Design Patterns

Gerwin van Dijken (gerwin.vandijken@inholland.nl)

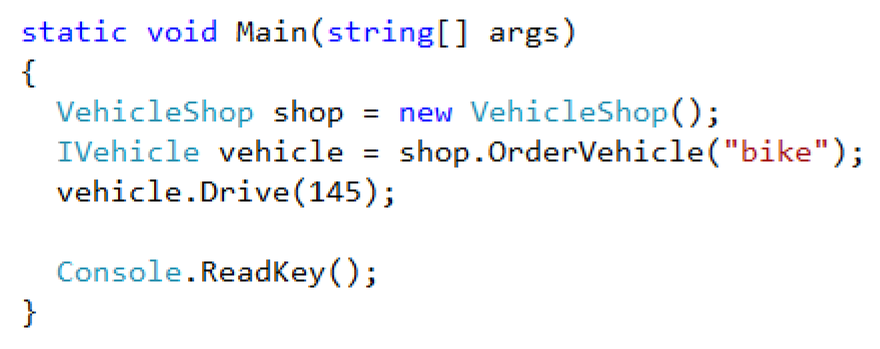
Program term 1.4

01 (wk-15)	abstract classes and interfaces
02 (wk-16)	Template Method pattern / Observer pattern
03 (wk-17)	MVC pattern
04 (wk-18)	*no classes*
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	repetition / practice exam

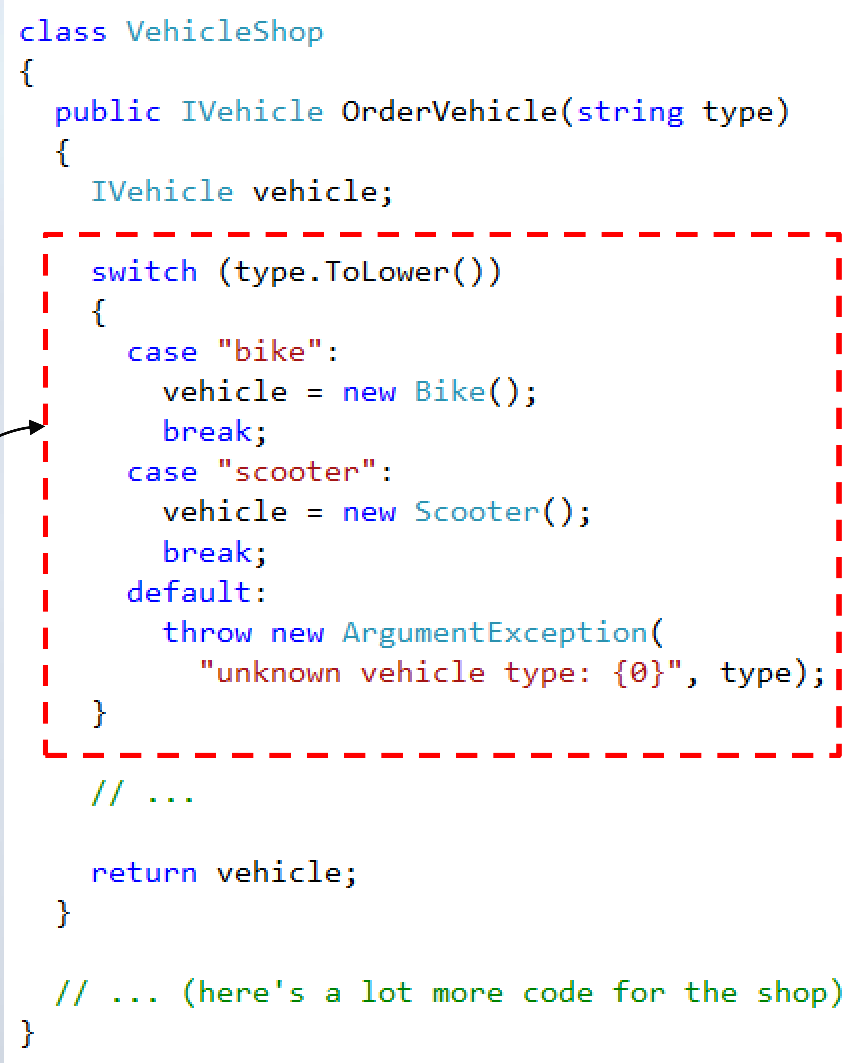
----

09 (wk-23)	exam *(computer assignments)*
10 (wk-24)	*retakes (courses term 1.3)*
11 (wk-25)	*retakes (courses term 1.4)*

(Simple) Factory



```
static void Main(string[] args)
{
  VehicleShop shop = new VehicleShop();
  IVehicle vehicle = shop.OrderVehicle("bike");
  vehicle.Drive(145);

  Console.ReadKey();
}
```



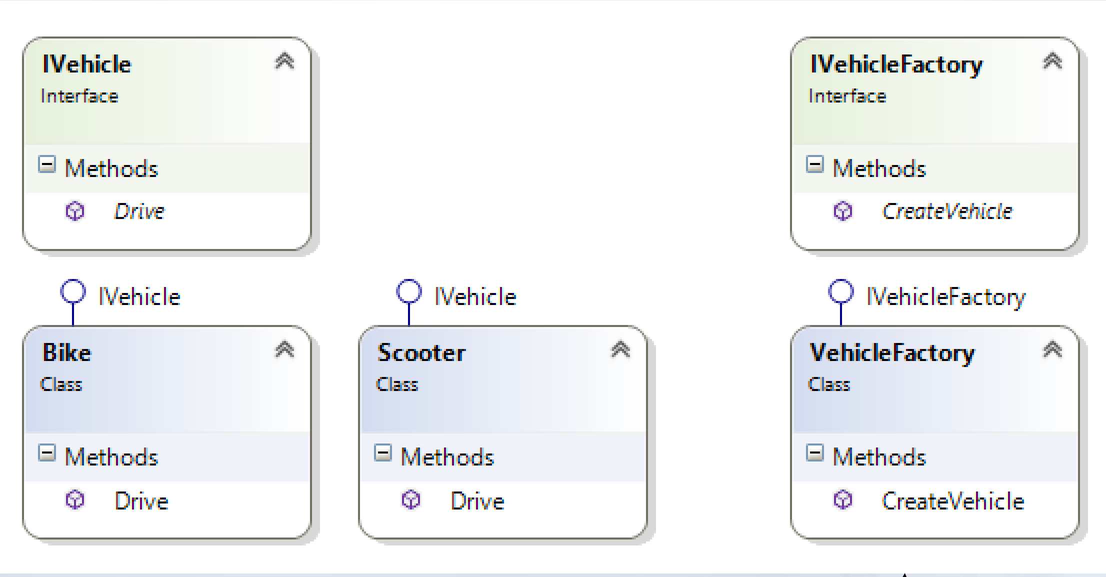
```
class VehicleShop
{
  public IVehicle OrderVehicle(string type)
  {
    IVehicle vehicle;

    switch (type.ToLower())
    {
      case "bike":
        vehicle = new Bike();
        break;
      case "scooter":
        vehicle = new Scooter();
        break;
      default:
        throw new ArgumentException(
          "unknown vehicle type: {0}", type);
    }

    // ...

    return vehicle;
  }

  // ... (here's a lot more code for the shop)
}
```

It looks like this code will change in the future…

… and this kind of code can often be found on several locations…

“identify the aspects that vary and separate them from what stays the same...”

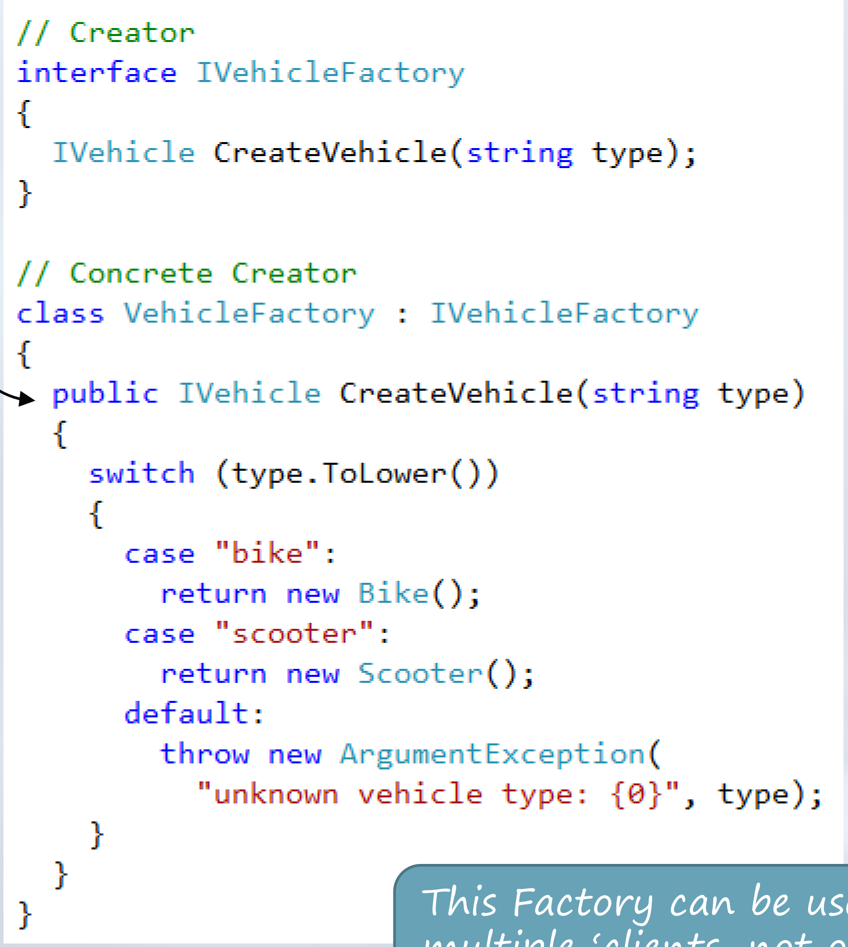
(Simple) Factory



We add a 'Factory' class which <u>sole</u> task is <u>to create objects</u> (vehicles) …

(Simple) Factory

… and we move the 'creation-code' to a method of this Factory-class.

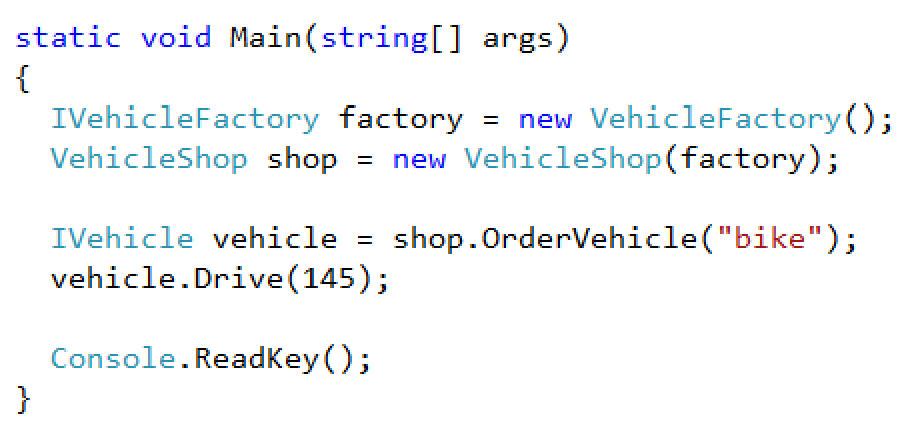


```
// Creator
interface IVehicleFactory
{
  IVehicle CreateVehicle(string type);
}

// Concrete Creator
class VehicleFactory : IVehicleFactory
{
  public IVehicle CreateVehicle(string type)
  {
    switch (type.ToLower())
    {
      case "bike":
        return new Bike();
      case "scooter":
        return new Scooter();
      default:
        throw new ArgumentException(
          "unknown vehicle type: {0}", type);
    }
  }
}
```

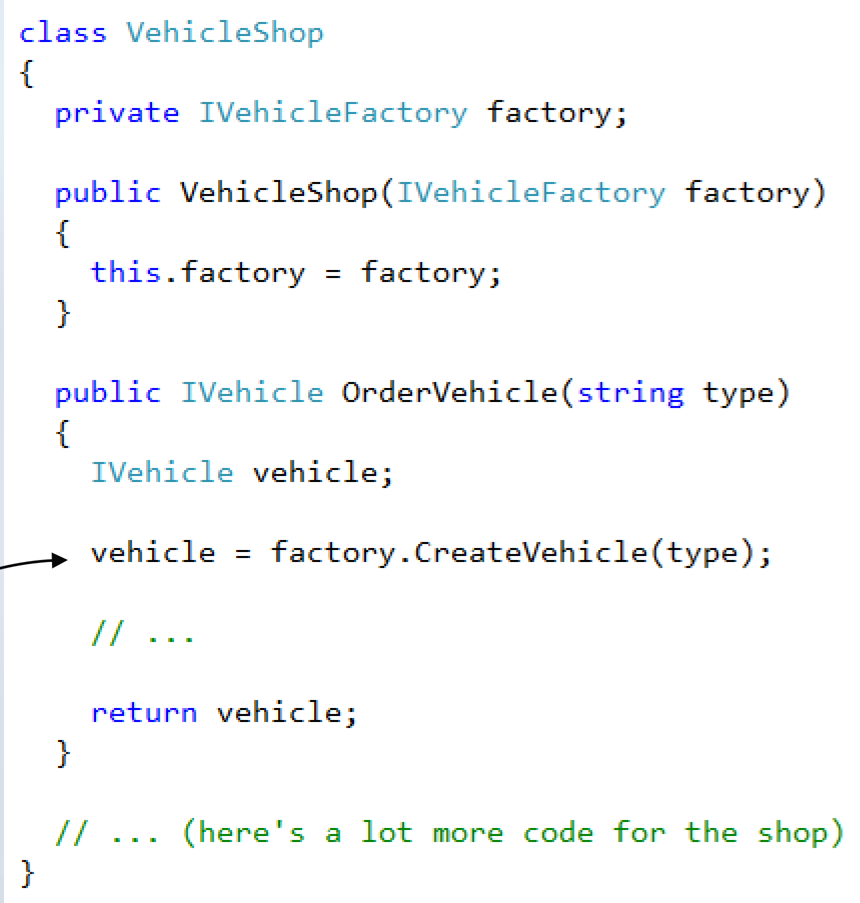
Future changes concerning the creation of Vehicles will now be done only inside this class.

This Factory can be used by multiple 'clients, not only by the VehicleShop…

(Simple) Factory



```
static void Main(string[] args)
{
  IVehicleFactory factory = new VehicleFactory();
  VehicleShop shop = new VehicleShop(factory);

  IVehicle vehicle = shop.OrderVehicle("bike");
  vehicle.Drive(145);

  Console.ReadKey();
}
```

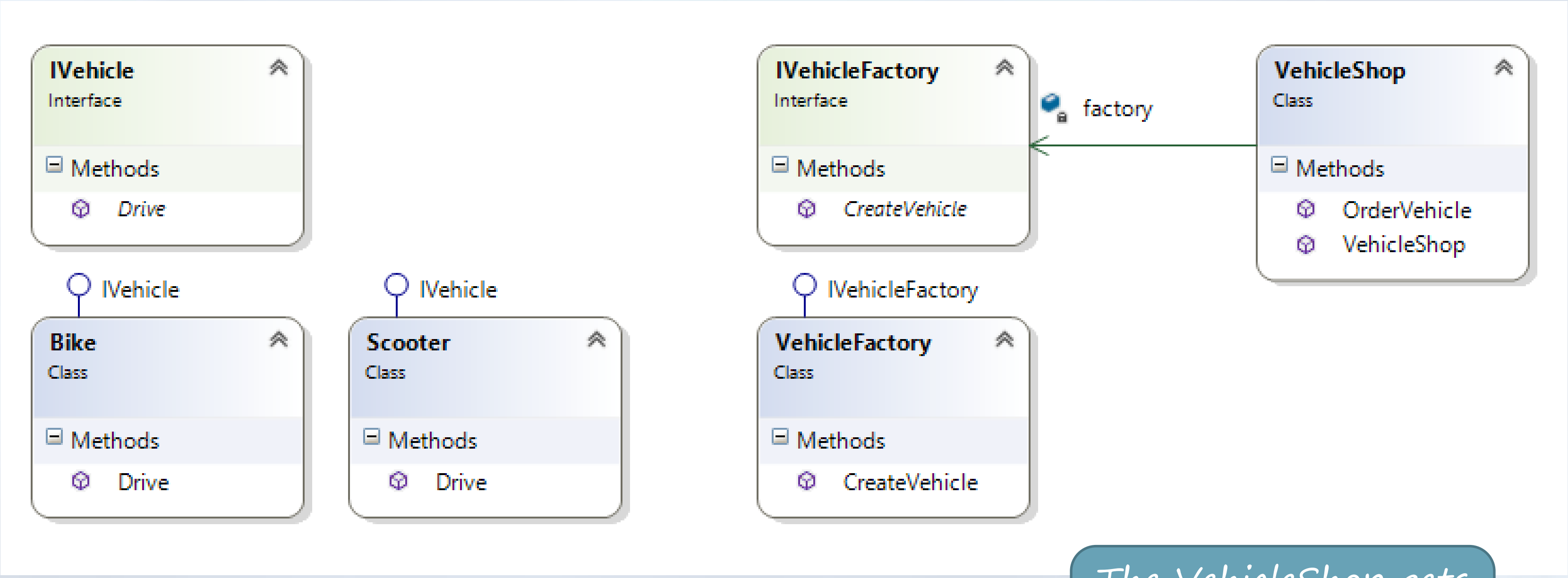


```
class VehicleShop
{
  private IVehicleFactory factory;

  public VehicleShop(IVehicleFactory factory)
  {
    this.factory = factory;
  }

  public IVehicle OrderVehicle(string type)
  {
    IVehicle vehicle;

    vehicle = factory.CreateVehicle(type);

    // ...

    return vehicle;
  }

  // ... (here's a lot more code for the shop)
}
```

The VehicleShop wil now use the factory to create objects (Vehicles).

Now the VehicleShop does not need to be changed when other 'vehicles' need to be created in the future.

(Simple) Factory



IVehicle
Interface
Methods
Drive

IVehicle
Bike
Class
Methods
Drive

IVehicle
Scooter
Class
Methods
Drive

IVehicleFactory
Interface
Methods
CreateVehicle

factory

VehicleShop
Class
Methods
OrderVehicle
VehicleShop

IVehicleFactory
VehicleFactory
Class
Methods
CreateVehicle

The VehicleShop gets objects (vehicles) through the factory.

Factory Method

The *Factory Method (GoF)*: *'Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.'*

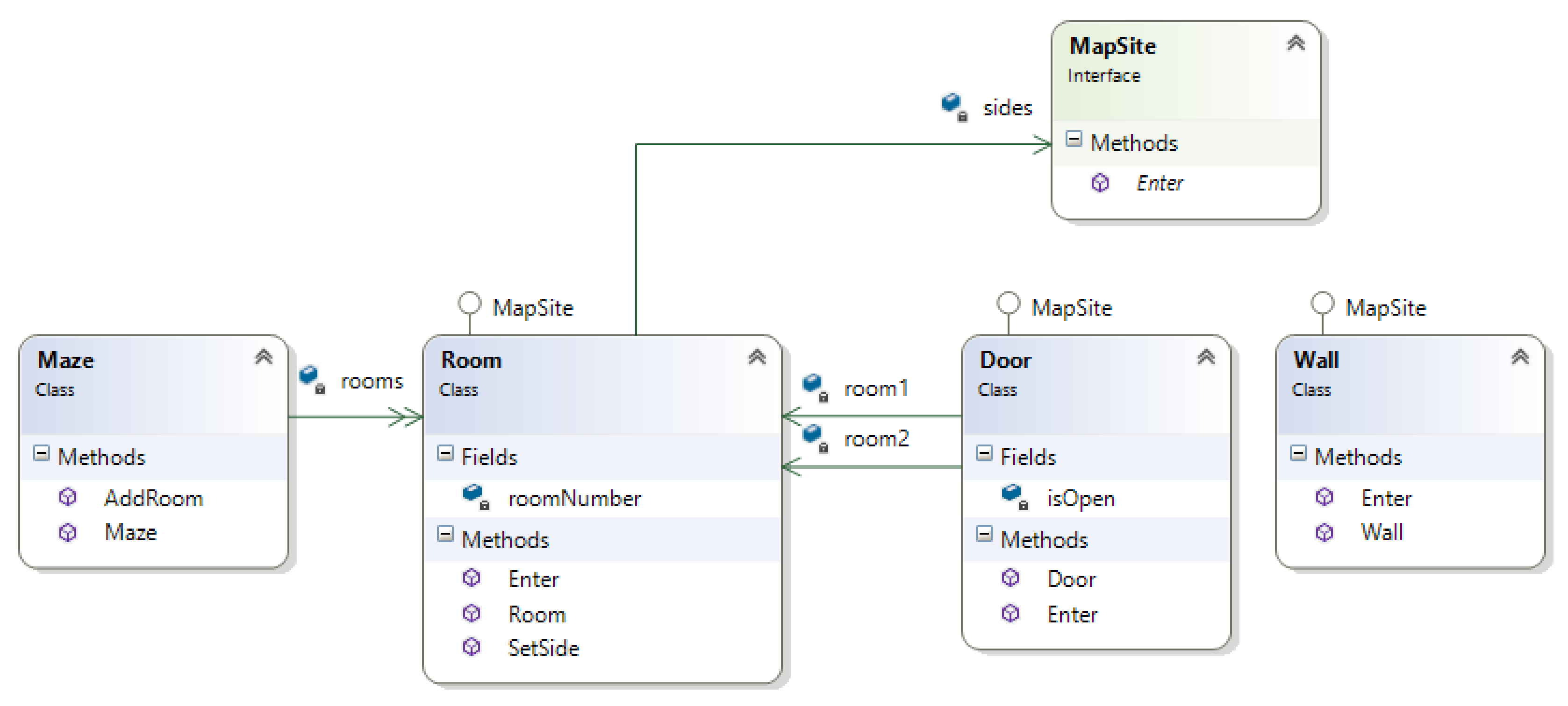
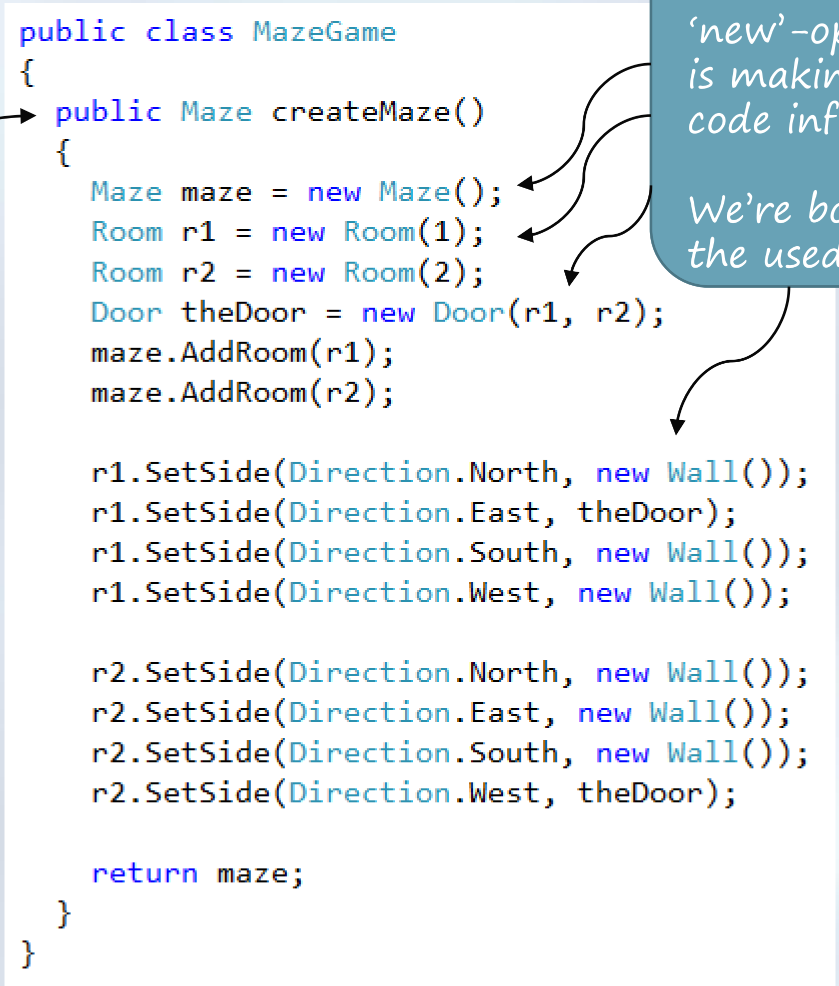


Image we have a maze with several rooms; each room has 4 sides (Wall, Door or Room).

Factory Method

We have a class MazeGame, that can create a Maze.



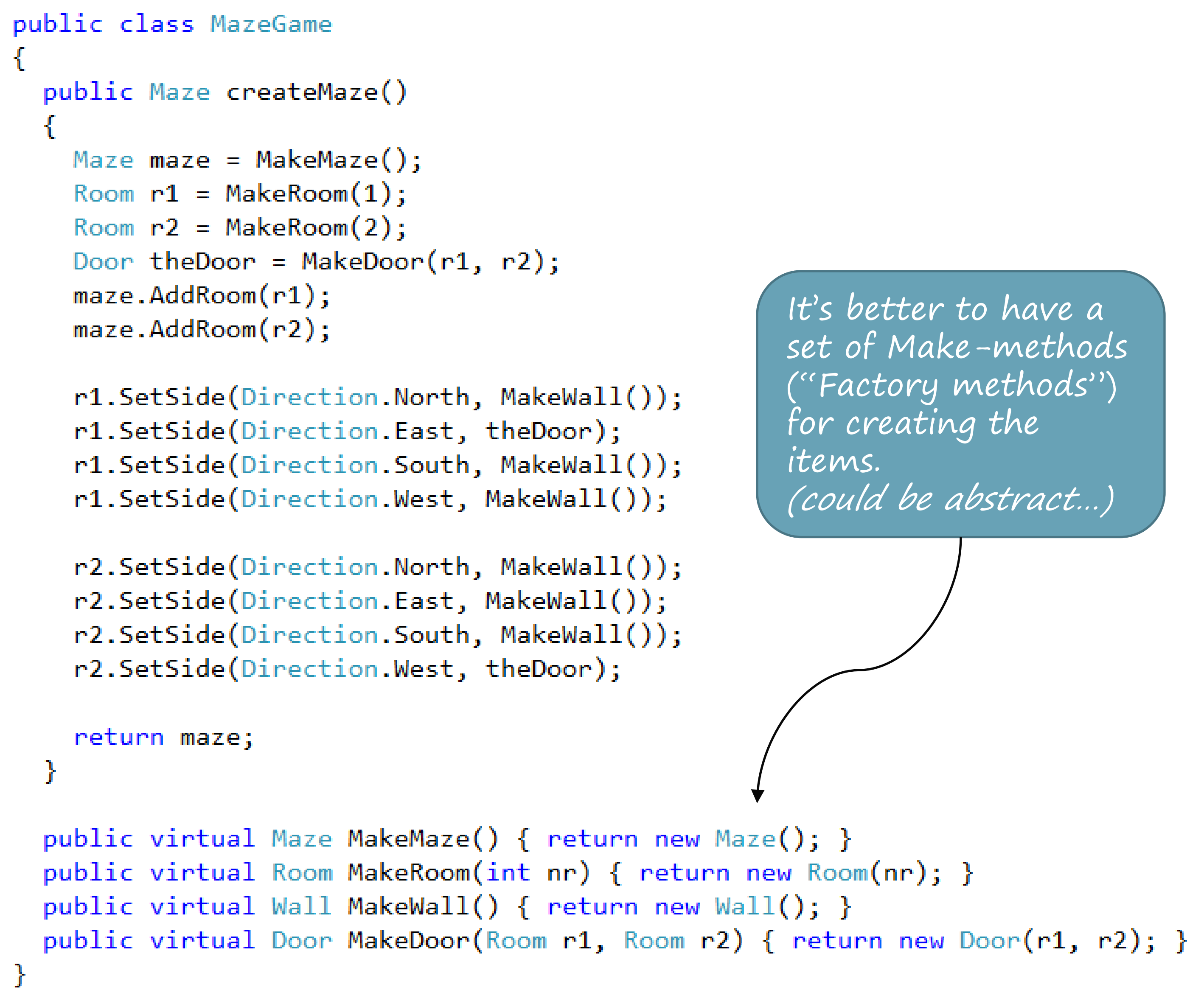
```
public class MazeGame
{
  public Maze createMaze()
  {
    Maze maze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door theDoor = new Door(r1, r2);
    maze.AddRoom(r1);
    maze.AddRoom(r2);

    r1.SetSide(Direction.North, new Wall());
    r1.SetSide(Direction.East, theDoor);
    r1.SetSide(Direction.South, new Wall());
    r1.SetSide(Direction.West, new Wall());

    r2.SetSide(Direction.North, new Wall());
    r2.SetSide(Direction.East, new Wall());
    r2.SetSide(Direction.South, new Wall());
    r2.SetSide(Direction.West, theDoor);

    return maze;
  }
}
```

The use of the 'new'-operator is making this code inflexible…

We're bound to the used classes!

Now image we want to create a different kind of Maze, with the same layout but with e.g. 'enchanted' Rooms and Doors?

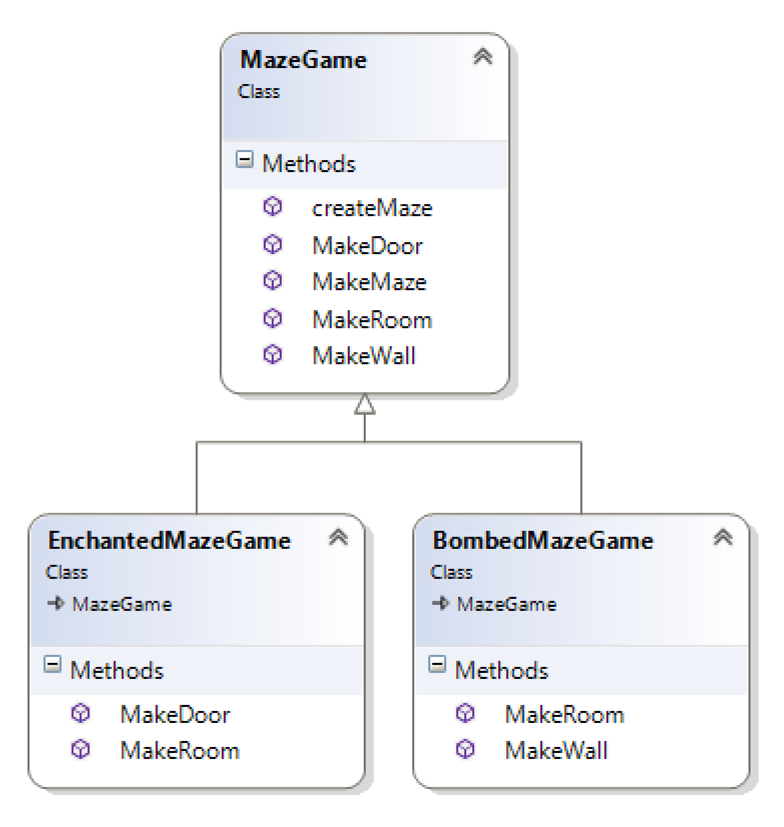
Factory Method



```
public class MazeGame
{
  public Maze createMaze()
  {
    Maze maze = MakeMaze();
    Room r1 = MakeRoom(1);
    Room r2 = MakeRoom(2);
    Door theDoor = MakeDoor(r1, r2);
    maze.AddRoom(r1);
    maze.AddRoom(r2);

    r1.SetSide(Direction.North, MakeWall());
    r1.SetSide(Direction.East, theDoor);
    r1.SetSide(Direction.South, MakeWall());
    r1.SetSide(Direction.West, MakeWall());

    r2.SetSide(Direction.North, MakeWall());
    r2.SetSide(Direction.East, MakeWall());
    r2.SetSide(Direction.South, MakeWall());
    r2.SetSide(Direction.West, theDoor);

    return maze;
  }

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

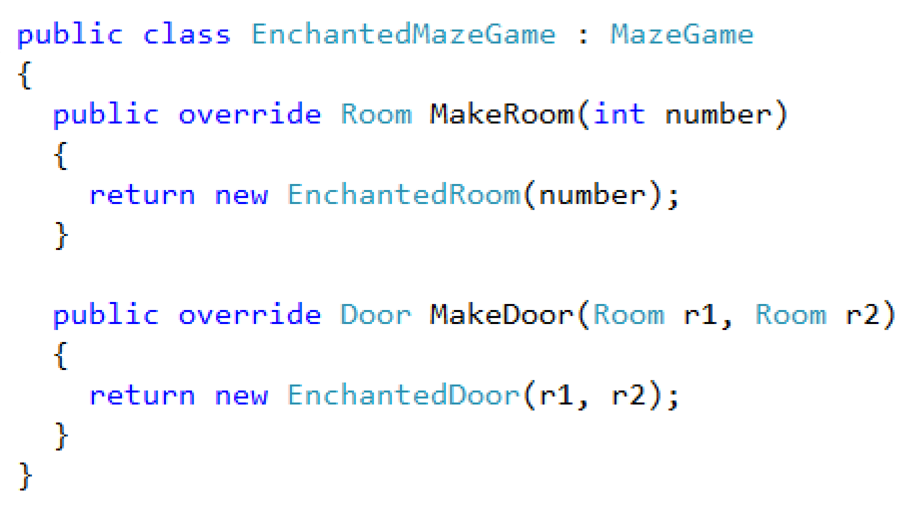
It's better to have a set of Make-methods ("Factory methods") for creating the items.
(could be abstract...)

Factory Method

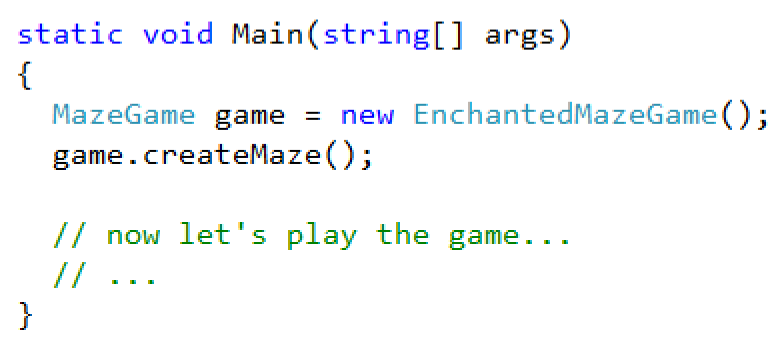


Factory Method

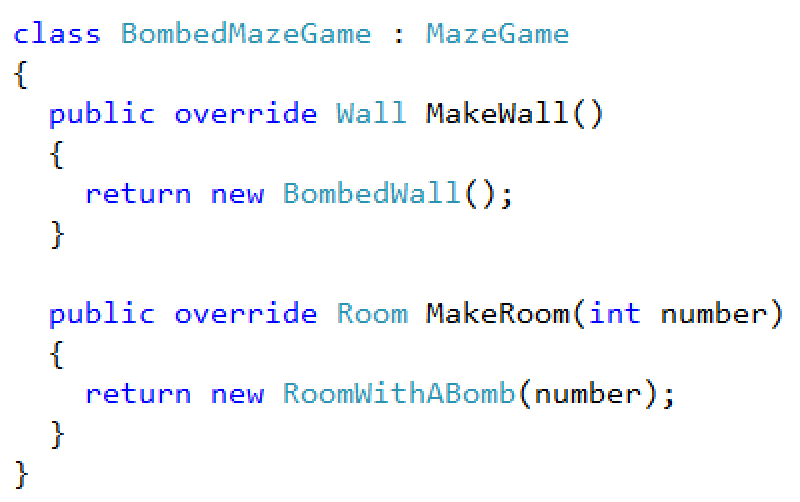
An "Enchanted" mazegame now only has to overwrite a few Factory-methods. The creation of the maze (createMaze) remains the same!



```
public class EnchantedMazeGame : MazeGame
{
  public override Room MakeRoom(int number)
  {
    return new EnchantedRoom(number);
  }

  public override Door MakeDoor(Room r1, Room r2)
  {
    return new EnchantedDoor(r1, r2);
  }
}
```



```
static void Main(string[] args)
{
  MazeGame game = new EnchantedMazeGame();
  game.createMaze();

  // now let's play the game...
  // ...
}
```

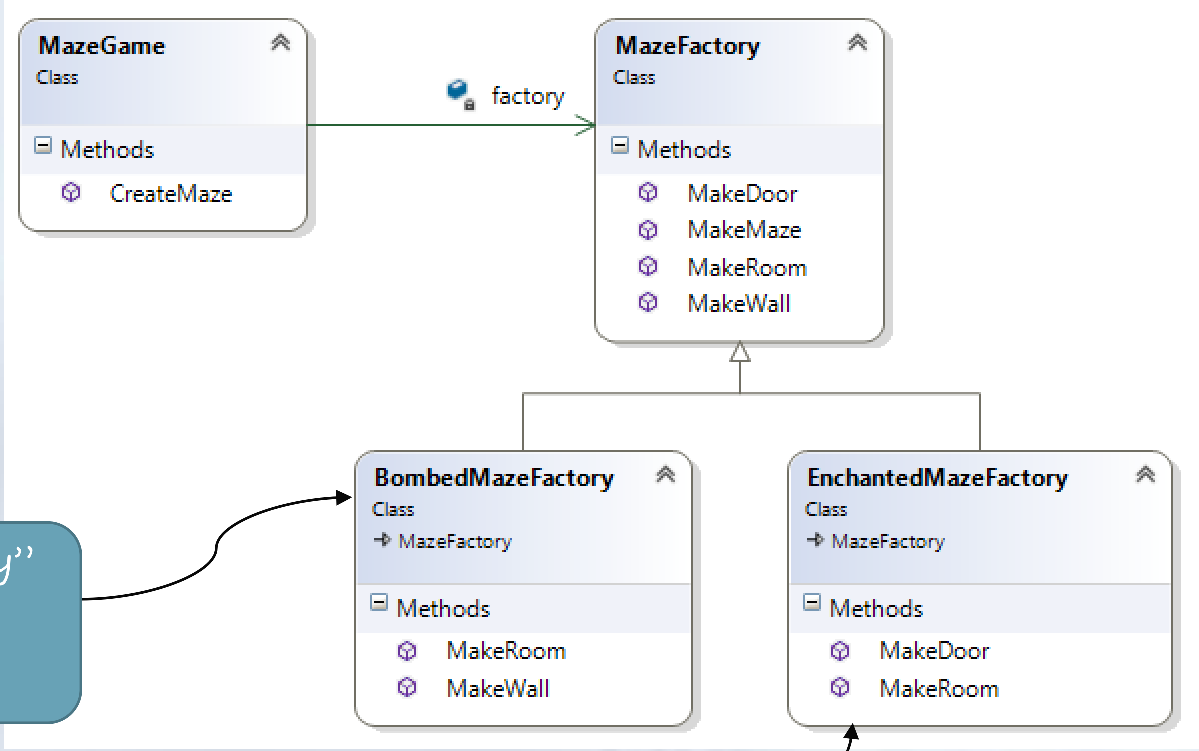


```
class BombedMazeGame : MazeGame
{
  public override Wall MakeWall()
  {
    return new BombedWall();
  }

  public override Room MakeRoom(int number)
  {
    return new RoomWithABomb(number);
  }
}
```

Abstract Factory

The Abstract Factory (GoF): *'Provide an interface for creating families of related or dependent objects without specifying their concrete classes.'*

*With "**Abstract Factory**" the objects are created through special factories.*



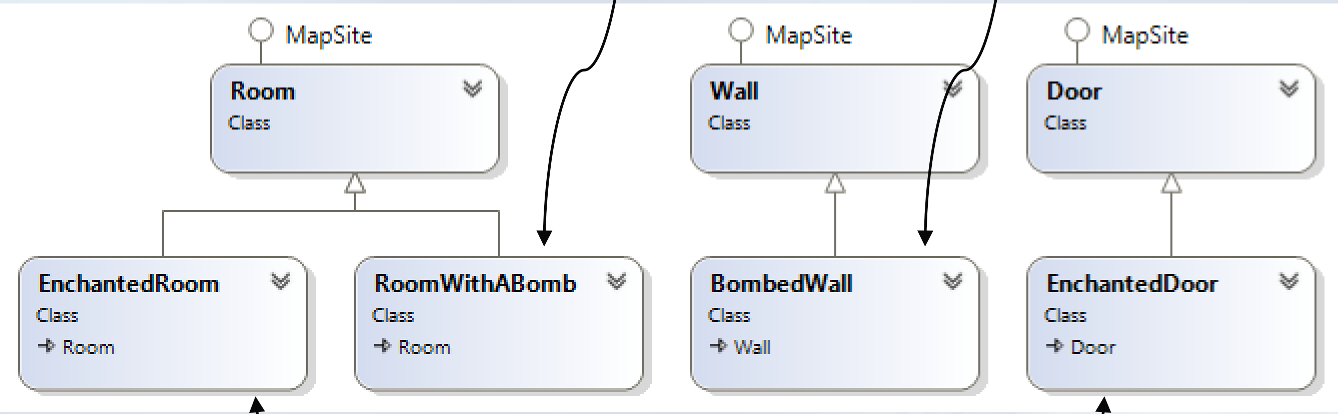
The "BombedMazeFactory" creates 'bombed' objects, like BombedWall and RoomWithABomb.

The "EnchantedMazeFactory" creates 'enchanted' objects, like EnchantedDoor and EnchantedRoom.

Abstract Factory

With the "***Abstract Factory***" complete sets of 'families' of related objects are defined.

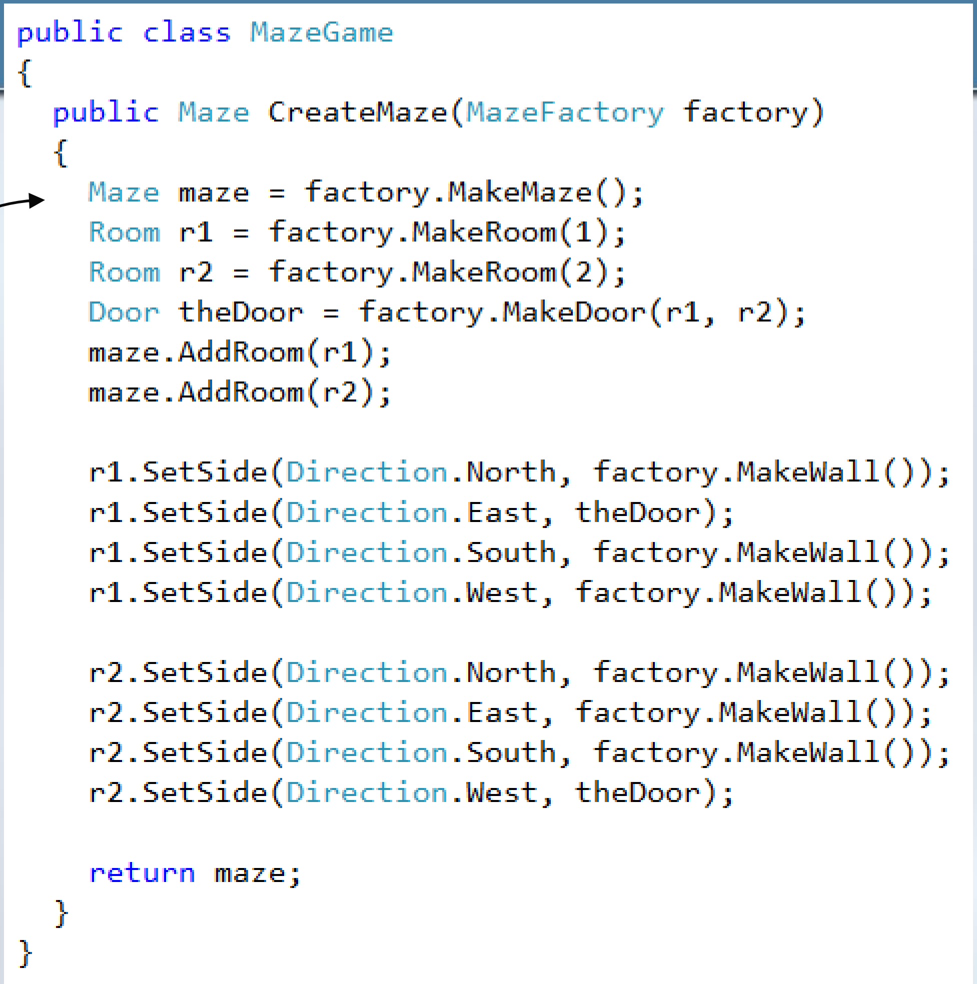
Here you see the family 'bombed' objects… (all created by the BombedMazeFactory).



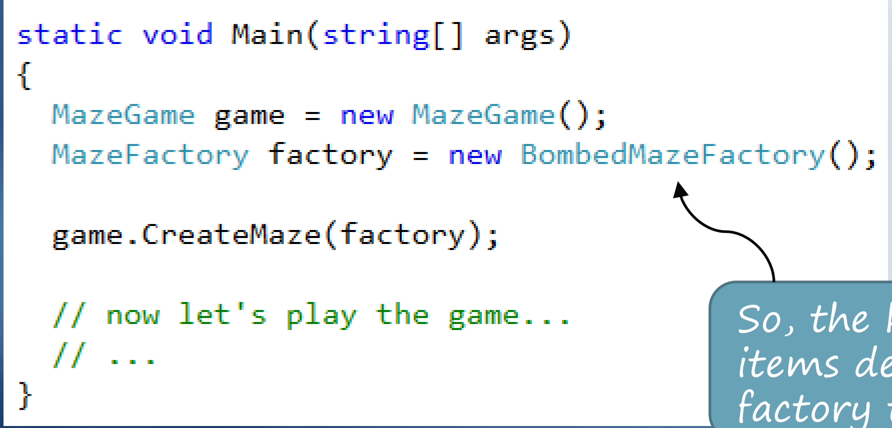
And here you see the family 'enchanted' objects… (all created by the EnchantedMazeFactory).

Abstract Factory

All maze-items (Room, Wall, …) are created through a factory.



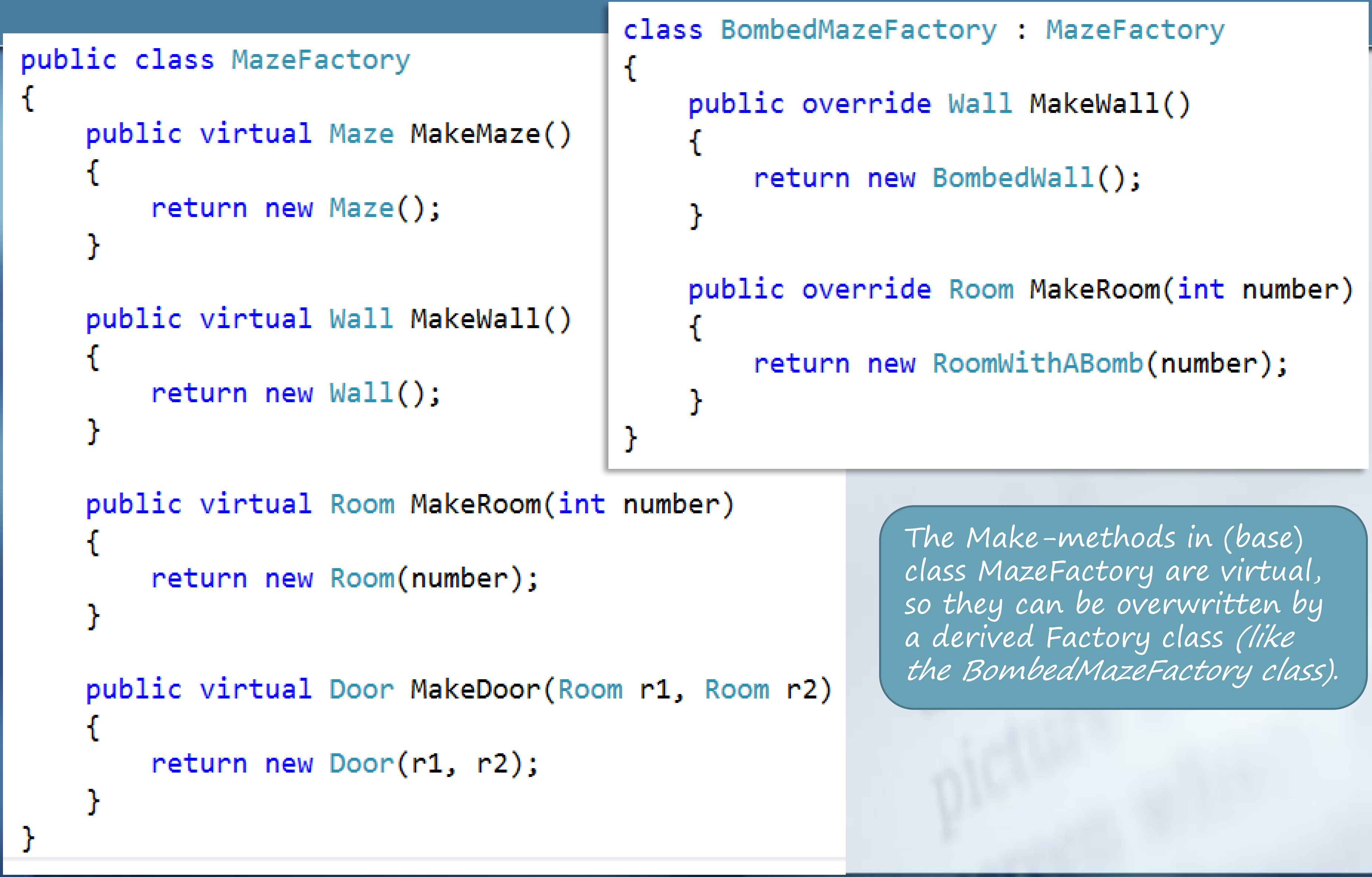
```
public class MazeGame
{
  public Maze CreateMaze(MazeFactory factory)
  {
    Maze maze = factory.MakeMaze();
    Room r1 = factory.MakeRoom(1);
    Room r2 = factory.MakeRoom(2);
    Door theDoor = factory.MakeDoor(r1, r2);
    maze.AddRoom(r1);
    maze.AddRoom(r2);

    r1.SetSide(Direction.North, factory.MakeWall());
    r1.SetSide(Direction.East, theDoor);
    r1.SetSide(Direction.South, factory.MakeWall());
    r1.SetSide(Direction.West, factory.MakeWall());

    r2.SetSide(Direction.North, factory.MakeWall());
    r2.SetSide(Direction.East, factory.MakeWall());
    r2.SetSide(Direction.South, factory.MakeWall());
    r2.SetSide(Direction.West, theDoor);

    return maze;
  }
}
```



```
static void Main(string[] args)
{
  MazeGame game = new MazeGame();
  MazeFactory factory = new BombedMazeFactory();

  game.CreateMaze(factory);

  // now let's play the game...
  // ...
}
```

So, the kind of maze-items depends on the factory that is used.

Abstract Factory



```
public class MazeFactory
{
    public virtual Maze MakeMaze()
    {
        return new Maze();
    }

    public virtual Wall MakeWall()
    {
        return new Wall();
    }

    public virtual Room MakeRoom(int number)
    {
        return new Room(number);
    }

    public virtual Door MakeDoor(Room r1, Room r2)
    {
        return new Door(r1, r2);
    }
}
```

```
class BombedMazeFactory : MazeFactory
{
    public override Wall MakeWall()
    {
        return new BombedWall();
    }

    public override Room MakeRoom(int number)
    {
        return new RoomWithABomb(number);
    }
}
```

The Make-methods in (base) class MazeFactory are virtual, so they can be overwritten by a derived Factory class (like the BombedMazeFactory class).

Assignments

- Moodle: ‘Week 6 assignments’