Software Reuse

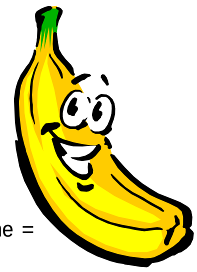
Design Pattern

Peter Sturm
Universität Trier

Analogie

- Erlernen einer Fremdsprache
  - Schritt 1: Rechtschreibung, Grammatik, Wortschatz, Semantik
  - Schritt 2: Vertiefung
    - Bildung vernünftiger Sätze, Absätze, ...
    - Frei Reden und Schreiben
    - Aufsätze, Gedichte, Romane, Pressebericht
  - Schritt 3: Meisterprüfung (= Alltag)
- Erlernen einer Programmiersprache
  - Schritt 1: Syntax, Schlüsselwörter, Semantik
  - Schritt 2: Erste while-Schleife, ...,
    Algorithmen und Datenstrukturen
  - Schritt 3: Ernsthafte Softwareentwicklung

Banane =

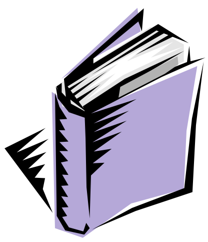


Design Pattern

- Entwurfsmuster
  - wesentliches Werkzeug für Schritt 2 und 3
  - Explizite Formulierung ist neu
  - Pattern-Kataloge
- Nachteile
  - Erlernen der Patterns (Eindeutigkeit)
- Vorteile
  - Experten bieten Lösungen für relevante Pattern
  - Qualitativ bessere Software
    - Weniger Fehler, Effizienz, Anpassbarkeit, Reuse, ...
  - Höherer Abstraktionsgrad in Diskussionen

Der Basiskatalog

- Pattern-Sammlungen
  - Allgemein
  - Spezifisch für Anwendungsfeld
- DAS Werk schlechthin:
  - Design Patterns
Elements of Reusable Object-Oriented Software
E. Gamma, R. Helm, R. Johnson, J. Vlissides
Addison-Wesley
  - Auch als CD erhältlich
- 23 Patterns
- Fester Katalogaufbau



Beispiel

- Factory Pattern
- Katalogstruktur
  - Name, Classification
  - Intent
  - Also Known As
  - Motivation
  - Applicability
  - Structure
  - Participants
  - Collaborations
  - Consequences
  - Implementation
  - Sample Code
  - Known Uses
  - Related Pattern

Factory Method — Class Creational

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As

Virtual Constructor

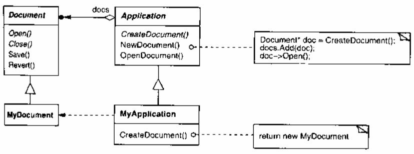
Motivation

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

Consider a framework for applications that can present multiple documents to the user. Two key abstractions in this framework are the classes Application and Document. Both classes are abstract, and clients have to subclass them to realize their application-specific implementations. To create a drawing application, for example, we define the classes DrawingApplication and DrawingDocument. The Application class is responsible for managing Documents and will create them as required—when the user selects Open or New from a menu, for example.

Because the particular Document subclass to instantiate is application-specific, the Application class can't predict the subclass of Document to instantiate—the Application class only knows *when* a new document should be created, not *what kind* of Document to create. This creates a dilemma: The framework must instantiate classes, but it only knows about abstract classes, which it cannot instantiate.

The Factory Method pattern offers a solution. It encapsulates the knowledge of which Document subclass to create and moves this knowledge out of the framework.



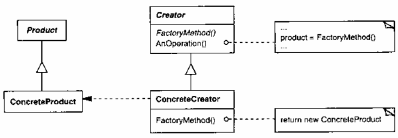
Application subclasses redefine an abstract CreateDocument operation on Application to return the appropriate Document subclass. Once an Application subclass is instantiated, it can then instantiate application-specific Documents without knowing their class. We call CreateDocument a **factory method** because it's responsible for "manufacturing" an object.

Applicability

Use the Factory Method pattern when

- a class can't anticipate the class of objects it must create.
- a class wants its subclasses to specify the objects it creates.
- classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure



Participants

- **Product** (Document)
  - defines the interface of objects the factory method creates.
- **ConcreteProduct** (MyDocument)
  - implements the Product interface.
- **Creator** (Application)
  - 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** (MyApplication)
  - overrides the factory method to return an instance of a ConcreteProduct.

Collaborations

- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct.

Consequences

Factory methods eliminate the need to bind application-specific classes into your code. The code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct classes.

A potential disadvantage of factory methods is that clients might have to subclass the Creator class just to create a particular ConcreteProduct object. Subclassing is fine when the client has to subclass the Creator class anyway, but otherwise the client now must deal with another point of evolution.

Here are two additional consequences of the Factory Method pattern:

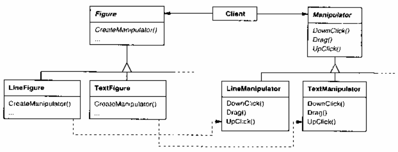
1. *Provides hooks for subclasses.* Creating objects inside a class with a factory method is always more flexible than creating an object directly. Factory Method gives subclasses a hook for providing an extended version of an object.

   In the Document example, the Document class could define a factory method called CreateFileDialog that creates a default file dialog object for opening an existing document. A Document subclass can define an application-specific file dialog by overriding this factory method. In this case the factory method is not abstract but provides a reasonable default implementation.

2. *Connects parallel class hierarchies.* In the examples we've considered so far, the factory method is only called by Creators. But this doesn't have to be the case; clients can find factory methods useful, especially in the case of parallel class hierarchies.

   Parallel class hierarchies result when a class delegates some of its responsibilities to a separate class. Consider graphical figures that can be manipulated interactively; that is, they can be stretched, moved, or rotated using the mouse. Implementing such interactions isn't always easy. It often requires storing and updating information that records the state of the manipulation at a given time. This state is needed only during manipulation; therefore it needn't be kept in the figure object. Moreover, different figures behave differently when the user manipulates them. For example, stretching a line figure might have the effect of moving an endpoint, whereas stretching a text figure may change its line spacing.

   With these constraints, it's better to use a separate Manipulator object that implements the interaction and keeps track of any manipulation-specific state that's needed. Different figures will use different Manipulator subclasses to handle particular interactions. The resulting Manipulator class hierarchy parallels (at least partially) the Figure class hierarchy:



The Figure class provides a CreateManipulator factory method that lets clients create a Figure's corresponding Manipulator. Figure subclasses override this method to return an instance of the Manipulator subclass that's right for them. Alternatively, the Figure class may implement CreateManipulator to return a default Manipulator instance, and Figure subclasses may simply inherit that default. The Figure classes that do so need no corresponding Manipulator subclass—hence the hierarchies are only partially parallel.

Notice how the factory method defines the connection between the two class hierarchies. It localizes knowledge of which classes belong together.

Implementation

Consider the following issues when applying the Factory Method pattern:

1. *Two major varieties.* The two main variations of the Factory Method pattern are (1) the case when the Creator class is an abstract class and does not provide an implementation for the factory method it declares, and (2) the case when the Creator is a concrete class and provides a default implementation for the factory method. It's also possible to have an abstract class that defines a default implementation, but this is less common.

   The first case *requires* subclasses to define an implementation, because there's no reasonable default. It gets around the dilemma of having to instantiate unforeseeable classes. In the second case, the concrete Creator uses the factory method primarily for flexibility. It's following a rule that says, "Create objects in a separate operation so that subclasses can override the way they're created." This rule ensures that designers of subclasses can change the class of objects their parent class instantiates if necessary.

2. *Parameterized factory methods.* Another variation on the pattern lets the factory method create *multiple* kinds of products. The factory method takes a

parameter that identifies the kind of object to create. All objects the factory method creates will share the Product interface. In the Document example, Application might support different kinds of Documents. You pass CreateDocument an extra parameter to specify the kind of document to create.

The Unidraw graphical editing framework [VL90] uses this approach for reconstructing objects saved on disk. Unidraw defines a `Creator` class with a factory method `Create` that takes a class identifier as an argument. The class identifier specifies the class to instantiate. When Unidraw saves an object to disk, it writes out the class identifier first and then its instance variables. When it reconstructs the object from disk, it reads the class identifier first.

Once the class identifier is read, the framework calls `Create`, passing the identifier as the parameter. `Create` looks up the constructor for the corresponding class and uses it to instantiate the object. Last, `Create` calls the object's `Read` operation, which reads the remaining information on the disk and initializes the object's instance variables.

A parameterized factory method has the following general form, where `MyProduct` and `YourProduct` are subclasses of `Product`:

```
class Creator {
public:
    virtual Product* Create(ProductId);
};

Product* Creator::Create (ProductId id) {
    if (id == MINE)  return new MyProduct;
    if (id == YOURS) return new YourProduct;
    // repeat for remaining products...

    return 0;
}
```

Overriding a parameterized factory method lets you easily and selectively extend or change the products that a Creator produces. You can introduce new identifiers for new kinds of products, or you can associate existing identifiers with different products.

For example, a subclass `MyCreator` could swap MyProduct and YourProduct and support a new `TheirProduct` subclass:

```
Product* MyCreator::Create (ProductId id) {
    if (id == YOURS)  return new MyProduct;
    if (id == MINE)   return new YourProduct;
        // N.B.: switched YOURS and MINE

    if (id == THEIRS) return new TheirProduct;

    return Creator::Create(id); // called if all others fail
}
```

Notice that the last thing this operation does is call `Create` on the parent class. That's because `MyCreator::Create` handles only `YOURS`, `MINE`, and

`THEIRS` differently than the parent class. It isn't interested in other classes. Hence `MyCreator` *extends* the kinds of products created, and it defers responsibility for creating all but a few products to its parent.

3. *Language-specific variants and issues.* Different languages lend themselves to other interesting variations and caveats.

   Smalltalk programs often use a method that returns the class of the object to be instantiated. A Creator factory method can use this value to create a product, and a ConcreteCreator may store or even compute this value. The result is an even later binding for the type of ConcreteProduct to be instantiated.

   A Smalltalk version of the Document example can define a `documentClass` method on `Application`. The `documentClass` method returns the proper `Document` class for instantiating documents. The implementation of `documentClass` in `MyApplication` returns the `MyDocument` class. Thus in class `Application` we have

```
clientMethod
    document := self documentClass new.

documentClass
    self subclassResponsibility
```

   In class `MyApplication` we have

```
documentClass
    ^ MyDocument
```

   which returns the class `MyDocument` to be instantiated to `Application`.

   An even more flexible approach akin to parameterized factory methods is to store the class to be created as a class variable of `Application`. That way you don't have to subclass `Application` to vary the product.

   Factory methods in C++ are always virtual functions and are often pure virtual. Just be careful not to call factory methods in the Creator's constructor—the factory method in the ConcreteCreator won't be available yet.

   You can avoid this by being careful to access products solely through accessor operations that create the product on demand. Instead of creating the concrete product in the constructor, the constructor merely initializes it to 0. The accessor returns the product. But first it checks to make sure the product exists, and if it doesn't, the accessor creates it. This technique is sometimes called **lazy initialization**. The following code shows a typical implementation:


```
class Creator {
public:
    Product* GetProduct();
protected:
    virtual Product* CreateProduct();
private:
    Product* _product;
};

Product* Creator::GetProduct () {
    if (_product == 0) {
        _product = CreateProduct();
    }
    return _product;
}
```

4. *Using templates to avoid subclassing.* As we've mentioned, another potential problem with factory methods is that they might force you to subclass just to create the appropriate Product objects. Another way to get around this in C++ is to provide a template subclass of Creator that's parameterized by the Product class:

```
class Creator {
public:
    virtual Product* CreateProduct() = 0;
};

template <class TheProduct>
class StandardCreator: public Creator {
public:
    virtual Product* CreateProduct();
};

template <class TheProduct>
Product* StandardCreator<TheProduct>::CreateProduct () {
    return new TheProduct;
}
```

   With this template, the client supplies just the product class—no subclassing of Creator is required.

```
class MyProduct : public Product {
public:
    MyProduct();
    // ...
};

StandardCreator<MyProduct> myCreator;
```

5. *Naming conventions.* It's good practice to use naming conventions that make it clear you're using factory methods. For example, the MacApp Macintosh application framework [App89] always declares the abstract operation that defines the factory method as `Class* DoMakeClass()`, where `Class` is the Product class.

Sample Code

The function CreateMaze (page 84) builds and returns a maze. One problem with this function is that it hard-codes the classes of maze, rooms, doors, and walls. We'll introduce factory methods to let subclasses choose these components.

First we'll define factory methods in MazeGame for creating the maze, room, wall, and door objects:

```
class MazeGame {
public:
    Maze* CreateMaze();

// factory methods:

    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

Each factory method returns a maze component of a given type. MazeGame provides default implementations that return the simplest kinds of maze, rooms, walls, and doors.

Now we can rewrite CreateMaze to use these factory methods:

```
Maze* MazeGame::CreateMaze () {
    Maze* aMaze = MakeMaze();

    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());

    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);
```



```
    return aMaze;
}
```

Different games can subclass MazeGame to specialize parts of the maze. MazeGame subclasses can redefine some or all of the factory methods to specify variations in products. For example, a BombedMazeGame can redefine the Room and Wall products to return the bombed varieties:

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();

    virtual Wall* MakeWall() const
        { return new BombedWall; }

    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

An EnchantedMazeGame variant might be defined like this:

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();

    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }

    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

Known Uses

Factory methods pervade toolkits and frameworks. The preceding document example is a typical use in MacApp and ET++ [WGM88]. The manipulator example is from Unidraw.

Class View in the Smalltalk-80 Model/View/Controller framework has a method defaultController that creates a controller, and this might appear to be a factory method [Par90]. But subclasses of View specify the class of their default controller by defining defaultControllerClass, which returns the class from which defaultController creates instances. So defaultControllerClass is the real factory method, that is, the method that subclasses should override.

A more esoteric example in Smalltalk-80 is the factory method parserClass defined by Behavior (a superclass of all objects representing classes). This enables a class

to use a customized parser for its source code. For example, a client can define a class SQLParser to analyze the source code of a class with embedded SQL statements. The Behavior class implements parserClass to return the standard Smalltalk Parser class. A class that includes embedded SQL statements overrides this method (as a class method) and returns the SQLParser class.

The Orbix ORB system from IONA Technologies [ION94] uses Factory Method to generate an appropriate type of proxy (see Proxy (207)) when an object requests a reference to a remote object. Factory Method makes it easy to replace the default proxy with one that uses client-side caching, for example.

Related Patterns

Abstract Factory (87) is often implemented with factory methods. The Motivation example in the Abstract Factory pattern illustrates Factory Method as well.

Factory methods are usually called within Template Methods (325). In the document example above, NewDocument is a template method.

Prototypes (117) don't require subclassing Creator. However, they often require an Initialize operation on the Product class. Creator uses Initialize to initialize the object. Factory Method doesn't require such an operation.

Die 23 Grundmuster

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Builder	Bridge	Command
		Prototype	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Creational Purpose

- Class
  - *Factory Method:* Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses
- Object
  - *Abstract Factory:* Provide an interface for creating families of related or dependent objects without specifying their concrete class.
  - *Builder:* Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - *Prototype:* Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - *Singleton:* Ensure a class only has one instance, and provide a global point of access to it.

Structural Purpose

- Class
  - *Adapter:* Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Object
  - *Adapter:* see above
  - *Bridge:* Decouple an abstraction from its implementation so that the two can vary independently.
  - *Composite:* Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
  - *Decorator:* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality

Structural Purpose (contd.)

- *Facade:* Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- *Flyweight:* Use sharing to support large numbers of fine-grained objects efficiently.
- *Proxy:* Provide a surrogate or placeholder for another object to control access to it.

Behavioral Purpose

- Class
  - Interpreter: Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
  - Template Method: Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- Object
  - Chain of Responsibility: Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
  - Command: Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Behavioral Purpose (contd.)

- Iterator: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
- Mediator: Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.
- Memento: Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.
- Observer: Define an one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- State: Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Behavioral Purpose (contd.)

- Strategy: Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Visitor: Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Beispiel: Decorator (Wrapper)

- Motivation
  - Zusätzliche bzw. veränderte Funktionalität für ein Objekt zur Laufzeit
  - Alternative zu Subclassing
  - “Minimal invasive” Eingriffe bei Reuse