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.

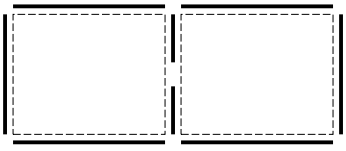
Recurring Themes

- 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

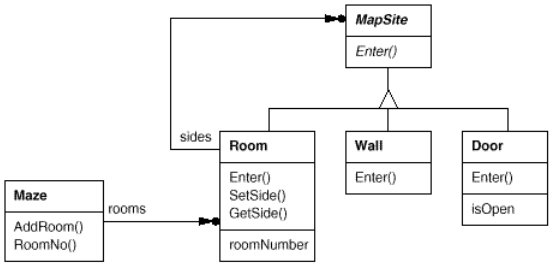
Running Example

- Building a maze for a computer game.



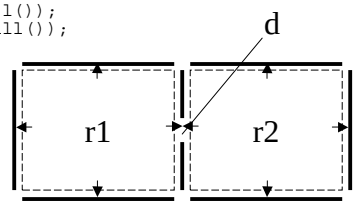
- A Maze is a set of Rooms.
- A Room knows its neighbours.
  - another room
  - a wall
  - a door

Maze Example



Creating Mazes

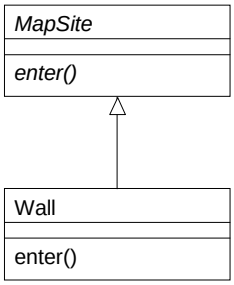
```
public class MazeGame
{
    public static void main(String args[]) {
        Maze m = new MazeGame().createMaze();
    }

    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

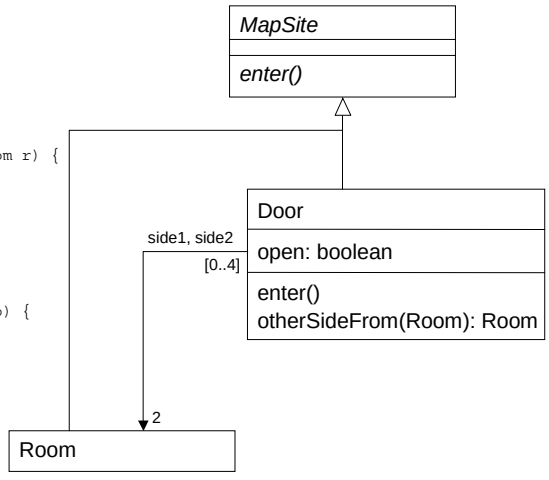
```
public abstract class MapSite
{
    public abstract void enter();
}
```

```
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;
}
```

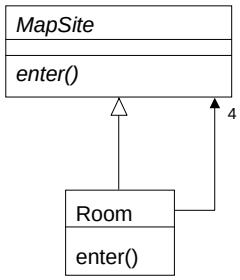


Maze Classes

```
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;
}
```

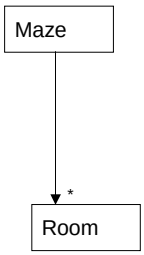

Maze Classes

```
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

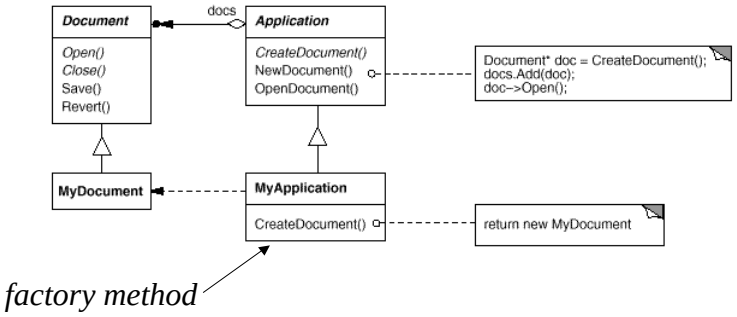
- Will make the maze more flexible.
  - easy to change the components of a maze
  - e.g., DoorNeedingSpell, EnchantedRoom
    - How can you change createMaze() so that it creates mazes with these different kind of classes?
    - Biggest obstacle is hard-coding of class names.

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

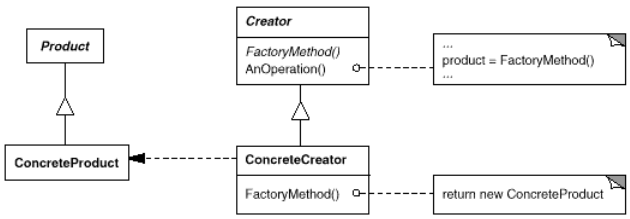
- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- a.k.a. Virtual Constructor
- e.g., app framework



Applicability

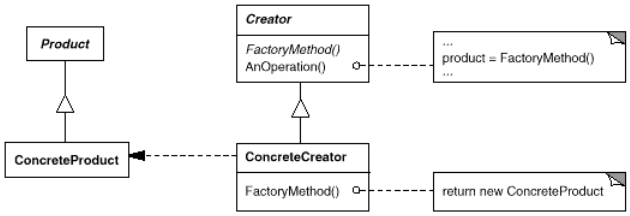
- Use when:
  - A class can't anticipate the kind of objects to create.
  - Hide the secret of which helper subclass is the current delegate.

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();
}
```

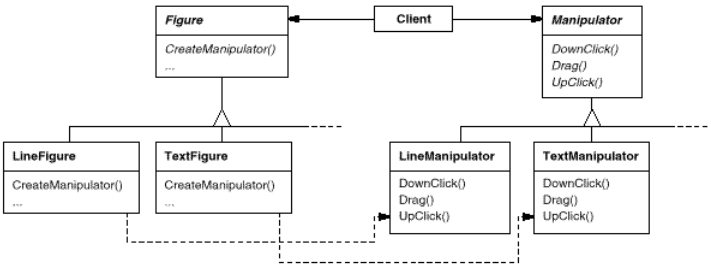
```
public static void main(String args[]) {
    Maze m = new BombedMazeGame().createMaze();
}
```


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"; }
}
```

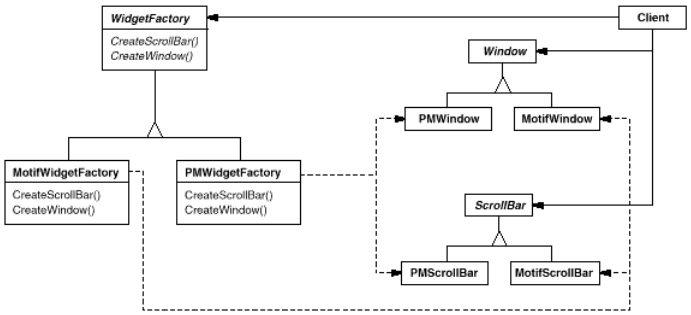

Implementation

- 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

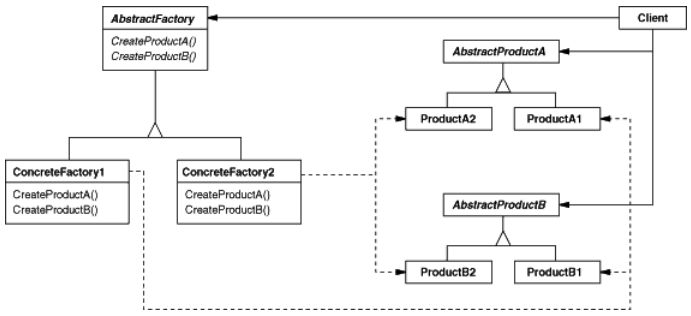
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- e.g., look-and-feel portability
  - independence
  - enforced consistency



Applicability

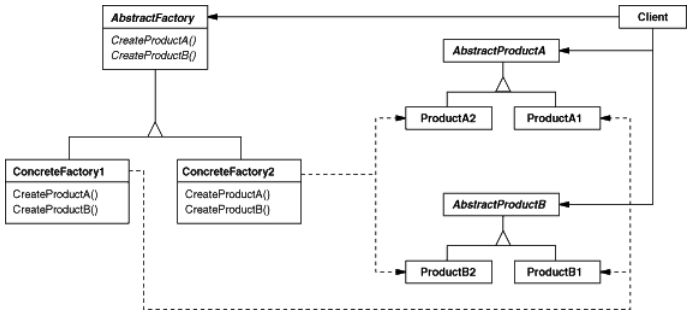
- Use when:
  - a system should be independent of how its products are created, composed, and represented
  - a system should be configured with one of multiple families of products.
  - a family of related product objects is designed to be used together, and you need to enforce this constraint.
  - 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



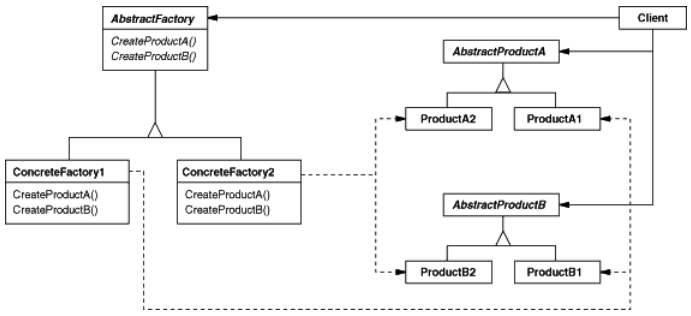
- AbstractFactory
  - declares an interface for operations that create product objects.
- ConcreteFactory
  - implements the operations to create concrete product objects.

Structure



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

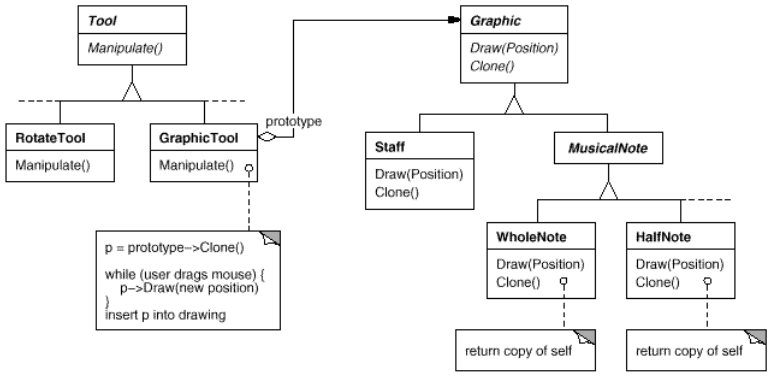
```
abstract class AbstractProduct {
    public abstract int geti();
}

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());
    }
}
```

Prototype

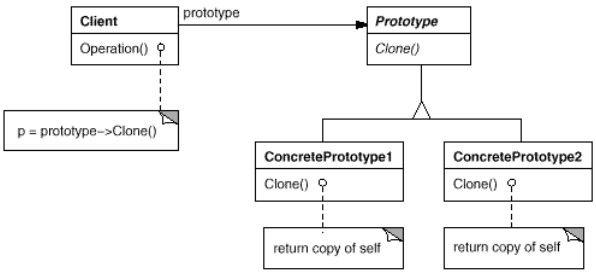
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - e.g., reduce # of classes (# of tools) by initializing a generic tool with a prototype



Applicability

- Use When:
  - the classes to be instantiated are specified at run-time
    - e.g., for dynamic loading
  - to avoid building a class hierarchy of factories to parallel the hierarchy of products
  - when instances can have only one of a few states
    - may be better to initialize once, and then clone prototypes

Structure



- 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

Sample Code

```
public class MazePrototypeFactory extends MazeFactory
{
    private Maze prototypeMaze;
    private Wall prototypeWall;
    private Room prototypeRoom;
    private Door prototypeDoor;

    public MazePrototypeFactory(Maze pm, Wall pw, Room pr, Door pd) {
        prototypeMaze = pm;
        prototypeWall = pw;
        prototypeRoom = pr;
        prototypeDoor = pd;
    }

    …
}
```

Sample Code

```
public class MazePrototypeFactory extends MazeFactory
{
    Wall makeWall() {
        Wall wall = null;
        try {
            wall = (Wall)prototypeWall.clone();
        } catch(CloneNotSupportedException e) { throw new Error(); }
        return wall;
    }
    Room makeRoom(int r) {
        Room room = null;
        try {
            room = (Room)prototypeRoom.clone();
        } catch(CloneNotSupportedException e) { throw new Error(); }
        room.initialize(r);
        return room;
    }
    …
}
```


Sample Code

```
public abstract class MapSite implements Cloneable
{
    public abstract void enter();

    public String toString() {
        return getClass().getName();
    }

    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

Sample Code

```
public class Door extends MapSite
{
    public Door(Room s1, Room s2) {
        initialize(s1,s2);
    }

    public void initialize(Room s1, Room s2) {
        side1 = s1;
        side2 = s2;
        open = true;
    }

    private Room side1;
    private Room side2;
    boolean open;

    …
}
```


Sample Code

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

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

    public Object clone() throws CloneNotSupportedException {
        Room r = (Room)super.clone();
        r.side = new MapSite[Direction.Num];
        return r;
    }
    …
    private int room_no;
    private MapSite[] side = new MapSite[Direction.Num];
}
```

Sample Code

```
public class EnchantedRoom extends Room
{
   public EnchantedRoom(int r, Spell s) {
       super(r);
       spell = s;
   }

   public Object clone() throws CloneNotSupportedException {
       EnchantedRoom r = (EnchantedRoom)super.clone();
       r.spell = new Spell();
       return r;
   }

   private Spell spell;
}
```


Sample Code

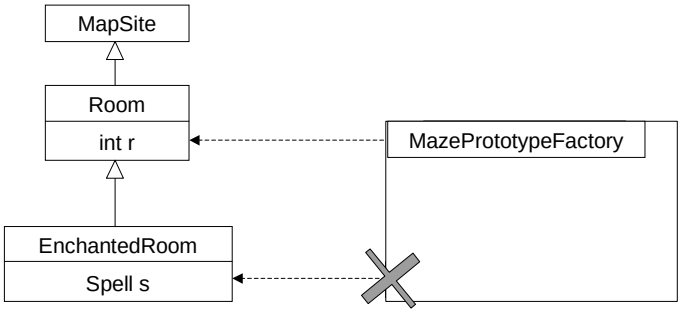
```
public static void main(String args[]) {
    MazeFactory mf = new MazePrototypeFactory(
                            new Maze(), new Wall(),
                            new Room(0), new Door(null,null));
    Maze m = new MazeGame().createMaze(mf);
}


public static void main(String args[]) {
    MazeFactory mf = new MazePrototypeFactory(
        new Maze(), new Wall(),
        (Room)Class.forName("EnchantedRoom").newInstance(),
        (Door)Class.forName("DoorNeedingSpell").newInstance());
    Maze m = new MazeGame().createMaze(mf);
}
```

Consequences

- Many of the same as AbstractFactory
- Can add and remove products at run-time
- new objects via new values
  - setting state on a prototype is analogous to defining a new class
- new structures
  - a multi-connected prototype + deep copy
- reducing subclassing
  - no need to have a factory or creator hierarchy
- dynamic load
  - cannot reference a new class's constructor statically
  - must register a prototype
- Disadvantage
  - implement clone() all over the place (can be tough).

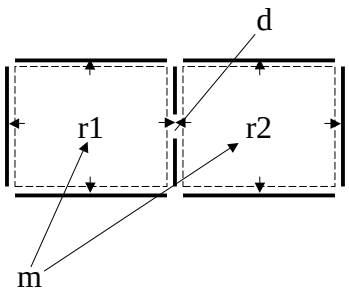
Consequences

- No parallel class hierarchy
  - awkward initialization



Implementation

- Use a prototype manager
  - store and retrieve in a registry (like Abstract Factory e.g.).
- Shallow versus deep copy
  - consider a correct implementation of clone() for Maze.
  - need a concept of looking up equivalent cloned rooms in the current maze



- m.clone()
  - r1.clone()
    - n.wall.clone()
    - e.door.clone()
      - r1.clone()
        - !!!

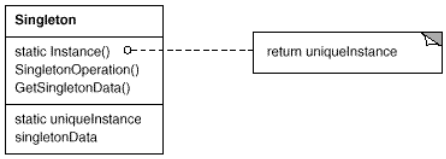
Singleton

- Ensure a class only has one instance, and provide a global point of access to it.
  - Many times need only one instance of an object
    - one file system
    - one print spooler
    - …
  - How do we ensure there is exactly one instance, and that the instance is easily accessible?
    - Global variable is accessible, but can still instantiate multiple instances.
    - make the class itself responsible

Applicability

- Use when:
  - there must be exactly one instance accessible from a well-known access point
  - the sole instance should be extensible via subclassing
    - clients should be able to use the extended instance without modifying their code

Structure



- Singleton
  - defines a class-scoped instance() operation that lets clients access its unique instance
  - may be responsible for creating its own unique instance

Sample Code

```
package penny.maze.factory;
public class MazeFactory {
    MazeFactory() { }

    private static MazeFactory theInstance = null;
    public static MazeFactory instance() {
        if( theInstance == null ) {
            String mazeKind =
                 AppConfig.getProperties().getProperty("maze.kind");
            if( mazeKind.equals("bombed") ) {
                theInstance = new BombedMazeFactory();
            } else if( mazeKind.equals("enchanted") ) {
                theInstance = new EnchantedMazeFactory();
            } else {
                theInstance = new MazeFactory();
            }
        }
        return theInstance;
    }
    …
}
```


Sample Code

file .mazerc:

```
# Maze application parameters
maze.kind=enchanted
```

Sample Code

```
import java.io.*;
import java.util.Properties;

public class AppConfig {
    public static Properties getProperties() {
        if( props == null ) {
            props = new Properties(defaults());
            try {
                props.load(new FileInputStream(".mazerc"));
            } catch(IOException e) {
                System.err.println("Cannot read .mazerc, using defaults");
            }
        }
        return props;
    }

    private static Properties defaults() {
        Properties p = new Properties();
        p.put("maze.kind", "bombed");
        return p;
    }

    private static Properties props = null;
}
```


Sample Code - More Dynamic

file .mazerc:

```
# Maze application parameters
maze.factory=EnchantedMazeFactory
```

Sample Code - More Dynamic

```
package penny.maze.factory;
public class MazeFactory {
    MazeFactory() { }

    private static MazeFactory theInstance = null;
    public static MazeFactory instance() {
        if( theInstance == null ) {
            String mazeFactory =
                 AppConfig.getProperties().getProperty("maze.factory");
            try{
                theInstance = (MazeFactory)
                              Class.forName(mazeFactory).newInstance();
            } catch(Exception e) {
                theInstance = new MazeFactory();
            }
        }
        return theInstance;
    }
    …
}
```


Consequences

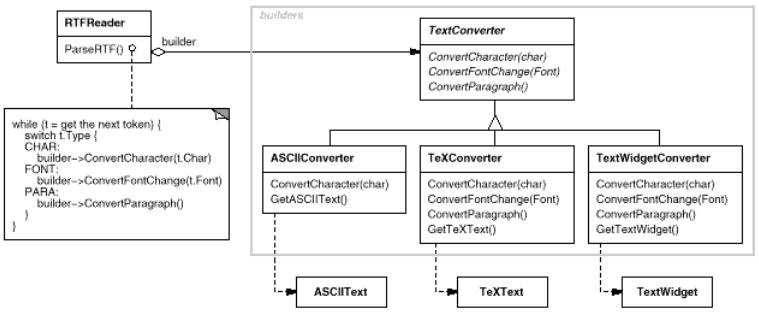
- Controlled access to sole instance.
  - Because singleton encapsulates the sole instance, it has strict control.
- Reduced name space
  - one access method only
- Variable # of instances
  - can change your mind to have e.g., 5 instances
- Easy to derive and select new classes
  - access controlled through a single point of entry

Implementation

- Ensuring a unique instance
  - can't define singleton as a global object!
    - no guarantee only one will get created
      - must prohibit instantiation
    - may not know the state settings at init time (prior to main())
    - must create whether used or not
  - can't define as a static object
    - all of the above +
      - C++ doesn't define the order of construction across translation units.
- Subclassing the singleton class
  - as shown
  - C++: implement Singleton::instance() in each sub-class, only link one in at link time.
  - registry of singletons: `instance("bombed")`
    - subclasses register in static initializers (or "init()" methods).

Builder

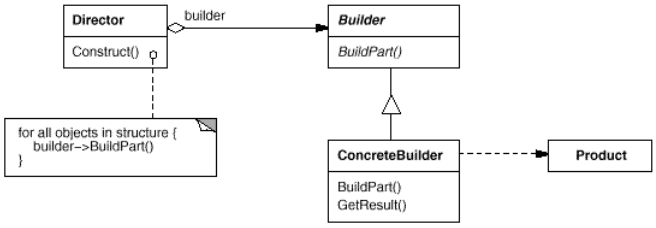
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - e.g., read in Rich Text Format, converting to may different formats on load.



Applicability

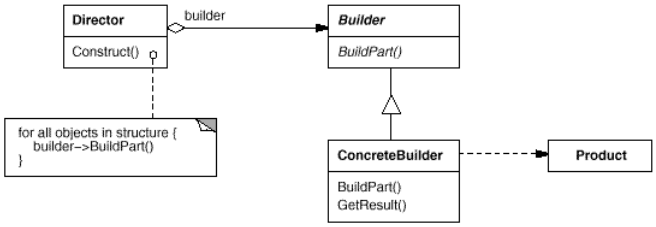
- Use When:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - the construction process must allow different representations for the object that's constructed

Structure



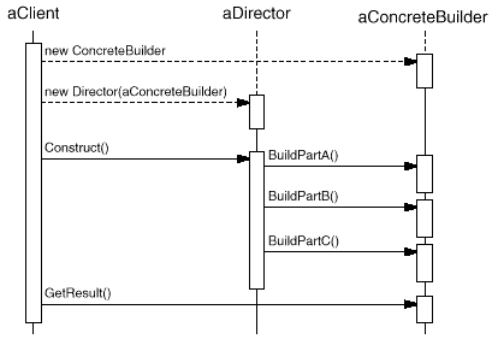
- Builder
  - specifies an abstract interface for creating parts of a Product object
- Concrete Builder
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product

Structure



- Director
  - constructs an object using the Builder interface
- Product
  - represents the complex object under construction.
  - includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

Collaborations



- The client creates the Director object and configures it with the Builder object.
- Director notifies the builder whenever a part of the product should be built.
- Builder handles requests from the director and adds parts to the product.
- The client retrieves the product from the builder.

Sample Code

```
public abstract class MazeBuilder {
    public void buildRoom(int r){}
    public void buildDoor(int r1, int direction, int r2){}
    public Maze getMaze(){return null;}
}


public class MazeGame {
    …
    public Maze createMaze(MazeBuilder b) {
        b.buildRoom(1);
        b.buildRoom(2);
        b.buildDoor(1, Direction.North, 2);
        return b.getMaze();
    }
    …
}
```


Sample Code

```
public class StandardMazeBuilder extends MazeBuilder
{
    private Maze currentMaze;

    public Maze getMaze() {
        if( currentMaze==null )
           currentMaze = new Maze();
        return currentMaze;
    }

    …
}
```

Sample Code

```
public class StandardMazeBuilder extends MazeBuilder
{
    …
    public void buildRoom(int r) {
        if( getMaze().getRoom(r) == null ) {
            Room room = new Room(r);
            getMaze().addRoom(room);
            for(int d = Direction.First; d <= Direction.Last; d++)
                room.setSide(d, new Wall());
         }
    }
    …
}
```


Sample Code

```
public class StandardMazeBuilder extends MazeBuilder
{
    …
    public void buildDoor(int r1, int d, int r2) {
        Room room1 = getMaze().getRoom(r1);
        Room room2 = getMaze().getRoom(r2);
        if( room1 == null ) {
            buildRoom(r1);
            room1 = getMaze().getRoom(r1);
        }
        if( room2 == null ) {
            buildRoom(r2);
            room2 = getMaze().getRoom(r2);
        }

        Door door = new Door(room1, room2);

        room1.setSide(d, door);
        room2.setSide(Direction.opposite(d), door);
    }
    …
}
```

Sample Code

```
public class CountingMazeBuilder extends MazeBuilder
{
    private int rooms = 0;
    private int doors = 0;

    public void buildDoor(int r1, int direction, int r2) {
        doors++;
    }

    public void buildRoom(int r) {
        rooms++;
    }

    public int getDoors() { return doors; }
    public int getRooms() { return rooms; }

}
```


Sample Code

```
public class MazeGame
{
    public static void main(String args[]) {
        MazeGame mg = new MazeGame();
        Maze m = mg.createMaze(new StandardMazeBuilder());
        System.out.println(m);

        CountingMazeBuilder cmb = new CountingMazeBuilder();
        mg.createMaze(cmb);
        System.out.println("rooms = "+cmb.getRooms());
        System.out.println("doors = "+cmb.getDoors());
    }

    …
}
```

Sample Code

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

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

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

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


Sample Code

```
public Maze createMaze(MazeBuilder b) {
    b.buildDoor(1, Direction.North, 2);
    return b.getMaze();
}
```

Consequences

- lets you vary a product's internal representation
- isolates code for construction and representation
- gives you control over the construction process

Implementation

- Assembly interface
  - sometimes can just append next element to structure
  - more often must lookup previously constructed elements
    - need an interface for doing this that hides Products
      - cookie of some sort
    - beware order of construction
- Product hierarchy?
  - often no great similarity
  - no great need
  - don’t use up a precious inheritance dimension
- abstract v.s. empty methods?
  - empty methods more generally useful
- User-installable product classes

Creational Patterns

- If createMaze() calls virtuals to construct components
  - Factory Method (class scoped)
- 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