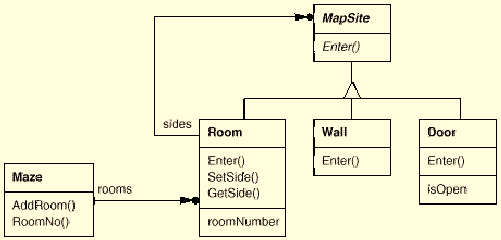
Creational Patterns

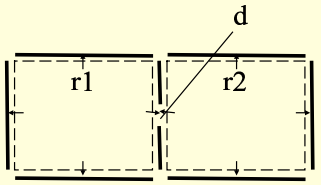
- Last week we introduced creational patterns.
- Snooping on the tutorials I'm not sure I did such a good job in lecture..
  - So now we have to have an interactive conversation
- What's the matter with constructors in createMaze?
  - Did they lie to us in csc148?
  - What did they leave out then?
  - What is new here?
  - What has this to do with OOA/OOD?

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;
    }
}
```



createMaze

- What does createMaze know?
  - It knows the structure of a particular maze game.
- Why is it more useful if it doesn't know about Constructors?
  - Because then we can reuse the code in createMaze to build mazes with the same structure but different components.
- This is the style of program that can be very effective using OO technique.
  - When interesting composite objects can be created from components that interact using a few relatively simple abstractions.

What if?

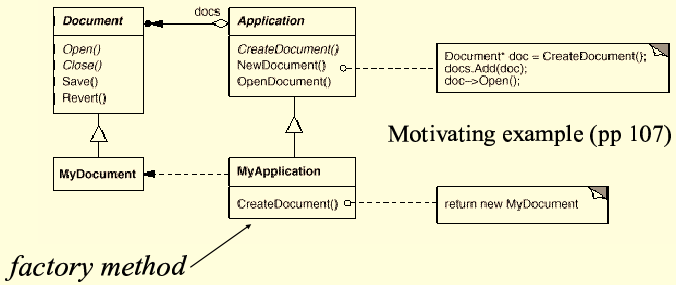
- What if instead of a trivial maze game it was a protein molecule with thousands of atoms?
- It sure would be nice to try out different simulations of the various atoms, forces, etc without having to get involved in the assembly of the atom. Over and Over.
- Suppose it's your job to extend the MazeGame with a new type of door.
  - How would a Pascal (or C) programmer do it?
  - With code copying?
  - Without code copying?
  - If you skipped the lecture -- we did this bit on the board.

Creational Patterns

- If createMaze() calls virtuals to construct components
  - Factory Method
- If createMaze() is uses a factory object to create rooms, walls, …
  - Abstract Factory
- If createMaze() is passed a object to create and connect-up mazes
  - Builder
- If createMaze is parameterized with various exemplars, or prototypes, of rooms, doors, walls, … which it clones 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



Motivating example (pp 107)

Factory Method 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() {
        …
    }
}
```

Factory Method 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;
}
```

Recall: these were constructors in "orange arrow" slide.

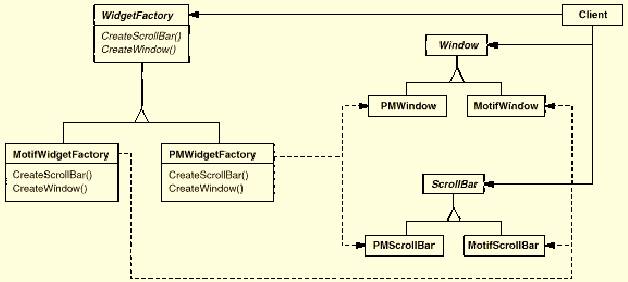
Sample Code

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

createMaze will create mazes with same structure but different components

Abstract Factory

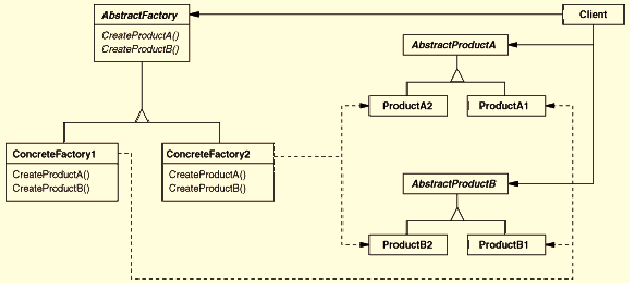
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- e.g., look-and-feel portability
  - independence
  - enforced consistency



Why Abstract Factory?

- What if it is essential for families of components to be managed separately? Makes sense to create all the family in one place -- a singleton factory object.
- Sometimes subclassing (like EnchantedMaze) is not desirable because we need the inheritance dimension for something else.

Structure



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

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);}
}
```

What if this class is abstract?
What if all the products are abstract?

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;
}
```

Now call methods on factory object

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());
    }
}
```


Looks good, but..

- You had better be happy with those abstract interfaces!
  - What if createMaze was almost right but not quite.
    - For instance, what if in some peculiar game a door depended on the something not exposed by the existing interface.
    - (Contrived) Say like the something in the basement.
  - Even if you code copied the createMaze method you wouldn't even know (for sure) which class of room to downcast to!

```
Door d  = factory.makeDoor(r1,r2);

..
r1.setSide(Direction.East, d);
..
r2.setSide(Direction.East, d);
```

  - Now what? Do we add a Basement parm to createDoor?
  - I don't mean to rain on the parade.. I just want to caution you that compositions created from abstract components lock you into the protocols you establish.

Implementation

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

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 i) { this.i = i; }

    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

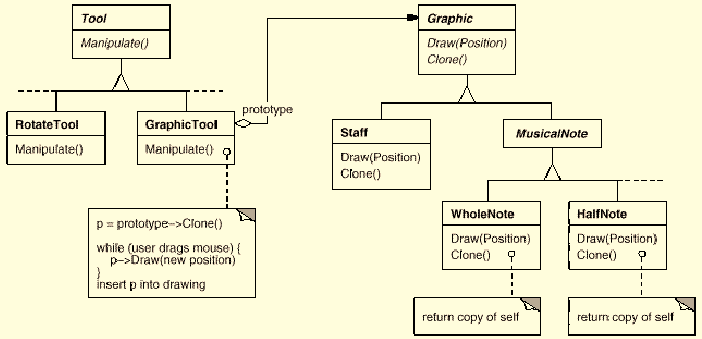
```
abstract class AbstractProduct {
    public abstract int geti();
}

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

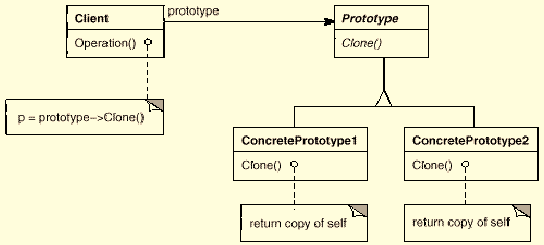
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.
  - e.g., reduce # of classes (# of tools) by initializing a generic tool with a prototype



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 product constructors require many methods but in fact only a few instances are required. Often it is not useful to expose many of the parameters to factory methods.
  - when instances can have only one of a few states
    - may be better to initialize once, and then clone prototypes
- Hence we clone generic "exemplars" and expect properties to be set as app runs.
- For instance, user preferences for GUI.
  - Start generic, provide preferences to change default.

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

Proto factory 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;
    }

    …
}
```

the "exemplars" or prototype instances

Proto factory 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;
    }
    …
}
```


Proto factory Sample Code -- MapSite

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

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

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

Copy construction is a complicated part of the OOP art that we will not delve into further at the moment.
The big issue is whether clone should really perform a "deep" or "shallow" copy.

Object.clone does shallow copy.
See javadoc for java.lang.Object.clone()

Proto factory Sample Code -- Door

```
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;

    …
}
```

Proto factory Sample Code - Room

```
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];
}
```


Proto factory Sample Code - Enchanted Room

```
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;
}
```


Proto factory Sample Code - mainline

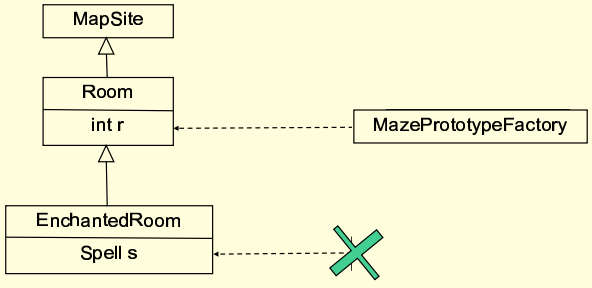
```
public static void main(String args[]) {
    MazeFactory mf = new MazePrototypeFactory(
                            new Maze(), new Wall(),
                            new Room(0), new Door(null,null));
    Maze m = new MazeGame().createMaze(mf);
}


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 + 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).

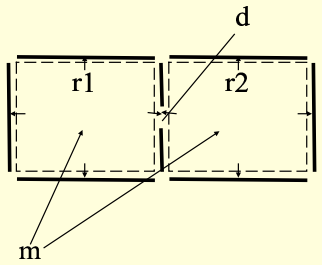
Consequences

- 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()
- !!!

Deep copy for composite objects can be very difficult

Prototype Manager Sample Code - data

file .mazerc:

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

Constructing concrete objects is now a matter of cloning and then setting properties. So why not read the properties from a structured file?

../demo/jfc/Notepad/resources/Notepad.properties

```
# file Menu definition
#
# Each of the strings that follow form a key to be
# used as the basis of a menu item definition.
#
# open ->  Notepad.openAction
# new  ->  Notepad.newAction
# save ->  Notepad.saveAction
# exit ->  Notepad.exitAction
file=new open save - exit
fileLabel=File
openLabel=Open
openImage=resources/open.gif
newLabel=New
newImage=resources/new.gif
saveLabel=Save
saveImage=resources/save.gif
exitLabel=Exit
```

Controls the creation of the interactive application.

Prototype Manager Sample Code - read

```
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
```

While we're at it, why not make the class dynamic also?

Java Background - `Class.forName(String)`

What does the following print?

```
class ClassForName {
    public static void main(String[] args){
        try{
            Class cl = Class.forName(
                                "java.lang.string");
               System.out.println("cl=" + cl);
               Object o = cl.newInstance();
               System.out.println("o=" + o);
           }catch(Exception e){
               System.out.println("oops. " + e );
           }
       }
   }
```

```
oops. java.lang.ClassNotFoundException: java.lang.string
```


Java Background - What was wrong?

```
class ClassForName {
    public static void main(String[] args){
        try{
            Class cl =  Class.forName(
                                "java.lang.String");
            System.out.println("cl=" + cl);
            Object o = cl.newInstance();
            System.out.println("o=" + o);
        }catch(Exception e){
            System.out.println("oops. " + e );
        }
    }
}
```

```
java.lang.string. The class name is capitalized.

Point is that reflection api's not for the faint of heart..
```

java.lang.Class.newInstance

What does this print?

```
class ClassForName {
    public static void main(String[] args){
        try{
            Class cl = Class.forName(
                                "java.lang.String");
               System.out.println("cl=" + cl);
               Object o = cl.newInstance();
               System.out.println(
                   "o.equals(\"\")=" +
                    o.equals(""));
           }catch(Exception e){
               System.out.println("oops. " + e );
           }
       }
   }
```

cl=class java.lang.String
o.equals("")=true

Sample Code - More Dynamic

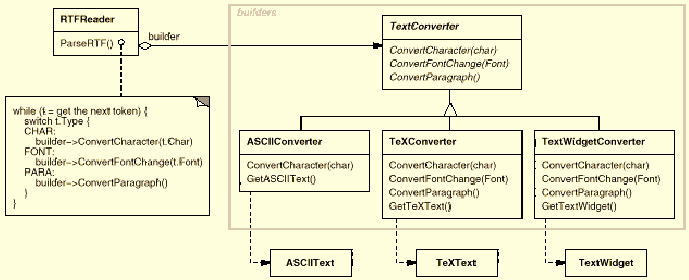
```
package penny.maze.factory;
public class MazeFactory {
    MazeFactory() { }

    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

Builder

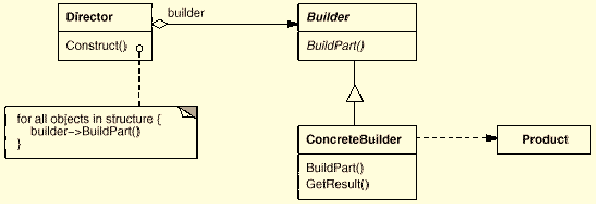
- Separate the construction of a complex object from its representation so that the same construction process can create different representations.
  - e.g., read in Rich Text Format, converting to may different formats on load.



Applicability

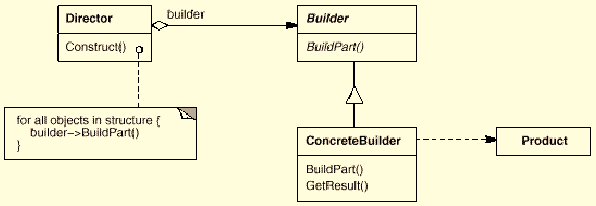
- Use When:
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled
  - the construction process must allow different representations for the object that's constructed
  - The way parts may be linked together may be different. (unlike our maze)

Structure



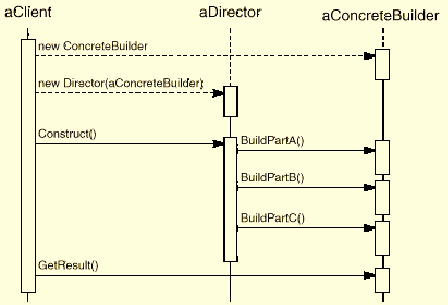
- Builder
  - specifies an abstract interface for creating parts of a Product object
- Concrete Builder
  - constructs and assembles parts of the product by implementing the Builder interface
  - defines and keeps track of the representation it creates
  - provides an interface for retrieving the product

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.

MazeBuilder 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();
    }
    …
}
```

This approach could be extended to read the maze from a file or some other complex representation. The builder knows how to assemble pieces

MazeBuilder Sample Code - lazy getMaze

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

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

    …
}
```


MazeBuilder Sample Code - buildRoom

```
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());
         }
    }
    …
}
```


MazeBuilder Sample Code - buildDoor

```
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);
    }
    …
}
```

MazeBuilder Sample Code - CountingMazeBuilder

```
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; }

}
```


MazeBuilder Sample Code - main

```
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());
    }

    …
}
```


Review factory createMaze..

```
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;
}
```


MazeBuilder Sample Code - createMaze

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

```
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);
}
```

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 clones 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