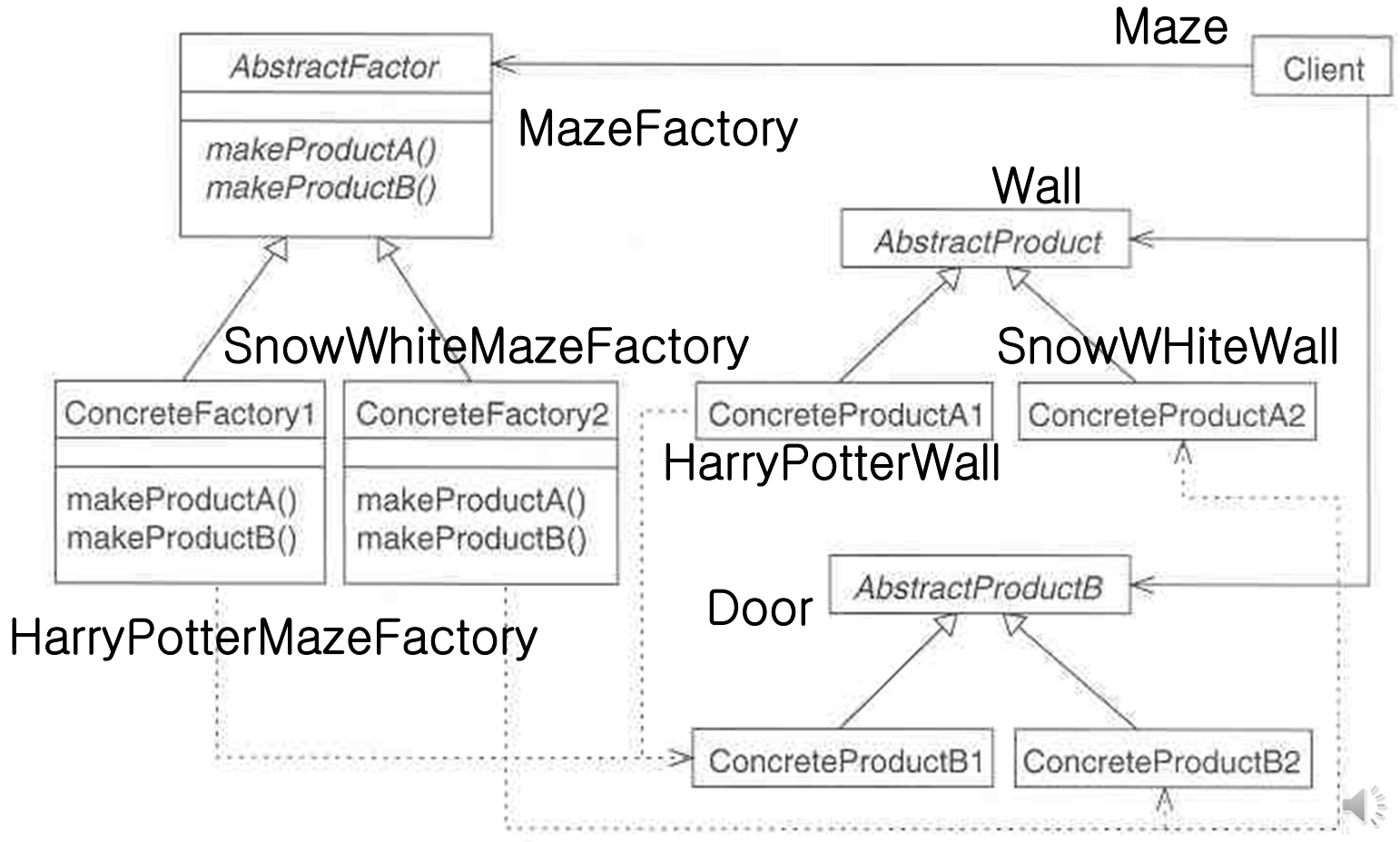
Design Pattern : Abstract Factory

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



10.2.3 Design Pattern : Factory Method

- Similarity : Factory Method Pattern & Abstract Factory
  - define methods(factory method) for creating instances of product
  - different themes are represented by subclasses that override the factory methods to create concrete products in respective themes

10.2.3 Design Pattern : Factory Method

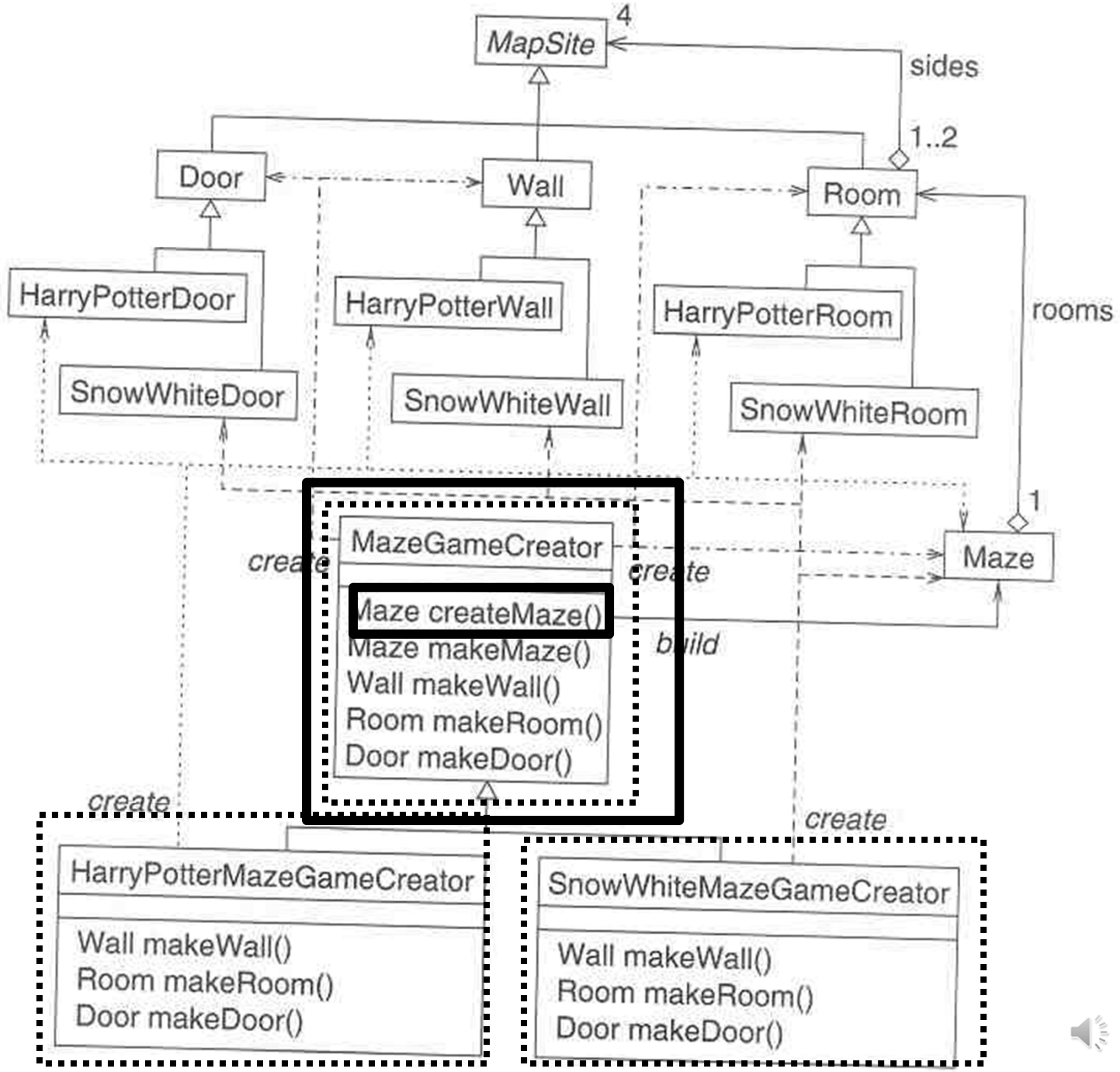
- Difference
  - the sole responsibility of the factories in the Abstract Factory pattern is to create products
  - Factory Method Pattern
    - also responsible for building a structure using the products created by the factory methods.

10.2.3 Design Pattern : Factory Method

- Difference
  - in the maze game, for creating map sites
    - 1) creating the map sites
    - 2) building the maze board using the parts
    - ➔ the Factory Method pattern combines these responsibilities into one class – the MazeGameCreator
    - ➔ Abstract Factory pattern : separate
      - a. the responsibilities for creating the map sites
      - b. building the maze board using the parts
      - assigns them to two classes.

Figure 10.5

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



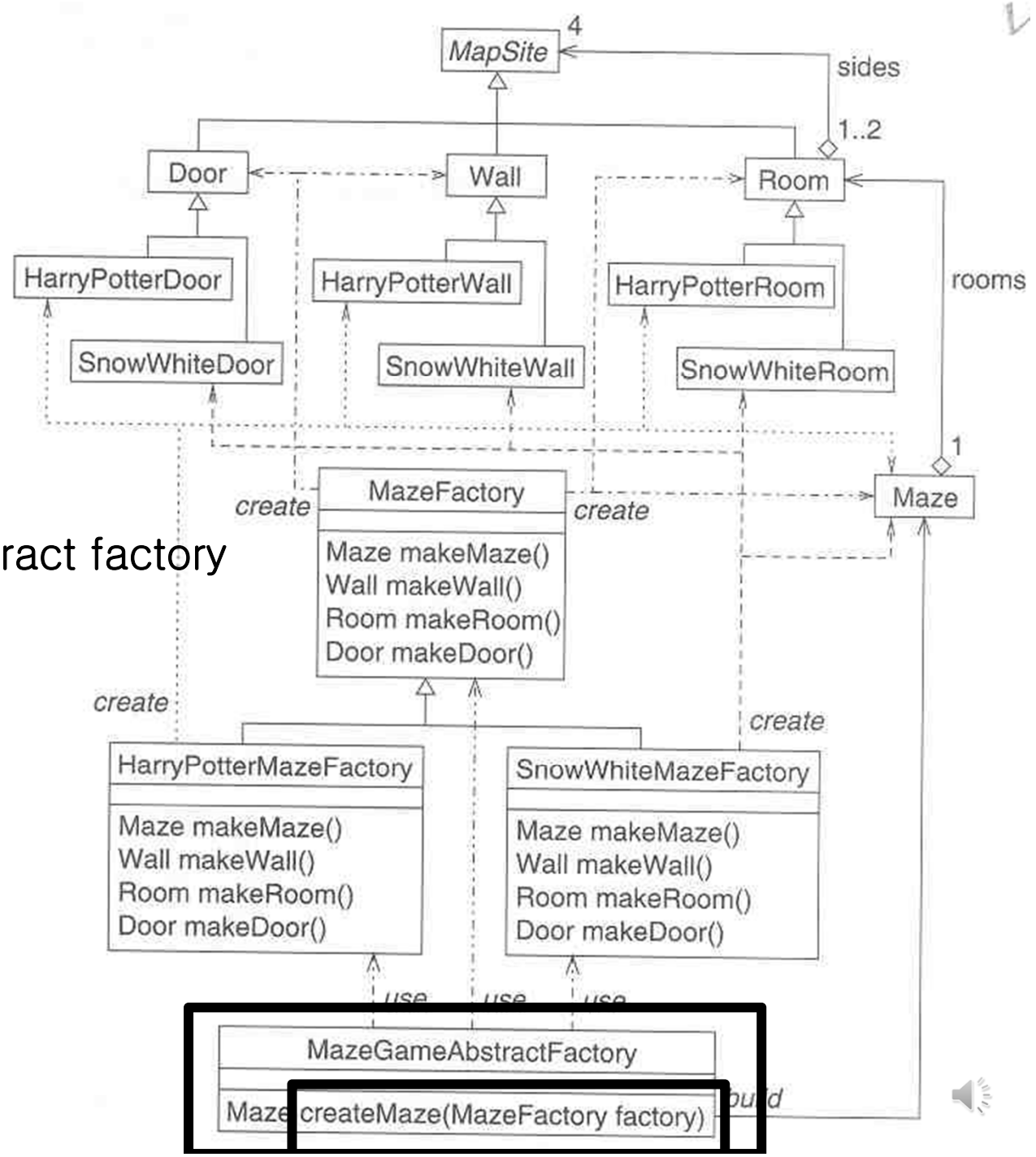
10.2

Figure 10.4

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

- Fig
  the

Abstract factory



```
// Class maze.MazeGameCreator

package maze;

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

/*
 *  Build a Maze game.
 *
 *  This design uses Factory Methods design pattern.
 *
 *  Compare this design with the one using Abstract
Factory design pattern:
 *    MazeGameAbstractFactory
 *
 */

public class MazeGameCreator {

  // This method must not be static
  public Maze createMaze() {
    Maze maze = makeMaze();
    Room room1 = makeRoom(1);
    Room room2 = makeRoom(2);
    Room room3 = makeRoom(3);
    Room room4 = makeRoom(4);
    Room room5 = makeRoom(5);
    Room room6 = makeRoom(6);
    Room room7 = makeRoom(7);
    Room room8 = makeRoom(8);
    Room room9 = makeRoom(9);
    Door door1 = makeDoor(room1, room2);
    Door door2 = makeDoor(room2, room3);
    Door door3 = makeDoor(room4, room5);
    Door door4 = makeDoor(room5, room6);
    Door door5 = makeDoor(room5, room8);
    Door door6 = makeDoor(room6, room9);
    Door door7 = makeDoor(room7, room8);
    Door door8 = 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, makeWall());
    room1.setSide(Direction.SOUTH, makeWall());
    room1.setSide(Direction.WEST, door1);

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

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

```
room4.setSide(Direction.NORTH, makeWall());
  room4.setSide(Direction.EAST, 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, makeWall());
  room5.setSide(Direction.WEST, door4);

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

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

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

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

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

   if (args.length > 0) {
     if ("Harry".equals(args[0])) {
            creator = new maze.harry.HarryPotterMazeGameCreator();
     } else if ("Snow".equals(args[0])) {
            creator = new maze.snow.SnowWhiteMazeGameCreator();
     }
   }
   if (creator == null) {
     creator = new MazeGameCreator();
   }
   maze = creator.createMaze();
   maze.setCurrentRoom(1);
   maze.showFrame("Maze -- Factory Method");
  }

}
```

```
C:₩Chapter10>java maze.MazeGameCreator 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
```

10.2.3 Design Pattern : Factory Method

- HerryPotterMazeGameCreator ,SnowWhiteMazeGameCreator Classes
  - the concrete creator
  - extend the creator classes, MazeGameCreator
  - override factory methods to create concrete map site objects in the respective themes.

```
// Class maze.HarryPotterMazeGameCreator

package maze.harry;

import maze.*;

public class HarryPotterMazeGameCreator extends MazeGameCreator {

  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);
  }

}
```

```
// Class maze.SnowWhiteMazeGameCreator

package maze.snow;

import maze.*;

public class SnowWhiteMazeGameCreator extends MazeGameCreator {

  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);
  }

}
```

10.2.4 Design Pattern : Prototype

- Shortcomings of the Abstract Factory and Factory Method patterns
  - 1) each product family is represented by a subclass
    - if there are many product families, there will also be many subclasses.
  - 2) it is possible to mix products from different product families, then potential number of combinations and the number of subclasses could explode

10.2.4 Design Pattern : Prototype

- Prototype pattern
  - prevent the explosion of the number of subclasses
  - objects are created by cloning a set of prototype.
  - objects in different product families can be created by using a different set of prototypes.
  - product families can be defined or altered at run time by choosing the desired prototype.
  - (abstract factory or factory method pattern, product families are defined in the form of a class and thus cannot be altered or configured at run time)

10.2.4 Design Pattern : Prototype

- Design Pattern : prototype

Design Pattern ***Prototype***

Category: Creational design pattern.

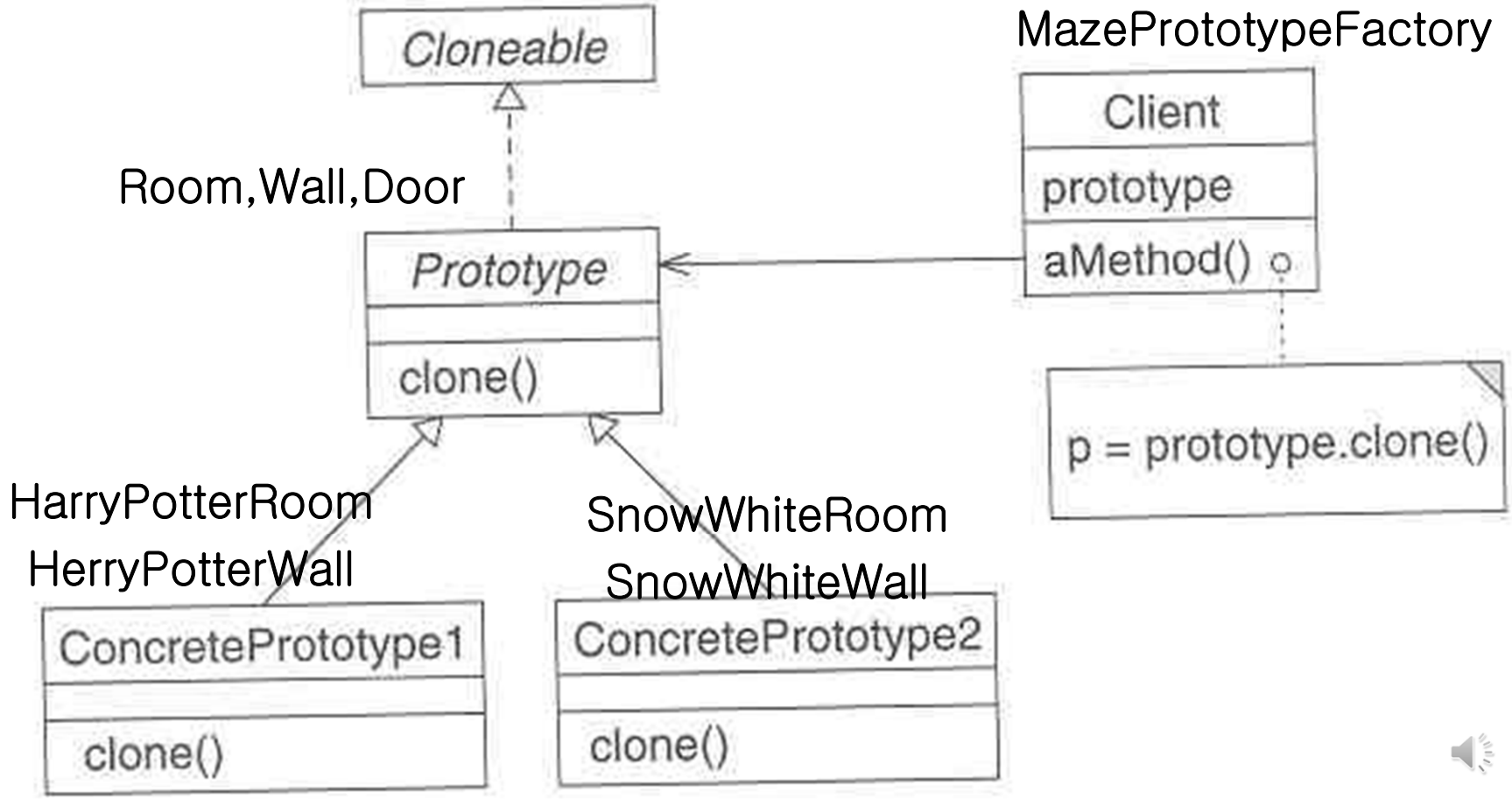
Intent: To specify the kinds of objects to create using a prototypical instance and create new instances by cloning this prototype.

Applicability: Use the Prototype design pattern

- when a system should be independent of how its components or products are created.
- when the classes to instantiate are specified at run time.
- to avoid building a class hierarchy of factories that parallels the class hierarchy of products.

10.2.4 Design Pattern : Prototype

- Structure of the Prototype (p. 496)



10.2.4 Design Pattern : Prototype

- The participants of the prototype design pattern

- *Prototype* (e.g., `Room`, `Wall`, `Door`), which defines interfaces of objects to be created, implements the `java.lang.Cloneable` interface and defines a public `clone()` method.
- *ConcretePrototype* (e.g., `HarryPotterRoom`, `HarryPotterWall`, `HarryPotterDoor`, `SnowWhiteRoom`, `SnowWhiteWall`, `SnowWhiteDoor`), which implements the *Prototype* interface and implements the `clone()` method.
- *Client* (e.g., `MazePrototypeFactory`), which creates new instances by cloning the prototype.

10.2.4 Design Pattern : Prototype

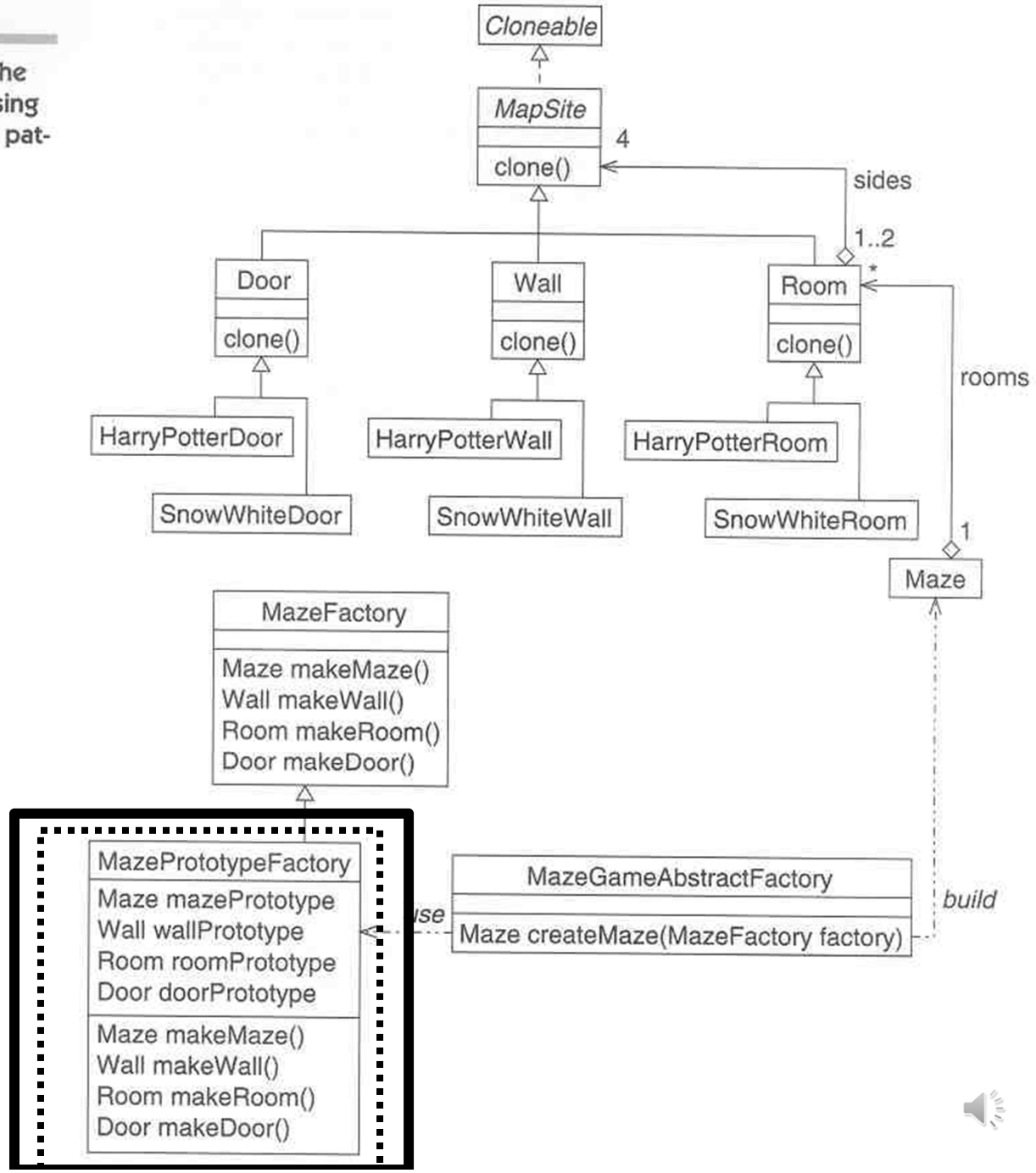
- Fig 10.6
  - this design is also an extension of the design using the Abstract Factory pattern
  - the MazePrototypeFactory
    - concrete factory in the Abstract Factory pattern
    - creates the map site objects using the prototype pattern
    - it contains a set of prototypes of the concrete map sites to be created
    - the concrete map site objects are created by cloning the corresponding prototypes.

10.2.4 [

- Figure 1
  the proto

Figure 10.6

A design of the maze game using the Prototype pattern.



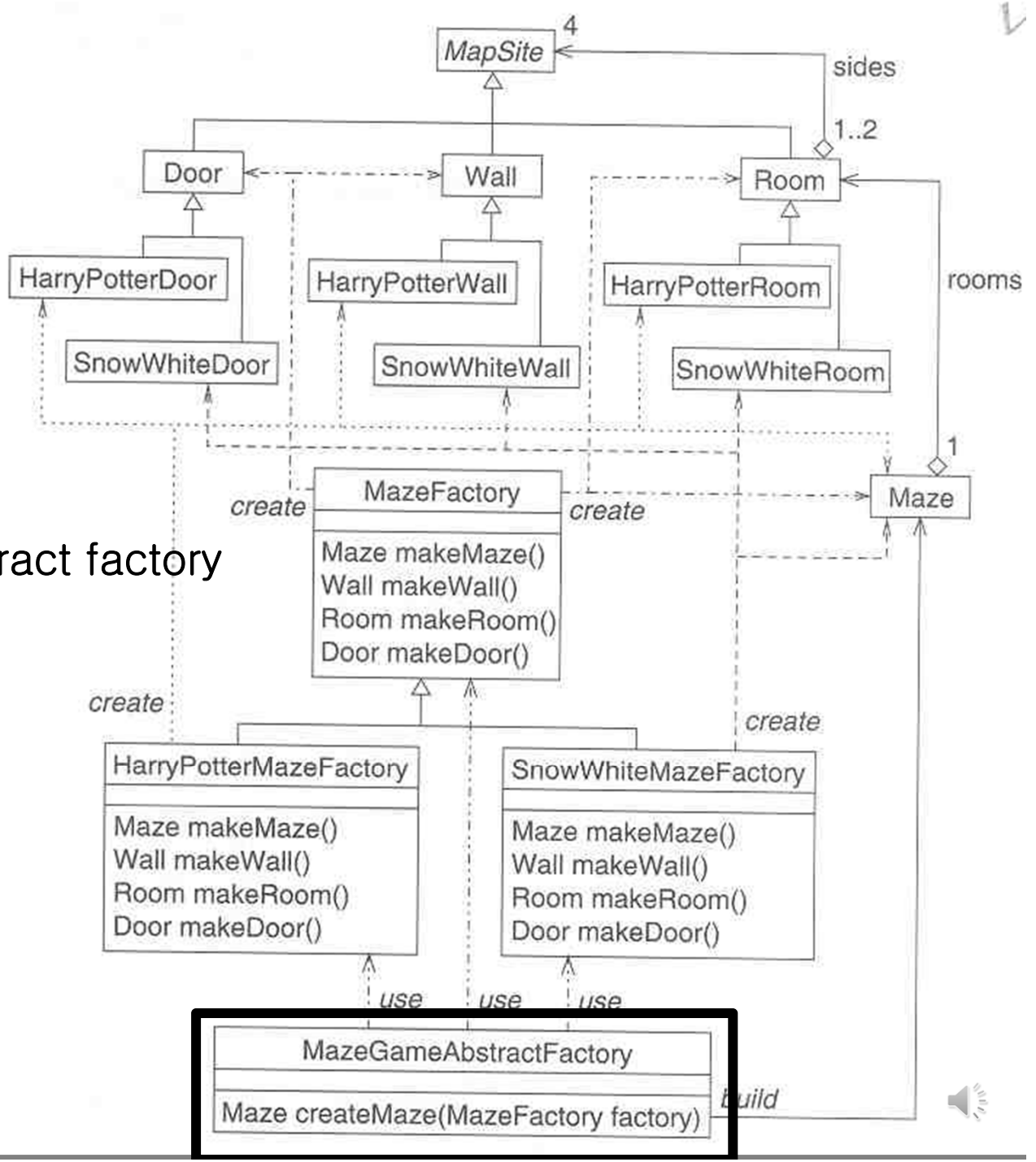
10.2

Figure 10.4

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

- Fig
  the

Abstract factory



```
// Class maze.MazePrototypeFactory
package maze;

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

public class MazePrototypeFactory extends
MazeFactory {

  public MazePrototypeFactory(Maze mazePrototype,
                              Wall wallPrototype,
                              Room roomPrototype,
                              Door doorPrototype) {
    this.mazePrototype = mazePrototype;
    this.wallPrototype = wallPrototype;
    this.roomPrototype = roomPrototype;
    this.doorPrototype = doorPrototype;
  }

  public Maze makeMaze() {   // p 486. p. 487 비교
    try {
      return (Maze) mazePrototype.clone();
    } catch (CloneNotSupportedException e) {

System.err.println("CloneNotSupportedException: "
+ e.getMessage());
    }
    return null;
  }

  public Wall makeWall() {
    try {
      return (Wall) wallPrototype.clone();
    } catch (CloneNotSupportedException e) {

System.err.println("CloneNotSupportedException: "
+ e.getMessage());
    }
    return null;
  }

  public Room makeRoom(int roomNumber) {
    try {
      Room room = (Room) roomPrototype.clone();
      room.setRoomNumber(roomNumber);
      return room;
    } catch (CloneNotSupportedException e) {

System.err.println("CloneNotSupportedException: "
+ e.getMessage());
    }
    return null;
  }

  public Door makeDoor(Room room1, Room room2)
{
    try {
      Door door = (Door) doorPrototype.clone();
      door.setRooms(room1, room2);
      return door;
    } catch (CloneNotSupportedException e) {

System.err.println("CloneNotSupportedException: "
+ e.getMessage());
    }
    return null;
  }
```

```
protected Maze mazePrototype;
 protected Wall wallPrototype;
 protected Room roomPrototype;
 protected Door doorPrototype;

 public static void main(String[] args) {
   Maze maze;
   MazePrototypeFactory    factory = null;
   MazeFactory    prototypeFactory = null;
   if (args.length > 0) {
     if ("Harry".equals(args[0])) {  // PrototypeFactory : concrete factory
             prototypeFactory = new maze.harry.HarryPotterMazeFactory();
     } else if ("Snow".equals(args[0])) {
             prototypeFactory = new maze.snow.SnowWhiteMazeFactory();
     }
   }
   if (prototypeFactory == null) {
     prototypeFactory = new MazeFactory();
   }
   factory = new MazePrototypeFactory(prototypeFactory.makeMaze(),
                                      prototypeFactory.makeWall(),
                                      prototypeFactory.makeRoom(0),
                                      prototypeFactory.makeDoor(null, null));

   maze = MazeGameAbstractFactory.createMaze(factory); // p. 488. Factory.*****
   maze.setCurrentRoom(1);
   maze.showFrame("Maze -- Prototype");
 }
}
```

```
package maze;

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

```
C:₩Chapter10>java maze.MazePrototypeFactory 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
```

10.2.4 Design Pattern : Prototype

- MazePrototypeFactory
  - can be invoked with an optional arguments
    - Harry or Snow
  - first creates a concrete factory, prototypeFactory
    - prototypeFactory is used to create prototype
  - then an instance of MazePrototypeFactory, the Prototype pattern-based factory is used as a concrete factory in the createMaze() method of the MazeGameAbstractFactory class to build the maze board
- 추가 : Prototype Design Pattern

10.2.4 Design Pattern : Prototype

- Configurable Universal Factory
  - reconfigurability
    - one of the advantages of using factories is reconfigurability
  - How to pass the configuration settings to program
    - 1) command-line arguments
    - 2) Operating system environment variables
      - plat dependent
    - 3) Java properties
      - property : mapping (keys, values)
      - can be stored in a text file
      - can be easily edited

10.2.4 Design Pattern : Prototype

- UniversalMazeFactory Class
  - can build maze games with various schemes and using various styles of factories
  - selects the scheme and factory based on the following two properties.
  - two properties
    - 1) maze.theme : indicates the theme of the game board, one of the game board, one of Harry, Snow, or Simple
    - 2) maze.prototype : indicates whether to use prototype-based factory, either true or false.

10.2.4 Design Pattern : Prototype

- how to specify these properties
  - 1) file named maze.properties
  - maze.theme = Harry
  - maze.prototype = true
  - 2) –D Command-line option
  - java –Dmaze.theme=Harry –Dmaze.prototype=true maze.UniversalMazeFactory

10.2.4 Design Pattern : Prototype

- UniversalMazeFactory Class
  - as a singleton
  - the static initialization block is executed
    - tries to find the properities file named maze.properties
  - getInstance() method
    - creates a factory based on the values of properties read in the initialization

// Class maze.UniversalMazeFactory

```
package maze;

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

public class UniversalMazeFactory extends MazeFactory {

  public static MazeFactory getInstance() {
    if (theInstance == null) {
      if (usePrototype) {
                MazeFactory factory = null;
                switch (theme) {
                case HARRY_PORTER_THEME:
                  factory = new maze.harry.HarryPotterMazeFactory();
                  break;
                case SNOW_WHITE_THEME:
                  factory = new maze.snow.SnowWhiteMazeFactory();
                  break;
                }
                if (factory == null) {
                  factory = new MazeFactory();
                }
                theInstance = new MazePrototypeFactory(factory.makeMaze(),
                                                       factory.makeWall(),
                                                       factory.makeRoom(0),
                                                       factory.makeDoor(null, null));
```

```
} else {
                switch (theme) {
                case HARRY_PORTER_THEME:
                  theInstance = new maze.harry.HarryPotterMazeFactory();
                  break;
                case SNOW_WHITE_THEME:
                  theInstance = new maze.snow.SnowWhiteMazeFactory();
                  break;
                default:
                  theInstance = new MazeFactory();
                  break;
                }
     }
    }
    return theInstance;
  }
protected UniversalMazeFactory() {}

  private static MazeFactory theInstance = null;

  private static final int SIMPLE_THEME = 0;
  private static final int HARRY_PORTER_THEME = 1;
  private static final int SNOW_WHITE_THEME = 2;
  private static boolean usePrototype = true;
  private static int theme = SIMPLE_THEME;
```

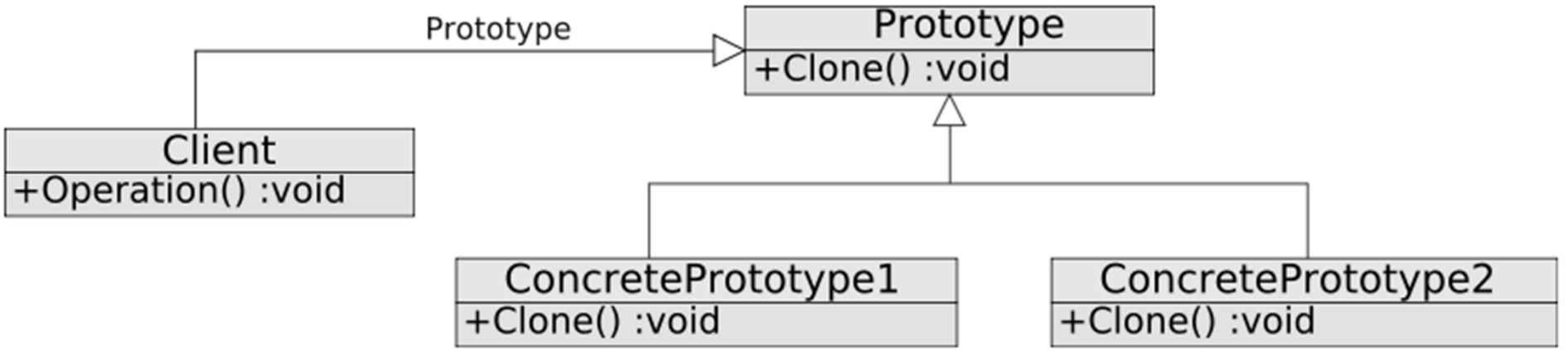
```
 static {
   Properties configProperties = new Properties();
   try {
     configProperties.load(new FileInputStream("maze.properties"));
   } catch (IOException e) {}
   String value;
   value = System.getProperty("maze.theme");
   if (value == null) {
     value = configProperties.getProperty("maze.theme");
   }
   if (value != null) {
     if ("Harry".equals(value)) {
               theme = HARRY_PORTER_THEME;
     } else if ("Snow".equals(value)) {
               theme = SNOW_WHITE_THEME;
     }
   }

   value = System.getProperty("maze.prototype");
   if (value == null) {
     value = configProperties.getProperty("maze.prototype");
   }
   if (value != null) {
     usePrototype = Boolean.getBoolean(value);
   }
 }

 public static void main(String[] args) {
   MazeFactory factory = UniversalMazeFactory.getInstance();
   Maze maze = MazeGameAbstractFactory.createMaze(factory);
   maze.setCurrentRoom(1);
   maze.showFrame("Maze -- Universal Factory");
 }
}
```

```
C:₩Chapter10>java -Dmaze.theme=Harry -Dmaze.prototype=true maze.UniversalMazeFa
tory
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
```

A prototype pattern

- a creational design pattern used in software development when the type of objects to create is determined by a prototypical instance, which is cloned to produce new objects. This pattern is used to:
  - avoid subclasses of an object creator in the client application, like the abstract factory pattern does.
  - avoid the inherent cost of creating a new object in the standard way (e.g., using the 'new' keyword) when it is prohibitively expensive for a given application.

- To implement the pattern, declare an abstract base class that specifies a pure virtual *clone()* method. Any class that needs a "polymorphic constructor" capability derives itself from the abstract base class, and implements the *clone()* operation.
- The client, instead of writing code that invokes the "new" operator on a hard-wired class name, calls the *clone()* method on the prototype, calls a factory method with a parameter designating the particular concrete derived class desired, or invokes the *clone()* method through some mechanism provided by another design pattern.


Prototype
Prototype
+Clone() :void
Client
+Operation() :void
ConcretePrototype1
+Clone() :void
ConcretePrototype2
+Clone() :void

```
/** Prototype Class **/
public class Cookie implements Cloneable {

   public Object clone() {
      try {
         Cookie copy = (Cookie)super.clone();

         //In an actual implementation of this pattern you might now change references to
         //the expensive to produce parts from the copies that are held inside the prototype.

         return copy;

      }
      catch(CloneNotSupportedException e) {
         e.printStackTrace();
         return null;
      }
   }
}

/** Concrete Prototypes to clone **/
public class CoconutCookie extends Cookie { }
```

```java
/** Client Class**/
public class CookieMachine {

    private Cookie cookie;//could have been a private Cloneable cookie;

    public CookieMachine(Cookie cookie) {
        this.cookie = cookie;
    }
    public Cookie makeCookie() {
        return (Cookie)cookie.clone();
    }
    public Object clone() { }

    public static void main(String args[]) {
        Cookie tempCookie =  null;
        Cookie prot = new CoconutCookie();
        CookieMachine cm = new CookieMachine(prot);
        for (int i=0; i<100; i++)
            tempCookie = cm.makeCookie();
    }
}
```

- 또다른 예제 Prototype 예제-F.ppt

PROTOTYPE

- Prototype 패턴의 유용성
  - 1) 객체를 생성하는 시점에 객체의 자료형을 정해주지 않아도 됨
  - 2) 실행 시간(Run-Time)에도 자료형에 무관하게 객체를 생성할 수 있음
  - 3) 생성되는 객체의 자료형과는 무관하게 동일한 형태로 동작이 가능함

- Prototype패턴
  - 클래스로부터 인스턴스를 생성하는 것이 아니라 인스턴스로부터 또 다른 인스턴스를 만들어내는 패턴

Prototype 패턴이 유용한 경우

- 1) 객체를 생성하는 방식이나 객체의 구성 형태, 표현 방식 등과는 무관하게 객체를 생성하고 싶을 때
  - 왜냐하면 Prototype 패턴은 직접 객체를 생성하는 방식이 아니라, 원본 객체를 Clone() 멤버 함수를 통해 복제하는 방식이므로 객체를 생성하는 방식이나 구성 형태, 표현 방식 등이 모두 Clone() 멤버 함수 내부에서 결정되고 Clien 입장에서는 전혀 신경쓸 필요가 없기 때문이다.
- 2) 생성할 객체가 실행 시간(Run-Time)에 결정될 경우
  - 생성할 객체가 실행 시간에 결정된다는 것은 실행 시간 때까지 어떤 자료형의 객체를 생성해야 할지를 알 수 없다는 것을 의미
  - 따라서 일반적인 형태로 객체의 자료형을 명시하고 객체를 생성하는 방식은 적용이 불가능
  - 반면 Prototype 패턴은 생성될 객체의 자료형을 정해줄 필요 없이 기존의 객체를 복제만 하는 것이므로, 실행 시간에 객체가 결정되어도 충분히 객체 생성을 수행할 수 있음

- 3) 생성될 객체의 클래스 구조와 병행해서 객체 생성을 위한 클래스들을 정의하고 싶지 않을 때 유용
  - Prototype 패턴은 별도의 객체 생성 클래스들이 불필요
- 4) 미리 각 상태별로 객체를 만들어두고, 필요할 경우 원하는 상태의 객체를 복제해서 사용할 수 있어 유용
  - 어떤 클래스의 객체들이 몇 개 안 되는 상태들 중 하나에 머무르는 형태일 경우 Prototype 패턴을 사용하면 미리 각 상태별로 객체를 만들어두고, 필요할 경우 원하는 상태의 객체를 복제해서 사용할 수 있어 유용.

- 5) 해당 객체의 상태 값을 변경시켜줌으로써 새로운 행위의 수행이 가능
  - 완전히 새로운 행위를 수행하는 객체를 생성하고자 할 때 새로운 클래스를 정의하지 않고, 기존 클래스로부터 객체를 생성해서 해당 객체의 상태 값을 변경시켜줌으로써 새로운 행위의 수행이 가능하게 하는 것이 효율적일 수 있다.
- 6) 어떤 객체의 생성이 각 부분 부분을 조합해서 생성되는 형태인 경우
  - 특히 전체 객체를 구성하는 각 부분들이 실행 시간(Run-Time)에 자료형이 결정될 경우에는 더욱 그렇다.
- 7) 실행 시간 환경(Run-Time Environment)에 의해 동적으로 로딩되는 클래스가 있는 경우 그 클래스의 객체를 생성하기 위해서는 Prototype 패턴을 적용하는 것이 유용
  - 왜냐하면 이 경우 클래스의 생성자(Constructor)를 통해 객체를 생성하고 싶어도 클래스가 동적으로 로딩되기 때문에 이를 정적으로 참조할 수 없기 때문이다.

Prototype 패턴의 장점

- 1) Prototype 패턴은 Abstract Factory나 Builder 패턴 등과는 달리 객체를 생성해주기 위해 별도의 클래스를 정의할 필요가 없음
  - Abstract Factory나 Builder 패턴의 경우에는 생성될 객체의 자료형에 따라 객체를 생성해주기 위한 클래스들을 정의해야 함
- 2) Prototype 패턴은 실행 시간(Run-Time)에도 생성할 객체의 종류를 추가 또는 삭제할 수 있다는 장점
  - 물론 이를 위해 별도의 Prototype Manager와 같은 객체가 필요할 수는 있으나, 동적으로 생성할 객체의 종류를 변경할 수 있다는 것은 일반적인 객체 생성 방식들이 가지지 못하는 큰 장점
  - 이와 비슷한 이유로 Prototype 패턴의 경우에는 Client가 생성할 객체의 종류를 객체 생성 시에 정해줄 필요가 없다는 장점도 있음

Prototype 패턴의 단점

- Prototype패턴의 가장 큰 단점은 생성될 객체들의 자료형인 클래스들이 모두 Clone() 멤버 함수를 구현해야 한다는 것
  - Clone() 멤버 함수의 구현은 때로 매우 까다로울 수 있음
  - 예를 들어 클래스 내부 구조가 순환 참조를 하고 있는 경우에는 Clone() 멤버 함수의 구현이 쉽지 않다.

10.2.5 Design Pattern : Builder

- createMaze (p. 492-뒷페이지)
  - constructs a maze board
  - lengthy and contains many code segments that perform identical tasks

- Builder design pattern
  - can be used to simplify the construction process by avoiding the duplication of similar code
  - shortening the code for the construction

```
// Class maze.MazeGameCreator  - P. 492

package maze;

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

/*
 *  Build a Maze game.
 *
 *  This design uses Factory Methods design pattern.
 *
 *  Compare this design with the one using Abstract
Factory design pattern:
 *    MazeGameAbstractFactory
 *
 */

public class MazeGameCreator {

  // This method must not be static
  public Maze createMaze() {
    Maze maze = makeMaze();
    Room room1 = makeRoom(1);
    Room room2 = makeRoom(2);
    Room room3 = makeRoom(3);
    Room room4 = makeRoom(4);
    Room room5 = makeRoom(5);
    Room room6 = makeRoom(6);
    Room room7 = makeRoom(7);
    Room room8 = makeRoom(8);
    Room room9 = makeRoom(9);
    Door door1 = makeDoor(room1, room2);
    Door door2 = makeDoor(room2, room3);
    Door door3 = makeDoor(room4, room5);
    Door door4 = makeDoor(room5, room6);
    Door door5 = makeDoor(room5, room8);
    Door door6 = makeDoor(room6, room9);
    Door door7 = makeDoor(room7, room8);
    Door door8 = 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, makeWall());
    room1.setSide(Direction.SOUTH, makeWall());
    room1.setSide(Direction.WEST, door1);

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

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

```
room4.setSide(Direction.NORTH, makeWall());
  room4.setSide(Direction.EAST, 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, makeWall());
  room5.setSide(Direction.WEST, door4);

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

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

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

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

10.2.5 Design Pattern : Builder

Design Pattern ***Builder***

Category: Creational design pattern.

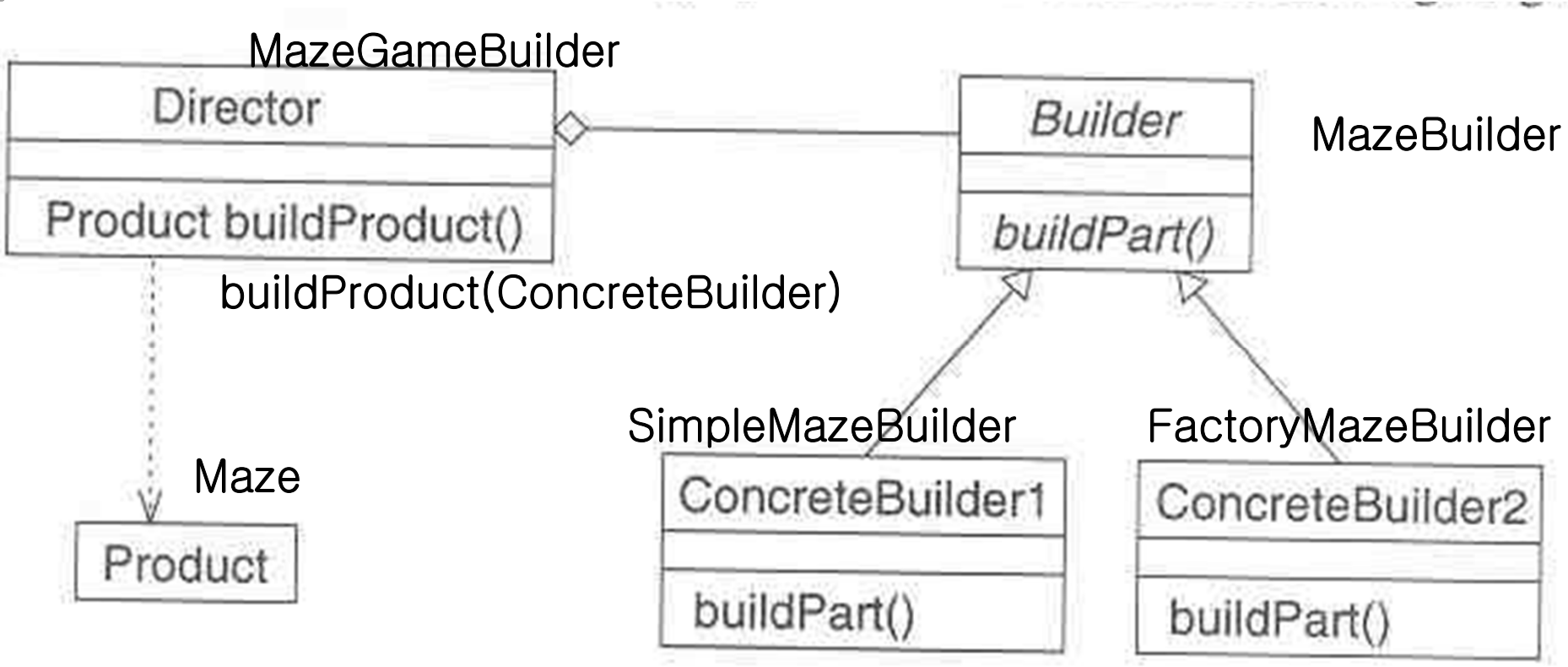
Intent: To separate the construction of a complex object from the implementation of its parts so that the same construction process can create complex objects from different implementations of parts.

Applicability: Use the Builder design pattern

- when the process for creating a complex object should be independent of the parts that make up the object.
- when the construction process should allow various implementations of the parts used for construction.

10.2.5 Design Pattern : Builder

- The Structure of the Builder design pattern



10.2.5 Design Pattern : Builder

- The participants of the Builder design pattern

- *Builder* (e.g., `MazeBuilder`), which defines an interface for creating parts of a *Product* object.
- *ConcreteBuilder* (e.g., `SimpleMazeBuilder`, `FactoryMazeBuilder`), which constructs and assembles parts of the product by implementing the *Builder* interface.
- *Director* (e.g., `MazeGameBuilder`), which constructs a *Product* object using the *Builder* interface.
- *Product* (e.g., `Maze`), which represents the complex object under construction.

10.2.5 De

Methods	Description
`newMaze()`	Start to build a new maze board.
`getMaze()()`	Retrieve the maze boards that have been completely built.
`buildRoom()`	Build a new room with walls on all four sides.
`buildDoor()`	Build a door between two rooms.

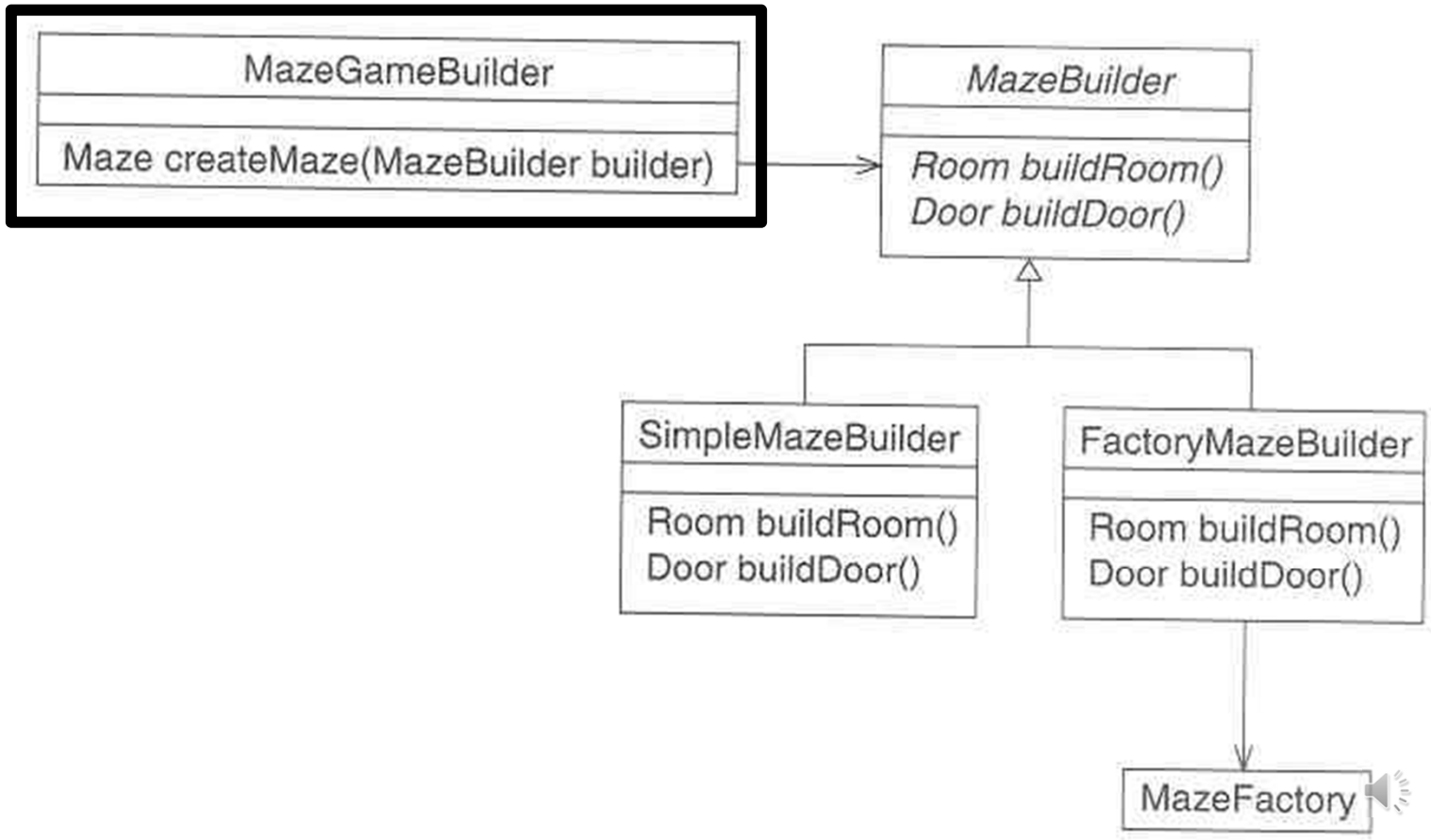


Figure 10.7

A design of the maze game using the Builder pattern.

// Class maze.MazeBuilder ➔ Builder inteface

```
package maze;

public interface MazeBuilder {

  /**
   *  Start to build a new Maze
   */
  public void newMaze();

  /**
   *  Fetch the Maze that have been built.
   */
  public Maze getMaze();

  /**
   *  Build a new room in the current Maze.
   */
  public void buildRoom(int roomNumber);

  /**
   *  Build a new door in the current Maze between two the specified rooms.
   *
   *  @param roomNumber1 specifies the room number of the first room
   *  @param roomNumber2 specifies the room number of the second room
   *  @param dir         specifies the side of the first room at which the door will be located
   *                     the second room will be on the other side of the door.
   *  @param open        whether the door is open or not
   */
  public void buildDoor(int roomNumber1, int roomNumber2,
                                          Direction dir, boolean open);

}
```

```
// Class maze.SimpleMazeBuilder
// concrete builder : constructs the map site objects in the default themes using the new operator
package maze;

public class SimpleMazeBuilder implements MazeBuilder {

  /**
   *  Start to build a new Maze
   */
  public void newMaze() {
    maze = new Maze();
  }

  /**
   *  Fetch the Maze that have been built.
   */
  public Maze getMaze() {
    return maze;
  }

  /**
   *  Build a new room in the current Maze.
   */
  public void buildRoom(int roomNumber) {
    if (maze == null) {
      newMaze();
    }
    Room room = new Room(roomNumber);
    for (Direction dir = Direction.first(); dir != null; dir = dir.next()) {
      room.setSide(dir, new Wall());
    }
    maze.addRoom(room);
  }
```

```
  /**
   *  Build a new door in the current Maze between two the specified rooms.
   *
   *  @param roomNumber1 specifies the room number of the first room
   *  @param roomNumber2 specifies the room number of the second room
   *  @param dir         specifies the side of the first room at which the door will be located
   *                     the second room will be on the other side of the door.
   */
  public void buildDoor(int roomNumber1, int roomNumber2, Direction dir, boolean open) {
    if (maze == null) {
      newMaze();
    }
    Room room1 = maze.findRoom(roomNumber1);
    Room room2 = maze.findRoom(roomNumber2);
    if (room1 != null &&
                room2 != null &&
                dir != null) {
      Door door = new Door(room1, room2);
      room1.setSide(dir, door);
      room2.setSide(dir.opposite(), door);
      door.setOpen(open);
    }
  }

  protected Maze maze;

}
```

```
// Class maze.FactoryMazeBuilder
// Concrete Builder
// uses a factory to construct the map site objects,
// so that different thems can be supported
package maze;

public class FactoryMazeBuilder implements MazeBuilder {

  public FactoryMazeBuilder(MazeFactory factory) {
    this.factory = factory;
  }

  /**
   *  Start to build a new Maze
   */
  public void newMaze() {
    maze = factory.makeMaze();
  }

  /**
   *  Fetch the Maze that have been built.
   */
  public Maze getMaze() {
    return maze;
  }
```

```
  /**
   *  Build a new room in the current Maze.
   */
  public void buildRoom(int roomNumber) {
    if (maze == null) {
      newMaze();
    }
    Room room = factory.makeRoom(roomNumber);
    for (Direction dir = Direction.first(); dir != null; dir
= dir.next()) {
      room.setSide(dir, factory.makeWall());
    }
    maze.addRoom(room);
  }
```

```
/**
   *  Build a new door in the current Maze between two the specified rooms.
   *
   *  @param roomNumber1 specifies the room number of the first room
   *  @param roomNumber2 specifies the room number of the second room
   *  @param dir         specifies the side of the first room at which the door will be located
   *                     the second room will be on the other side of the door.
   */
  public void buildDoor(int roomNumber1, int roomNumber2, Direction dir, boolean open) {
    if (maze == null) {
      newMaze();
    }
    Room room1 = maze.findRoom(roomNumber1);
    Room room2 = maze.findRoom(roomNumber2);
    if (room1 != null &&
                room2 != null &&
                dir != null) {
      Door door = factory.makeDoor(room1, room2);
      room1.setSide(dir, door);
      room2.setSide(dir.opposite(), door);
      door.setOpen(open);
    }
  }
 protected MazeFactory factory;
  protected Maze maze;

}
```

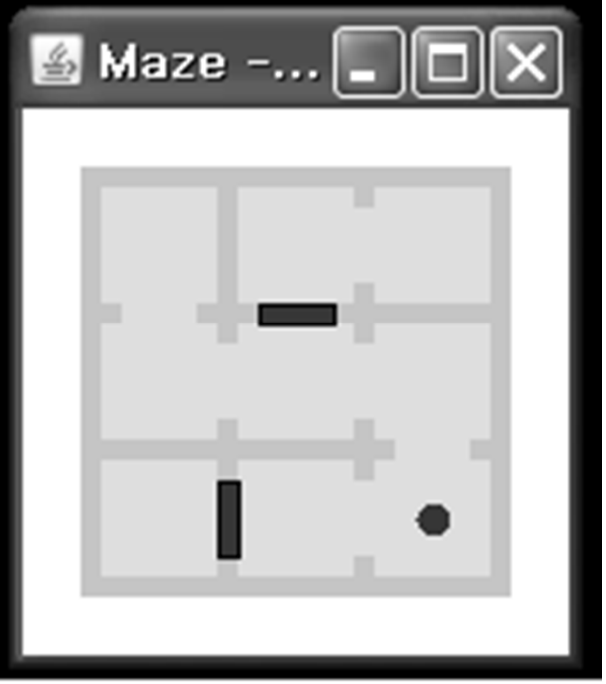
- MazeGameBuilder Class
  - the director in the Builder Pattern
  - constructs the maze board from the components build by the builder
  - the createMaze() method : construct the same 3*3 maze board constructed in the previous examples.
  - become much simpler and shorter

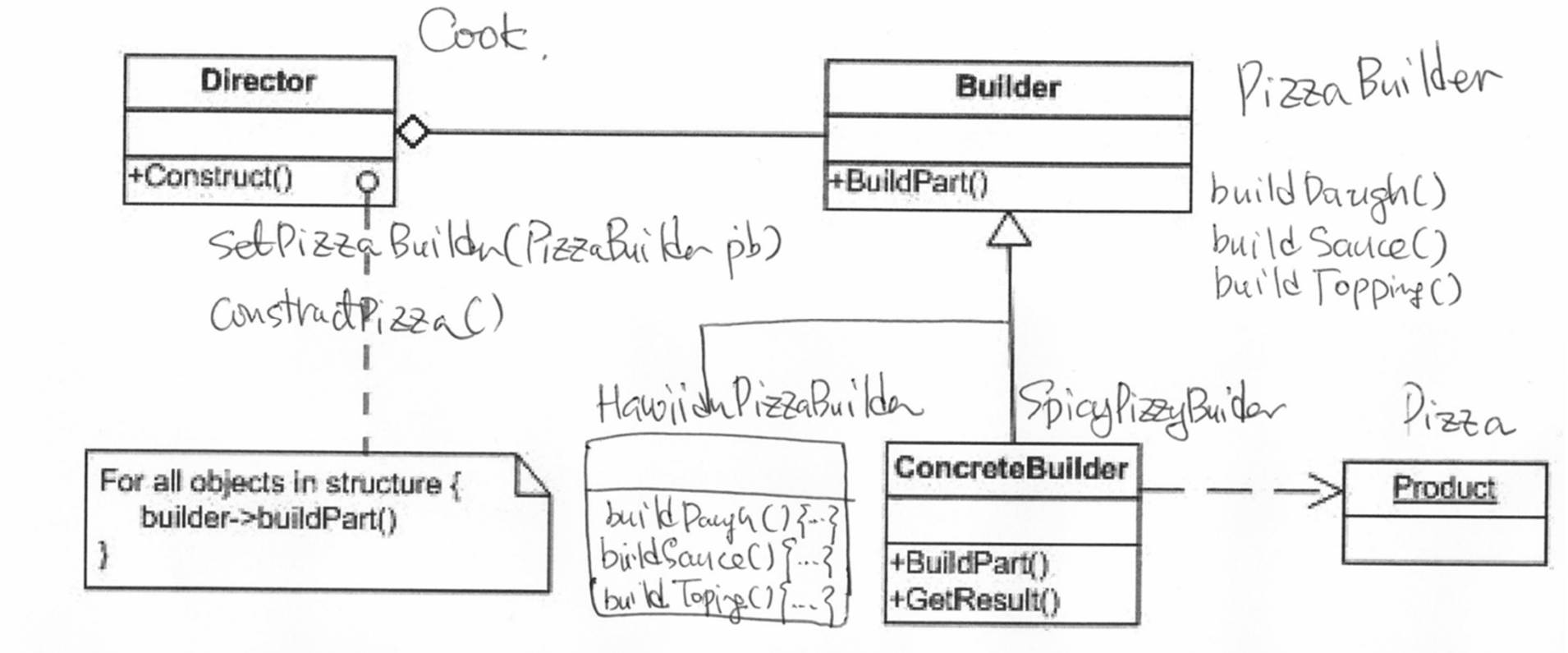
```
// Class maze.MazeGameBuilder
// director in the builder pattern
// constructs the maze board from the components build by the builder

// createMaze() : constructs the same 3*3 maze board constructed in the previous examples.
// Using Builder pattern, the createMaze becomes much simpler and shorter
package maze;

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

/*
 *  Build a Maze game.
 *
 *  This implmentation uses Builder and Abstract Factory design patterns.
 *
 *  Run the program as follows:
 *   java maze.MazeGameBuilder Harry
 *        -- uses the FactoryMazeBuilder with the HarryPotterMazeFactory
 *   java maze.MazeGameBuilder Snow
 *        -- uses the FactoryMazeBuilder with the SnowWhiteMazeFactory
 *   java maze.MazeGameBuilder Simple
 *        -- uses the FactoryMazeBuilder with the default MazeFactory
 *   java maze.MazeGameBuilder
 *        -- uses the SimpleMazeBuilder, which does not use a factory
 *
 */
```

```
public class MazeGameBuilder {

public static Maze createMaze(MazeBuilder builder)
{
    builder.newMaze();
    builder.buildRoom(1);
    builder.buildRoom(2);
    builder.buildRoom(3);
    builder.buildRoom(4);
    builder.buildRoom(5);
    builder.buildRoom(6);
    builder.buildRoom(7);
    builder.buildRoom(8);
    builder.buildRoom(9);

    builder.buildDoor(1, 2, Direction.WEST, true);
    builder.buildDoor(2, 3, Direction.WEST, false);
    builder.buildDoor(4, 5, Direction.WEST, true);
    builder.buildDoor(5, 6, Direction.WEST, true);
    builder.buildDoor(5, 8, Direction.NORTH, false);
    builder.buildDoor(6, 9, Direction.NORTH, true);
    builder.buildDoor(7, 8, Direction.WEST, true);
    builder.buildDoor(1, 4, Direction.NORTH, true);

   return builder.getMaze();
  }
```

```
public static void main(String[] args) {
   Maze maze;
   MazeBuilder builder;
   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();
     } else if ("Default".equals(args[0])) {
                factory = new MazeFactory();
     }
   }
   if (factory != null) {
     builder = new FactoryMazeBuilder(factory);
   } else {
     builder = new SimpleMazeBuilder();
   }
   maze = createMaze(builder);
   maze.setCurrentRoom(1);
   maze.showFrame("Maze -- Builder");
 }

}
```

```
C:₩Chapter10>java maze.MazeGameBuilder Default
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
```





- ((Builder Design Pattern))
  - The intention is to abstract steps of construction of objects so that different implementations of these steps can construct different representations of objects

- Builder
  - Abstract interface for creating objects (product).
- Concrete Builder
  - Provide implementation for Builder. Construct and assem ble parts to build the objects.
- Director
  - responsible for managing the correct sequence of object creation.
  - receives a Concrete Builder as a parameter
  - and executes the necessary operations on it.
- Product
  - The final object that will be created by the Director using Builder.

```
/** "Product" */
class Pizza {
        private String dough = "";
        private String sauce = "";
        private String topping = "";

        public void setDough(String dough) {
                this.dough = dough;
        }

        public void setSauce(String sauce) {
                this.sauce = sauce;
        }

        public void setTopping(String topping) {
                this.topping = topping;
        }
}
```

```
/** "Abstract Builder" */
abstract class PizzaBuilder {
        protected Pizza pizza;

        public Pizza getPizza() {
                return pizza;
        }

        public void createNewPizzaProduct() {
                pizza = new Pizza();
        }

        public abstract void buildDough();

        public abstract void buildSauce();

        public abstract void buildTopping();
}
```

```
/** "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
        public void buildDough() {
                pizza.setDough("cross");
        }

        public void buildSauce() {
                pizza.setSauce("mild");
        }

        public void buildTopping() {
                pizza.setTopping("ham+pineapple");
        }
}
/** "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
        public void buildDough() {
                pizza.setDough("pan baked");
        }

        public void buildSauce() {
                pizza.setSauce("hot");
        }

        public void buildTopping() {
                pizza.setTopping("pepperoni+salami");
        }
}
```

```
/** "Director" */
class Cook {
        private PizzaBuilder pizzaBuilder;

        public void setPizzaBuilder(PizzaBuilder pb) {
                pizzaBuilder = pb;
        }

        public Pizza getPizza() {
                return pizzaBuilder.getPizza();
        }

        public void constructPizza() {
                pizzaBuilder.createNewPizzaProduct();
                pizzaBuilder.buildDough();
                pizzaBuilder.buildSauce();
                pizzaBuilder.buildTopping();
        }
}
```

```
/** A given type of pizza being constructed. */
public class BuilderExample {
        public static void main(String[] args) {
                Cook cook = new Cook();
                PizzaBuilder hawaiianPizzaBuilder = new HawaiianPizzaBuilder();
                PizzaBuilder spicyPizzaBuilder = new SpicyPizzaBuilder();

                cook.setPizzaBuilder(hawaiianPizzaBuilder);
                cook.constructPizza();

                Pizza pizza = cook.getPizza();
        }
}
```

- 또다른 예제와 분석 : Builder Pattern
  - Builder 예제-F.ppt

BUILDER 패턴

- Builder 패턴
  - 객체를 생성하되, 그 객체를 구성하는 부분 부분(ex. Dough, Souce,Topping)을 먼저 생성하고, 이를 조합함으로써 전체 객체(Pizza)를 생성하며, 생성할 객체의 종류가 손쉽게 추가, 확장(Ex.불고기 피자)이 가능하게 고안된 클래스 설계
  - 복합 객체의 생성 과정(Ex. 불고기피자)과 표현 방법(Dough, Source, Topping)을 분리하여 동일한 생성 절차에서 서로 다른 표현 결과(Ex.불고기 피자)를 만들 수 있게 하는 패턴

Builder 패턴이 유용한 경우

- 복잡한 객체를 생성하는 데 있어 그 객체를 구성하는 부분 부분들을 생성한 후 그것을 조합해도 객체의 생성이 가능할 때, 즉 객체의 부분 부분을 생성하는 것(Dough, Topping) 과 그들을 조합해서 전체 객체를 생성하는 것(MakePizza)이 서로 독립적으로 이루어질 수 있는 경우.

- 서로 다른 표현 방식을 가지는 객체(하와이안 피자, 불고기 피자) 를 동일한 방식(세가지 요소로 피자를 만듬)으로 생성하고 싶을 때.

Builder 패턴의 장점

- Builder 클래스는 Director 클래스에게 객체를 생성할 수 있는 인터페이스를 제공한다. 대신 생성되는 객체의 내부 구조나 표현 방식, 조합 방법 등은 Builder 클래스 내부로 숨긴다. 이런 형태이기 때문에 Builder 패턴에서는 생성되는 객체의 내부 구조를 변경시킬 필요가 있을 경우 기존 소스 코드에는 영향을 주지 않고, 새로운 하위 클래스를 추가 정의하면 원하는 작업을 수행할 수 있는 장점이 있다.
- Builder 패턴은 객체를 생성하는 부분과 그것을 실제 표현하고 구성하는 부분을 분리시켜줌으로써 서로 간의 독립성을 기할 수 있게 해준다. 이를 통해 Builder와 Director 클래스는 서로 독립적으로 수정되거나 재사용될 수 있다.
- Builder 패턴의 경우에는 객체를 한꺼번에 생성하는 것이 아니라, 부분 부분을 생성한 후 최종 결과를 얻어가는 방식이므로, 객체 생성 과정을 좀더 세밀히 제어할 수 있다.

Builder 패턴의 단점

- Builder 패턴은 새로운 종류의 객체(Ex. 김치피자)를 추가하기는 쉬우나, 객체를 구성하는 각 부분(새로운 구성요소)들을 새롭게 추가하기는 어렵다. 따라서 Builder 패턴을 적용하고자 할 경우에는 생성되는 객체를 구성하는 부분들을 명확히 해서 추가하거나 수정해야 할 부분이 없게 하는 것이 중요하다.