Creational Patterns

- Patterns used to abstract the process of instantiating objects.
  - class-scoped patterns
    - uses inheritance to choose the class to be instantiated
      - Factory Method
  - object-scoped patterns
    - uses delegation
      - Abstract Factory
      - Builder
      - Prototype
      - Singleton

Importance

- Becomes important as emphasis moves towards dynamically composing smaller objects to achieve complex behaviours.
  - need more than just instantiating a class
  - need consistent ways of creating related objects.

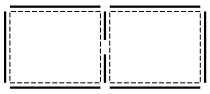
Recurring Themes

- Hide the secret about which concrete classes the system uses.
- Hide the secret about how instances are created and associated.

- Gives flexibility in
  - what gets created
  - who creates it
  - how it gets created
  - when it get gets created

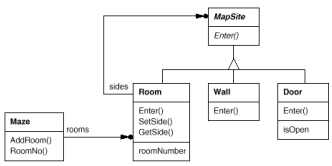
Running Example

- Building a maze for a computer game.



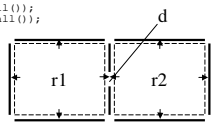
- A Maze is a set of Rooms.
- A Room knows its neighbours.
  - another room
  - a wall
  - a door

Maze Example



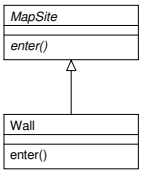
Creating Mazes

```
public class MazeGame
{
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze();
    }

    public Maze createMaze() {
        Room r1 = new Room(1);
        Room r2 = new Room(2);
        Door d  = new Door(r1,r2);

        r1.setSide(Direction.North, new Wall());
        r1.setSide(Direction.East, d);
        r1.setSide(Direction.West, new Wall());
        r1.setSide(Direction.South, new Wall());

        r2.setSide(Direction.North, new Wall());
        r2.setSide(Direction.West, d);
        r2.setSide(Direction.East, new Wall());
        r2.setSide(Direction.South, new Wall());

        Maze m  = new Maze();
        m.addRoom(r1);
        m.addRoom(r2);
        return m;
    }
}
```



Maze Classes

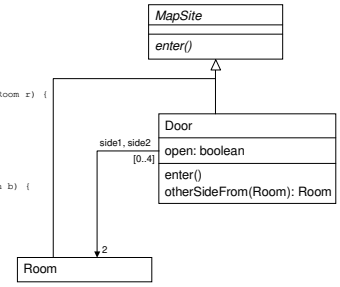
```
public abstract class MapSite
{
    public abstract void enter();
}
```




```
public class Wall extends MapSite
{
    public void enter() {
    }
}
```


Maze Classes

```
public class Door extends MapSite
{
    Door(Room s1, Room s2) {
        side1 = s1;
        side2 = s2;
    }

    public void enter() {
    }

    public Room otherSideFrom(Room r) {
        if( r == side1 )
            return side2;
        else if( r == side2 )
            return side1;
        else
            return null;
    }

    public void setOpen(boolean b) {
        open = b;
    }

    public boolean getOpen() {
        return open;
    }

    private Room side1;
    private Room side2;
    boolean open;
}
```

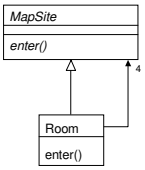


Maze Classes

```
public class Direction
{
    public final static int First = 0;
    public final static int North = First;
    public final static int South = North+1;
    public final static int East = South+1;
    public final static int West = East+1;
    public final static int Last = West;
    public final static int Num = Last-First+1;
}
```

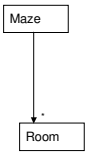

Maze Classes

```
public class Room extends MapSite
{
    public Room(int r) {
        room_no = r;
    }

    public void enter() {
    }

    public void setSide(int direction, MapSite ms) {
        side[direction] = ms;
    }

    public MapSite getSide(int direction) {
        return side[direction];
    }

    public void setRoom_no(int r) {
        room_no = r;
    }

    public int getRoom_no() {
        return room_no;
    }

    private int room_no;
    private MapSite[] side = new MapSite[Direction.Num];
}
```



Maze Classes

```
import java.util.Vector;

public class Maze
{
    public void addRoom(Room r) {
        rooms.addElement(r);
    }

    public Room getRoom(int r) {
        return (Room)rooms.elementAt(r);
    }

    public int numRooms() {
        return rooms.size();
    }

    private Vector rooms = new Vector();
}
```



Maze Creation

```
public Maze createMaze() {
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door d  = new Door(r1,r2);

    r1.setSide(Direction.North, new Wall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, new Wall());
    r1.setSide(Direction.South, new Wall());

    r2.setSide(Direction.North, new Wall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, new Wall());
    r2.setSide(Direction.South, new Wall());

    Maze m  = new Maze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```

Maze Creation

- Fairly complex member just to create a maze with two rooms.
- Obvious simplification:
  - Room() could initialize sides with 4 new Wall()
  - That just moves the code elsewhere.
- Problem lies elsewhere: *inflexibility*
  - Hard-codes the maze creation
  - Changing the layout can only be done by re-writing, or overriding and re-writing.

Creational Patterns Benefits

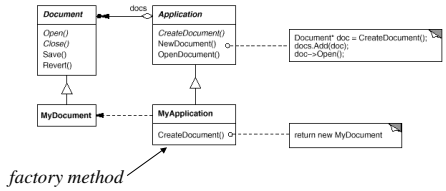
- Will make the maze more flexible.
  - easy to change the components of a maze
  - e.g., DoorNeedingSpell, EnchantedRoom
    - How can you change createMaze() so that it creates mazes with these different kind of classes?
    - Biggest obstacle is hard-coding of class names.

Creational Patterns

- If createMaze() calls virtuals to construct components
  - Factory Method
- If createMaze() is passed a parameter object to create rooms, walls, …
  - Abstract Factory
- If createMaze() is passed a parameter object to create and connect-up mazes
  - Builder
- If createMaze is parameterized with various prototypical rooms, doors, walls, … which it copies and then adds to the maze
  - Prototype
- Need to ensure there is only one maze per game, and everybody can access it, and can extend or replace the maze without touching other code.
  - Singleton

Factory Method

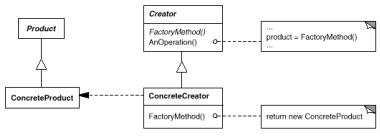
- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- a.k.a. Virtual Constructor
- e.g., app framework



Applicability

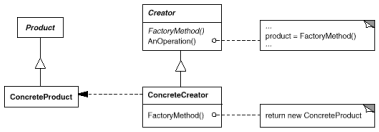
- Use when:
  - A class can't anticipate the kind of objects to create.
  - Hide the secret of which helper subclass is the current delegate.

Structure



- Product
  - defines the interface of objects the factory method creates
- ConcreteProduct
  - implements the Product interface

Structure



- Creator
  - declares the factory method which return a Product type.
  - [define a default implementation]
  - [call the factory method itself]
- ConcreteCreator
  - overrides the factory method to return an instance of a ConcreteProduct

Sample Code

```
public class MazeGame {
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze();
    }

    private Maze makeMaze() { return new Maze(); }
    private Wall makeWall() { return new Wall(); }
    private Room makeRoom(int r) { return new Room(r); }
    private Door makeDoor(Room r1, Room r2) { return new Door(r1,r2); }

    public Maze createMaze() {
        …
    }
}
```


Sample Code

```
public Maze createMaze() {
    Room r1 = makeRoom(1);
    Room r2 = makeRoom(2);
    Door d  = makeDoor(r1,r2);

    r1.setSide(Direction.North, makeWall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, makeWall());
    r1.setSide(Direction.South, makeWall());

    r2.setSide(Direction.North, makeWall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, makeWall());
    r2.setSide(Direction.South, makeWall());

    Maze m  = makeMaze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```


Sample Code

```
public class BombedMazeGame extends MazeGame
{
    private Wall makeWall() { return new BombedWall(); }
    private Room makeRoom(int r) { return new RoomWithABomb(r); }
}


public class EnchantedMazeGame extends MazeGame
{
    private Room makeRoom(int r)
        { return new EnchantedRoom(r, castSpell()); }
    private Door makeDoor(Room r1, Room r2)
        { return new DoorNeedingSpell(r1,r2); }
    private Spell castSpell()
        { return new Spell(); }
}
```


Sample Code

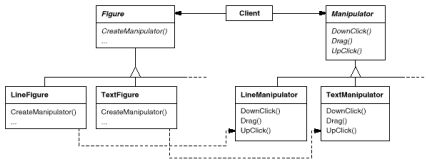
```
public static void main(String args[]) {
    Maze m = new EnchantedMazeGame().createMaze();
}


public static void main(String args[]) {
    Maze m = new BombedMazeGame().createMaze();
}
```


Consequences

- Advantage:
  - Eliminates the need to bind to specific implementation classes.
    - Can work with any user-defined ConcreteProduct classes.
- Disadvantage:
  - Uses an inheritance dimension
  - Must subclass to define new ConcreteProduct objects
    - interface consistency required

Consequences

- Provides hooks for subclasses
  - always more flexible than direct object creation
- Connects parallel class hierarchies
  - hides the secret of which classes belong together



Implementation

- Two major varieties
  - creator class is abstract
    - *requires* subclass to implement
  - creator class is concrete, and provides a default implementation
    - *optionally allows* subclass to re-implement
- Parameterized factory methods
  - takes a class id as a parameter to a generic make() method.
  - (more on this later)
- Naming conventions
  - use 'makeXXX()' type conventions (e.g., MacApp – DoMakeClass())
- Can use templates instead of inheritance
- Return class of object to be created
  - or, store as member variable

Question

- What gets printed?

```
public class Main {
    public static void main(String args[])
        { new DerivedMain(); }
    public String myClass()
        { return "Main"; }
}

class DerivedMain extends Main {
    public DerivedMain()
        { System.out.println(myClass()); }
    public String myClass()
        { return "DerivedMain"; }
}
```

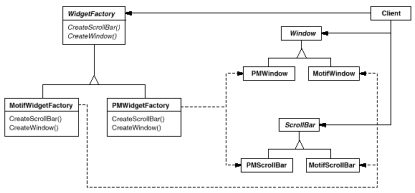

Implementation

- Lazy initialization
  - In C++, subclass vtable pointers aren't installed until after parent class initialization is complete.
    - DON'T CREATE DURING CONSTRUCTION!
    - can use lazy instatiation:

```
Product getProduct() {
    if( product == null ) {
        product = makeProduct();
    }
    return product;
}
```


Abstract Factory

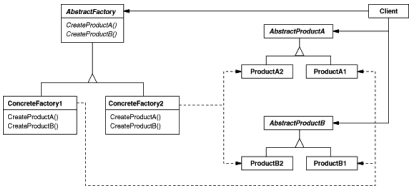
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- e.g., look-and-feel portability
  - independence
  - enforced consistency



Applicability

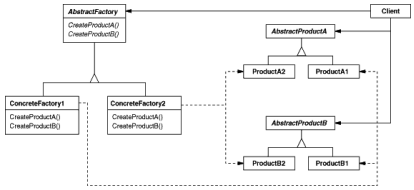
- Use when:
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
  - you want to hide and reuse awkward or complex details of construction

Structure



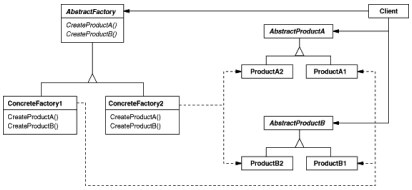
- AbstractFactory
  - declares an interface for operations that create product objects.
- ConcreteFactory
  - implements the operations to create concrete product objects.

Structure



- AbstractProduct
  - declares an interface for a type of product object.
- ConcreteFactory
  - defines a product to be created by the corresponding concrete factory.
  - implements th AbstractProduct interface.

Structure



- Client
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes.

Sample Code

```
public class MazeFactory {
    Maze makeMaze() { return new Maze(); }
    Wall makeWall() { return new Wall(); }
    Room makeRoom(int r) { return new Room(r); }
    Door makeDoor(Room r1, Room r2) { return new Door(r1,r2);}
}
```


Maze Creation (old way)

```
public Maze createMaze() {
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door d  = new Door(r1,r2);

    r1.setSide(Direction.North, new Wall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, new Wall());
    r1.setSide(Direction.South, new Wall());

    r2.setSide(Direction.North, new Wall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, new Wall());
    r2.setSide(Direction.South, new Wall());

    Maze m  = new Maze();
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```


Sample Code

```
public Maze createMaze(MazeFactory factory) {
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door d  = factory.makeDoor(r1,r2);

    r1.setSide(Direction.North, factory.makeWall());
    r1.setSide(Direction.East, d);
    r1.setSide(Direction.West, factory.makeWall());
    r1.setSide(Direction.South, factory.makeWall());

    r2.setSide(Direction.North, factory.makeWall());
    r2.setSide(Direction.East, d);
    r2.setSide(Direction.West, factory.makeWall());
    r2.setSide(Direction.South, factory.makeWall());

    Maze m  = factory.makeMaze()
    m.addRoom(r1);
    m.addRoom(r2);
    return m;
}
```

Sample Code

```
public class EnchantedMazeFactory extends MazeFactory {
    public Room makeRoom(int r) {
        return new EnchantedRoom(r, castSpell());
    }

    public Door makeDoor(Room r1, Room r2) {
        return new DoorNeedingSpell(r1,r2);
    }

    private protected castSpell() {
        // randomly choose a spell to cast;
        …
    }
}
```


Sample Code

```
public class MazeGame
{
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze(new MazeFactory());
    }
}

public class MazeGame
{
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze(new EnchantedMazeFactory());
    }
}
```


Consequences

- It isolates concrete classes
  - Helps control the classes of objects that an application creates.
  - Isolates clients from implementation classes
  - Clients manipulate instances through abstract interfaces
  - Product class names are isolated in the implementation of the concrete factory
    - they do not appear in the client code

Consequences

- It makes exchanging product families easy
  - The class of a concrete factory appears only once in the app.
    - where it's instantiated
  - Easy to change the concrete factory an app uses.
  - The whole product family changes at once

Consequences

- It promotes consistency among products
  - When products are designed to work together, it's important that an application use objects only from one family at a time.
  - AbstractFactory makes this easy to enforce.

Consequences

- Supporting new kinds of products is difficult.
  - Extending AbstractFactory to produce new product types isn't easy
    - extend factory interface
    - extend all concrete factories
    - add a new abstract product
    - + the usual (implement new class in each family)

Implementation

- Factories as Singletons
  - An app typically needs only one instance of a ConcreteFactory per product family.
  - Best implemented as a Singleton

Implementation

- Defining extensible factories
  - Hard to extend to new product types
  - Add parameter to operations that create products
    - need only `make()`
    - less safe
    - more flexible
    - easier in languages that have common subclass
      - e.g. java Object
    - easier in more dynamically-typed languages
      - e.g., Smalltalk
    - all products have same abstract interface
      - can downcast – not safe
      - classic tradeoff for a very flexible/extensible interface

Implementation

- Creating the products
  - AbstractFactory declares an interface for product creation
  - ConcreteFactory implements it. How?
    - Factory Method
      - virtual overrides for creation methods
      - simple
      - requires new concrete factories for each family, even if they only differ slightly
    - Prototype
      - concrete factory is initialized with a prototypical instance of each product in the family
      - creates new products by cloning
      - doesn't require a new concrete factory class for each product family
      - variant: can register class objects

Prototype-based Implementation

```
abstract class AbstractProduct implements Cloneable {
    public abstract int geti();
    public abstract Object clone();
}

class ConcreteProduct extends AbstractProduct
{
    public ConcreteProduct(int j) { i = j; }

    public Object clone() { return new ConcreteProduct(i); }

    public int geti() { return i; }

    private int i;
}
```


Prototype-based Implementation

```
import java.util.Hashtable;

public class ConcreteFactory {
    void addProduct(AbstractProduct p, String name) {
        map.put(name, p);
    }

    AbstractProduct make(String name) {
        return (AbstractProduct)
            ((AbstractProduct)map.get(name)).clone();
    }

    private Hashtable map = new Hashtable();
}
```


Prototype-based Implementation

```
public class Main {
    public static void main(String args[]) {
        ConcreteFactory f = new ConcreteFactory();
        f.addProduct(new ConcreteProduct(42), "ap");
        AbstractProduct p = f.make("ap");
        System.out.println(p.geti());
    }
}
```

Class Registration Implementation

```
abstract class AbstractProduct {
    public abstract int geti();
}

class ConcreteProduct extends AbstractProduct {
    public int geti() { return i; }
    private int i = 47;
}

public class ConcreteFactory {
    void addProduct(Class c, String name) {
        map.put(name, c);
    }

    Product make(String name) throws Exception {
        Class c = (Class)map.get(name);
        return (Product) c.newInstance();
    }

    private Hashtable map = new Hashtable();
}
```


Class Registration Implementation

```
public class Main {
    public static void main(String args[]) throws Exception {
        ConcreteFactory f = new ConcreteFactory();
        f.addProduct(Class.forName("ConcreteProduct"), "ap");
        AbstractProduct p = f.make("ap");
        System.out.println(p.geti());
    }
}
```