currentForm.close();
            } catch (Exception ex) {
                System.out.println("Could not create form.");
                System.exit(1);
            }
        }
    }
```

Related Patterns

Abstract Factory is often implemented with factory methods.

Factory methods are usually called within template methods.