Object –Oriented Software Development Using Java

Chap 10. More Design Patterns

한동대학교 전자전산학부
김 기석 교수

- peterkim@handong.edu
- http://www.handong.edu

10. More Design Patterns

- Chapter Overview
  - an idiom for type-safe enumeration types
  - important design patterns
    - Abstract Factory
    - Prototype
    - Builder
    - Command
    - Adapter

10.1 Type-Safe Enumeration Types

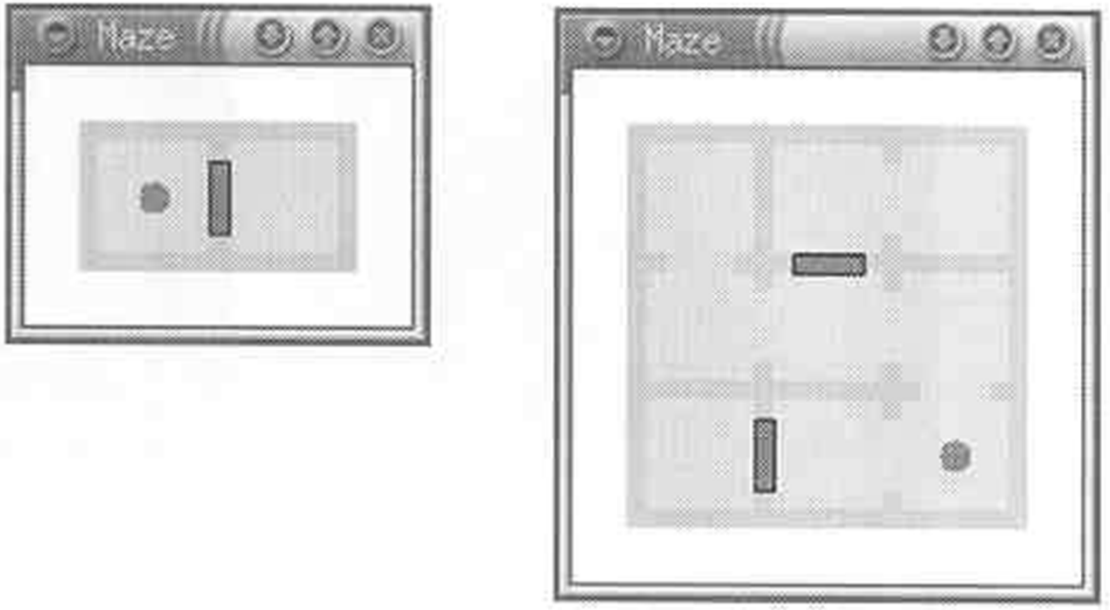
- 10.1.1 A Simple Maze Game
  - an n * m grid
  - two instances of the maze game
    - 1) 1 * 2 maze game
    - 2) 3 * 3 maze game
  - Each square on the board is a room
  - Adjacent rooms are separated by either a wall or a door between them
  - the doors can be either open or closed
  - a player represented by a dot
  - the player can move from room to room through open doors

10.1.1 A Simple Maze Game

- Figure 10.1 A simple maze game

Figure 10.1

A simple maze game.



10.1.2 Enumeration Types

- Implementation idiom for enumeration types in Java
  - An Enumeration type : a finite set of distinct values
  - two enumeration types for the maze game
    - 1) Orientation, (the wall or a door) with values of horizontal and vertical
    - 2) Direction, with values of north, south, east and west
- enumeration types for C/C++
  - provide language support
  - distinct from all other types
  - their values are all distinct and noninterchangeable
  - proper use can be checked at compile time.

10.1.2 Enumeration Types

- Java enumeration types

```
public enum Day {
    SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
    THURSDAY, FRIDAY, SATURDAY
}
```

- 1) Simple approach
  - treat enumeration types as integer types
  - define integer constants for the values of enumeration types

Types

```
public interface Orientation {
    public static final int HORIZONTAL = 0;
    public static final int VERTICAL   = 1;
}

public interface Direction {
    public static final int NORTH = 0;
    public static final int EAST  = 1;
    public static final int SOUTH = 2;
    public static final int WEST  = 3;
}
```

- workable solution
- shortcomings
  - 1) it is not type-safe
  - 2) error-prone

```
int orientation; // an enumeration type variable for orientation
int direction;   // an enumeration type variable for direction

orientation = Orientation.VERTICAL; // okay
direction = Direction.EAST;         // okay
```

- shortcomings -Ex1) not type-safe (P 467)
  - make no sense, but legal

```
orientation = Direction.WEST;
direction = Orientation.HORIZONTAL;

orientation = 6;
direction = -1;
```

  - variables and values of different enumeration types are interchangable, the values may not even be meaningful

10.1.2 Enumeration Types

- shortcomings – Ex 2) The values of enumeration types are numbers
  - When directly printing these values, they are cryptic

```
direction = Direction.EAST;
System.out.println("The current direction is: " + direction);
```

This prints out the following message, which is cryptic and unreadable:

```
The current direction is: 1
```

  - To make the printout more readable,
  -

10.1.2 Enumeration Types

- ➔ Solution) the printing of values

```
public interface Orientation {
   public static final int HORIZONTAL = 0;
   public static final int VERTICAL   = 1;

   public static String name(int v) {
      if (v == HORIZONTAL)
         return "Horizontal";
      else if (v == VERTICAL)
         return "Vertical";
      else
         return null;
   }
}

//...
direction = Direction.EAST;
System.out.println("The current direction is: " +
                   Orientation.name(direction));
```

10.1.3 Unordered Type-safe Enumeration Idiom

- type-safe enumeration idiom
  - each enumeration type is defined as a class
  - each value of the enumeration type is associated with a descriptive name rather than an integer value
  - toString() : returns the descriptive name of each value
  - the constructor of the class is private
    - prevents any other instances of the class from being created outside the class
    - since an enumeration type consists of only a fixed finite number of distinct values
    - the instances referenced by the constants defined in this class are the only instances of this class in existence

10.1.3 Unordered Type-safe Enumeration Idiom

- type-safe enumeration idiom

```
// Enumeration Type maze.Orientation
package maze;

/*  An enumeration type  */
  public class Orientation {
  public static final Orientation VERTICAL = new Orientation("Vertical");
  public static final Orientation HORIZONTAL = new Orientation("Horizontal");

  public String toString() {
    return name;
  }
  private Orientation(String name) {
    this.name = name;
  }
  private final String name;

}
```

10.1.3 Unordered Type-safe Enumeration Idiom

- Unordered type-safe enumeration idiom
  - each enumeration type may only have a fixed number of distinct values.
  - the enumeration types are type-safe; that is, variables and values of different enumeration types are not interchangeable.
  - the values can be printed with descriptive names.
    - ex) printout

```
Orientation orientation = Orientation.VERTICAL;
System.out.println("The current orientation is: " + orientation);
```

```
The current orientation is: Vertical
```

10.1.3 Unordered Type-safe Enumeration Idiom

- Identity comparison
  - since the enumeration type has only a fixed number of distinct values, equality of two values(instances) is the same as the identity of the two values
  - Identity comparison(equality of references) can be used for comparing the equality of values

```
Orientation orientation1 = . . . ;
Orientation orientation2 = . . . ;
if (orientation1 == orientation2) {
   // both have the same orientation
}
```


10.1.4 Ordered Type-safe enumeration Idiom

- ordered Type-safe enumeration Idiom
  - ex) a definition of the ordered enumeration
  - each value is also associated with an ordinal number
  - first(), next()

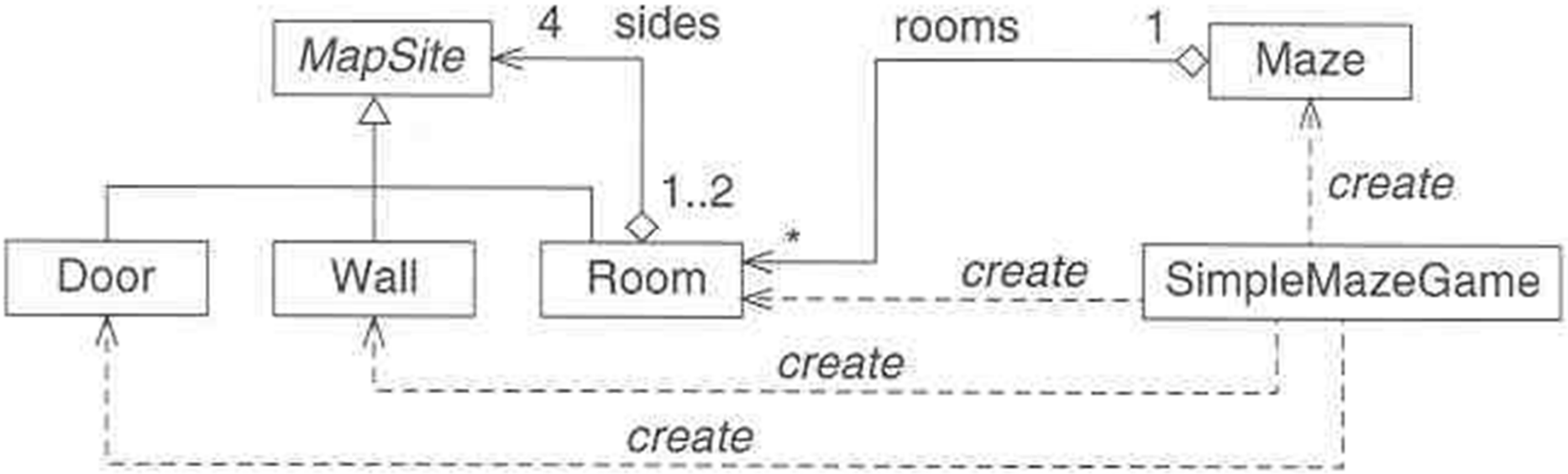
```
for (Direction dir = Direction.first(); dir != null; dir = dir.next()) {
    // process each direction
}
```

```
// Ordered enumeration type maze.Direction
package maze;
/*
 *  An enumeration type
 */
public class Direction implements Comparable {

  public static final Direction NORTH = new Direction("North");
  public static final Direction EAST  = new Direction("East");
  public static final Direction SOUTH = new Direction("South");
  public static final Direction WEST  = new Direction("West");

  public String toString() {
    return name;
  }

  public int getOrdinal() {
    return ordinal;
  }

  public int compareTo(Object o) {
    if (o instanceof Direction) {
      return ordinal - ((Direction) o).getOrdinal();
    }
    return 0;
  }

  public static Direction first() {
    return values[0];
  }
```

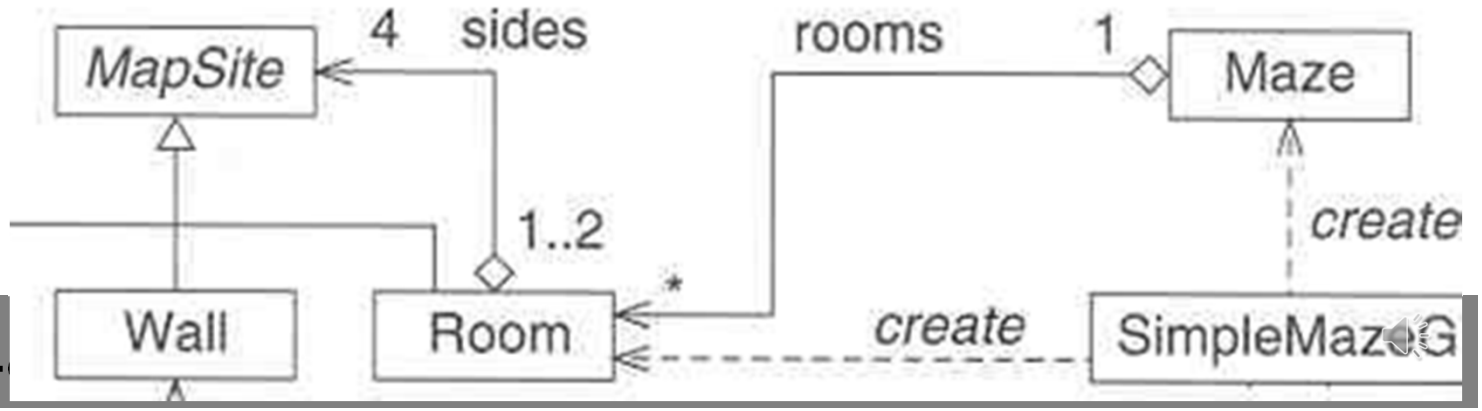
```
public Direction next() {
    if (ordinal < values.length - 1) {
      return values[ordinal + 1];
    } else {
      return null;
    }
  }

  public Direction opposite() {
    return values[(ordinal + 2) % 4];
  }


  private Direction(String name) {
    this.name = name;
  }

  private static int nextOrdinal = 0;
  private final String name;
  private final int ordinal = nextOrdinal++;

  private static final Direction[] values = { NORTH, EAST, SOUTH, WEST };

}
```

10.2 Creational Design Patterns

- 10.2.1 A Simple Design of the Maze Game
  - support only a default style of the maze game
  - Fig 10.2 the design of the simple maze game

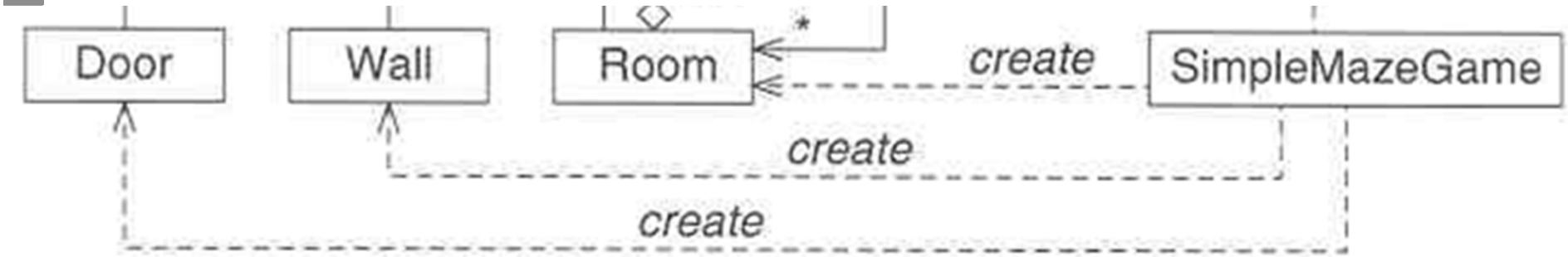


- Navigation arrow _____>
  - ?? MapSite & Room …. 4: 1..2
    - sides : Room Class의 p. 473. Protected MapSite[] sides = new MapSite[4];
  - ?? Room & Maze
    - rooms : Maze Class의 p. 476. Protected List Rooms = new ArrayList();



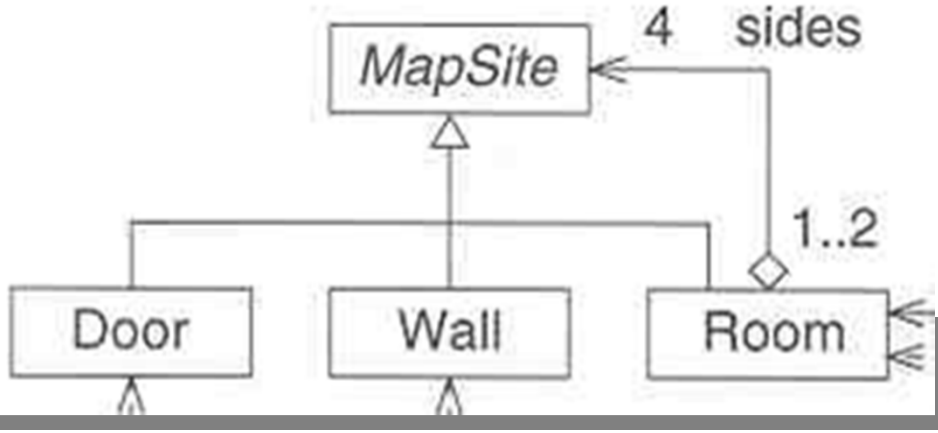
- Dependence ---→
  - SimpleMazeGame Class

```
Door door = new Door(room1, room2);

room1.setSide(Direction.NORTH, newWall());
room1.setSide(Direction.EAST, door);
```



10.2.1 A Simple Design of the Maze Game

- The Map Sites
  - the interface MapSite
    - defines the common behavior of the map sites
    - mapSite : fixed objects : rooms, walls, doors
    - draw() : draws the visual appearance of each map site
    - enter() : defines the reaction of each map site when the player attempts to enter the map site.
    -



```
// Interface maze.MapSite

package maze;

import java.awt.*;

public interface MapSite extends Cloneable {

  public Object clone() throws CloneNotSupportedException;
  public void enter(Maze maze);
  public void draw(Graphics g, int x, int y, int w, int h);

}
```

10.2.1 A Simple Design of the Maze Game

- Room class
  - each room has a room number
  - each room has four sides, which must be either walls or doors
  - when a player enters a room, a sound will be played.

```
// Class maze.Room
package maze;

import java.awt.*;
import java.applet.AudioClip;

public class Room implements MapSite {

  public static final Color ROOM_COLOR = new
Color(152, 251, 152);
  public static final Color PLAYER_COLOR =
Color.red;

  public Room(int roomNumber) {
    this.roomNumber = roomNumber;
  }

  public Object clone() throws
CloneNotSupportedException {
    Room room = (Room) super.clone();
    room.sides = new MapSite[4];
    for (int i = 0; i < 4; i++) {
      if (sides[i] != null) {
                room.sides[i] = (MapSite) sides[i].clone();
      }
    }
    return room;
  }

  public MapSite getSide(Direction dir) {
    if (dir != null) {
      return sides[dir.getOrdinal()];
    }
    return null;
  }
```

```
 public void setSide(Direction dir, MapSite site) {
    if (dir != null) {
      sides[dir.getOrdinal()] = site;
      if (site instanceof Door) {
                Door door = (Door) site;
                if (dir == Direction.NORTH ||
                    dir == Direction.SOUTH) {

door.setOrientation(Orientation.HORIZONTAL);
                } else {

door.setOrientation(Orientation.VERTICAL);
                }
      }
    }
  }

  public void setRoomNumber(int roomNumber) {
    this.roomNumber = roomNumber;
  }

  public int getRoomNumber() {
    return roomNumber;
  }

  public Point getLocation() {
    return location;
  }

  public void setLocation(Point location) {
    this.location = location;
  }
```

```
public void enter(Maze maze) {
    maze.setCurrentRoom(this);
    gong.play();
  }

  public boolean isInRoom() {
    return inroom;
  }

  public void setInRoom(boolean inroom) {
    this.inroom = inroom;
  }
 public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(ROOM_COLOR);
    g.fillRect(x, y, w, h);
    if (inroom) {
      g.setColor(PLAYER_COLOR);
      g.fillOval(x + w / 2 - 5, y + h / 2 - 5, 10, 10);
    }
  }

  protected int roomNumber = 0;
  protected boolean inroom = false;  // whether the play is in this room
  protected MapSite[] sides = new MapSite[4];
  protected Point location = null;

  protected static AudioClip gong = util.AudioUtility.getAudioClip("audio/gong.au");
  //protected static AudioClip spacemusic = util.AudioUtility.getAudioClip("audio/spacemusic.au");
}
```

10.2.1 A Simple Design of the Maze Game

- Door class
  - Each door connects two rooms
  - when the player tries to enter a door that is open, the player will enter the room on the other side of the door
  - when a player tries to enter a door that is closed, the player will remain in the current room and a smashing sound will be played.

```
// Class maze.Door

package maze;

import java.awt.*;
import java.applet.AudioClip;

public class Door implements MapSite {

  public Door(Room room1, Room room2) {
    this.room1 = room1;
    this.room2 = room2;
  }

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

  public boolean isOpen() {
    return open;
  }

  public void setOpen(boolean open) {
    this.open = open;
  }
```

```
public void setRooms(Room room1, Room room2) {
   this.room1 = room1;
   this.room2 = room2;
 }

 public Orientation getOrientation() {
   return orientation;
 }

 public void setOrientation(Orientation orientation) {
   this.orientation = orientation;
 }

 public Room otherSideFrom(Room room) {
   if (room != null) {
     if (room == room1) {
           return room2;
     } else if (room == room2) {
           return room1;
     }
   }
   return null;
 }
```

```
public void enter(Maze maze) {
   if (open) {
     Room otherRoom = otherSideFrom(maze.getCurrentRoom());
     if (otherRoom != null) {
            otherRoom.enter(maze);
     }
   } else {
     ding.play();
   }
 }
```

```
 public void draw(Graphics g, int x, int y, int w, int h) {
   g.setColor(Wall.WALL_COLOR);
   g.fillRect(x, y, w, h);
   if (orientation == Orientation.VERTICAL) {
     y += 2 * w; h -= 4 * w;
   } else {
     x += 2 * h; w -= 4 * h;
   }
   if (open) {
     g.setColor(Room.ROOM_COLOR);
     g.fillRect(x, y, w, h);
   } else {
     g.setColor(Color.red);
     g.fillRect(x, y, w, h);
     g.setColor(Color.black);
     g.drawRect(x, y, w, h);
   }
 }

 protected Room room1;
 protected Room room2;
 protected boolean open;
 protected Orientation orientation;

 protected static AudioClip ding = util.AudioUtility.getAudioClip("audio/ding.au");

}
```

10.2.1 A Simple Design of the Maze Game

- Wall class
  - A wall could be between two rooms or on the edge of the maze board
  - Walls are impassable
  - when a player tries to enter a wall, the player will remain in the current room and a smashing round will be played.

```
// Class maze.Wall
package maze;

import java.awt.*;
import java.applet.AudioClip;

public class Wall implements MapSite {

  public static final Color WALL_COLOR = Color.orange;

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

  public void enter(Maze maze) {
    hurts.play();
  }

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(WALL_COLOR);
    g.fillRect(x, y, w, h);
  }

  protected static AudioClip hurts =
util.AudioUtility.getAudioClip("audio/that.hurts.au");

}
```

10.2.1 A Simple Design of the Maze Game

- The Maze Board : The Maze Class
  - The Maze Class
  - the maze board consists of a list of rooms
  - curRoom : current room
  - two major responsibilities
    - 1) to draw the maze board (** Focusing)
    - 2) to control the movement of the player (➔10.3.1)

```
// Class maze.Maze

package maze;

import java.util.Stack;
import java.util.List;
import java.util.ArrayList;
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class Maze implements Cloneable {

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

  public void addRoom(Room room) {
    if (room != null) {
      rooms.add(room);
    }
  }
```

```
public Room findRoom(int roomNumber) {
  for (int i = 0; i < rooms.size(); i++) {
    Room room = (Room) rooms.get(i);
    if (roomNumber == room.getRoomNumber()) {
          return room;
    }
  }
  return null;
}
public void setCurrentRoom(int roomNumber) {
  Room room = findRoom(roomNumber);
  setCurrentRoom(room);
}
public void setCurrentRoom(Room room) {
  if (room != curRoom) {
    if (curRoom != null) {
          curRoom.setInRoom(false);
    }
    if (room != null) {
          room.setInRoom(true);
          curRoom = room;
    }
    if (view != null) {
          view.repaint();
    }
  }
}
```

```
public Room getCurrentRoom() {
   return curRoom;
 }

public void draw(Graphics g) {
    (draw the maze board on the view component)

}

 protected List rooms = new ArrayList();  // the list of rooms that comprise
                                          // the board

  protected Room curRoom = null;  // the room in which the player is

  protected Component view;   // The GUI component representing the board

  (.....)


 }
```

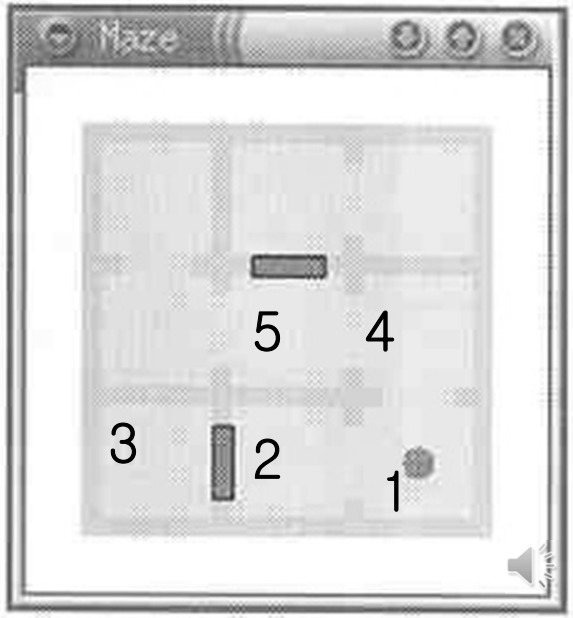
10.2.1 A Simple Design of the Maze Game

- SimpleMazeGame Class
  - main driver of the maze program
  - responsible for creating the layout of maze games
  - createMaze() : creates a small maze board with two rooms
  - createLargeMaze() : creates a large maze board with nine rooms
  - Invoked with an optional argument Large

```
// Class maze.SimpleMazeGame

package maze;

import java.awt.*;
import javax.swing.*;

/*
 *  Build a Maze game.
 *
 *  This implmentation does not use any design patterns.
 *  Compare the createMaze() method of this class to the ccorresponding method of the following
 *  classes which create the same Maze using different creational design patterns.
 *    - MazeGameAbstractFactory: using Abstract Factory design pattern
 *    - MazeGameBuilder: using Builder design pattern
 *
 *  Run the program as follows:
 *    java maze.SimpleMazeGame Large
 *          -- create a large Maze game
 *    java maze.SimpleMazeGame
 *          -- create a small Maze game
 */
public class SimpleMazeGame {
```

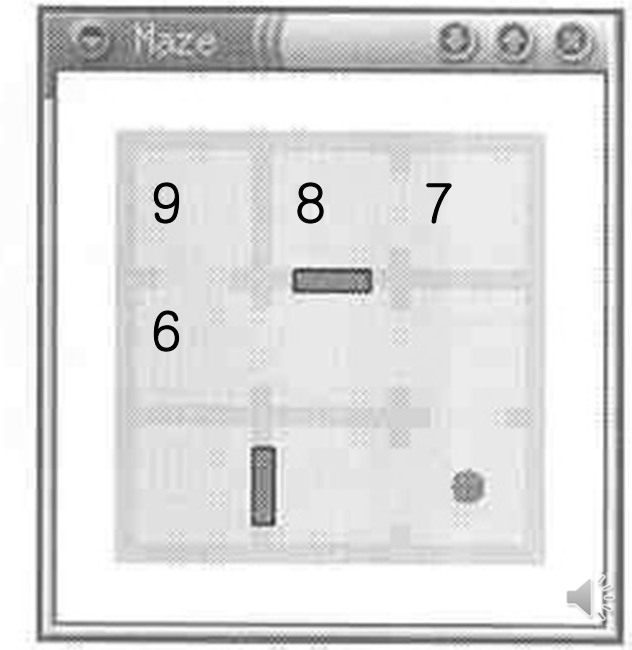
```
  /**
   *  Creates a small maze with 2 rooms.
   */
  public static Maze createMaze() {
    Maze maze = new Maze();
    Room room1 = new Room(1);
    Room room2 = new Room(2);
    Door door = new Door(room1, room2);

    room1.setSide(Direction.NORTH, new Wall());
    room1.setSide(Direction.EAST, door);
    room1.setSide(Direction.SOUTH, new Wall());
    room1.setSide(Direction.WEST, new Wall());

    room2.setSide(Direction.NORTH, new Wall());
    room2.setSide(Direction.EAST, new Wall());
    room2.setSide(Direction.SOUTH, new Wall());
    room2.setSide(Direction.WEST, door);

    maze.addRoom(room1);
    maze.addRoom(room2);

    return maze;
  }
```

```
/**
 *  Creates a large maze with 9 rooms.
 */
public static Maze createLargeMaze() {
  Maze maze = new Maze();
  Room room1 = new Room(1);
  Room room2 = new Room(2);
  Room room3 = new Room(3);
  Room room4 = new Room(4);
  Room room5 = new Room(5);
  Room room6 = new Room(6);
  Room room7 = new Room(7);
  Room room8 = new Room(8);
  Room room9 = new Room(9);
  Door door1 = new Door(room1, room2);
  Door door2 = new Door(room2, room3);
  Door door3 = new Door(room4, room5);
  Door door4 = new Door(room5, room6);
  Door door5 = new Door(room5, room8);
  Door door6 = new Door(room6, room9);
  Door door7 = new Door(room7, room8);
  Door door8 = new Door(room1, room4);

  door1.setOpen(true);
  door2.setOpen(false);
  door3.setOpen(true);
  door4.setOpen(true);
  door5.setOpen(false);
  door6.setOpen(true);
  door7.setOpen(true);
  door8.setOpen(true);
  room1.setSide(Direction.NORTH, door8);
  room1.setSide(Direction.EAST, new Wall());
  room1.setSide(Direction.SOUTH, new Wall());
  room1.setSide(Direction.WEST, door1);
  room2.setSide(Direction.NORTH, new Wall());
  room2.setSide(Direction.EAST, door1);
  room2.setSide(Direction.SOUTH, new Wall());
  room2.setSide(Direction.WEST, door2);

  room3.setSide(Direction.NORTH, new Wall());
  room3.setSide(Direction.EAST, door2);
  room3.setSide(Direction.SOUTH, new Wall());
  room3.setSide(Direction.WEST, new Wall());

  room4.setSide(Direction.NORTH, new Wall());
  room4.setSide(Direction.EAST, new Wall());
  room4.setSide(Direction.SOUTH, door8);
  room4.setSide(Direction.WEST, door3);

  room5.setSide(Direction.NORTH, door5);
  room5.setSide(Direction.EAST, door3);
  room5.setSide(Direction.SOUTH, new Wall());
  room5.setSide(Direction.WEST, door4);
```



```
    room6.setSide(Direction.NORTH, door6);
    room6.setSide(Direction.EAST, door4);
    room6.setSide(Direction.SOUTH, new Wall());
    room6.setSide(Direction.WEST, new Wall());

    room7.setSide(Direction.NORTH, new Wall());
    room7.setSide(Direction.EAST, new Wall());
    room7.setSide(Direction.SOUTH, new Wall());
    room7.setSide(Direction.WEST, door7);

    room8.setSide(Direction.NORTH, new Wall());
    room8.setSide(Direction.EAST, door7);
    room8.setSide(Direction.SOUTH, door5);
    room8.setSide(Direction.WEST, new Wall());

    room9.setSide(Direction.NORTH, new Wall());
    room9.setSide(Direction.EAST, new Wall());
    room9.setSide(Direction.SOUTH, door6);
    room9.setSide(Direction.WEST, new Wall());

    maze.addRoom(room1);
    maze.addRoom(room2);
    maze.addRoom(room3);
    maze.addRoom(room4);
    maze.addRoom(room5);
    maze.addRoom(room6);
    maze.addRoom(room7);
    maze.addRoom(room8);
    maze.addRoom(room9);

    return maze;
  }
```

```
  public static void main(String[] args) {
    Maze maze;
    if (args.length > 0 &&
                "Large".equals(args[0])) {
      maze = createLargeMaze();
    } else {
      maze = createMaze();
    }
    maze.setCurrentRoom(1);
    maze.showFrame("Maze");
  }


}
```



```
C:₩Chapter10>^V

C:₩Chapter10>java maze.SimpleMazeGame
Maze.Draw(): offset=0, 0
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: 1, 0
Key pressed
Key press ignored
```


```
C:₩Chapter10>java maze.SimpleMazeGame Large
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```

10.2.1 A Simple Design of the Maze Game

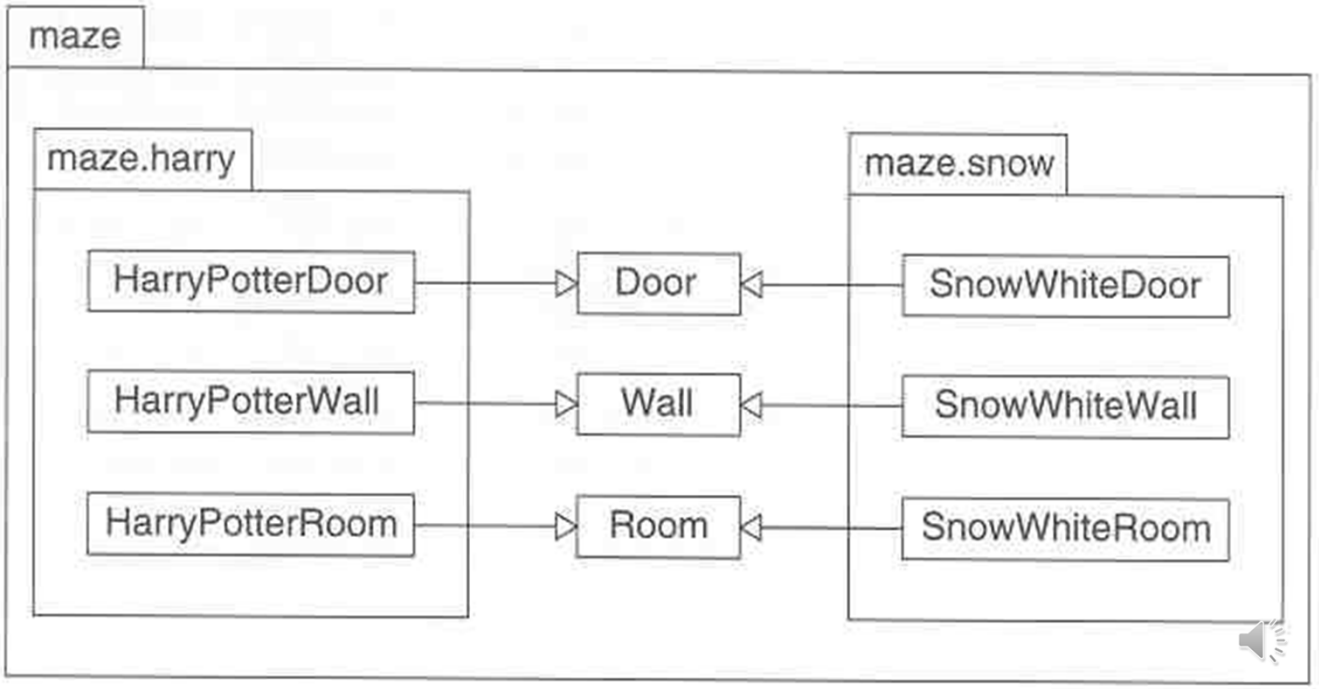
- The Themes
  - support different looks or themes of the game
  - the rooms, walls, and doors could be drawn in different shapes or colors
  - the reactions to entering rooms,walls and doors could alose have different sound or animation effects.
  - Each theme consists of a set of rooms, walls, and doors.

10.2.1 A Simple Design of the Maze Game

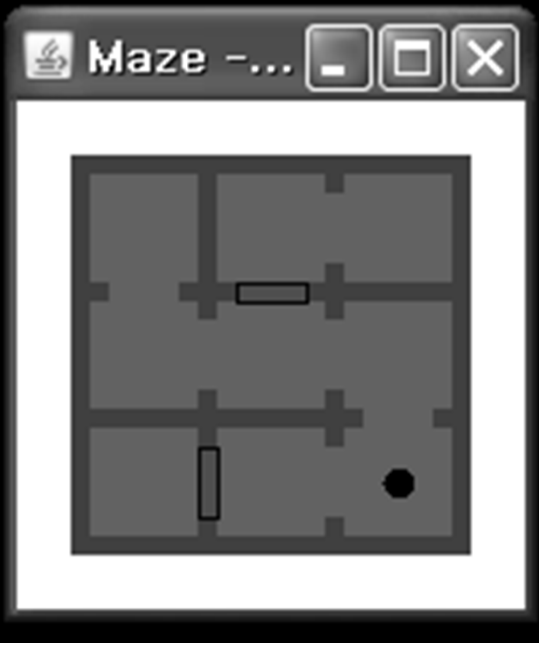
- Two new themes
  - 1) Harry Potter theme
  - 2) Snow White theme
- subpackage maze.harry, maze.snow

Figure 10.3

Two themes of the maze game.

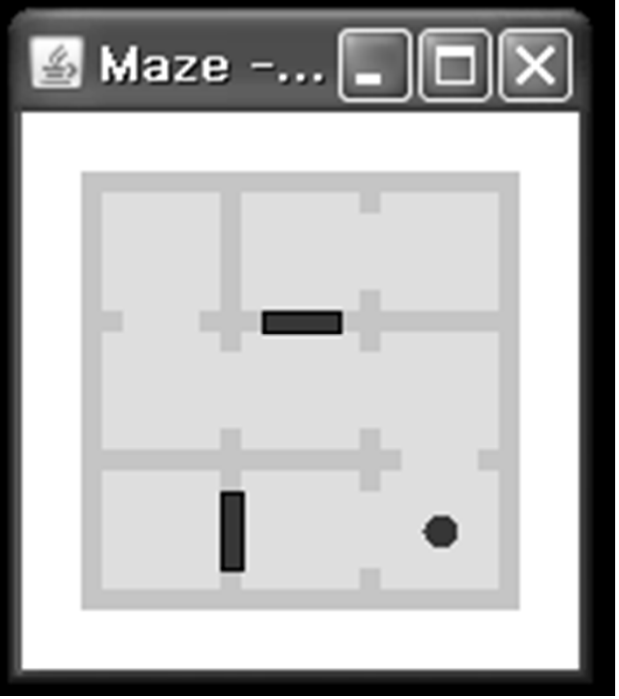


```
C:₩Chapter10>java maze.MazeGameAbstractFactory Harry
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```




```
C:₩Chapter10>java maze.MazeGameAbstractFactory Snow
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```

```
C:₩Chapter10>java maze.MazeGameAbstractFactory
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```



```
// Class maze.harry.HarryPotterRoom

package maze.harry;

import java.awt.*;
import java.applet.AudioClip;
import maze.*;

class HarryPotterRoom extends Room {

  public static final Color ROOM_COLOR = new Color(85, 107, 47);
  public static final Color PLAYER_COLOR = Color.black;

  public HarryPotterRoom(int roomNumber) {
    super(roomNumber);
  }

  public void enter(Maze maze) {
    maze.setCurrentRoom(this);
    adapt.play();
  }

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(ROOM_COLOR);
    g.fillRect(x, y, w, h);
    if (inroom) {
      g.setColor(PLAYER_COLOR);
      g.fillOval(x + w / 2 - 5, y + h / 2 - 5, 10, 10);
    }
  }

  protected static AudioClip adapt = util.AudioUtility.getAudioClip("audio/adapt-or-die.au");

}
```

```
// Class maze.harry.HarryPotterDoor

package maze.harry;

import java.awt.*;
import maze.*;

class HarryPotterDoor extends Door {

  public HarryPotterDoor(Room room1, Room room2) {
    super(room1, room2);
  }

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(HarryPotterWall.WALL_COLOR);
    g.fillRect(x, y, w, h);
    if (orientation == Orientation.VERTICAL) {
      y += 2 * w; h -= 4 * w;
    } else {
      x += 2 * h; w -= 4 * h;
    }
    if (open) {
      g.setColor(HarryPotterRoom.ROOM_COLOR);
      g.fillRect(x, y, w, h);
    } else {
      g.setColor(new Color(139, 69, 0));
      g.fillRect(x, y, w, h);
      g.setColor(Color.black);
      g.drawRect(x, y, w, h);
    }
  }

}
```

```
// Class maze.harry.HarryPotterWall

package maze.harry;

import java.awt.*;
import maze.*;

class HarryPotterWall extends Wall {

  public static final Color WALL_COLOR = new Color(178, 34, 34);

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(WALL_COLOR);
    g.fillRect(x, y, w, h);
  }

}
```

// Class maze.snow.SnowWhiteRoom

```
package maze.snow;

import java.awt.*;
import java.applet.AudioClip;
import maze.*;

class SnowWhiteRoom extends Room {

  public static final Color ROOM_COLOR = new Color(255, 218, 185);
  public static final Color PLAYER_COLOR = Color.white;

  public SnowWhiteRoom(int roomNumber) {
    super(roomNumber);
  }

  public void enter(Maze maze) {
    maze.setCurrentRoom(this);
    tiptoe.play();
  }

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(ROOM_COLOR);
    g.fillRect(x, y, w, h);
    if (inroom) {
      g.setColor(PLAYER_COLOR);
      g.fillOval(x + w / 2 - 5, y + h / 2 - 5, 10, 10);
    }
  }

  protected static AudioClip tiptoe = util.AudioUtility.getAudioClip("audio/tiptoe.thru.the.tulips.au");

}
```

```
// Class maze.snow.SnowWhiteDoor

package maze.snow;

import java.awt.*;
import maze.*;

class SnowWhiteDoor extends Door {

  public SnowWhiteDoor(Room room1, Room room2) {
    super(room1, room2);
  }

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(SnowWhiteWall.WALL_COLOR);
    g.fillRect(x, y, w, h);
    if (orientation == Orientation.VERTICAL) {
      y += 2 * w; h -= 4 * w;
    } else {
      x += 2 * h; w -= 4 * h;
    }
    if (open) {
      g.setColor(SnowWhiteRoom.ROOM_COLOR);
      g.fillRect(x, y, w, h);
    } else {
      g.setColor(Color.orange);
      g.fillRect(x, y, w, h);
      g.setColor(Color.black);
      g.drawRect(x, y, w, h);
    }
  }

}
```

```
// Class maze.snow.SnowWhiteWall

package maze.snow;

import java.awt.*;
import maze.*;

class SnowWhiteWall extends Wall {

  public static final Color WALL_COLOR = new Color(255, 20, 147);

  public void draw(Graphics g, int x, int y, int w, int h) {
    g.setColor(WALL_COLOR);
    g.fillRect(x, y, w, h);
  }

}
```

10.2.1 A Simple Design of the Maze Game

- How to make the themes easily interchangeable and to make it easy to add additional themes.
  - First Solution) to create the maze boards of different themes in a way similar to the createMaze() and createLargeMaze() in SimpleMazeGame
    - the maze boards in the two new themes can be created as follows
    - p 483.

```
public static Maze createHarryPotterMaze() {
    Maze maze = new Maze();
    Room room1 = new HarryPotterRoom(1);
    Room room2 = new HarryPotterRoom(2);
    Door door = new HarryPotterDoor(room1, room2);

    room1.setSide(Direction.NORTH, new HarryPotterWall());
    room1.setSide(Direction.EAST, door);

    room1.setSide(Direction.SOUTH, new HarryPotterWall());
    room1.setSide(Direction.WEST, new HarryPotterWall());

    room2.setSide(Direction.NORTH, new HarryPotterWall());
    room2.setSide(Direction.EAST, new HarryPotterWall());
    room2.setSide(Direction.SOUTH, new HarryPotterWall());
    room2.setSide(Direction.WEST, door);

    maze.addRoom(room1);
    maze.addRoom(room2);

    return maze;
}
```


```
public static Maze createSnowWhiteMaze() {
    Maze maze = new Maze();
    Room room1 = new SnowWhiteRoom(1);
    Room room2 = new SnowWhiteRoom(2);
    Door door = new SnowWhiteDoor(room1, room2);

    room1.setSide(Direction.NORTH, new SnowWhiteWall());
    room1.setSide(Direction.EAST, door);
    room1.setSide(Direction.SOUTH, new SnowWhiteWall());
    room1.setSide(Direction.WEST, new SnowWhiteWall());

    room2.setSide(Direction.NORTH, new SnowWhiteWall());
    room2.setSide(Direction.EAST, new SnowWhiteWall());
    room2.setSide(Direction.SOUTH, new SnowWhiteWall());
    room2.setSide(Direction.WEST, door);

    maze.addRoom(room1);
    maze.addRoom(room2);

    return maze;
}
```

10.2.1 A Simple Design of the Maze Game

- review of first solution
  - undesirable solution
  - duplication of largely identical code segments
- Second solution) to create objects in different themes and to switch easily between themes
  - various creational design patterns
    - Abstract Factory
    - Factory Method
    - Prototype
    - Builder

10.2.2 Design Pattern : Abstract Factory

- Abstract Factory
  - creational design pattern
  - creating a set of related and compatible products from several interchangeable product families
  - the product instances
    - are not created using the new operator
    - are created using a set of methods defined in a factory class, which represents a product family.
  - the different factories representing different product families.. Share a common interface .. Abstract factory interface
  - a client can easily switch from one factory, to another

10.2.2 Design Pattern : Abstract Factory

- Design Pattern : abstract Factory (p. 485)

Design Pattern ***Abstract Factory***

Category: Creational design pattern.

Intent: To provide an interface for creating a family of related or dependent objects without specifying their concrete classes.

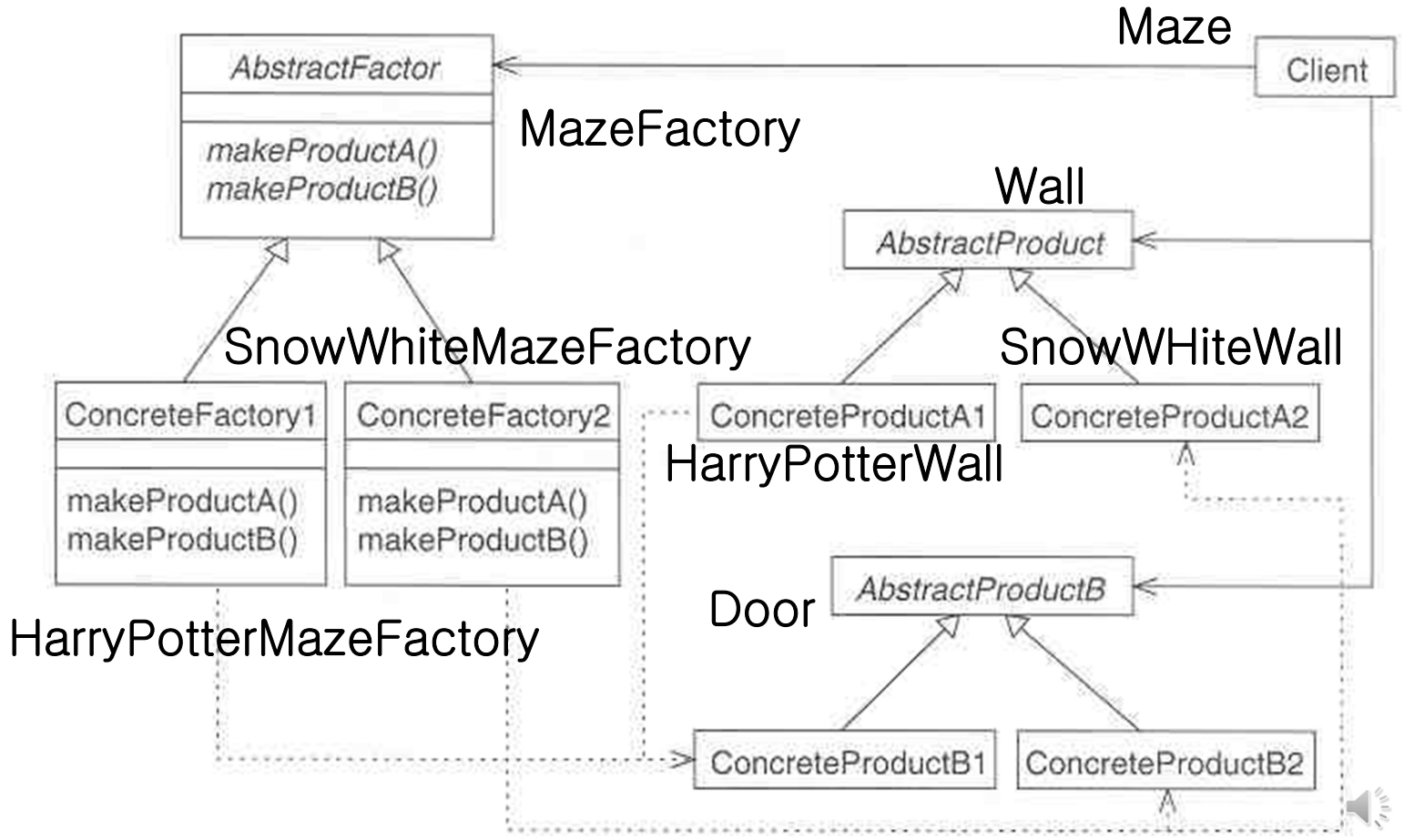
Also Known As: Kit.

Applicability: Use the Abstract Factory design pattern

- when a system should be independent of how its components or products are created.
- when a system should be configurable with one of multiple interchangeable families of products.
- when a family of related products should not be mixed with similar products from different families.
- when only the interfaces of the products are exposed, while the implementation of the products is not revealed.

10.2.2 Design Pattern : Abstract Factory

- The structure of the Abstract Factory design pattern (p. 485)



10.2.2 Design Pattern : Abstract Factory

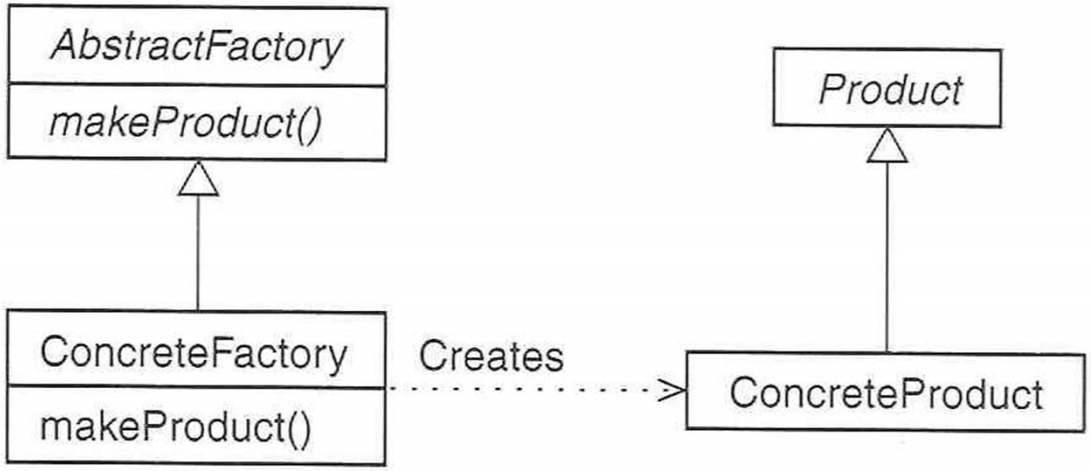
- P 486. The participants of the Abstract Factory design.
  - *AbstractFactory* (e.g., `MazeFactory`), which defines methods that create abstract products (e.g. `makeRoom()`, `makeWall()`, `makeDoor()`) and may provide default implementation.
  - *ConcreteFactory* (e.g., `HarryPotterMazeFactory`, `SnowWhiteMazeFactory`), which implements the methods to create concrete instances of *ConcreteProduct*. Each *ConcreteFactory* creates products in the same product family.
  - *AbstractProduct* (e.g., `Room`, `Wall`, `Door`), which defines an interface for a type of product and may provide default implementation.
  - *ConcreteProduct* (e.g., `HarryPotterRoom`, `HarryPotterWall`, `HarryPotterDoor`, `SnowWhiteRoom`, `SnowWhiteWall`, `SnowWhiteDoor`), which defines a product to be created by the corresponding concrete factory and implements the *AbstractProduct* interface.
  - *Client* (e.g., `Maze`), which uses only *AbstractFactory* and *AbstractProduct*.

((7.4.3 Design Pattern : Factory))

- Design Pattern: Factory

Design Pattern *Factory*

Category: Creational design pattern.
Intent: Define an interface for creating objects but let subclasses decide which class to instantiate and how.
Applicability: The Factory design pattern should be used when a system should be independent of how its products are created.



((7.4.3 Design Pattern : Factory))

- The participants in the Factory design pattern

- *Product* (e.g., `SortAlgorithm`), which defines an interface of the objects that the factory will create;
- *ConcreteProduct* (e.g., `QuickSortAlgorithm`), which implements the `Product` interface;
- *AbstractFactory* (e.g., `AlgorithmFactory`), which defines a factory method (e.g., `makeSortAlgorithm()`) that returns an object of type `Product`; and
- *ConcreteFactory* (e.g., `StaticSortAlgorithmFactory`), which overrides the factory method to return an instance of `ConcreteProduct`.

10.2.2 Design Pattern : Abstract Factory

- In the SImpleMazeGame[p. 477]
  - map site objects are created using the new operator
  - inflexible, inextensible

- Using Abstract Factory Design
  - Fig 10.4
  - the abstract products : the map site classes Room, Wall, Door
  - there are abstract products with default implementations
  - Concrete products : HarryPotterRoom, HarryPotterWall.. Etc
  - The MazeFactory class
    - defines the abstract factory
    - provide a default implementation that returns the map site objects in the default theme.

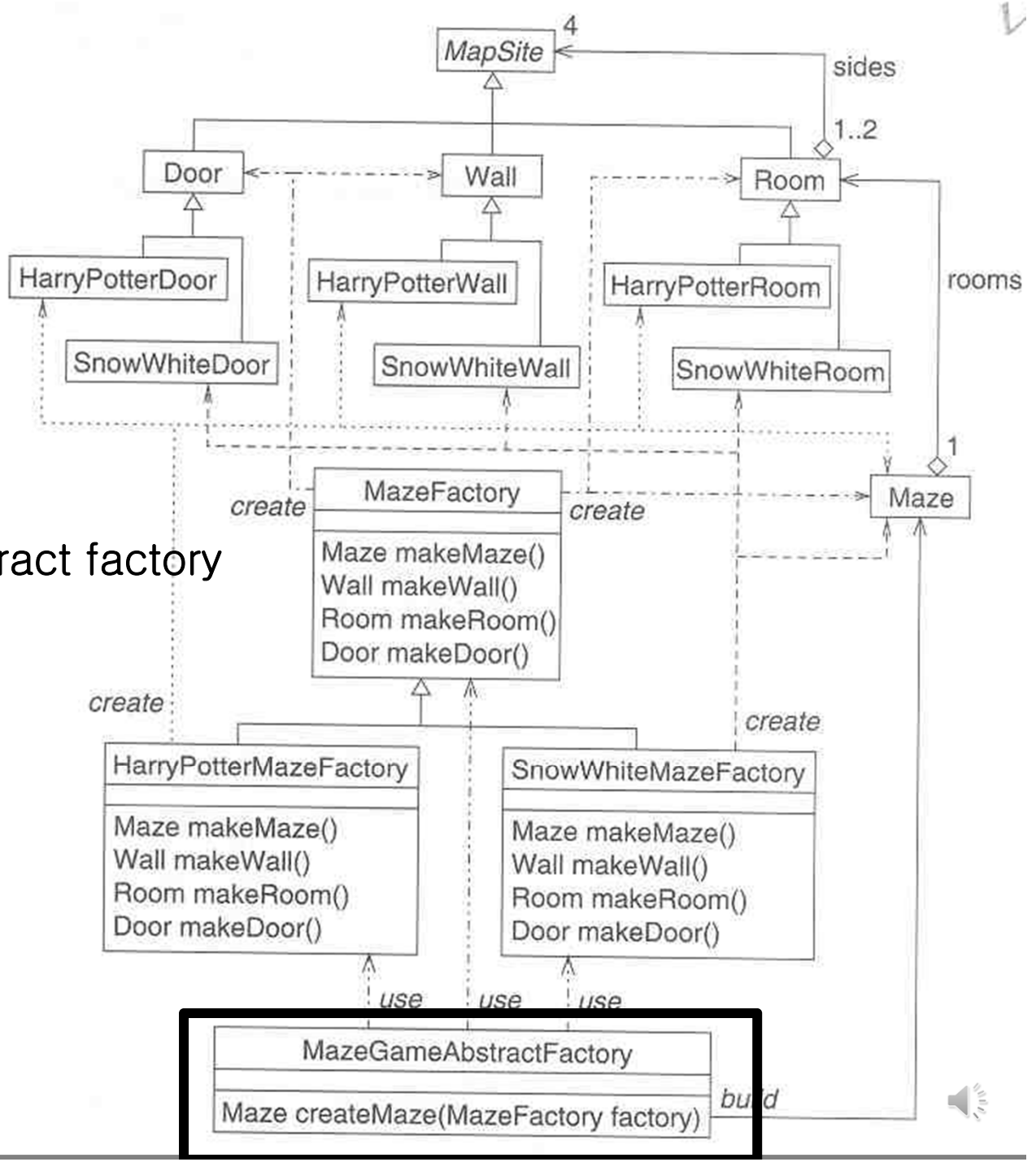
10.2

Figure 10.4

A design of the maze game using the Abstract Factory pattern.

- Figu
  the

Abstract factory



```
// Abstract factory maze.mazeFactory

package maze;

public class MazeFactory {

  public Maze makeMaze() {
    return new Maze();
  }

  public Wall makeWall() {
    return new Wall();
  }

  public Room makeRoom(int roomNumber) {
    return new Room(roomNumber);
  }

  public Door makeDoor(Room room1, Room room2) {
    return new Door(room1, room2);
  }

}

// provides a default implementation that returns the map site objects in the
default theme.
```

```
// Concrete factory maze.HarryPotterMazeFactory

package maze.harry;

import maze.*;

public class HarryPotterMazeFactory extends MazeFactory {

  public Wall makeWall() {
    return new HarryPotterWall();
  }

  public Room makeRoom(int roomNumber) {
    return new HarryPotterRoom(roomNumber);
  }

  public Door makeDoor(Room room1, Room room2) {
    return new HarryPotterDoor(room1, room2);
  }

}
```

```
// Concrete factory maze.SnowWhiteMazeFactory

package maze.snow;

import maze.*;

public class SnowWhiteMazeFactory extends MazeFactory {

  public Wall makeWall() {
    return new SnowWhiteWall();
  }

  public Room makeRoom(int roomNumber) {
    return new SnowWhiteRoom(roomNumber);
  }

  public Door makeDoor(Room room1, Room room2) {
    return new SnowWhiteDoor(room1, room2);
  }

}
```

10.2.2 Design Pattern : Abstract Factory

- MazeGameAbstractFactory
  - main class of this implementation of the maze game using the AbstractFactory pattern
  - client role in the Abstract Factory Pattern
  - createMaze() method
    - create 3*3 maze board
    - takes a parameter of MazeFactory object – the abstract factory
  - Switching means using different concrete factories
    - no code duplication

```
// Maze game(client)
maze.MazeGameAbstractFactory

package maze;

import java.awt.*;
import javax.swing.*;

/*
 *  Build a Maze game.
 *
 *  This implmentation uses Abstract Factory design
pattern.
 *
 *  Run the program as follows:
 *   java maze.MazeGameAbstractFactory Harry
 *        -- uses the HarryPotterMazeFactory
 *   java maze.MazeGameAbstractFactory Snow
 *        -- uses the SnowWhiteMazeFactory
 *   java maze.MazeGameAbstractFactory
 *        -- uses the default MazeFactory
 *
 */
public class MazeGameAbstractFactory {

  public static Maze createMaze(MazeFactory factory)
{
    Maze maze = factory.makeMaze();
    Room room1 = factory.makeRoom(1);
    Room room2 = factory.makeRoom(2);
    Room room3 = factory.makeRoom(3);
    Room room4 = factory.makeRoom(4);
    Room room5 = factory.makeRoom(5);
    Room room6 = factory.makeRoom(6);
    Room room7 = factory.makeRoom(7);
Room room8 = factory.makeRoom(8);
    Room room9 = factory.makeRoom(9);
    Door door1 = factory.makeDoor(room1, room2);
    Door door2 = factory.makeDoor(room2, room3);
    Door door3 = factory.makeDoor(room4, room5);
    Door door4 = factory.makeDoor(room5, room6);
    Door door5 = factory.makeDoor(room5, room8);
    Door door6 = factory.makeDoor(room6, room9);
    Door door7 = factory.makeDoor(room7, room8);
    Door door8 = factory.makeDoor(room1, room4);

 door1.setOpen(true);
    door2.setOpen(false);
    door3.setOpen(true);
    door4.setOpen(true);
    door5.setOpen(false);
    door6.setOpen(true);
    door7.setOpen(true);
    door8.setOpen(true);

    room1.setSide(Direction.NORTH, door8);
    room1.setSide(Direction.EAST,
factory.makeWall());
    room1.setSide(Direction.SOUTH,
factory.makeWall());
    room1.setSide(Direction.WEST, door1);

  room2.setSide(Direction.NORTH,
factory.makeWall());
    room2.setSide(Direction.EAST, door1);
    room2.setSide(Direction.SOUTH,
factory.makeWall());
    room2.setSide(Direction.WEST, door2);
```

```
 room3.setSide(Direction.NORTH,
factory.makeWall());
    room3.setSide(Direction.EAST, door2);
    room3.setSide(Direction.SOUTH,
factory.makeWall());
    room3.setSide(Direction.WEST,
factory.makeWall());

    room4.setSide(Direction.NORTH,
factory.makeWall());
    room4.setSide(Direction.EAST,
factory.makeWall());
    room4.setSide(Direction.SOUTH, door8);
    room4.setSide(Direction.WEST, door3);

    room5.setSide(Direction.NORTH, door5);
    room5.setSide(Direction.EAST, door3);
    room5.setSide(Direction.SOUTH,
factory.makeWall());
    room5.setSide(Direction.WEST, door4);

    room6.setSide(Direction.NORTH, door6);
    room6.setSide(Direction.EAST, door4);
    room6.setSide(Direction.SOUTH,
factory.makeWall());
    room6.setSide(Direction.WEST,
factory.makeWall());

    room7.setSide(Direction.NORTH,
factory.makeWall());
    room7.setSide(Direction.EAST,
factory.makeWall());
    room7.setSide(Direction.SOUTH,
factory.makeWall());
    room7.setSide(Direction.WEST, door7);
```

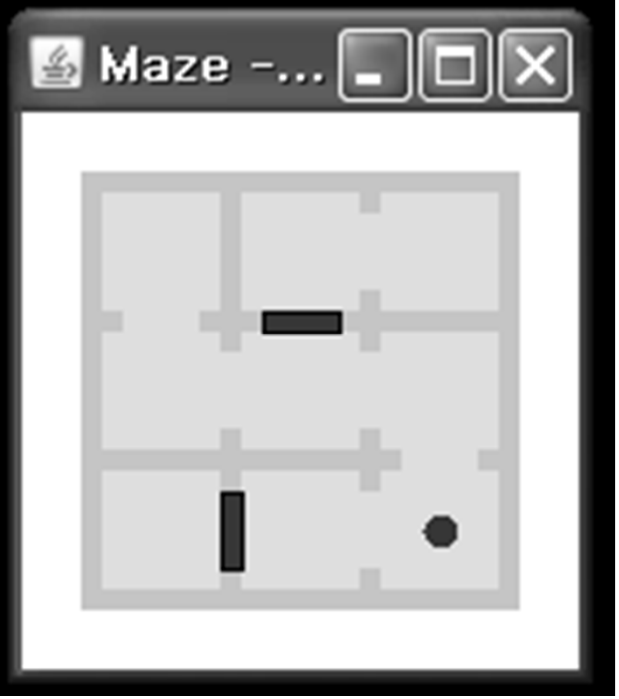
```
 room8.setSide(Direction.NORTH,
factory.makeWall());
    room8.setSide(Direction.EAST, door7);
    room8.setSide(Direction.SOUTH, door5);
    room8.setSide(Direction.WEST,
factory.makeWall());

    room9.setSide(Direction.NORTH,
factory.makeWall());
    room9.setSide(Direction.EAST,
factory.makeWall());
    room9.setSide(Direction.SOUTH, door6);
    room9.setSide(Direction.WEST,
factory.makeWall());

    maze.addRoom(room1);
    maze.addRoom(room2);
    maze.addRoom(room3);
    maze.addRoom(room4);
    maze.addRoom(room5);
    maze.addRoom(room6);
    maze.addRoom(room7);
    maze.addRoom(room8);
    maze.addRoom(room9);

    return maze;
  }
```

```
public static void main(String[] args) {
   Maze maze;
   MazeFactory factory = null;

   if (args.length > 0) {
     if ("Harry".equals(args[0])) {
            factory = new maze.harry.HarryPotterMazeFactory();
     } else if ("Snow".equals(args[0])) {
            factory = new maze.snow.SnowWhiteMazeFactory();
     }
   }
   if (factory == null) {
     factory = new MazeFactory();
   }
   maze = createMaze(factory);
   maze.setCurrentRoom(1);
   maze.showFrame("Maze -- Abstract Factory");
  }

}
```

```
C:₩Chapter10>java maze.MazeGameAbstractFactory Harry
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```


```
C:₩Chapter10>java maze.MazeGameAbstractFactory Snow
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```

```
C:₩Chapter10>java maze.MazeGameAbstractFactory
Maze.Draw(): offset=-2, -2
Maze.Draw(): Room 1 location: 0, 0
Maze.Draw(): Room 2 location: -1, 0
Maze.Draw(): Room 3 location: -2, 0
Maze.Draw(): Room 4 location: 0, -1
Maze.Draw(): Room 5 location: -1, -1
Maze.Draw(): Room 6 location: -2, -1
Maze.Draw(): Room 7 location: 0, -2
Maze.Draw(): Room 8 location: -1, -2
Maze.Draw(): Room 9 location: -2, -2
```



10.2.2 Design Pattern : Abstract Factory

- MazeGameAbstractFactory
  - can be invoked with an optional argument : Harry for Harry Potter theme, or Snow for the Snow White theme.
  - first create a concrete factory for the specified theme.
  - Another benefit of the Abstract Factory pattern
    - allows the implementation of the concrete products to be hidden from the client.
    - the classes representing concrete products are in separate packages, maze.harry, maze.sow
      - these classes are nonpublic
    - only concrete factory needs to be public ➔ encapsulation