Creational Patterns

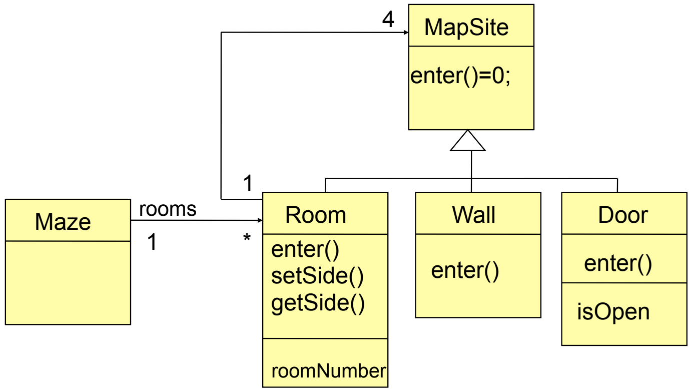
Overview of creational patterns

- Abstract the instantiation process
- Help make a system independent of how its objects are created, composed, represented

- Class creational pattern
  - uses inheritance to vary the class that's instantiated
  - *Factory Method*

- Object creational pattern
  - delegates instantiation to another object
  - *Abstract Factory, Prototype, Singleton, Builder*

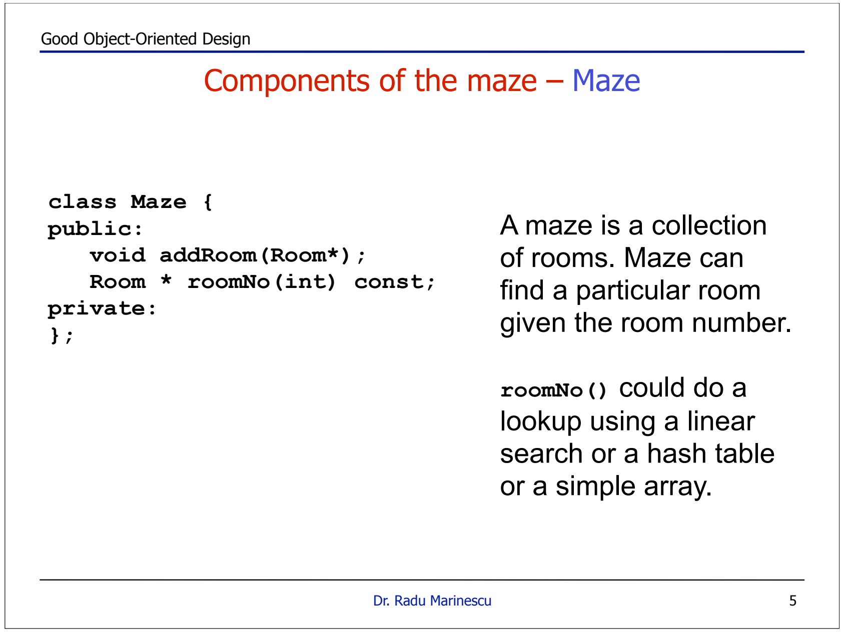
Class Diagram for the Maze



Common abstract class for all Maze Components

```
enum Direction {North, South, East, West};

class MapSite {
public:
   virtual void enter() = 0;
};
```

- Meaning of enter() depends on what you are entering.
  - room → location changes
  - door → if door is open go in; else hurt your nose ;)

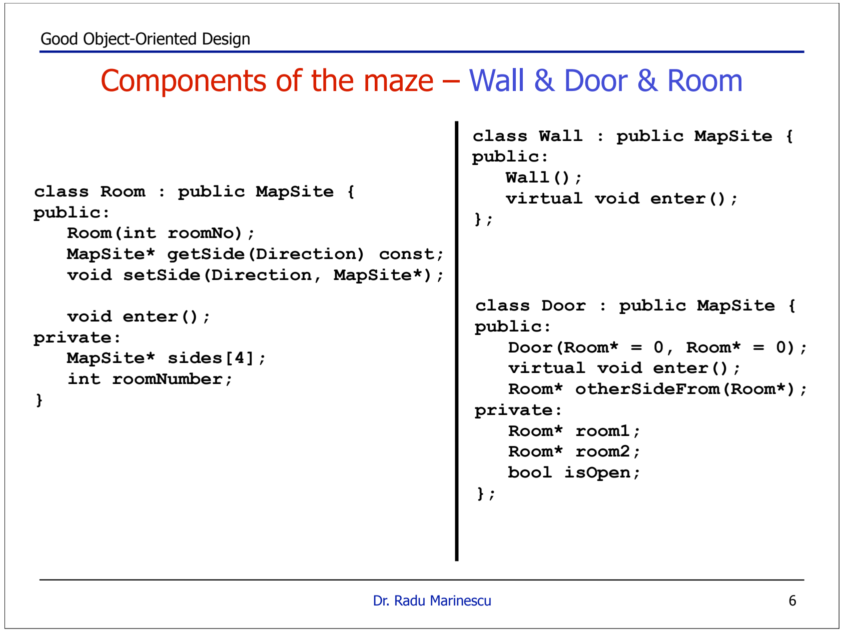


Components of the maze – Maze

```
class Maze {
public:
   void addRoom(Room*);
   Room * roomNo(int) const;
private:
};
```

A maze is a collection of rooms. Maze can find a particular room given the room number.

`roomNo()` could do a lookup using a linear search or a hash table or a simple array.

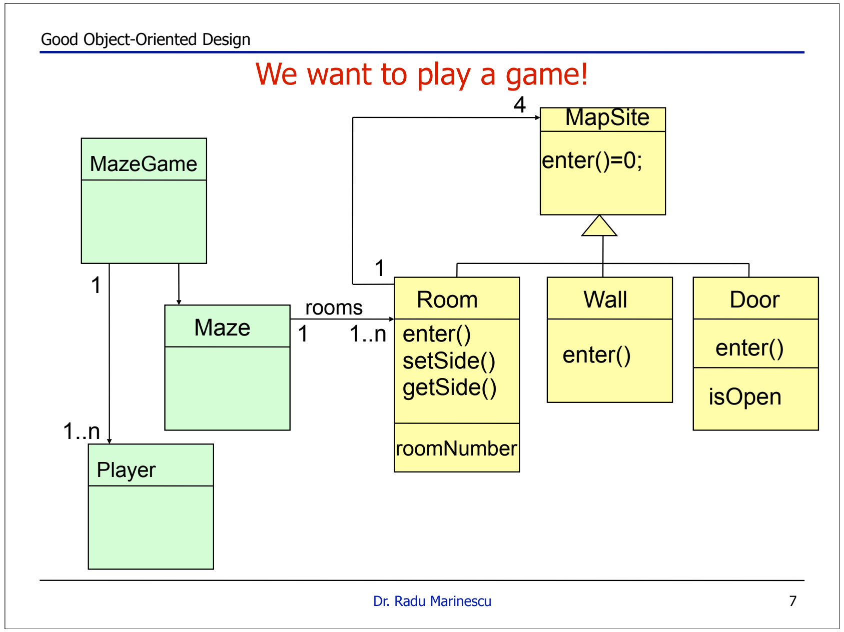


Components of the maze – Wall & Door & Room

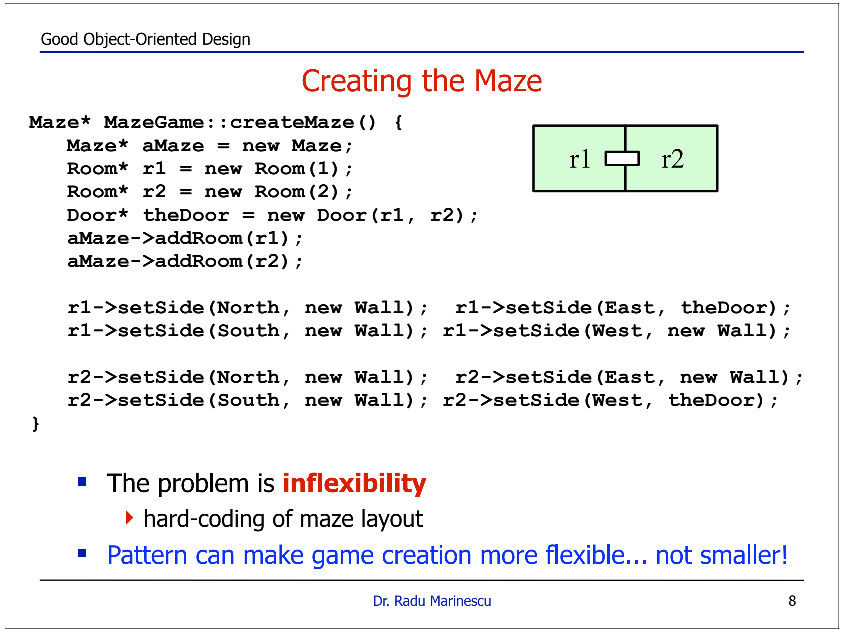
```
class Room : public MapSite {
public:
   Room(int roomNo);
   MapSite* getSide(Direction) const;
   void setSide(Direction, MapSite*);

   void enter();
private:
   MapSite* sides[4];
   int roomNumber;
}
```

```
class Wall : public MapSite {
public:
   Wall();
   virtual void enter();
};
```

```
class Door : public MapSite {
public:
   Door(Room* = 0, Room* = 0);
   virtual void enter();
   Room* otherSideFrom(Room*);
private:
   Room* room1;
   Room* room2;
   bool isOpen;
};
```



We want to play a game!



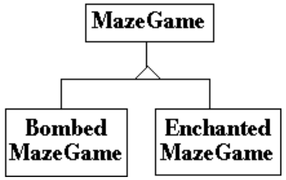
Creating the Maze

```
Maze* MazeGame::createMaze() {
   Maze* aMaze = new Maze;
   Room* r1 = new Room(1);
   Room* r2 = new Room(2);
   Door* theDoor = new Door(r1, r2);
   aMaze->addRoom(r1);
   aMaze->addRoom(r2);

   r1->setSide(North, new Wall);  r1->setSide(East, theDoor);
   r1->setSide(South, new Wall); r1->setSide(West, new Wall);

   r2->setSide(North, new Wall);  r2->setSide(East, new Wall);
   r2->setSide(South, new Wall); r2->setSide(West, theDoor);
}
```

- The problem is **inflexibility**
  - hard-coding of maze layout
- Pattern can make game creation more flexible... not smaller!

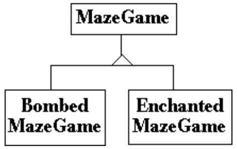
We want Flexibility in Maze Creation

- Be able to vary the kinds of mazes
  - Rooms with bombs
  - Walls that have been bombed
  - Enchanted rooms
    - Need a spell to enter the door!



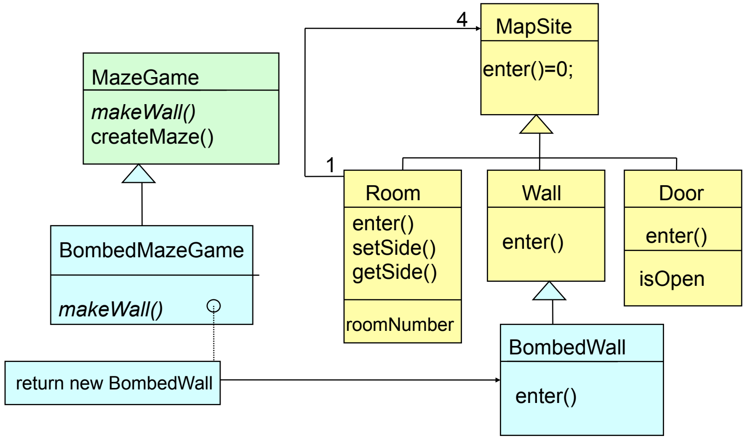
Idea 1: Subclass MazeGame, override createMaze

```
Maze* BombedMazeGame::createMaze() {
   Maze* aMaze = new Maze;
   Room* r1 = new RoomWithABomb(1);
   Room* r2 = new RoomWithABomb(2);
   Door* theDoor = new Door(r1, r2);
   aMaze->addRoom(r1);
   aMaze->addRoom(r2);

   r1->setSide(North, new BombedWall);
   r1->setSide(East, theDoor);
   r1->setSide(South, new BombedWall);
   r1->setSide(West, new BombedWall);
   // etc...etc...
}
```



- Lots of code duplication... :((

Idea 2: Use a Factory Method



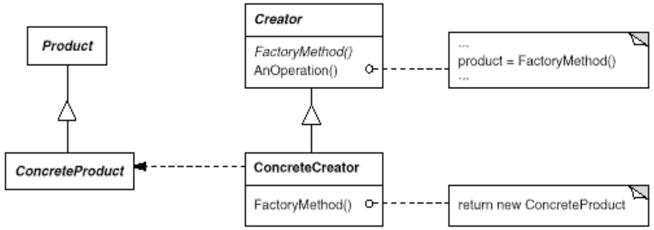
```
Maze* MazeGame::CreateMaze () {
        Maze* aMaze = makeMaze();

        Room* r1 = makeRoom(1);
        Room* r2 = makeRoom(2);
        Door* theDoor = makeDoor(r1, r2);

        aMaze->addRoom(r1);
        aMaze->addRoom(r2);

        r1->SetSide(North, makeWall());
        r1->SetSide(East, theDoor);
        r1->SetSide(South, makeWall());
        r1->SetSide(West, makeWall());

        r2->SetSide(North, makeWall());
        r2->SetSide(East, makeWall());
        r2->SetSide(South, makeWall());
        r2->SetSide(West, theDoor);

        return aMaze;
 }
```

Factory Method

Basic Aspects

- 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
- Also Known As
  - Virtual Constructor
- Applicability
  - A class can't anticipate the class of objects it must create
  - A class wants its subclasses to specify the objects it creates
  - Classes delegate responsibility to one of several helper subclasses

Structure



Participants & Collaborations

- Product
  - defines the interface of objects that will be created by the FM
  - Concrete Product implements the interface
- Creator
  - declares the FM, which returns a product of type Product.
    - may define a default implementation of the FM
    - may call the FM to create a product
- ConcreteCreator
  - overrides FM to provide an instance of ConcreteProduct

Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate ConcreteProduct

Consequences

- Eliminate binding of application specific classes into your code.
  - creational code only deals with the Product interface
- Provide hooks for subclassing
  - subclasses can change this way the product that is created
- Clients might have to subclass the Creator just to create a particular ConcreteProduct object.

Implementation Issues

- Varieties of Factory Methods
  - Creator class is **abstract**
    - does not provide an implementation for the FM it declares
    - requires subclasses
  - Creator is a **concrete** class
    - provides default implementation
    - FM used for flexibility
    - Create objects in a separate operation so that subclasses can override it
- Parametrization of Factory Methods
  - A variation on the pattern lets the factory method create multiple kinds of products
  - a parameter identifies the type of Product to create
  - all created objects share the Product interface

Parameterizing the Factory

```
class Creator {
public:
   virtual Product * create(productId);
};

Product* Creator::create(ProductId id) {
   if (id == MINE) return new MyProduct;
   if (id == YOURS) return new YourProduct;
}

Product * MyCreator::create(ProductId id) {
   if (id == MINE) return new YourProduct;
   if (id == YOURS) return new MyProduct;
   if (id == THEIRS) return new TheirProduct;
   return Creator::create(id); // called if others fail
}
```

- selectively extend or change products that get created

Static Factory Method

```
abstract class Shape {
  public abstract void draw();
  public abstract void erase();
  public static Shape factory(String type) {
    if(type.equals("Circle")) return new Circle();
    if(type.equals("Square")) return new Square();
    throw new RuntimeException(
      "Bad shape creation: " + type);
  }
}

class Circle extends Shape {
  Circle() {} // Package-access constructor
  public void draw() {
    System.out.println("Circle.draw");
  }
  public void erase() {
    System.out.println("Circle.erase");
  }
}
```

Java: forName and Factory Methods

```
class Creator {
     public Product FactoryMethod(String productType) {
        Class productClass = Class.forName(productType);
        return (Product) productClass.newInstance();
     }
}


Product theBest = new Creator().FactoryMethod("ProductA");

theBest.newInstance();
```

```
import java.util.*;

class AbstractFactory {
    public Product make(String c) {
     try {
        Class prod = Class.forName(c);
        return (Product) prod.newInstance();
     }
     catch(Exception e) {
        System.out.println("Error");
        System.exit(1);
        return null;
     }
    }
}
```

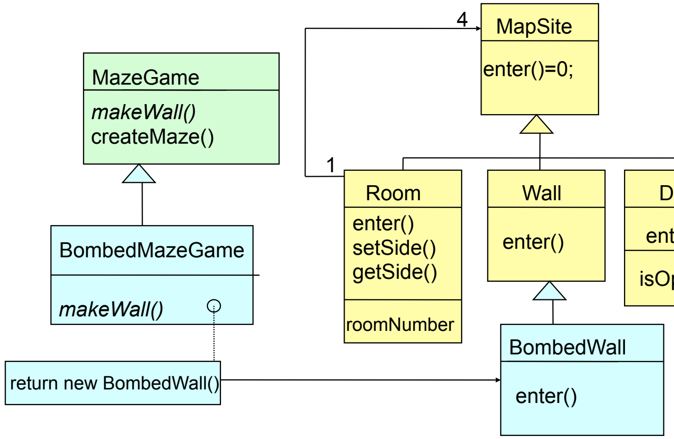
```
abstract class Product {
    abstract public void doSomething();
}

class ProductA extends Product {
    public void doSomething() {
     System.out.println("ProductA");
    }
}

class ProductB extends Product {
    public void doSomething() {
     System.out.println("ProductB");
    }
}
```

```
class Main {
    public static void main(String[] args) {
     AbstractFactory af = new AbstractFactory();

     af.make(args[0]).doSomething();
    }
}
```

C++: Templates to Avoid Subclassing

```
template <class ProductType>
class Creator
     {
     public:
          virtual Product* FactoryMethod();
     }


template <class ProductType>
Product* Creator::FactoryMethod( ) {
     return new ProductType();
}

// ....
Creator<ConcreteProduct> theBest;
Product* bestProduct = theBest.FactoryMethod();
```

Revisiting the Solution to the Maze Problem...



Idea 3:
Factory Method in Product

- Make the product responsible for creating itself
  - e.g. let the Door know how to construct an instance of it rather than the MazeGame
- The client of the product needs a reference to the "creator"
  - specified in the constructor
- see next slide...

```
class Room : public MapSite  {
  public:
   virtual Room* makeRoom(int no)  {
           return new Room(no);
      }
   // ...
};

class RoomWithBomb : public Room {
  public:
   Room* makeRoom(int no)  {
           return new RoomWithBomb();
      }
   // ...
};

// ...
```

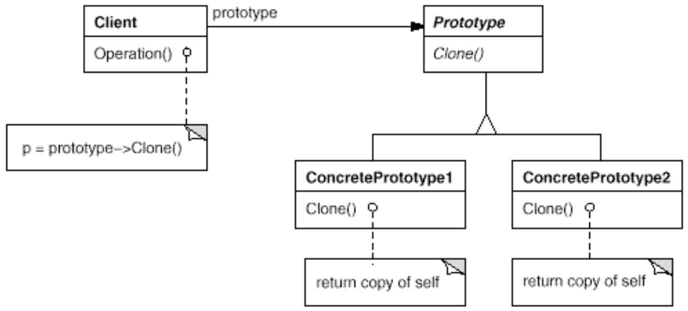
```
class MazeGame {
  protected:
   Room* roomMaker;
   // ...
  public:
   MazeGame(Room* rfactory) {
     roomMaker = rfactory;
   }

   public Maze* CreateMaze() {
     Maze aMaze = new Maze();

     Room r1 = roomMaker->makeRoom( 1 );
     // ...
};
```

The Prototype Pattern

Basic Aspects

- Intent
  - Specify the kinds of objects to create using a prototypical instance
  - Create new objects by copying this prototype
- Applicability
  - when a client class should be independent of how its products are created, composed, and represented **and**
  - when the classes to instantiate are specified at run-time

Structure



Participants & Collaborations

- Prototype
  - declares an interface for cloning itself.
- ConcretePrototype
  - implements an operation for cloning itself.
- Client
  - creates a new object by asking a prototype to clone itself.
- A client asks a prototype to clone itself.
- The client class must initialize itself in the constructor
  - with the proper concrete prototype.

Consequences

- Adding and removing products at run-time
- Reduced subclassing
  - avoid parallel hierarchy for creators
- Each subclass of Prototype must implement `clone`
  - difficult when classes already exist or
  - internal objects don't support copying or have circular references

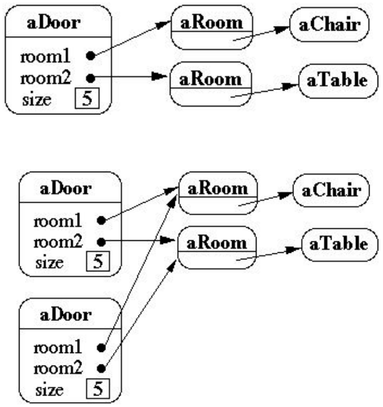
Implementation Issues

- Using a Prototype manager
  - number of prototypes isn't fixed
    - keep a registry → **prototype manager**
  - clients instead of knowing the prototype know a manager
    - associative store
- Initializing clones
  - heterogeneity of initialization methods
  - write an `Initialize` method
- Implementing the `clone` operation
  - shallow vs. deep copy

Shallow Copy vs. Deep Copy

Original

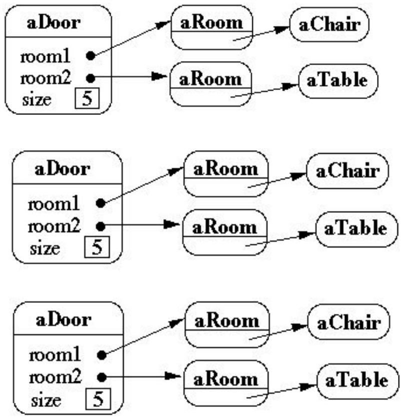
Shallow Copy



Shallow Copy vs. Deep Copy (2)

Original

Deep Copy



Cloning in C++ – Copy Constructors

```
class Door {
  public:
    Door();
    Door( const Door&);
    virtual Door* clone() const;
    virtual void Initialize( Room*, Room* );
 private:
    Room* room1; Room* room2;
};

//Copy constructor
Door::Door ( const Door& other )  {
  room1 = other.room1; room2 = other.room2;
}

Door* Door::clone() {
  return new Door( *this );
}
```

Cloning in Java – `Object clone()`

`protected Object clone() throws CloneNotSupportedException`

- Creates a clone of the object.
  - allocate a new instance and,
  - place a *bitwise clone* of the current object in the new object.

```
class Door implements Cloneable  {
  public void Initialize( Room a, Room b) {
    room1 = a; room2 = b;
  }

  public Object clone() throws CloneNotSupportedException {
          return super.clone();
  }
  Room room1, room2;
}
```

```
class Room : public MapSite  {
  public:
   virtual Room* makeRoom(int no)  {
           return new Room(no);
     }
   // ...
};

class RoomWithBomb : public Room {
  public:
   Room* makeRoom(int no)  {
           return new RoomWithBomb();
     }
   // ...
};

// ...
```

Is this a Prototype?

```
class MazeGame {
  protected:
   Room* roomMaker;
   // ...
  public:
   MazeGame(Room* rfactory) {
     roomMaker = rfactory;
   }

   public Maze* CreateMaze() {
     Maze aMaze = new Maze();

     Room r1 = roomMaker->makeRoom( 1 );
     // ...
};
```

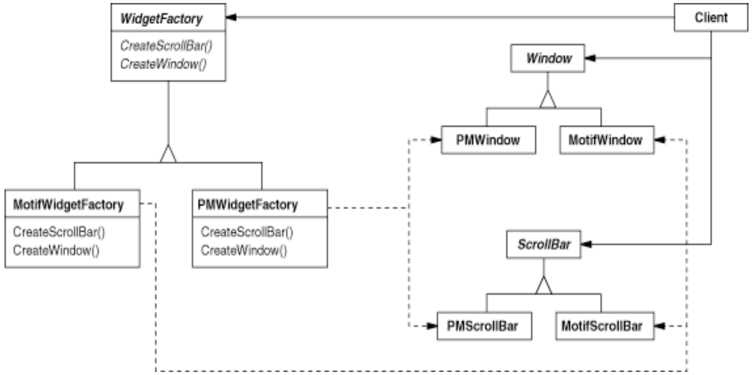
```
class MazePrototypeFactory {
public:
  MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

  virtual Maze* MakeMaze() const;
  virtual Room* MakeRoom(int) const;
  virtual Wall* MakeWall() const;
  virtual Door* MakeDoor(Room*, Room*) const;
private:
  Maze* _prototypeMaze; Room* _prototypeRoom;
  Wall* _prototypeWall; Door* _prototypeDoor;
};

MazePrototypeFactory::MazePrototypeFactory (
  Maze* m, Wall* w, Room* r, Door* d) {
   _prototypeMaze = m; _prototypeWall = w;
   _prototypeRoom = r; _prototypeDoor = d;
}


Wall* MazePrototypeFactory::MakeWall () const {
  return _prototypeWall->Clone();
}

Door* MazePrototypeFactory::MakeDoor (
  Room* r1, Room *r2) const {
     Door* door = _prototypeDoor->Clone();
     door->Initialize(r1, r2);
     return door;
}
```

Creating a maze for a game..........

```
MazePrototypeFactory simpleMazeFactory
(
  new Maze, new Wall, new Room, new Door
);

MazeGame game;
Maze* maze =
game.CreateMaze(simpleMazeFactory);
```

Abstract Factory

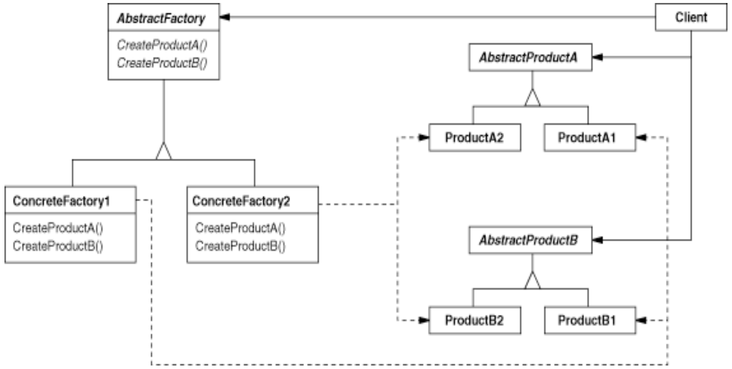
Introductive Example



Basic Aspects

- Intent
  - Provide an interface for creating **families of related or dependent objects** without specifying their concrete classes
- Applicability
  - System should be independent of how its products are created, composed and represented
  - System should be configured with one of multiple families of products
  - Need to **enforce** that a family of product objects is used together

Structure



Participants & Collaborations

- Abstract Factory
  - declares an interface for operations to create abstract products
- ConcreteFactory
  - implements the operations to create products
- AbstractProduct
  - declares an interface for a type of product objects
- ConcreteProduct
  - declares an interface for a type of product objects
- Client
  - uses only interfaces decl. by AbstractFactory and AbstractProduct
- A single instance of a ConcreteFactory created.
  - create products having a particular implementation

Consequences

- Isolation of concrete classes
  - appear in `ConcreteFactories` not in client's code
- Exchanging of product families becomes easy
  - a `ConcreteFactory` appears only in one place
    - easy to change
- Promotes consistency among products
  - all products in a family change **at once**, and change **together**
- Supporting new kinds of products is difficult
  - requires a change in the interface of AbstractFactory
  - ... and consequently all subclasses

Implementation Issues

- Factories as Singletons
  - to assure that only one ConcreteFactory per product family is created
- Creating the Products
  - collection of Factory Methods
  - can be also implemented using Prototype
    - define a prototypical instance for each product in ConcreteFactory
- Defining Extensible Factories
  - a single factory method with parameters
  - more flexible, less safe!

Creating Products...

- ...using own factory methods

```
abstract class WidgetFactory {
   public Window createWindow();
   public Menu createMenu();
   public Button createButton();
}


class MacWidgetFactory extends WidgetFactory {
     public Window createWindow()
       { return new MacWindow() }
     public Menu createMenu()
       { return new MacMenu() }
     public Button createButton()
       { return new MacButton()  }
}
```

Creating Products...

- ... using product's factory methods
  - subclass just provides the concrete products in the constructor
  - spares the re-implementation of FM's in subclasses

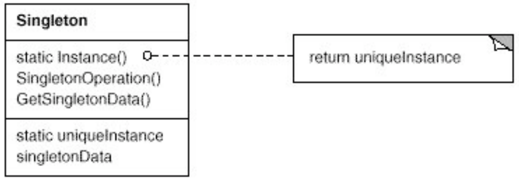
```
abstract class WidgetFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;

   public Window createWindow()
      { return windowFactory.createWindow() }
   public Menu createMenu();
      { return menuFactory.createWindow() }
   public Button createButton()
      { return buttonFactory.createWindow() }
}

class MacWidgetFactory extends WidgetFactory {
   public MacWidgetFactory() {
      windowFactory = new MacWindow();
      menuFactory = new MacMenu();
      buttonFactory = new MacButton();
   }
}
```

Singleton

Basics

- Intent
  - Ensure a class has only one instance and provide a global point of access to it
- Applicability
  - want exactly one instance of a class
  - accessible to clients from one point
  - want the instance to be extensible
  - can also allow a countable number of instances
  - improvement over global namespace
  - better than static class:
    - can't change mind
    - methods never virtual

Structure of the Pattern



Put constructor in private/protected data section

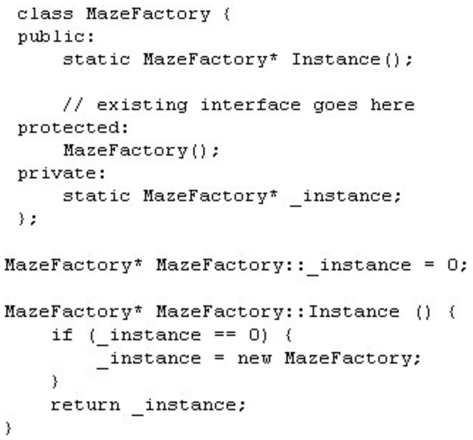
Participants and Collaborations

- Singleton
  - defines an **`Instance`** method that becomes the single "gate" by which clients can access its unique instance.
    - **`Instance`** is a class method (static member function in C++)
  - may be responsible for creating its own unique instance
- Clients access Singleton instances solely through the ***`Instance`*** method

Consequences

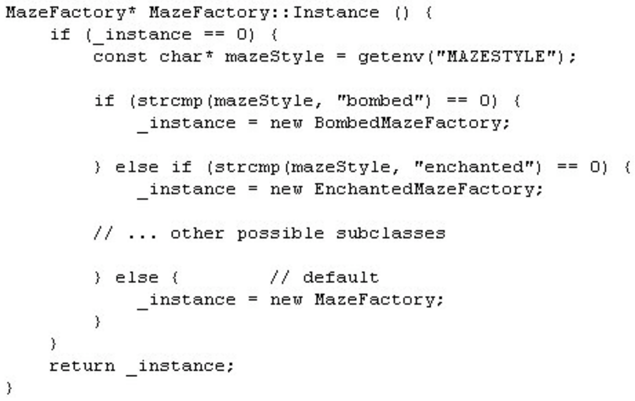
- Controlled access to sole instance
- Permits refinement of operations and representation
- Permits a variable (but precise) number of instances
- Reduced global name space

Making a single MazeFactory



```
 class MazeFactory {
 public:
     static MazeFactory* Instance();

     // existing interface goes here
 protected:
     MazeFactory();
 private:
     static MazeFactory* _instance;
 };

MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

What if there are subclasses of MazeFactory?



```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;

        // ... other possible subclasses

        } else {        // default
            _instance = new MazeFactory;
        }
    }
    return _instance;
}
```