Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 4: Factory Method design pattern

Contents

4.1 Introduction

This chapter will introduce the Factory Method design pattern. The pattern provides a structure whereby the creation of objects is delegated to subclasses in such doing not needing to specify the class that the object belongs to.

4.2 Programming preliminaries

When an object is instantiated (created), a constructor is called. If a constructor, with the specific parameter list, is not defined for the class the default constructor is called. When an object goes out of scope (deleted, destroyed), the destructor is called. The behaviour exhibited by the constructor and destructor is to handle memory and attributes. The constructor allocates memory for attributes and assigns values to attributes, while the destructor must release any memory that has been assigned during the life-time of the object [1].

In Lecture Notes L04 - Template Method Pattern, constructors were used. The lecture note showed how to make use of member-list initialisation in order to assign values to the attributes of the class. It also illustrated the use of the member-list initialisation technique to call the constructor of the base class in an inheritance relationship. In this section, the concepts of object construction, destruction and initialisation will be explained in more detail.

4.2.1 Constructors

The constructor is a member function defined in the class. The name of the constructor is the same as that of the class. The constructor can take parameters, but does not have a return type. A constructor that does not take any parameters is called the default constructor. If a constructor, or the default constructor, has not been defined for the class, the compiler will automatically generate a default constructor so that objects of the class can be created.

Constructors are used to initialise class member variables (attributes) and other setup-type requirements for the object. Initialising of member variables can be done either in the body of the class or in the member-list initialisation of the constructor. The choice between body or member-list is simple, variables that do not require memory to be allocated on the heap can be initialised using the member-list, otherwise they should be initialised in the body. Superclass constructors must be called in the member-list in order for any superclass member variables to be initialised in a controlled manner.

4.2.2 Destructors

The job of a destructor is to release any memory that the object might have acquired during its lifetime. As with constructors, if a destructor has not been explicitly defined, the compiler will define a default destructor for the class. Unlike constructors, only one destructor is needed per class.

The name of the destructor is the same as the class and it takes no parameters. To distinguish the destructor from the default constructor, a destructor is define with a tilde (∼) before its member function name.

4.2.3 An example

Let us revisit the Employee example given in L04 - Template Method Pattern and apply the understanding we have gained with regards to constructors and destructors.

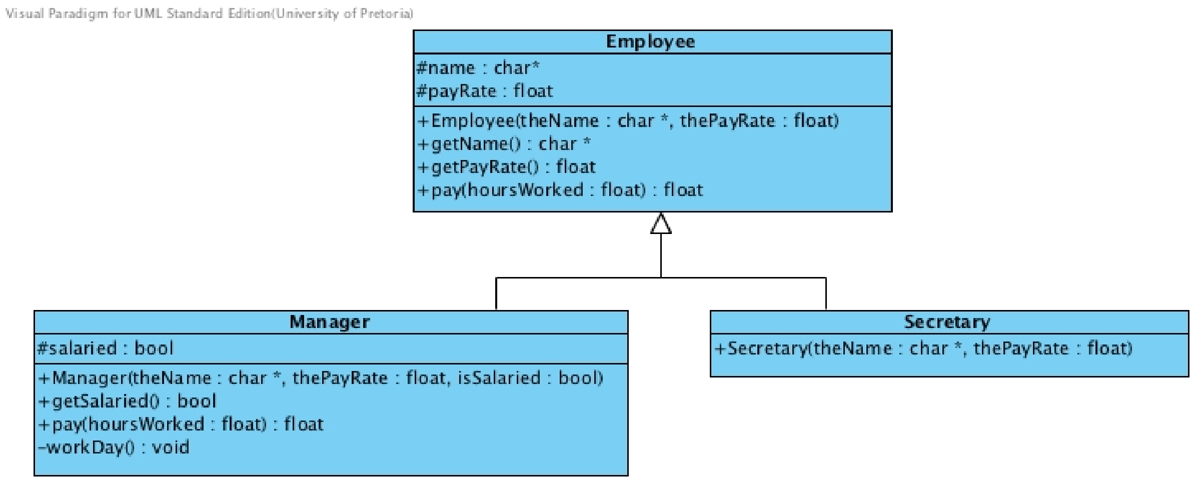
Visual Paradigm for UML Standard Edition(University of Pretoria)
Employee
#name : char*
#payRate : float
+Employee(theName : char *, thePayRate : float)
+getName() : char *
+getPayRate() : float
+pay(hoursWorked : float) : float
Manager
#salaried : bool
+Manager(theName : char *, thePayRate : float, isSalaried : bool)
+getSalaried() : bool
+pay(hoursWorked : float) : float
-workDay() : void
Secretary
+Secretary(theName : char *, thePayRate : float)

Figure 1: Employee class diagram as presented in L04

The constructor parameter that is going to give the most problems in `theName` in `Employee`. `theName` would have been allocated on the heap by the client. Merely assigning `theName` to `name` will result in `name` pointing to the same memory location as what `theName` does. Any changes made to the memory being pointed to will result in both variables, the one in the client and the one in the `Employee` hierarchy, changing value. In order to make sure that this does not occur, the class attribute `name` needs independent memory allocated to it. The following implementation of the constructor does just this.

```
Employee :: Employee(char* theName,
                     float thePayRate) : payRate(thePayRate)
{
    name = new char[strlen(theName)+1];
    strcpy(name, theName);
}
```

Having allocated the memory in the class, the class needs to take responsibility to delete the memory when it goes out of scope. It is therefore necessary to define the default constructor in the class and provide the implementation for it. The class definition will include the following as a public member:

```
  ~Employee();
```

The implementation of the destructor follows:

```
Employee::~Employee(){
    delete [] name;
}
```

4.3 Factory Method Pattern

4.3.1 Identification

Name	Classification	Strategy
Factory Method	Creational	Inheritance (Class)
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. ([2]:107)		

4.3.2 Structure

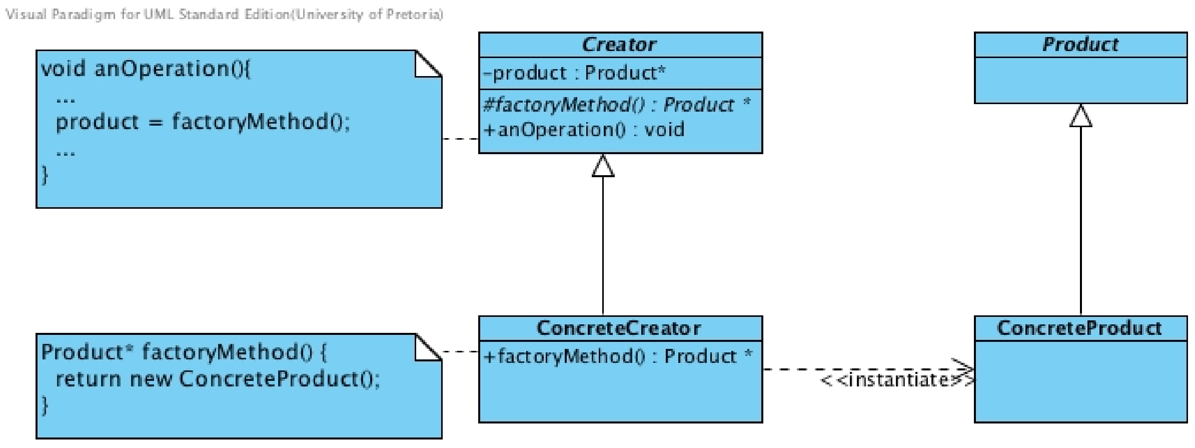
Visual Paradigm for UML Standard Edition(University of Pretoria)
void anOperation(){
...
product = factoryMethod();
...
}
Creator
-product : Product*
#factoryMethod() : Product *
+anOperation() : void
Product
Product* factoryMethod() {
return new ConcreteProduct();
}
ConcreteCreator
+factoryMethod() : Product *
<<instantiate>>
ConcreteProduct

Figure 2: The structure of the Factory Method Pattern

4.3.3 Problem

The factory method essentially wraps the class construction into an operation with a descriptive name and requires the concrete creator to make the decision as to which product constructor is going to be called consequently resulting in the creation of the product. The pattern solves the problem of having a tight coupling between objects that create product and the product objects themselves.

4.3.4 Participants

Creator

- declares the factory method which returns a product object
- default factory method implementations may return a default concrete product

ConcreteCreator

- overrides the factory method to return an instance of the product

Product

- defines the product interface for the factory method to create

ConcreteProduct

- implements the interface for the product

4.4 Factory Method Pattern Explained

4.4.1 Clarification

The creator is not sure what class of product is to be created and delegates this responsibility to it subclasses. It is the responsibility of the the concrete creator classes to create specific product. This results in the parallel hierarchies of Creator and Product with the dependencies between the hierarchies on the concrete level, ConcreteCreator uses a ConcreteProduct.

4.4.2 Code improvements achieved

Objects are created in an orderly fashion. Central management of the object creation process exists. This could be used with great effect to control the life-time of the object and ensure that as with creation, deletion is also conducted in an orderly way.

4.4.3 Implementation Issues

The factory method can be implemented either by defining the creator class as abstract or as concrete. In the case of an abstract creator class, a concrete creator per product must be defined or the concrete creator needs to be parameterised to produce the correct concrete product. When the creator class is defined as a concrete class it must provide default implementations for all operations it defines.

4.4.4 Common Misconceptions

Using only a wrapper with a descriptive name for the construction process [3], does not mean that a Factory Method design pattern has been used. Consider the `ComplexNumber` class in Figure 3 that participated as the originator in the Memento pattern described in L03. There is no distinction in this class with regards to cartesian or polar coordinates and implementing a constructor that can distinguish between these is not feasible as an extra parameter will be required to make the distinction. The best would be to provide a public operation, with a descriptive name, that indicates the co-ordinate system being used as parameters for the creation of the object which returns an instance of `ComplexNumber`.

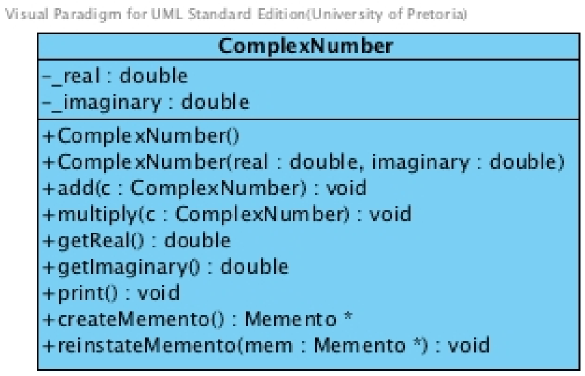
Visual Paradigm for UML Standard Edition(University of Pretoria)
ComplexNumber
-_real : double
-_imaginary : double
+ComplexNumber()
+ComplexNumber(real : double, imaginary : double)
+add(c : ComplexNumber) : void
+multiply(c : ComplexNumber) : void
+getReal() : double
+getImaginary() : double
+print() : void
+createMemento() : Memento *
+reinstateMemento(mem : Memento *) : void

Figure 3: UML class diagram of ComplexNumber

A suggestion for the implementation of these co-ordinate specific operations is given below. The visibility of the constructor of the class that accepts two doubles as parameters can be changed to protected in order to ensure that it is not inadvertently called with an incorrect co-ordinate system.

```
ComplexNumber* ComplexNumber::fromCartesian(double real, double imaginary)
{
  return new ComplexNumber(real,imaginary);
}

ComplexNumber* ComplexNumber::fromPolar(double modulus, double angle)
{
  return new ComplexNumber(modulus*cos(angle),modulus*sin(angle));
}
```

In order for this to be considered as an implementation of a Factory Method design pattern, the `ComplexNumber` class needs to inherit from an abstract class, say `Number`. The determining of the type of co-ordinate system should be left to the concrete creator which forms part of the parallel factory hierarchy that needs to be defined. Refer to the example given in section 4.5 for a suggestion to implement the `ComplexNumber` class as a product of the Factory Method design pattern.

4.4.5 Related Patterns

Template Method
The Factory Method may make use of Template Method in both the Product and the Creator hierarchies.

Abstract Factory
The Factory Method may be used in the implementation of the Abstract Factory design pattern.

Prototype
Factory Methods can be used to initialise prototypical objects. The prototype also can be used instead of the factory method to avoid large parallel hierarchies.

Singleton
In only one instance of a concrete factory is required, the concrete factory can be made a Singleton.

4.5 Example

This example can be combined with the Memento example given in Lecture Note 03. For clarity, all references to the Memento have been removed in order to illustrate only the Factory Method design pattern. Figure 4 shows the relationships between the classes and the structure of each of the classes participating in the Factory Method design pattern.

The corresponding pattern participants for this example are:

- Creator: `NumberGenerator` - an abstract class defining the factory method `generateNumber`. `nextNumber` is a template method operation that forms part of the Template Method design pattern in the example.
- ConcreteCreator: `ComplexNumberGenerator` - produces a `ComplexNumber` product object. The instantiation of the product object is dependent on the co-ordinate system encapsulated in the creator hierarchy.
- Product: `Number` - provides the interface for numbers.
- ConcreteProduct: `ComplexNumber` - defined exactly as it was for the Memento example, except for the removal of the Memento specific operations and the inheritance relationship with the `Number` class.

4.5.1 Implementation notes

To successfully implement the design pattern for the given example, the following should be noted:

Virtual destructor
`NumberGenerator` defines the interface to generate different number types, specifically a complex number in this example. Instantiating an object of `ComplexNumberGenerator`

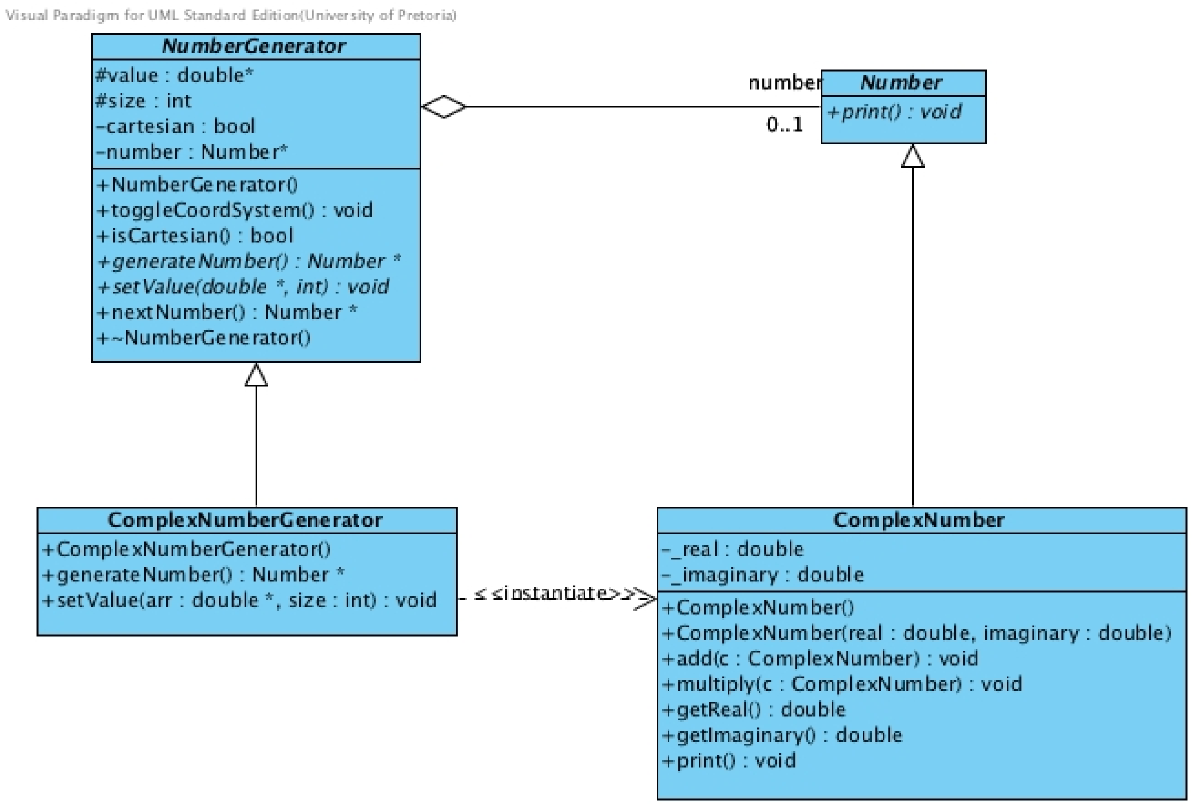
Visual Paradigm for UML Standard Edition(University of Pretoria)
NumberGenerator
#value : double*
#size : int
-cartesian : bool
-number : Number*
+NumberGenerator()
+toggleCoordSystem() : void
+isCartesian() : bool
+generateNumber() : Number *
+setValue(double *, int) : void
+nextNumber() : Number *
+~NumberGenerator()
number
0..1
Number
+print() : void
ComplexNumberGenerator
+ComplexNumberGenerator()
+generateNumber() : Number *
+setValue(arr : double *, size : int) : void
<<instantiate>>
ComplexNumber
-_real : double
-_imaginary : double
+ComplexNumber()
+ComplexNumber(real : double, imaginary : double)
+add(c : ComplexNumber) : void
+multiply(c : ComplexNumber) : void
+getReal() : double
+getImaginary() : double
+print() : void

Figure 4: UML class diagram for the example of the Factory Method design pattern

reserves heap memory that has been defined in the corresponding base class. In order to successfully clear the memory when an object of `ComplexNumberGenerator` goes out of scope, the destructor of the base class `NumberGenerator` must be defined as virtual. This destructor must then deallocate the heap memory defined by it that instantiating classes in the hierarchy would have allocated. The definition and implementation of `NumberGenerator` is given in the listing that follows.

```
class NumberGenerator
{
  public:

    NumberGenerator()
    {
      number = 0;
      cartesian = true;
      value = 0;
      size = 0;
    };

    void toggleCoordSystem()
    {
      cartesian = !cartesian;
    };
    bool isCartesian() {
      return cartesian;
    };

    virtual Number* generateNumber() = 0;
    virtual void setValue(double*,int) = 0;

    Number* nextNumber() {
      number = generateNumber();
      return number;
    };

    virtual ~NumberGenerator()
    {
      if (number != 0) {
        number = 0;
      }
      if (size != 0) {
        delete [] value;
        value = 0;
      }
    }
  protected:
    double* value;
    int size;
  private:
    bool     cartesian;
    Number* number;
};
```

Calling the constructor of the base class

In order to initialise the attributes of `ComplexNumberGenerator` that are defined in the base class, the constructor of the base class must be called by the constructor of the derived class. Implementing this is trivial and can be accomplished by using member-list initialisation. The implementation of the constructor is given by:

```
ComplexNumberGenerator::ComplexNumberGenerator() : NumberGenerator()
{
}
```

Primitive operation implementation

Notice the memory management applied to the array of values derived from the base class in the code for the primitive operations of the template method for the `ComplexNumberGenerator` given below. As complex number is implemented in terms of the cartesian co-ordinate system, it is necessary that the generator for complex number class does the conversion of polar to cartesian.

```
Number* ComplexNumberGenerator::generateNumber()
{
    if (size == 0) {
      value = new double[2];
      value[0] = 0;
      value[1] = 0;
      size = 2;
    }
    if (isCartesian())
      return new ComplexNumber(value[0], value[1]);
    else
      return new ComplexNumber(value[0]*cos(value[1]),
                               value[0]*sin(value[1]));
  };

  void ComplexNumberGenerator::setValue(double* arr, int size) {
    if (this->size != 0) {
      delete [] value;
      this->size = 0;
    }
    value = new double[size];
    value[0] = arr[0];
    value[1] = arr[1];
    this->size = size;
  };
```

4.5.2 Main program

An example of a test program is given. Notice that the responsibility of deleting objects of `Number` is left to the client of the factory method. It is also important to note that the client never directly instantiates an object of `ComplexNumber`, it is the job of the corresponding generator to do so. It is a good habit to ensure that all heap memory is deallocated in the reverse order of allocation.

```
int main()
{
  double* valueList;

  valueList = new double[2];
  valueList[0] = 3;
  valueList[1] = 8;

  NumberGenerator* factory = new ComplexNumberGenerator();

  Number* one = 0;
  Number* two = 0;

  one = factory->nextNumber();
  one->print();
  factory->toggleCoordSystem();

  factory->setValue(valueList,2);
  two = factory->nextNumber();

  one->print();
  two->print();

  delete two;
  delete one;

  delete factory;
  delete [] valueList;

  return 0;
}
```

4.6 Exercises

1. Consider the class diagram presented in Figure 5 and answer the questions that follow:

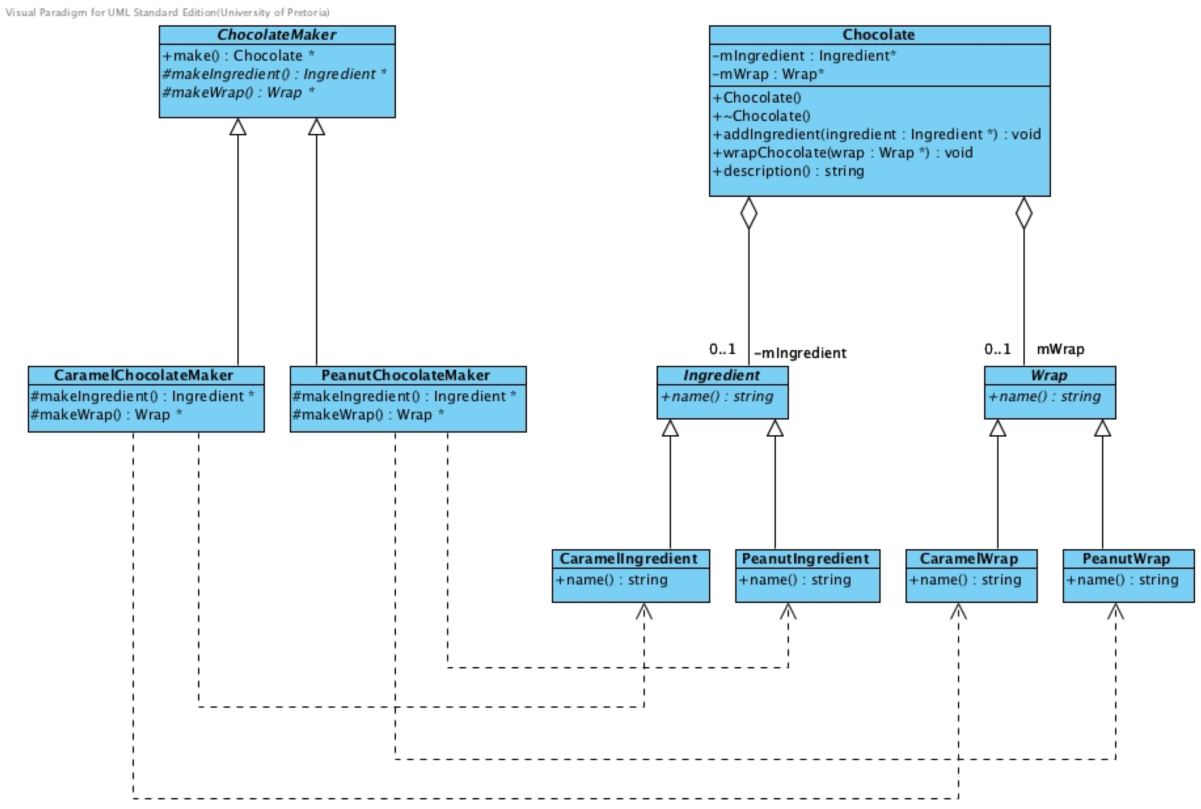

Figure 5: Chocolate Factory

   (a) Identify the participants.
   (b) Has a template method been used in combination with the factory method?
   (c) Write a client program that makes use of this factory method hierarchy.

2. Combine the Memento implementation with the Factory Method implementation discussed here. Hint: You should adapt the memento to internalise the state of any number object.

References

[1] Tony Gaddis. *Starting out with C++: from control structures through objects.* Pearson Education, seventh edition, 2012.

[2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software.* Addison-Wesley, Reading, Mass, 1995.

[3] Wikipedia. Factory method pattern — wikipedia, the free encyclopedia, 2011. URL `http://en.wikipedia.org/w/index.php?title=Factory_method_pattern`. [Online; accessed 10 August 2011].