Factory method pattern

In class-based programming, 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.

# Overview

The Factory Method design pattern is one of the "Gang of Four" design patterns that describe how to solve recurring design problems to design flexible and reusable object-oriented software, that is, objects that are easier to implement, change, test, and reuse.

The Factory Method design pattern is used instead of the regular class constructor for keeping within the SOLID principle of programming, decoupling the construction of objects from the objects themselves. This has the following advantages and is useful for the following cases, among others:

* Allows construction of classes with a component of a type that has not been predetermined, but only defined in an "interface", or which is defined as a dynamic type.

Thus, for example, a class Vehicle that has a member Motor of interface IMotor, but no concrete type of Motor defined in advance, can be constructed by telling the Vehicle constructor to use an ElectricMotor or a GasolineMotor. The Vehicle constructor code then calls a Motor factory method, to create the desired Motor that complies with the IMotor interface.

* Allows construction of subclasses to a parent whose component type has not been predetermined, but only defined in an interface, or which is defined as a dynamic type.

For example, a class Vehicle with a member Motor defined with a dynamic type, can have subclasses of type ElectricPlane and OldCar each constructed with a different type of Motor. This can be accomplished by constructing the subclasses with a Vehicle factory method, while supplying the motor type. In cases like this the constructor may be hidden.

* Allows for more readable code in cases where multiple constructors exist, each for a different reason.

For example if there are two constructors Vehicle(make:string, motor:number) and Vehicle(make:string, owner:string, license:number, purchased:date) a more readable construction of the classes would be to use Vehicle.CreateOwnership(make:string, owner:string, license:number, purchased: date) vs Vehicle.Create(make:string, motor:number)

* Allows a class to defer instantiation to subclasses, and to prevent direct instantiation of an object of the parent class type.

For example, a Vehicle can be prevented from being instantiated directly since it has no constructor, and only subclasses like ElectricPlane or OldCar can be created by calling the Vehicle (static) factory method in the subclass constructor or initializer.

Creating an object directly within the class that requires or uses the object is inflexible because it commits the class to a particular object and makes it impossible to change the instantiation independently of the class. A change to the instantiator would require a change to the class code which we would rather not touch. This is referred to as *code coupling* and the Factory method pattern assists in *decoupling* the code.

The Factory Method design pattern is used by first defining a separate operation, a *factory method*, for creating an object, and then using this *factory method* by calling it to create the object. This enables writing of subclasses that decide how a parent object is created and what type of objects the parent contains.  
See the UML class diagram below.

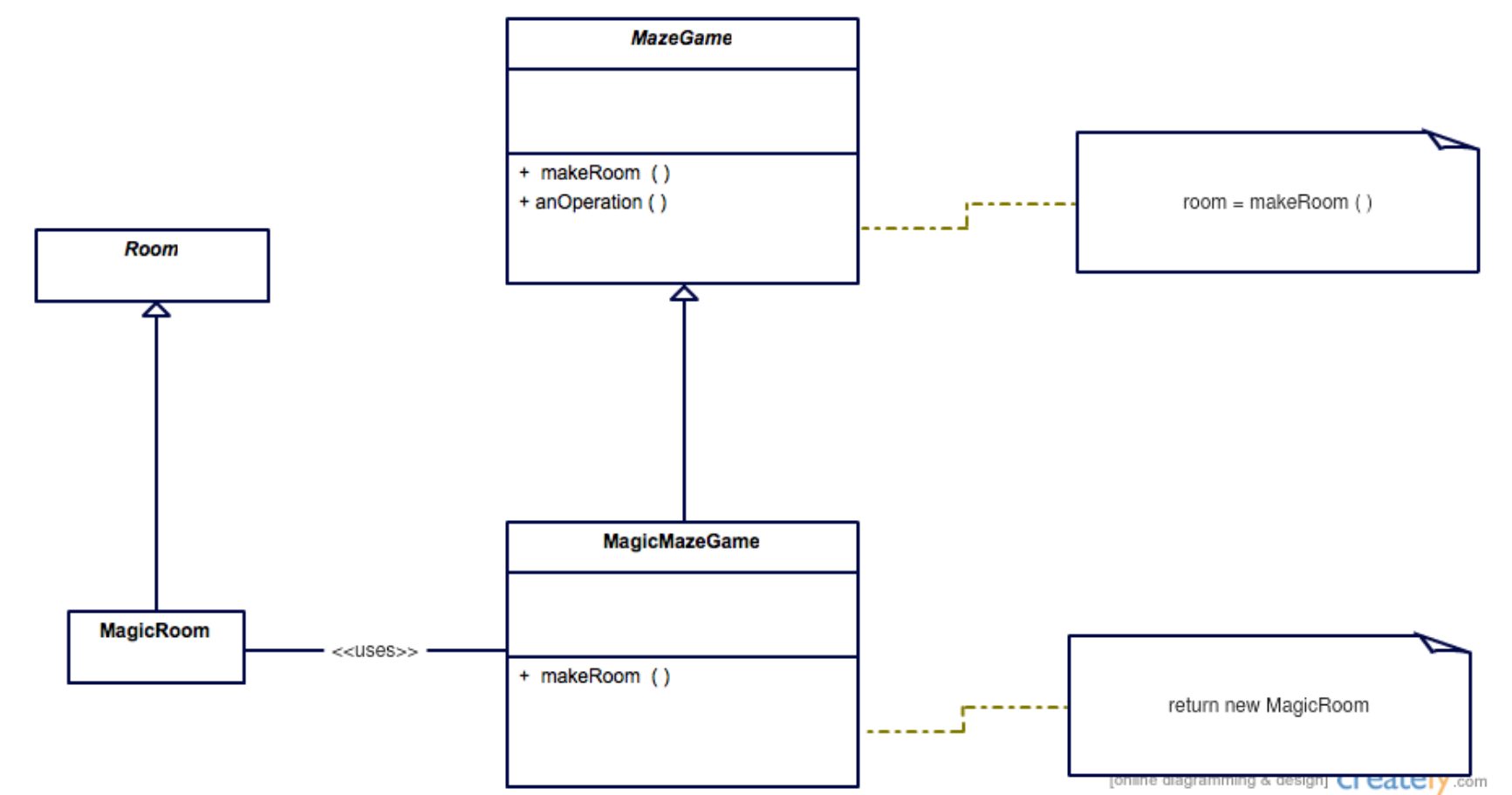
# Definition

"Define an interface for creating an object, but let subclasses decide which class to instantiate. The Factory method lets a class defer instantiation it uses to subclasses." (Gang Of Four)

Creating an object often requires complex processes not appropriate to include within a composing object. The object's creation may lead to a significant duplication of code, may require information not accessible to the composing object, may not provide a sufficient level of abstraction, or may otherwise not be part of the composing object's concerns. The factory method design pattern handles these problems by defining a separate method for creating the objects, which subclasses can then override to specify the derived type of product that will be created.

The factory method pattern relies on inheritance, as object creation is delegated to subclasses that implement the factory method to create objects.

# Structure



Type to enter a caption.

# Implementations in c++

class Room{

public:

virtual void connect(Room \*room) = 0;

};

class MagicRoom : public Room {

public:

MagicRoom() {

cout << "creating MagicRoom" << endl;

}

void connect(Room\* room) override {

}

};

class MazeGame {

private:

vector<Room \*> rooms;

public:

virtual Room\* makeRoom() = 0;

~MazeGame() {

for (Room\* ptr : rooms) {

cout << "deleting room" << endl;

delete ptr;

}

}

// TO-DO implement destructor

void anOperation() {

Room\* room1 = makeRoom();

Room\* room2 = makeRoom();

room1->connect(room2);

rooms.push\_back(room1);

rooms.push\_back(room2);

}

};

class MagicMazeGame : public MazeGame {

public:

MagicMazeGame() {

anOperation();

}

Room\* makeRoom() override {

return new MagicRoom();

}

};