**Creational Patterns**

# Creational Patterns

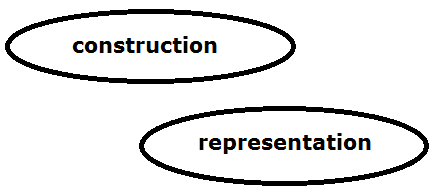
Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Problem with basic object creation:

* could result in design problems or
* added complexity to the design

# Builder

Builder Creational Pattern is used to separate the construction of a complex object from its representation so that the same construction process can create different objects representations.



Builder Creational Pattern

## Problem:

We want to construct a complex object, however we do not want to have a complex constructor member or one that would need many arguments.

## Solution:

Define an intermediate object whose member functions define the desired object part by part before the object is available to the client. Builder Pattern lets us defer the construction of the object until all the options for creation have been specified.

Builder design pattern describes how to solve such problems:

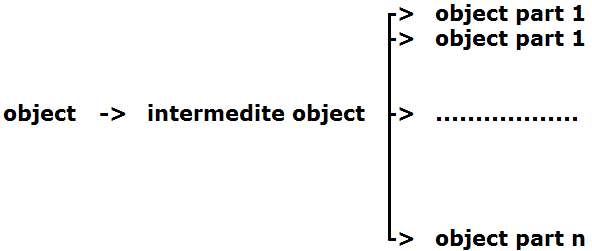
* Encapsulate creating and assembling the parts of a complex object in a separate Builder object.
* A class delegates object creation to a Builder object instead of creating the objects directly.

## Advantages

* Allows you to vary a product’s internal representation
* Encapsulates code for construction and representation
* Provides control over steps of construction process

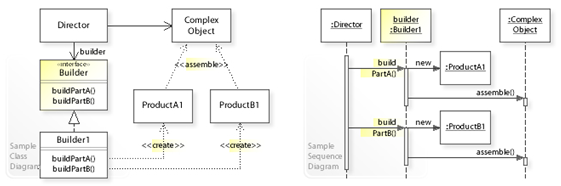
## Disadvantages

* Requires creating a separate ConcreteBuilder for each different type of product
* Requires the builder classes to be mutable
* Data members of class aren't guaranteed to be initialized
* Dependency injection may be less supported

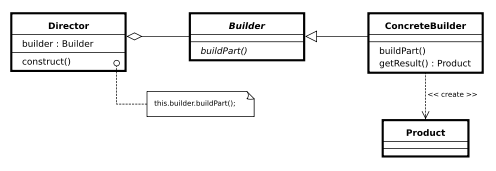


Builder Creational Patterns

## Structure



UML class and sequence diagram for the Builder design pattern



Class diagram

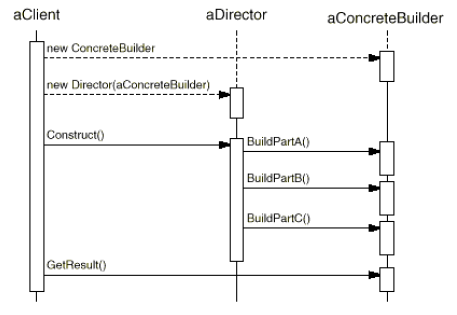
**Builder:** Abstract interface for creating objects (product)

**ConcreteBuilder:** Provides implementation for Builder. It is an object able to construct other objects. Constructs and assembles parts to build the objects

**Director:** constructs an object using the Builder interface

**Product:** represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled

Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

****

Interaction Diagram

## Example:



# Factory

A utility class that creates an instance of a class from a family of derived classes.

The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created.

This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

**Also known as Virtual Constructor**

Define an interface for creating an object, but let subclasses decide which class to instantiate.

Solves problems like:

* How can an object be created so that subclasses can redefine which class to instantiate?
* How can a class defer instantiation to subclasses?

Describes how to solve such problems:

* Define a separate operation (factory method) for creating an object
* Create an object by calling a factory method

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.**

The creation is deferred to run-time.

1. Client knows about abstract base class but not concrete subclass.
2. Run-time creation of objects.
3. Subclasses create objects.

Actually, the factory method is a way of circumventing following limitations of C++ constructor:

**No return type**: A constructor cannot return a result, which means that we cannot signal an error during object initialization. The only way of doing it is to throw an exception from a constructor.

**Naming**: A constructor should have the same name as the class, which means we cannot have two constructors that both take a single argument.

**Compile time bound**: At the time when we create an object, we must specify the name of a concrete class which is known at compile time. There is no way of dynamic binding constructors at run time.

**There is no virtual constructor**: We cannot declare a virtual constructor. If we are constructing derived object, the compiler calls the base class constructor first, and the derived class hasn't been initialized yet. This is the reason why we cannot call virtual methods from the constructor.

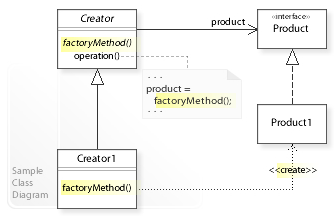
## What's the difference between Abstract Factory Pattern and Factory Method?

* Abstract Factory design pattern creates Factory
* Factory design pattern creates Products

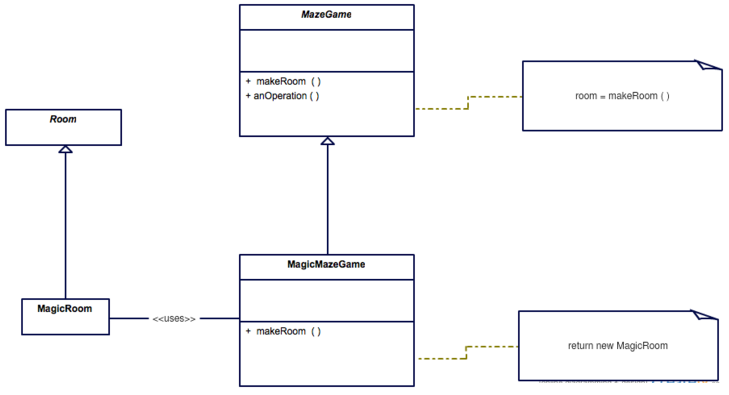
## Factory Method - Abstract Creator Class

## Factory Method - Concrete Creator Class

## Structure



UML class diagram



Class Diagram

## Application

1. In ADO.NET, IDbCommand.CreateParameter is an example of the use of factory method to connect parallel class hierarchies.
2. In Qt, QMainWindow::createPopupMenu is a factory method declared in a framework that can be overridden in application code.
3. In Java, several factories are used in the javax.xml.parsers package. e.g. javax.xml.parsers.DocumentBuilderFactory

or

javax.xml.parsers.SAXParserFactory.

# Abstract Factory

# Prototype

# Singleton

# END