Design Patterns

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

Programma periode 1.4

01 (wk-15)	abstracte classes en interfaces
02 (wk-16)	Template Method pattern / Observer pattern
03 (wk-17)	MVC pattern
04 (wk-18)	*geen lessen (meivakantie)*
05 (wk-19)	Strategy pattern / Adapter pattern
06 (wk-20)	Singleton pattern / State pattern
07 (wk-21)	Factory patterns
08 (wk-22)	herhaling / proeftentamen

---

09 (wk-23)	tentamen (praktijk)
10 (wk-24)	*hertentamens (vakken periode 1.3)*
11 (wk-25)	*hertentamens (vakken periode 1.4)*

(Simple) Factory

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

  Console.ReadKey();
}
```

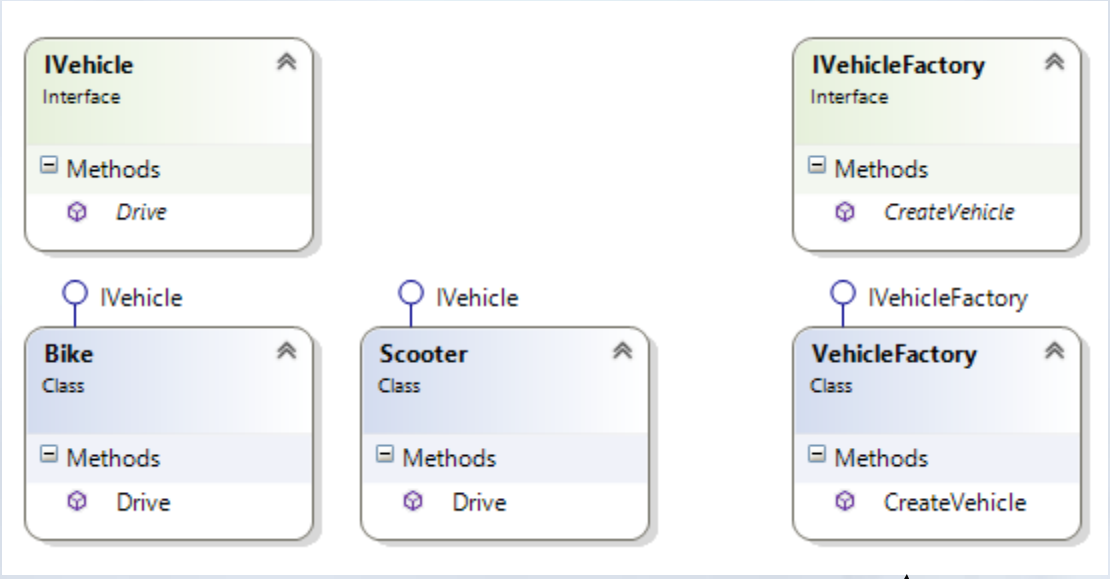
Het ziet er naar uit dat deze code in de toekomst aangepast zal worden...

En dit soort code staat vaak op meerdere plekken...

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

```
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)
}
```

(Simple) Factory



We voegen een 'Factory' class toe die zich alleen bezig houdt met het aanmaken van objecten (vehicles) …

(Simple) Factory

… en verplaatsen de 'creation-code' naar een methode binnen deze 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);
    }
  }
}
```

Wijzigingen omtrent het aanmaken van Vehicles zal nu alleen binnen deze class plaatsvinden.

Deze Factory kan door meerdere 'clients' gebruikt worden, niet alleen door de VehicleShop…

(Simple) Factory

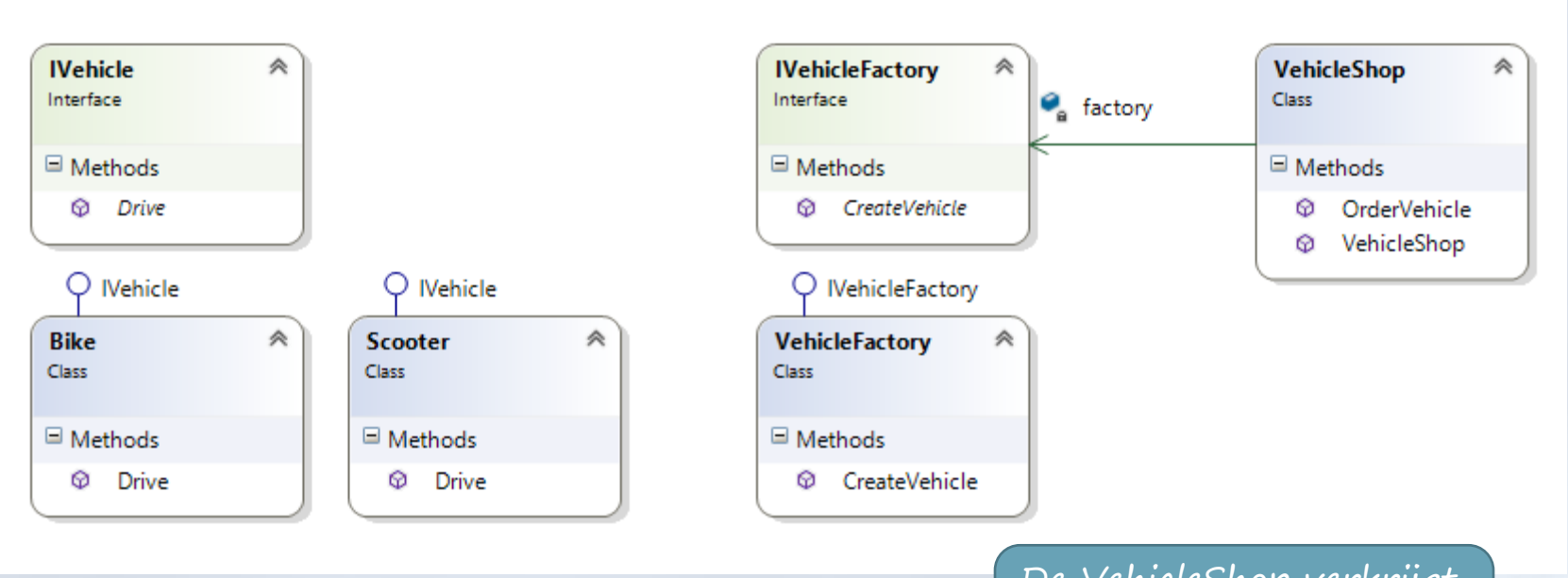
```
static void Main(string[] args)
{
  IVehicleFactory factory = new VehicleFactory();
  VehicleShop shop = new VehicleShop(factory);

  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)
}
```

De VehicleShop gebruikt nu de factory om objecten (Vehicles) aan te maken.

Hierdoor hoeft de VehicleShop niet meer aangepast worden als er andere 'vehicles' aangemaakt moeten worden in de toekomst....

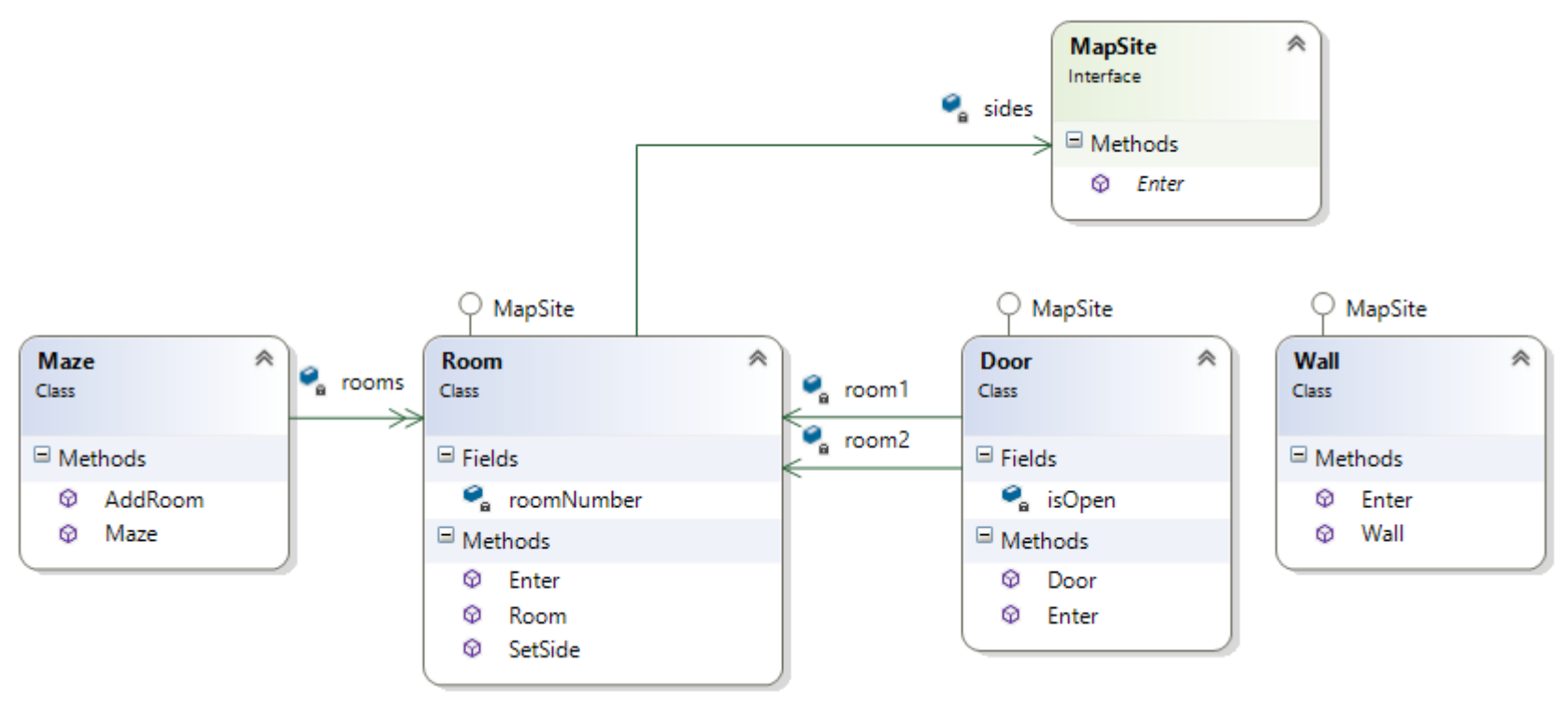
(Simple) Factory



De VehicleShop verkrijgt objecten (vehicles) via de 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.'*



Stel we hebben een maze met verschillende rooms; elke room heeft 4 sides (Wall, Door or Room).

Factory Method

Verder hebben we een MazeGame class, die een Maze kan aanmaken.

```
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;
  }
}
```

Het gebruik van de 'new'-operator maakt deze code in-flexible...

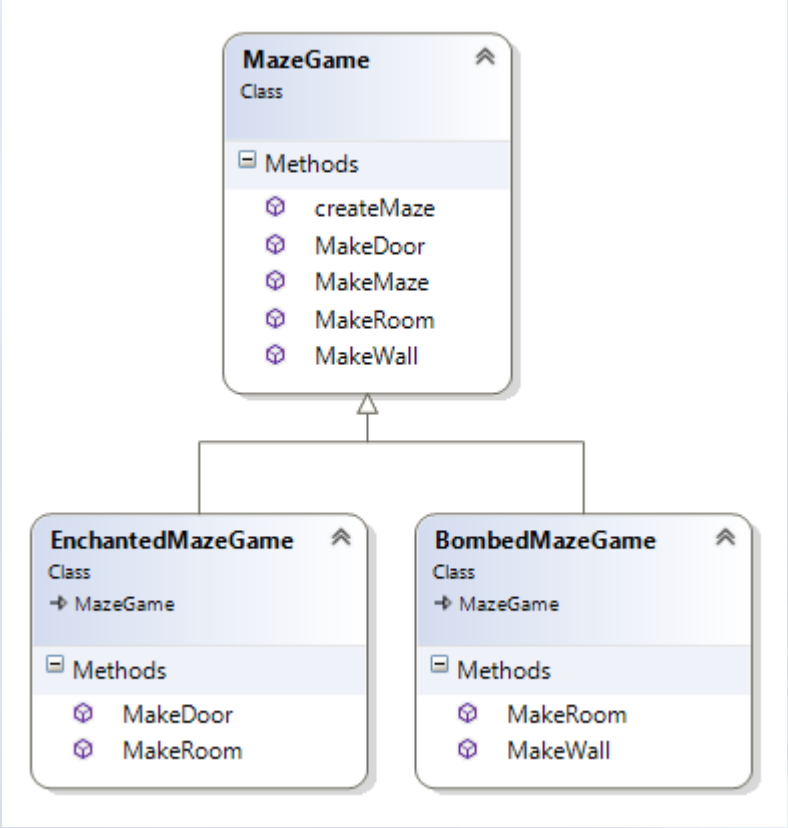
Maar stel nu dat we een ander soort Maze willen maken met dezelfde indeling maar met bv 'enchanted' Rooms en Doors?

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); }
}
```

We kunnen beter een set van Make-methoden ("Factory methods") definieren om daarmee de items aan te maken. (eventueel abstract...)

Factory Method



Factory Method

Een "Enchanted" mazegame hoeft nu alleen een aantal Factory-methods te overschrijven.
Het aanmaken v/d maze (createMaze) zelf blijft ongewijzigd!

```
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);
  }
}
```

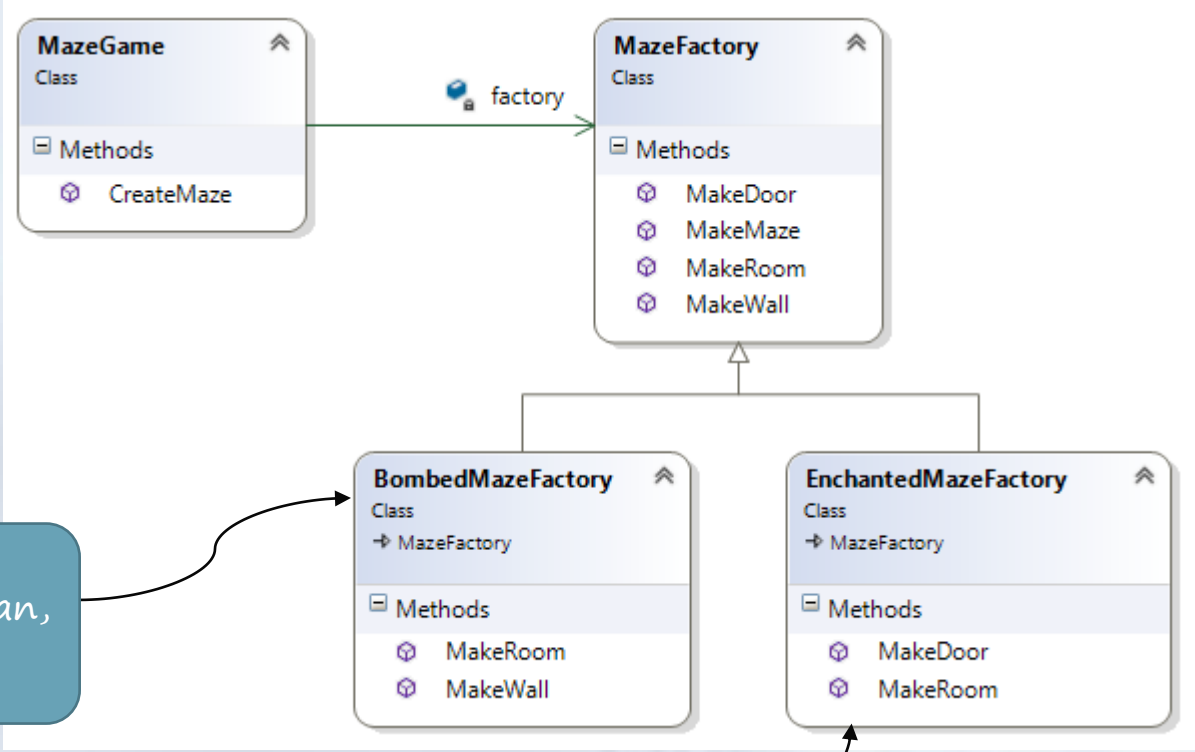
```
static void Main(string[] args)
{
  MazeGame game = new EnchantedMazeGame();
  game.createMaze();

  // 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.'*

Bij de "Abstract Factory" worden de objecten via speciale factories aangemaakt.



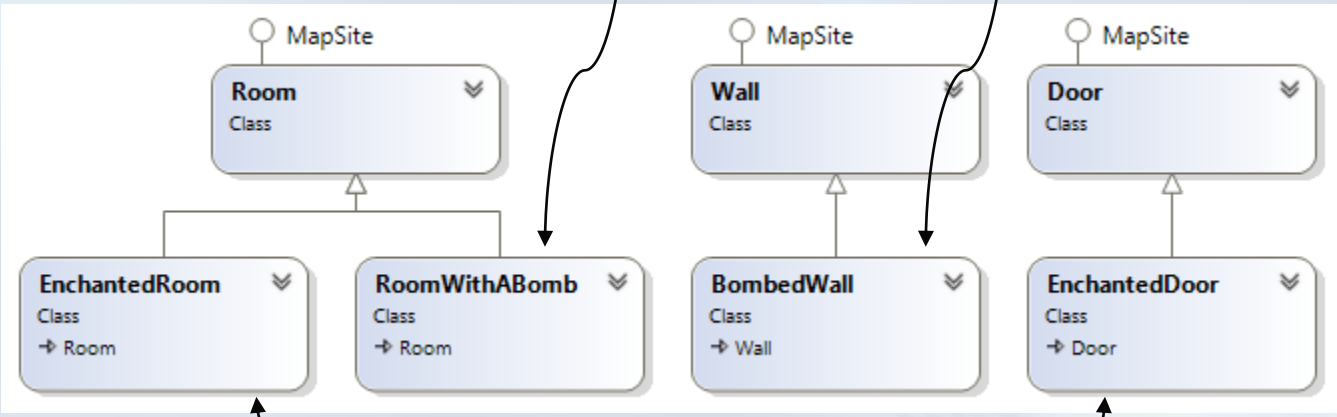
De "BombedMazeFactory" maakt 'bombed' objecten aan, zoals BombedWall and RoomWithABomb.

De "EnchantedMazeFactory" maakt 'enchanted' objecten aan, zoals EnchantedDoor and EnchantedRoom.

Abstract Factory

Bij de "**Abstract Factory**" heb je hele 'families' van gerelateerde objecten.

Zo heb je hier de familie 'bombed' objecten... (allemaal aangemaakt door de BombedMazeFactory)



En hier de familie 'enchanted' objecten... (allemaal aangemaakt door de EnchantedMazeFactory)

Abstract Factory

Via een factory worden alle maze-items (Room, Wall, ...) aangemaakt.

```
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...
  // ...
}
```

Wat voor maze-items we krijgen, hangt dus af van de factory die we gebruiken.

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);
    }
}
```

De Maak-methoden in de (base) MazeFactory class zijn virtual, zodat ze overschreven kunnen worden door een afgeleide Factory class (zoals de BombedMazeFactory class).

Opdrachten

- Zie Moodle: ‘Week 6 opdrachten’