객체지향 설계와 패턴

Lecture #9: 생성 패턴(1)

Eun Man Choi
emchoi@dgu.ac.kr

학습 목표

- 생성패턴
  - 동기, 목적
  - 종류
- 팩토리 패턴과 그 적용
- 싱글톤 패턴과 그 적용
- 프로토타입 패턴과 그 적용
- 실습 문제

생성 패턴

- 생성 패턴
  - 객체의 생성과정을 추상화하여 *유연성(flexibility)*을 높인다.
  - 실제 객체와 객체생성 부분을 분리함으로써 결합도(coupling)를 낮춘다.
- 종류
  - 팩토리 패턴
  - 싱글톤 패턴
  - 프로토타입 패턴
  - 추상 팩토리 패턴
  - 빌더 패턴

Pattern #1 - 팩토리 패턴

생성자만으로는 개별 객체 생성이 적합하지 않은 경우 사용

- 목적
  - 객체생성을 위한 인터페이스를 정의하는데 있다.
  - 어떤 클래스의 인스턴스를 생성할지에 대한 결정은 서브클래스에서 이루어지도록 인스턴스의 책임을 미룸
- 결과
  - 다양한 형태의 객체를 반환하는 융통성을 갖게 됨

팩토리 패턴을 사용하는 이유

- 생성자 사용할 때의 문제

```
Duck duck;
if (picnic) {
  duck = new MallardDuck();
} else if (hunting) {
  duck = new DecoyDuck();
} else if (inBathTub) {
  duck = new RubberDuck();
}
```

새로운 타입이 추가될 때 문제가 됨

사례 #1 - 피자 스토어

```
Pizza orderPizza(String type){
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if type.equals("greek")) {
    pizza = new GreekPizza();
    } else if type.equals("pepperoni")) {
    pizza = new PepperoniPizza();
    }
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box()
}
```

- 만일 피자 스토어에서 주문하는 피자 종류를 바꾸기로 결정한다면 orderPizza 메소드는 바뀌어야

사례 #1- 피자 스토어 생성자

```
Pizza orderPizza(String type){
    if (type.equals("cheese")) {
      pizza = new CheesePizza();
    } else if type.equals("greek")) {
    pizza = new GreekPizza();
    } else if type.equals("pepperoni")) {
    pizza = new PepperoniPizza();
    }
   pizza.prepare();
   pizza.bake();
   pizza.cut();
   pizza.box()
}
```

변동이 있을 부분

고정된 부분

사례 #1– 객체 생성을 캡슐화

```
if (type.equals("cheese")) {
   pizza = new CheesePizza();
} else if type.equals("greek")) {
pizza = new GreekPizza();
} else if type.equals("pepperoni")) {
pizza = new PepperoniPizza();
}
```

SimplePizzaFactory 클래스가 담당

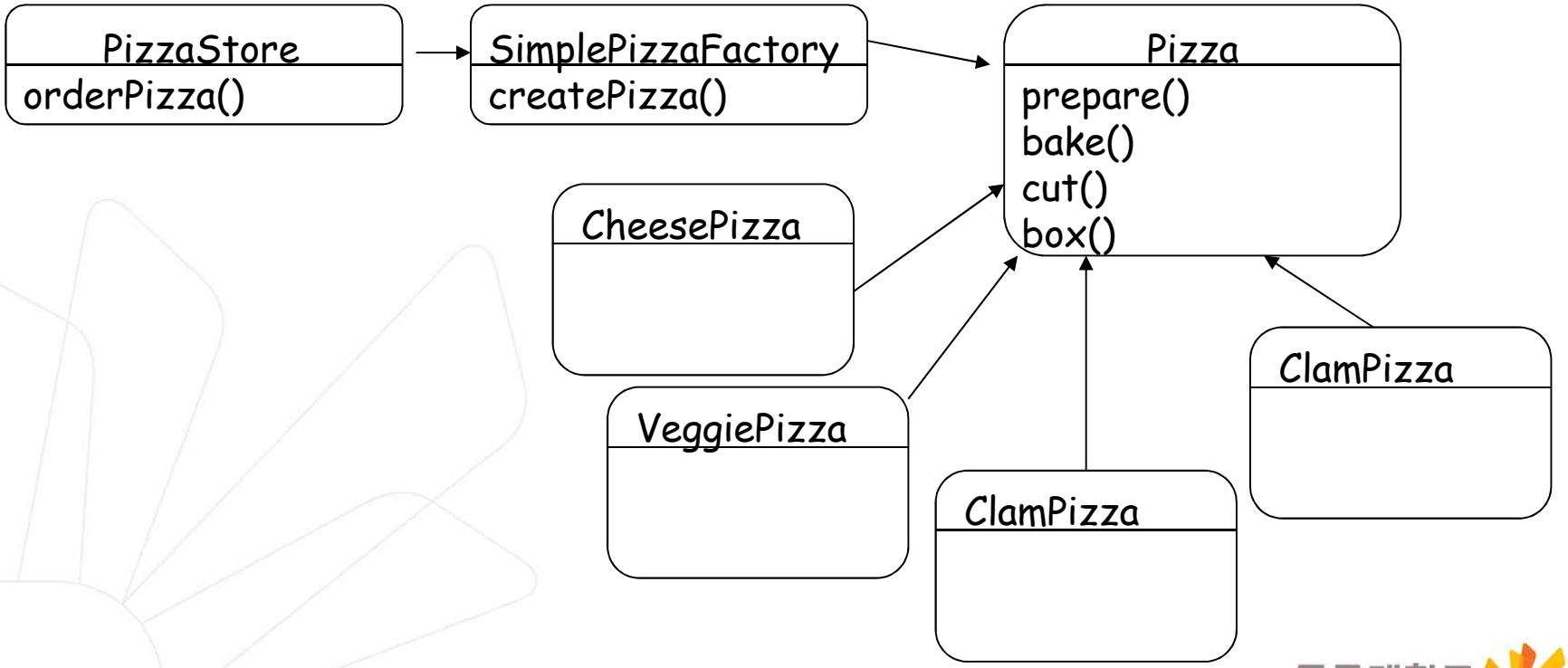
```
public class SimplePizzaFactory {
    public Pizza createPizza(String type) {
        Pizza pizza = null;
        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }
        return pizza;
    }
}
```

사례 #1 – PizzaStore 클래스 수정

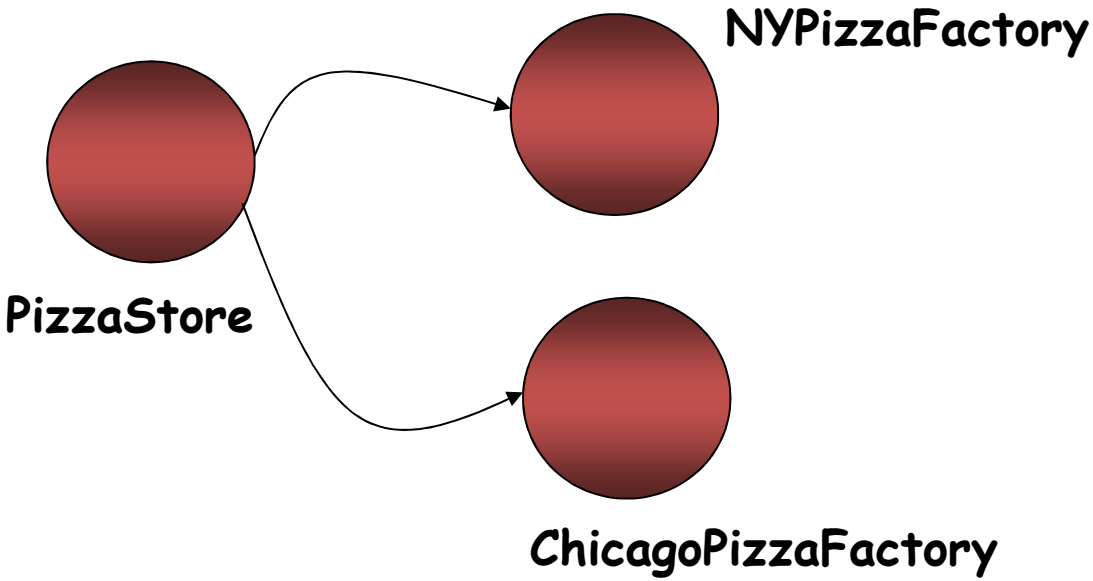
```
public class PizzaStore {
    SimplePizzaFactory factory;
    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

SimpleFactory 전체 사례

```
SimplePizzaFactory factory = new SimplePizzaFactory();
PizzaStore store = new PizzaStore(factory);
Pizza pizza = store.orderPizza("cheese");
```



여러 Factory 생성



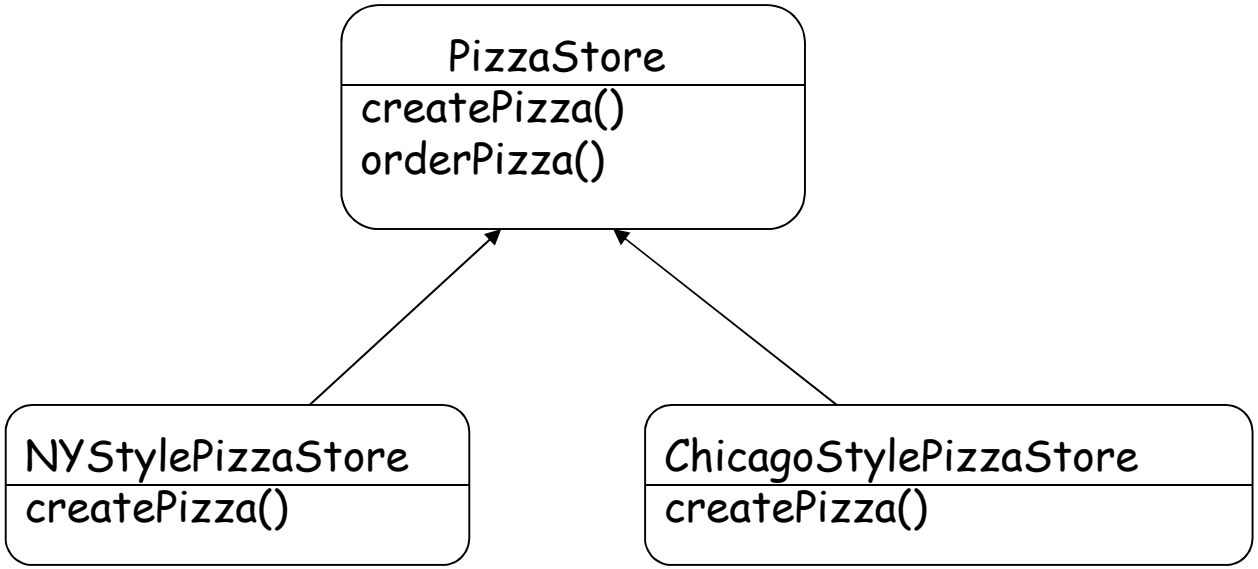

```
NYPizzaFactory nyFactory = new NYPizzaFactory();
PizzaStore nyStore = new PizzaStore(nyFactory);
Pizza pizza = nyStore.orderPizza("cheese");

ChicagoPizzaFactory chicagoFactory = new ChicagoPizzaFactory();
PizzaStore chicagoStore = new PizzaStore(chicagoFactory);
Pizza pizza = chicagoStore.orderPizza("cheese");
```

다른 방법 – 추상 메소드

- PizzaStore를 위한 프레임워크

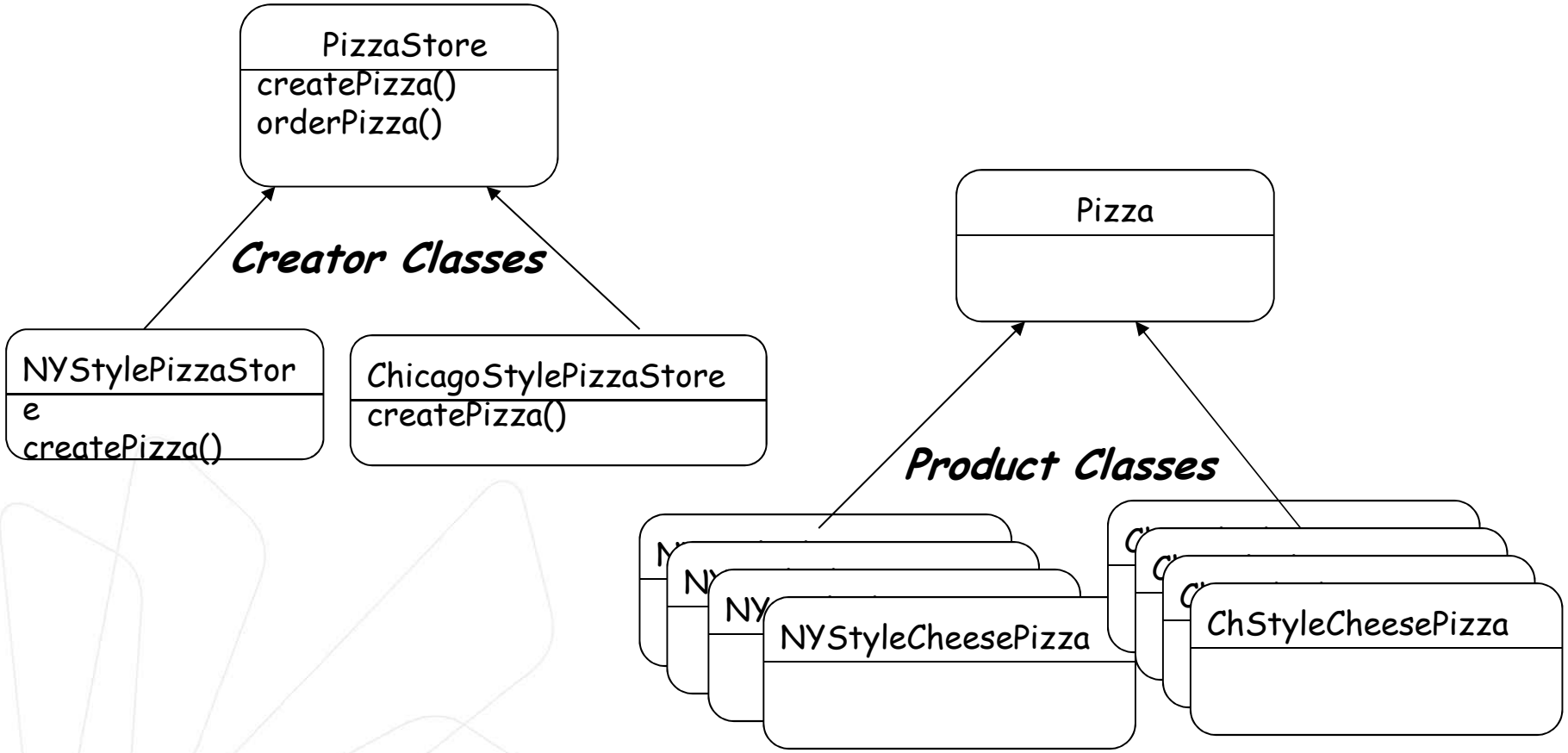
```
public abstract class PizzaStore {
  abstract Pizza createPizza(String item);

  public Pizza orderPizza(String type) {
      Pizza pizza = createPizza(type);
      pizza.prepare();
      pizza.bake();
      pizza.cut();
      pizza.box();
      return pizza;
   }
}
```

서브클래스를 정할 수 있게



- 팩토리 메소드는 객체 생성을 다루며 이를 서브클래스에 캡슐화 함
- 슈퍼 클래스의 클라이언트 코드에서 서브 클래스에 대하여 발생하는 객체생성을 떼내 올(decoupled) 수 있음

팩토리 메소드 패턴



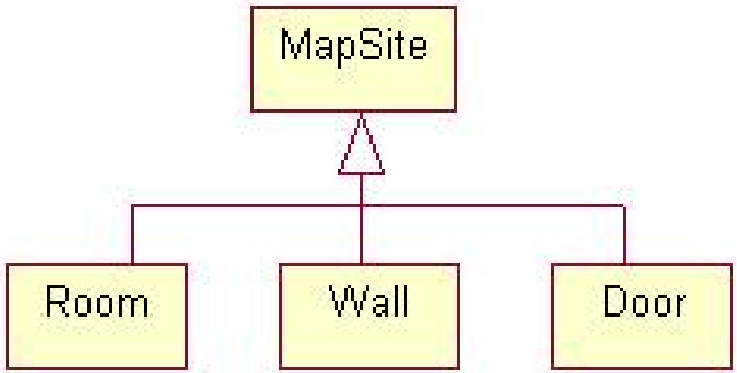
팩토리 메소드 패턴 정의

- 팩토리 메소드 패턴은 객체를 생성하기 위한 인터페이스를 정의한 것
- 어떤 클래스가 인스턴스로 될지는 서브클래스가 결정하게
- 팩토리 메소드는 서브클래스에게 인스턴스화를 연기시킨 것

사례 #2 : 미로 찾기

- 예제:컴퓨터 게임 미로찾기 만들기
  - 단순 미로 찾기
  - 미로(maze)는 방(room)과 벽(wall)과 문(door)으로 구성
  - room은 동서남북 각 방향으로 wall이나 door가 연결되어 있음.
  - room과 room간의 연결은 하나의 door로 연결
  - maze는 복수 개의 room으로 구성
- 클래스 다이어그램
  - 클래스: Maze, Room, Wall, Door,

    → Room, Wall, Door 를 미로의 공통 구성요소로 간주하여 MapSite 로 추상화

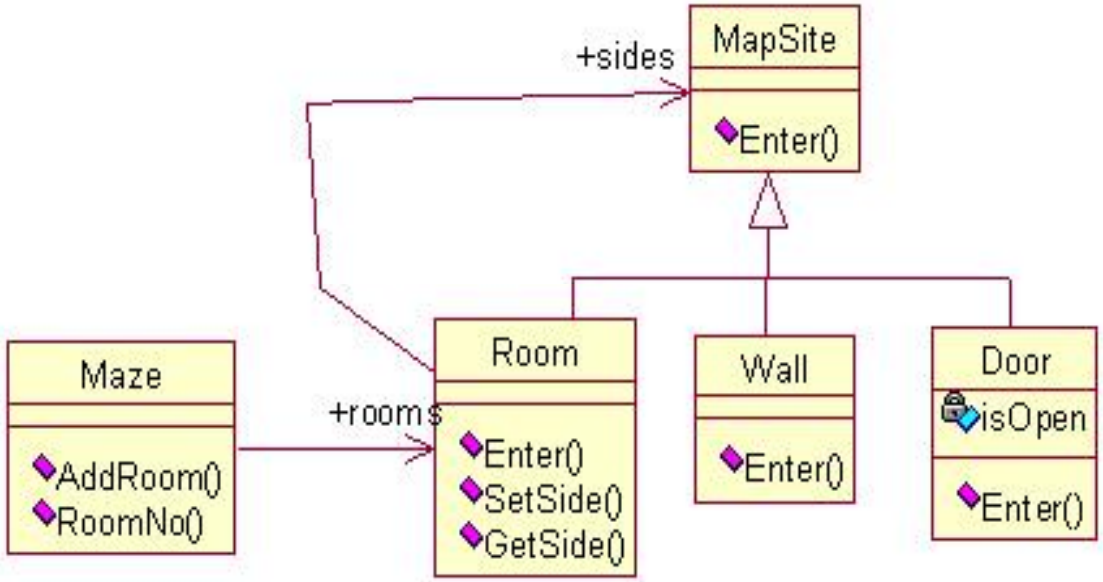


패턴 미적용 사례

- MapSite의 메소드
  - Enter(): Player의 위치를 변경.
  - 만일 MapSite가 Room 객체라면 Player가 있는 방을 변경
  - 만일 MapSite 가 Door 객체라면 Door가 열려져 있는 경우엔 다른 Room으로 이동할 수 있지만, 잠겨있을 경우엔 이동할 수 없음.
    → *Door 가 열려져 있는지 여부를 표현하기 위한 속성 필요*
- Room의 메소드
  - 동서남북 각 방향으로 Wall이나 Door를 지정하기 위한 Method 필요.
    → *SetSide(), GetSide()*
- Maze의 메소드
  - Room을 추가하기 위한 Method 필요
    → *AddRoom()*

패턴 미적용 사례

- Maze를 위한 클래스 다이어그램



패턴 미적용 사례 - 코드

```
class MapSite {
public:
    virtual void Enter() = 0;
};

class Room : public MapSite {
public:
    Room(int = 0);
    Room(const Room&);

    virtual Room* Clone() const;
    void InitializeRoomNo(int);

    MapSite* GetSide(Direction);
    void SetSide(Direction, MapSite

    virtual void Enter();
private:
    MapSite* _sides[4];
    int _roomNumber;
};
```

```
class Wall : public MapSite {
public:

    Wall();
    Wall(const Wall&);
    virtual Wall* Clone() const;
    virtual void Enter();
};

class Door : public MapSite {
public:
    Door(Room* = 0, Room* = 0);
    Door(const Room&);

    virtual Door* Clone() const;
    void Initialize(Room*, Room*);

    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
    bool _isOpen;
};
```

```
class Maze {
public:
    Maze();
    Maze(const Maze&);
    Room* RoomNo(int);
    void AddRoom(Room*);

    virtual Maze* Clone() const;
private:
    // ...
};
```

패턴 미적용 사례 - 코드

```
Maze* MazeGame::CreateMaze() {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door *theDoor = new Door(r1, r2);

    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);

    r1->SetSide(North, new Wall);
    r1->SetSide(East, theDoor);
    r1->SetSide(South, new Wall);
    r1->SetSide(West, new Wall);
    r2->SetSide(North, new Wall);
    r2->SetSide(East, new Wall);
    r2->SetSide(South, new Wall);
    r2->SetSide(West, theDoor);

    return aMaze;
}
```

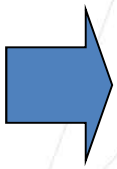
- **MazeGame::CreateMaze()**
  - **Maze 생성 (aMaze)**
  - **Room을 두 개 생성 (r1, r2)**
  - **Door을 하나 생성 (theDoor)**
  - **aMaze에 r1, r2 추가**
  - **r1은 동쪽에 theDoor를 연결하고 나머지 방향에는 Wall을 연결**
  - **r2는 서쪽에 theDoor를 연결하고 나머지 방향에는 Wall을 연결**

패턴 미적용 사례 - 문제점

- 문제점:
  - 미로의 구조가 하드 코딩 되어 있기 때문에 미로 구조가 변경될 때마다 코드 (MazeGame::CreateMaze())를 수정하여야 한다.
  - 기존의 미로를 다음의 요소들이 추가되는 "마법의 미로" 로 바꾸려 한다.
    - 주문을 외면 열리는 문 (DoorNeedingSpell)
    - 특별 아이템이 숨어있는 방 (EnchantedRoom)

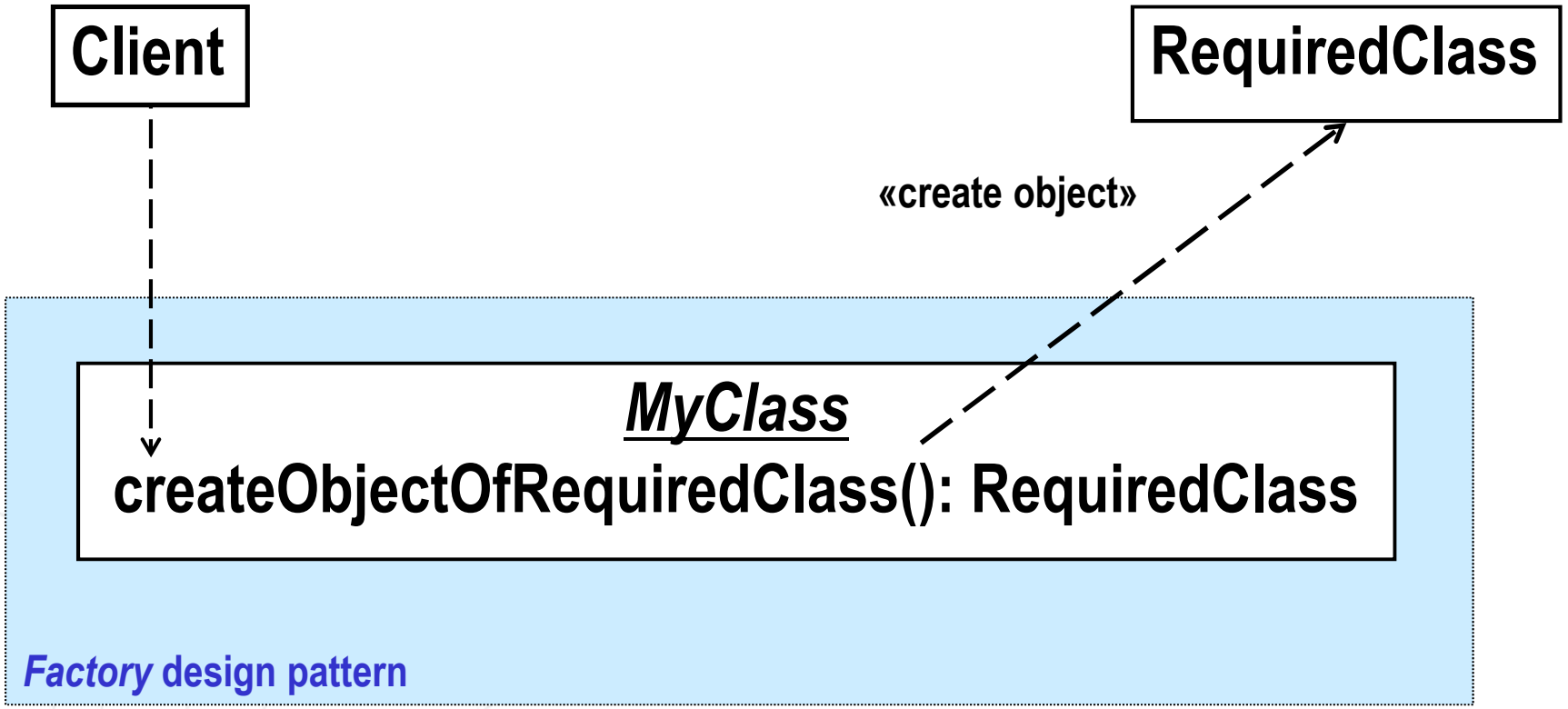
  → *어떤 코드들이 변경되어야 하는가?*

  → *쉽게 변경할 수 있는가?*

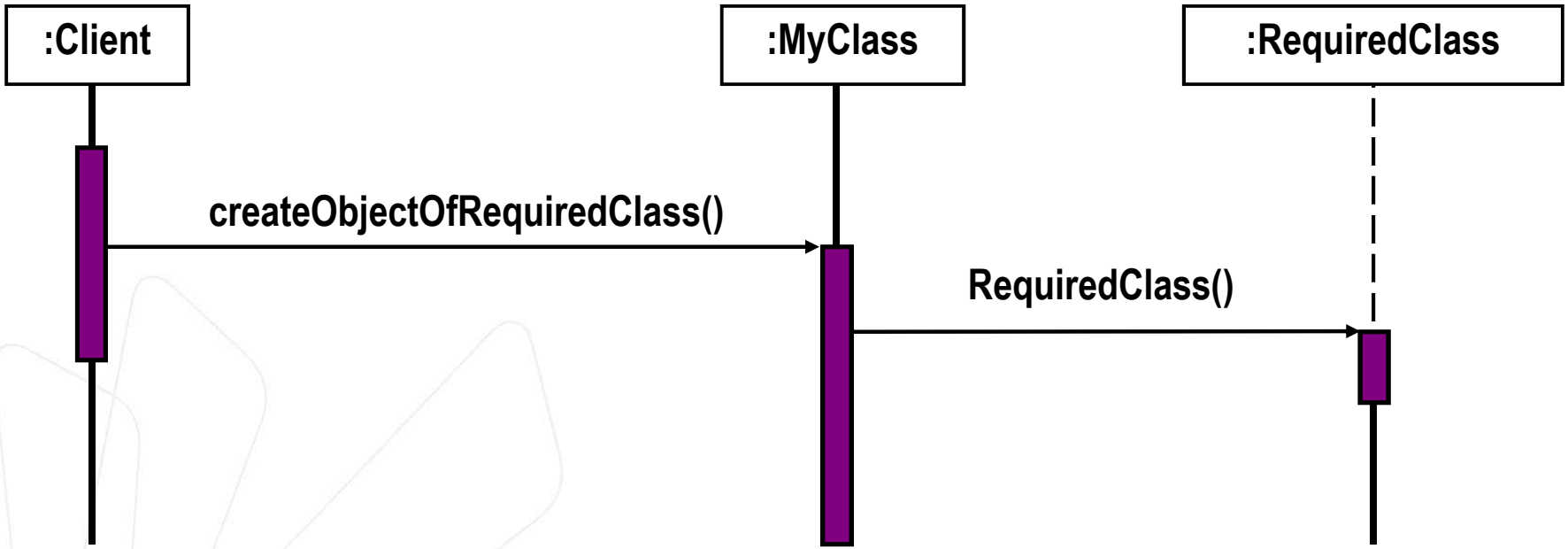


- 객체들의 생성과 그 과정이 하드 코딩 되어 있기 때문에 변화에 대처하기 힘들다.
- Creational Patterns을 이용함으로써 이러한 문제점을 해결할 수 있다.

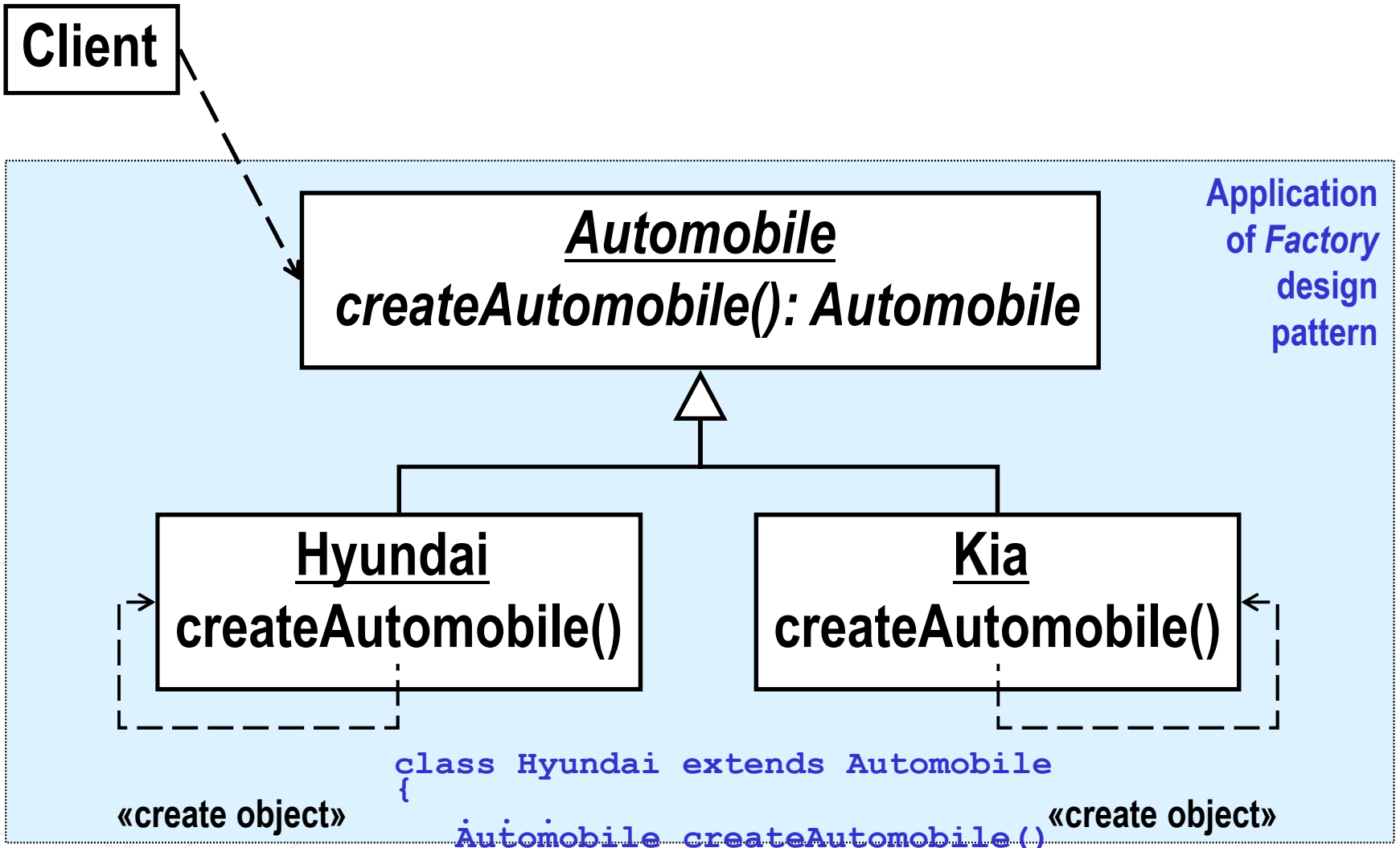
팩토리 패턴의 개념(정적 모형)



팩토리 패턴의 개념(동적 모형)



팩토리 패턴의 간단한 예제




```
class Hyundai extends Automobile
{
    . . .
    Automobile createAutomobile()
    {
        return new Hyundai();
    }
    ...
}
```

팩토리 패턴의 예: 메일 생성 시스템

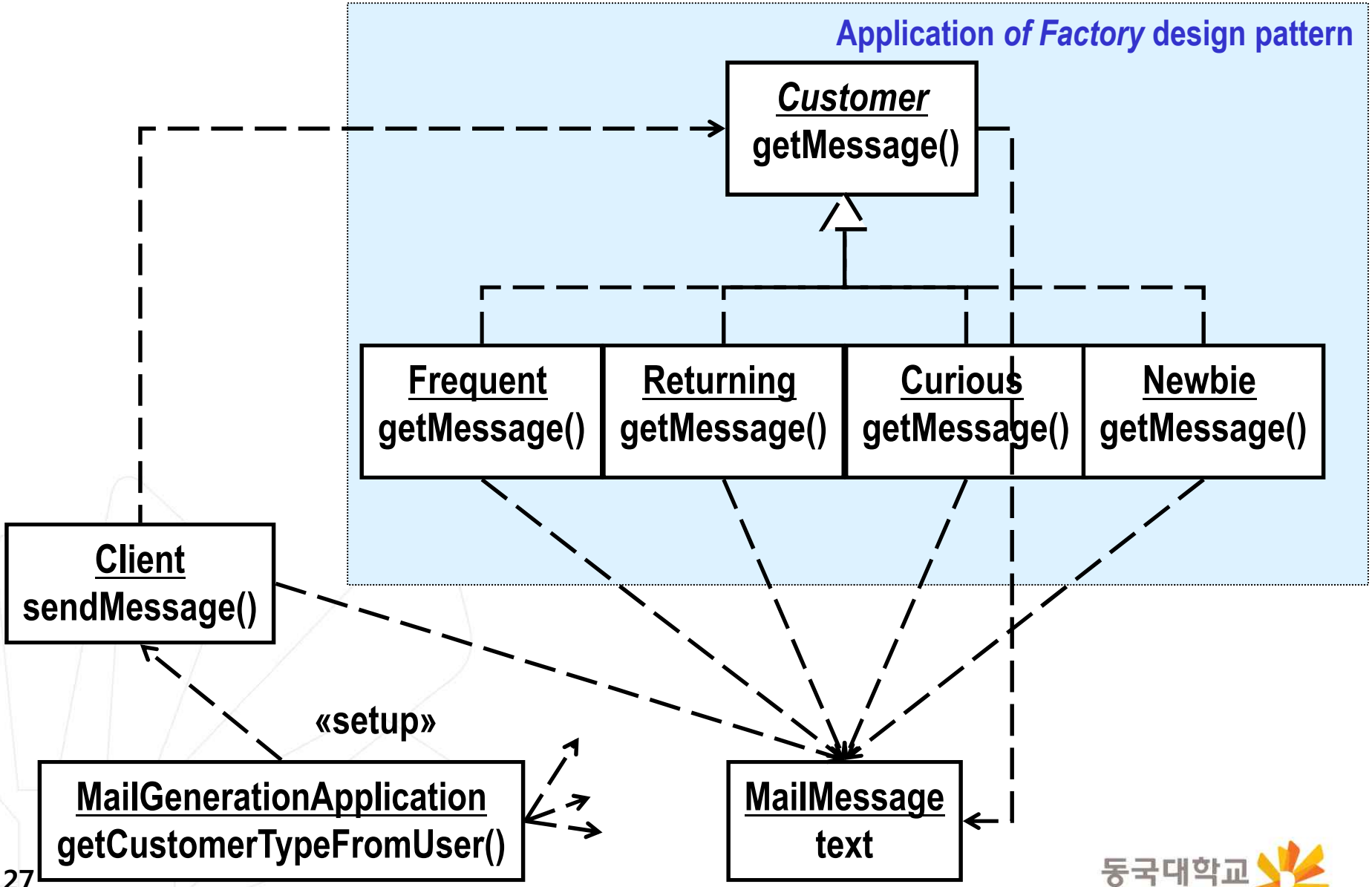
- E-mail 생성 시스템
  - 다양한 고객에 맞추어진 메일 메시지를 생성하는 시스템

```
Please pick a type of customer from one of the following:
curious
returning
frequent
newbie
returning
This message will be sent:

Losts of material intended for all customers ...
... a (possibly long) message for returning customers ...
```

사례 #3: 메일 시스템

1. 메일 생성 시스템은 고객의 이메일 주소를 요청한다.
2. 사용자는 고객의 이메일 주소를 입력한다.
3. 메일 생성 시스템은 고객 타입의 목록을 보여주며, 어떤 타입의 고객 메일 메시지인지를 묻는다.
4. 사용자는 의도한 고객 타입을 입력한다.
5. 메일 생성 시스템은 고객의 타입을 모니터에 다시 출력해준다.
6. 메일 생성 시스템은 모든 고객에게 보내지는 부분과 사용자에 의해 요구에 맞춰진 메일로 구성된 메일 메시지를 모니터에 출력한다.
7. 메일 생성 시스템은 메시지를 보낸다.

이메일 생성 시스템



클라이언트 클래스

```
class Client
{
    public Client() {
       super();
    }

    public void sendMessage(Customer aCustomer) {
        MailMessage mailMessage = aCustomer.getMessage();

      System.out.println("This message will be sent:");
        System.out.println(mailMessage.getText());
    }
}

abstract class Customer
{
    public MailMessage getMessage()
    {   return new MailMessage( "Losts of material intended for all    customers
   ..." );
    }
}
```

애플리케이션

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

class MailGenerationApplication {
   private static Client client = new Client();
      public MailGenerationApplication() {
        super();
      }

   public static void main(String[ ] args) {
      Customer customer =
   getCustomerTypeFromUser();
        MailGenerationApplication.client.sendMessage(cu
   stomer); // let the client do its job
      }

      private static Customer
   getCustomerTypeFromUser() {
        String customerType = "newbie"; // default
        Hashtable customerTypeTable = new
   Hashtable(); // keys strings to Customer types
        customerTypeTable.put("frequent", new
   Frequent());
        customerTypeTable.put("returning", new
   Returning());
        customerTypeTable.put("curious", new Curious());
        customerTypeTable.put("newbie", new Newbie());

        System.out.println(
           "Please pick a type of customer from one of the
   following:");
        for (Enumeration enumeration =
   customerTypeTable.keys();
             enumeration.hasMoreElements(); ) {
          System.out.println(enumeration.nextElement());
     }

        try { // pick up user's input
          BufferedReader bufReader =
             new BufferedReader(new
   InputStreamReader(System.in));
          customerType = bufReader.readLine();
        }
        catch (IOException e) {
          System.out.println(e);
        }
```

애플리케이션

```
        Customer customerSelected = (Customer)  customerTypeTable.get(customerType);
        if (customerSelected != null)
            return customerSelected;
        else { // use default if user input bad
            System.out.println("Sorry: Could not understand your input: newbie customer
    assumed.");
         return new Newbie();
       }
    } // end getCustomerTypeFromUser()
}
```

팩토리 패턴이 적용된 코드

```
class MailMessage
{
    String text = "No text chosen yet";
    public MailMessage()
    {    super();
    }
    public MailMessage( String aText )
    {
        this();
        text = aText;
    }
    public String getText()
    {    return text;
    }
    public void setText( String aString )
    {    text = aString;
    }
}
```

```
class Frequent extends Customer
{
     public Frequent()
     {    super();
     }
     public MailMessage getMessage()
     {
          String frequentCustomerMessage = "...
    a (possibly long) message for
               frequent customers ...";
          String messageTextForAllCustomers =
    ( super.getMessage() ).getText();
                    return new MailMessage( "\n" +
    messageTextForAllCustomers + "\n" +
               frequentCustomerMessage );
     }
}
```

팩토리 패턴 사례: 미로 게임

- **Sample Code**
  - **Slide 7,8에 있는 Maze 소스코드의 경우**
    - **미로의 구성요소(Room, Door, Wall) 생성이 hard-coding 되어 있다.**

```
Maze* MazeGame::CreateMaze() {
    Maze* aMaze = new Maze;
    Room* r1 = new Room(1);
    Room* r2 = new Room(2);
    Door *theDoor = new Door(r1, r2);

    …
    return aMaze;
}
```

Factory Method pattern을 적용한다면?

팩토리 패턴 사례: 미로 게임

```
class MazeGame {
public:
    Maze* CreateMaze();

    // factory methods:
    virtual Maze* MakeMaze() const
        { return new Maze; }
    virtual Room* MakeRoom(int n) const
        { return new Room(n); }
    virtual Wall* MakeWall() const
        { return new Wall; }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new Door(r1, r2); }
};
```

override될 factory method

factory method를 이용하여 객체 생성

```
Maze* MazeGame::CreateMaze() {
    Maze* aMaze = MakeMaze();
    Room* r1 = MakeRoom(1);
    Room* r2 = MakeRoom(2);
    Door* theDoor = MakeDoor(r1, r2);
    aMaze->AddRoom(r1);
    aMaze->AddRoom(r2);
    r1->SetSide(North, MakeWall());
    r1->SetSide(East, theDoor);
    r1->SetSide(South, MakeWall());
    r1->SetSide(West, MakeWall());
    r2->SetSide(North, MakeWall());
    r2->SetSide(East, MakeWall());
    r2->SetSide(South, MakeWall());
    r2->SetSide(West, theDoor);
    return aMaze;
}
```

팩토리 패턴 사례: 미로 게임

```
class BombedMazeGame : public MazeGame {
public:
    BombedMazeGame();
    virtual Wall* MakeWall() const
        { return new BombedWall; }
    virtual Room* MakeRoom(int n) const
        { return new RoomWithABomb(n); }
};
```

factory method

```
class EnchantedMazeGame : public MazeGame {
public:
    EnchantedMazeGame();
    virtual Room* MakeRoom(int n) const
        { return new EnchantedRoom(n, CastSpell()); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
        { return new DoorNeedingSpell(r1, r2); }
protected:
    Spell* CastSpell() const;
};
```

// Client Code

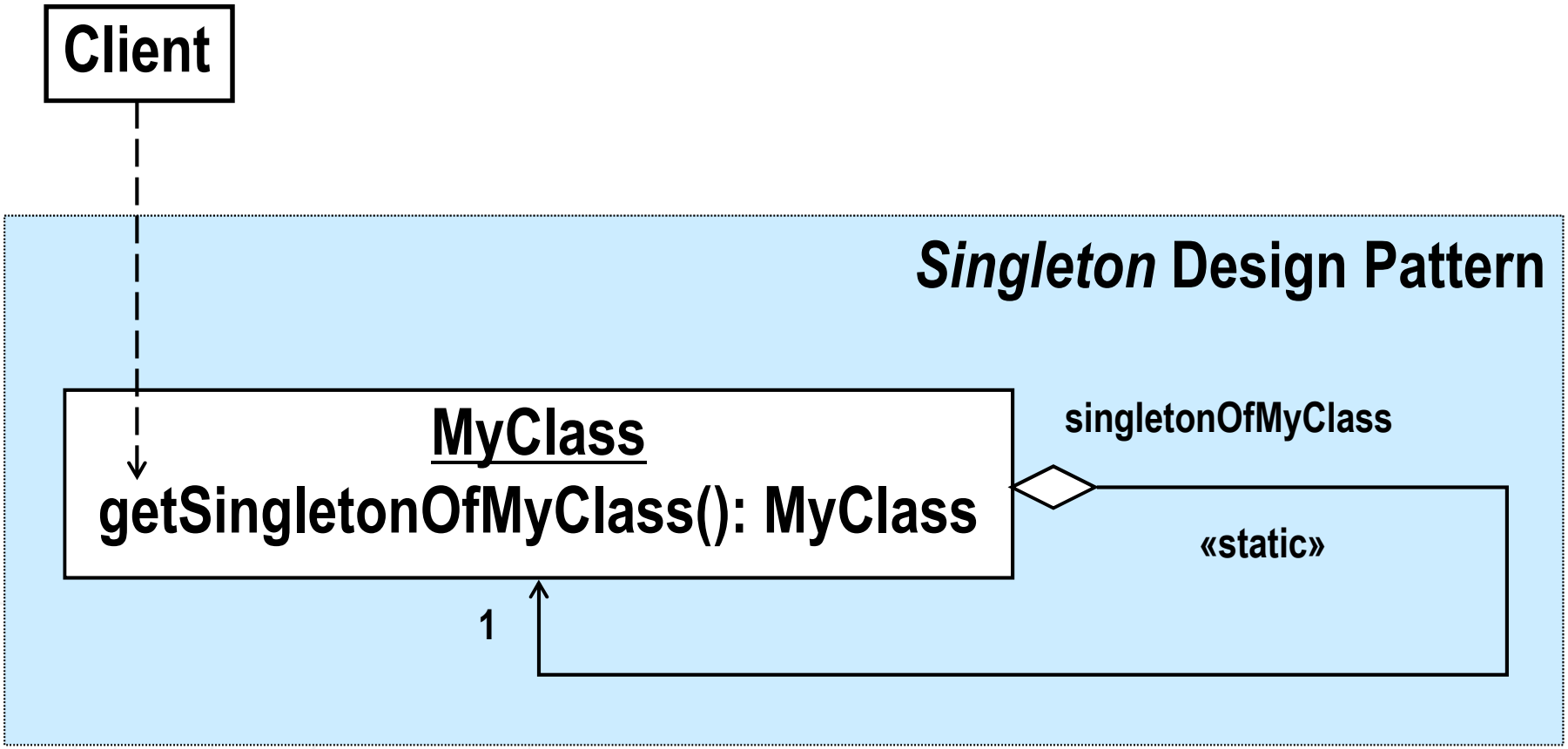
```
BombedMazeGame game1;
EnchantedMazeGame game2;

game1.CreateMaze();
game2.CreateMaze();
```

Pattern #2 – 싱글톤 패턴

어떤 클래스, S의 인스턴스를 단 하나만 만들고 싶을 때.
어플리케이션 전체에 꼭 하나만 필요한 경우.

- 패턴의 핵심
  - S의 생성자를 private으로 만들고, S 안에 private 정적 속성을 정의한다. 이를 접근하는 public 함수를 제공한다.
  - 싱글톤은 오직 한 개의 객체만 존재하려는 목적이 있어 더 이상 만들려는 생성자의 호출을 안전하게 막아야 한다.

싱글톤 패턴 – 클래스 다이어그램



싱글톤 패턴 – MyClass

1. MyClass 클래스 안에 MyClass라는 이름의 private 정적 변수를 선언한다.
   - private static MyClass singletonOfMyClass = new MyClass();

2. MyClass의 생성자를 private으로 만든다.
   - private MyClass() { /* .... constructor code .... */ };

3. 멤버를 접근하는 정적 public 메소드를 정의한다.

```
public static MyClass getSingletonOfMyClass()
{
    return singletonOfMyClass;
}
```

사례 #1 – 보고서 문제

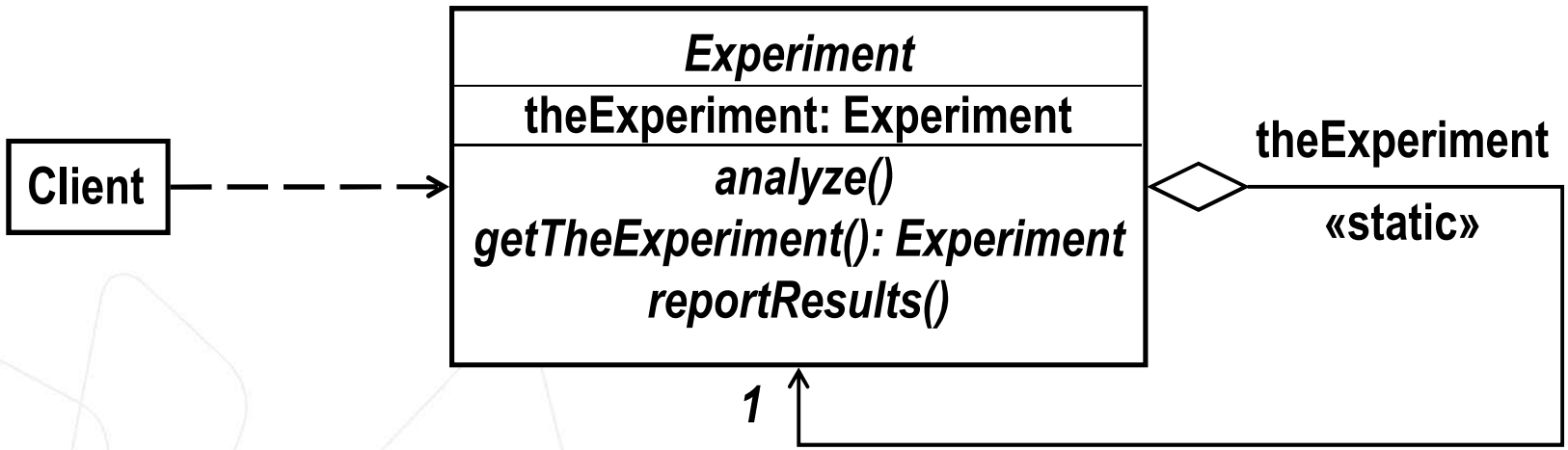
- 연구실의 실험 결과 평가 어플리케이션
  - 정확히 하나의 experiment 객체만이 실시간에 존재함을 보장하여야 함
  - 어플리케이션에서 experiment를 접근할 때 다음과 같이 표시

Output

```
Noting that the Experiment singleton referenced 1 times so far
The analysis shows that the experiment was a resounding success. . . .
```

사례 #1 – 보고서 문제

- 클래스 다이어그램



사례 #1 – 코드

```
class Client
{
    public Client()
    {    super();
    }
    public static void main( String[] args )
    {
        Experiment experiment =
    Experiment.getTheExperiment();
        experiment.analyze();
        experiment.reportResults();
    }
}
```

```
class Experiment
{
        private static final Experiment theExperiment
    = new Experiment();
          String result = "Experiment result not yet
    assigned";   // result of the experiment

        private static int numTimesReferenced = 0;

        private Experiment() {
          super();
        }

        public synchronized void analyze() {
          theExperiment.result =
                        "... The analysis shows that the
    experiment was a resounding success. ....";
        }
```

사례 #1 – 코드

```
    public static Experiment getTheExperiment() {
      ++numTimesReferenced;
      System.out.println
            ("Noting that the Experiment singleton
referenced " +
          numTimesReferenced + " times so far");

      return theExperiment;
    }

    public void reportResults() {
      System.out.println(result);
    }
}
```

사례 #2 – 미로 게임

- Sample Code
  - Abstract Factory pattern의 Sample Code 에서
    - MazeFactory 부분을 Singleton 클래스로 작성
    - 그러나, MazeFactory::Instance() 함수에서 실제 생성되는 Factory는 환경변수 “MAZESTYLE” 에 따라 MazeFactory, BombedMazeFactory, EnchantedMazeFactory 등이 생성되도록 구현한다.
  - 변경되는 부분
    - MazeFactory
    - Client Code

사례 #2 – 미로 게임

```
class MazeFactory {
public:
    static MazeFactory* Instance();
    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};

MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");
        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;
        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;
        // ... other possible subclasses
        } else
            _instance = new MazeFactory;
    }
    return _instance;
}
```

// Client Code

```
setenv("MAZESTYLE","bombed",1);
MazeFactory *factory = MazeFactory::Instance();
MazeGame game;

game.CreateMaze(factory);
```

환경변수 값에 따라
실제 factory 객체를 결정

Pattern #3 - 프로토타입 패턴

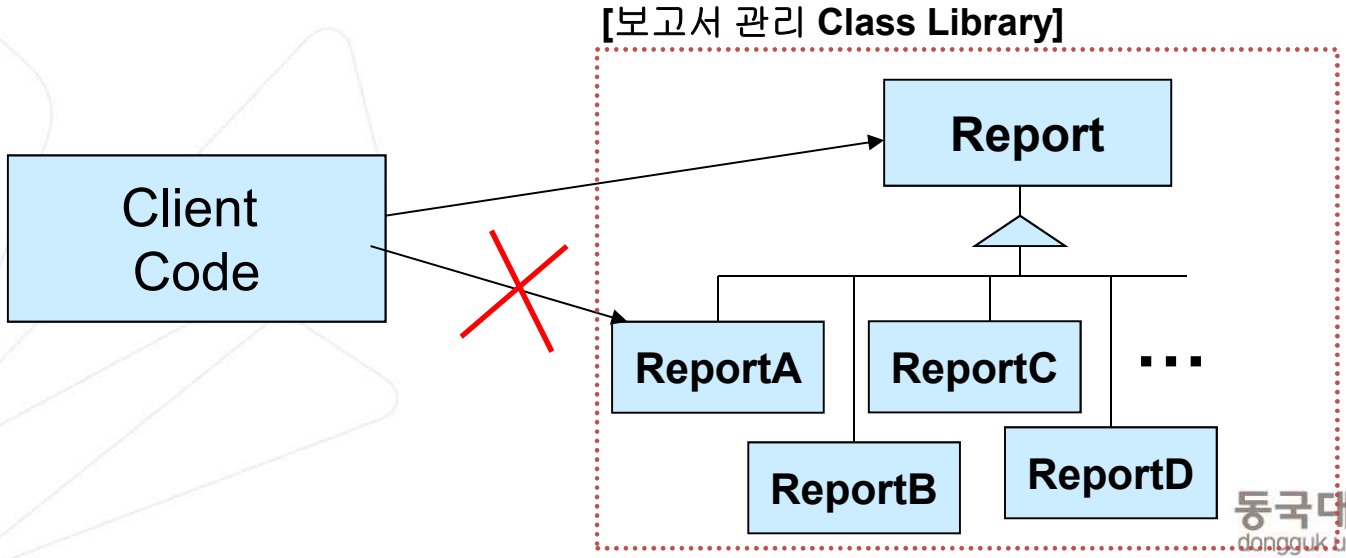
런타임에 그 타입이 결정되는 거의 동일한 객체의 집합을 만들려고 할 때 적용.

- 가정
  - 프로토타입이 될 인스턴스를 이미 알고 있어야 함
  - 새로운 인스턴스가 필요할 때 언제든지 이를 클론화
- 인스턴스를 만드는 방법
  - New Something()
  - MyPart anotherMyPart = MyPartPrototype.clone();

프로토타입 패턴: 동기

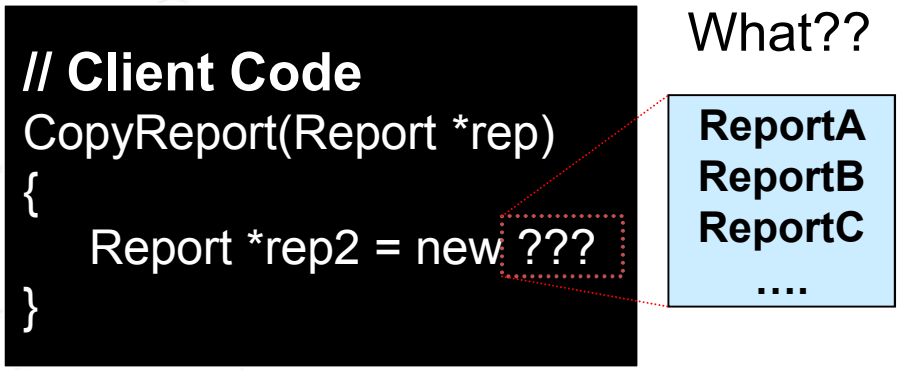
A회사는 자사의 주력 제품인 "보고서 관리 Class Library"의 upgrade 프로젝트를 결정하였으며, 그 개발 팀의 시스템 설계자로 당신이 선발되었다. upgrade는 주로 고객(library를 이용하는 개발자)의 불만 사항을 해결하는데 초점이 맞추어져 있다.

- **"보고서 관리 Class Library" 의 현재 상황**
  개발자는 실제 보고서 객체(ReportA, ReportB, ...) 를 직접 다루지 않고 모두 Report 객체로 간주하여 이용한다.
  즉, 실제 보고서 객체는 외부에서 접근하지 못한다.



프로토타입 패턴: 동기

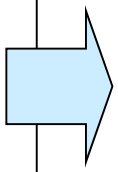
- 고객(개발자)의 불만사항
  - 특정 Report 객체를 넘겨받아 이를 이용하여 작업을 수행한다.
  - 그러나, *넘겨받은 Report 객체와 동일한 Report 객체를 생성해야 되는 경우*엔 어떻게 해야 되는가?
  - 예를 들면 현재 Client Code에서 특정 Report 객체(실제로는 ReportA 객체라고 가정)를 가지고 작업 중인데, 이 보고서의 복사본을 생성하기 위해서는 어떻게 해야 하는가?
    (Client Code는 이 객체가 ReportA임을 알 수 없다.)



프로토타입 패턴: 동기

- **불만사항 해결방안 1**

```
// Client Code
CopyReport(Report *rep)
{
    Report *rep2;
    switch ( rep->getType() )
    {
        case REPORTA:
            rep2 = new ReportA;
            break;
        case REPORTB:
            rep2 = new ReportB;
            break;
        case REPORTC:
            rep3 = new ReportC;
            break;
        …
    }
    // use rep2.
}
```



[문제점]

1. **Client Code**에게 구체적 **Report** 객체로의 접근을 허용해야 한다.
2. **Client Code**에서 모든 구체적 **Report** 객체를 알고 있어야 한다.
3. 새로운 구체적 **Report** 객체가 추가될 때 마다 **Client Code** 가 수정되어야 한다.
4. **rep2** 객체가 적합한 클래스로 생성은 되지만, **rep**의 내용을 또 다시 복사해야 한다.

프로토타입 패턴: 동기

- **불만사항 해결방안 2:**
  - **특정 객체를 복사해야 되는 경우, 객체 생성과 내용 복사기능을 해당 객체에 맡긴다.**

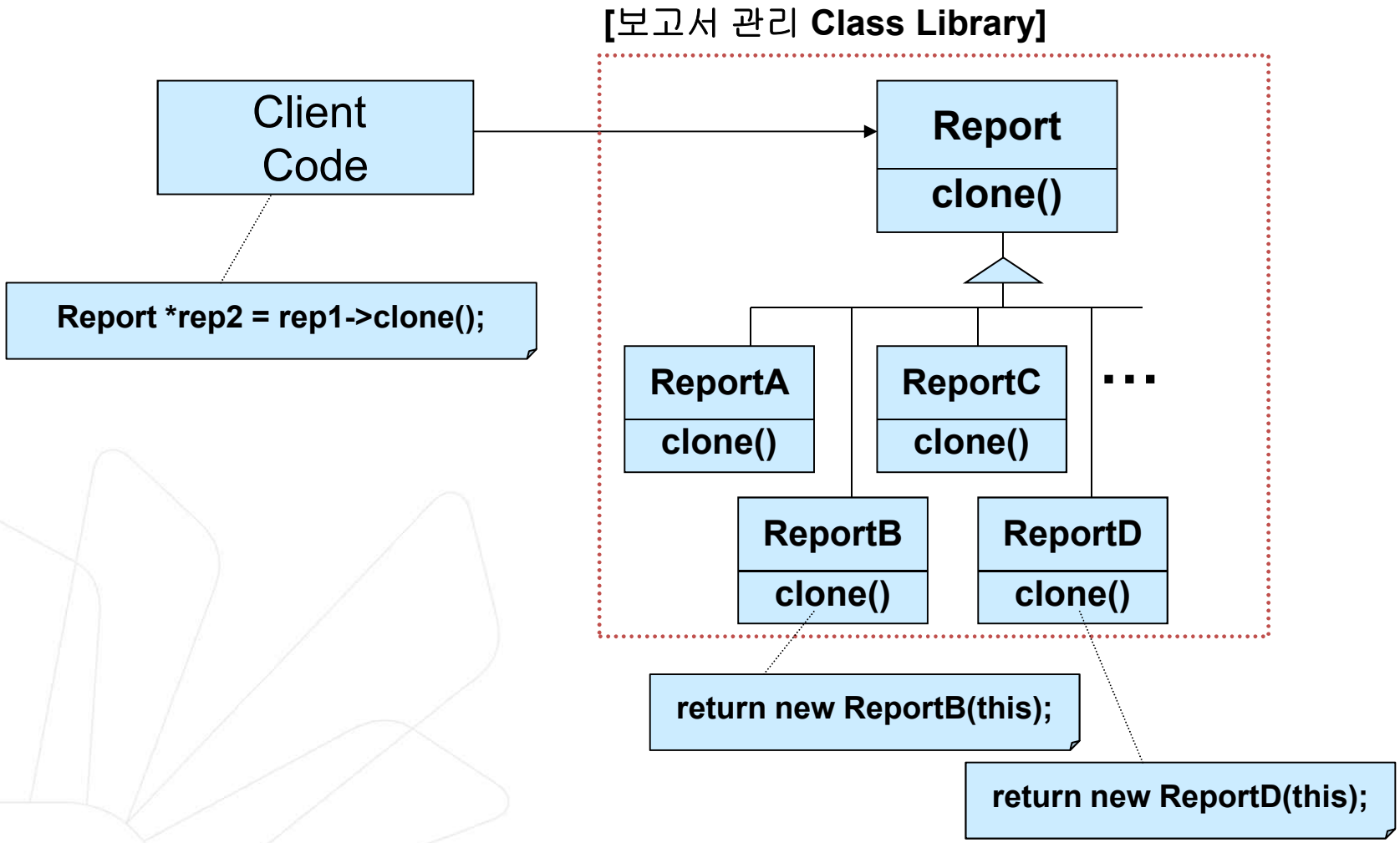
```
// Client Code
CopyReport(Report *rep)
{
    Report *rep2 = rep->clone();
    // use rep2
    …
}
```

clone함수는 실제 구체 객체와 동일한 내용을 가지는 객체를 생성해서 반환하는 기능제공

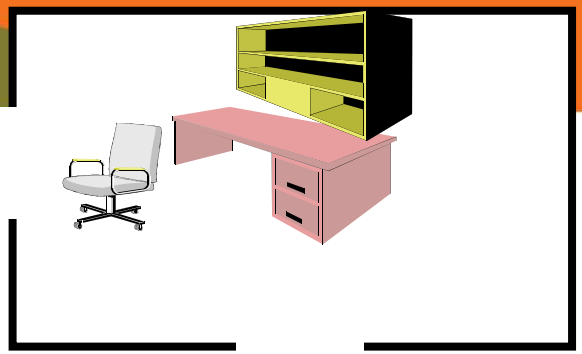
1. **Client Code**에게 구체적 **Report** 객체로의 접근을 허용하지 않아도 된다.
2. **Client Code**에서 모든 구체적 **Report** 객체를 알 필요가 없다.
3. 새로운 구체적 **Report** 객체가 추가되더라도 기존 **Client Code**가 수정될 필요는 없다.
4. **clone** 함수 내에서 **rep**의 내용의 복사작업을 수행한다.

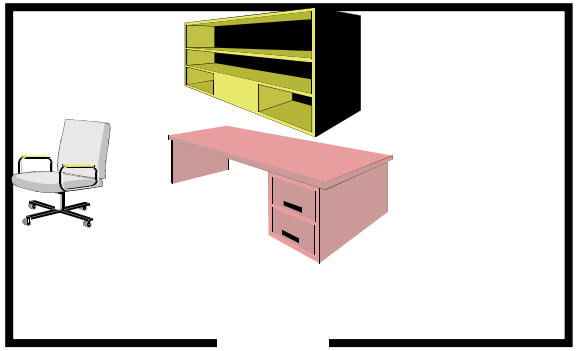
48

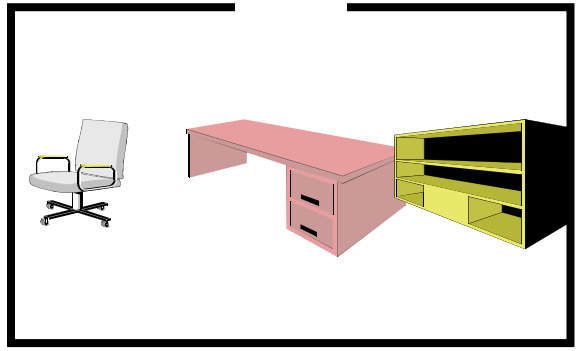
프로토타입 패턴: 동기

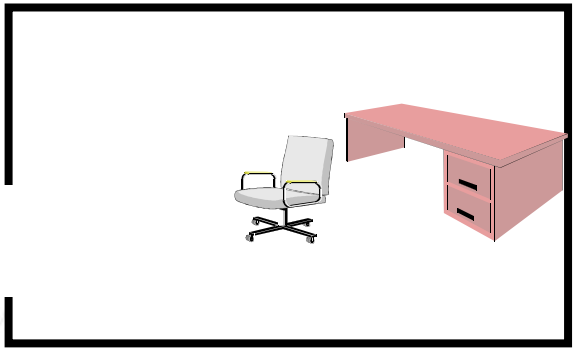


프로토타입 사례









Furniture color

Click on choice of desk:

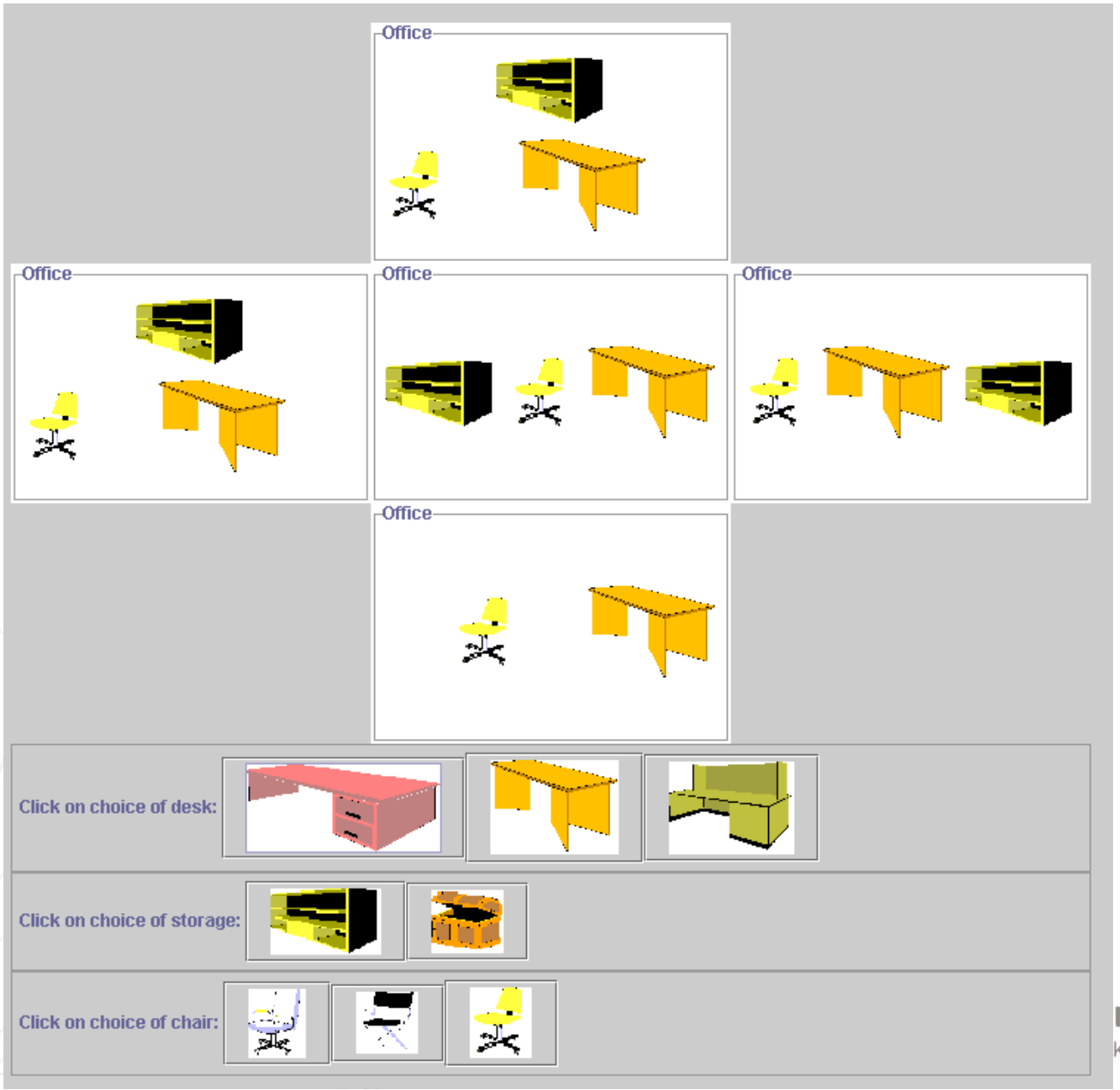
Click on choice of storage:

Click on choice of chair:

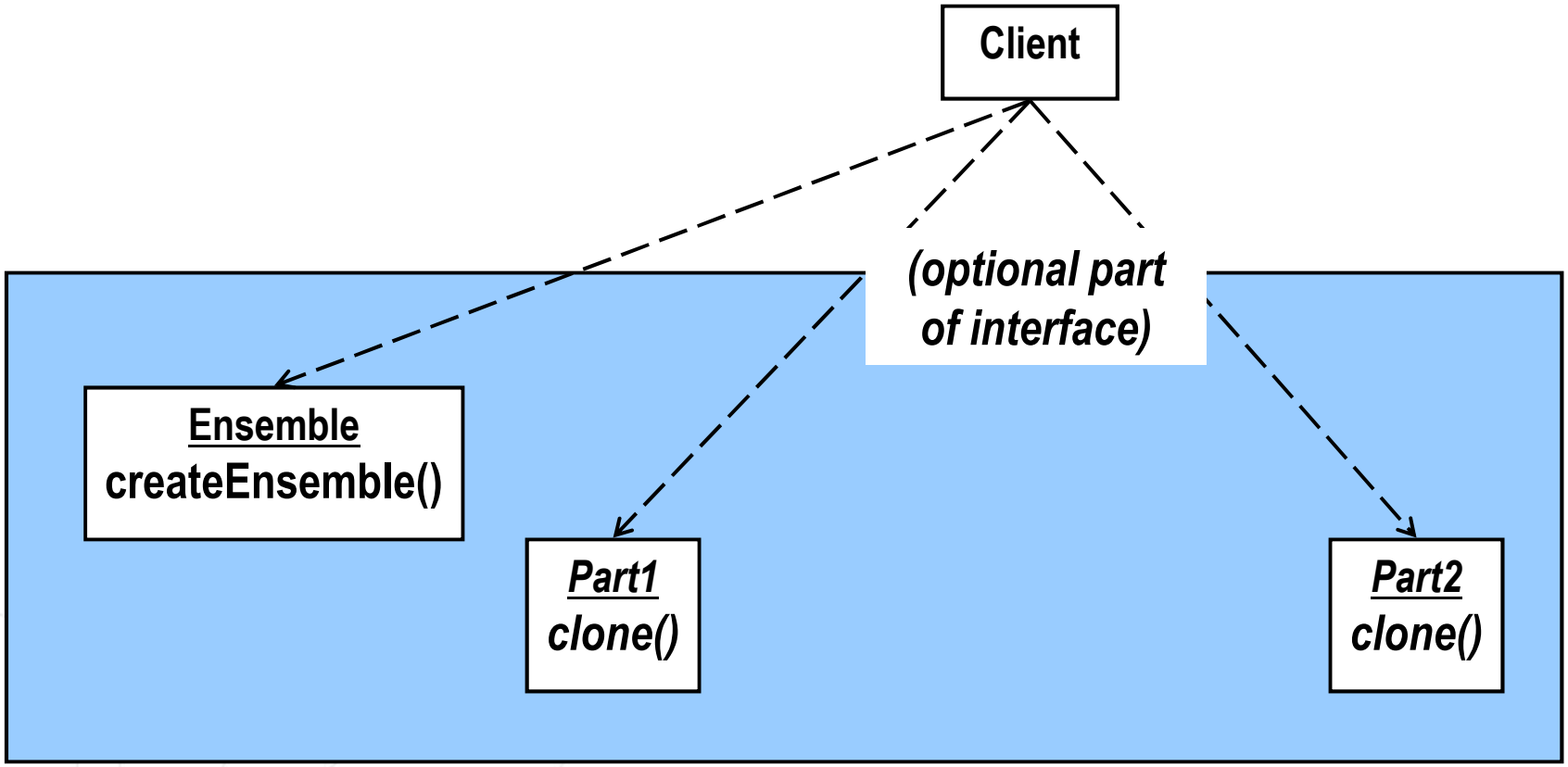
Furniture hardware type

colonial

프로토타입 사례

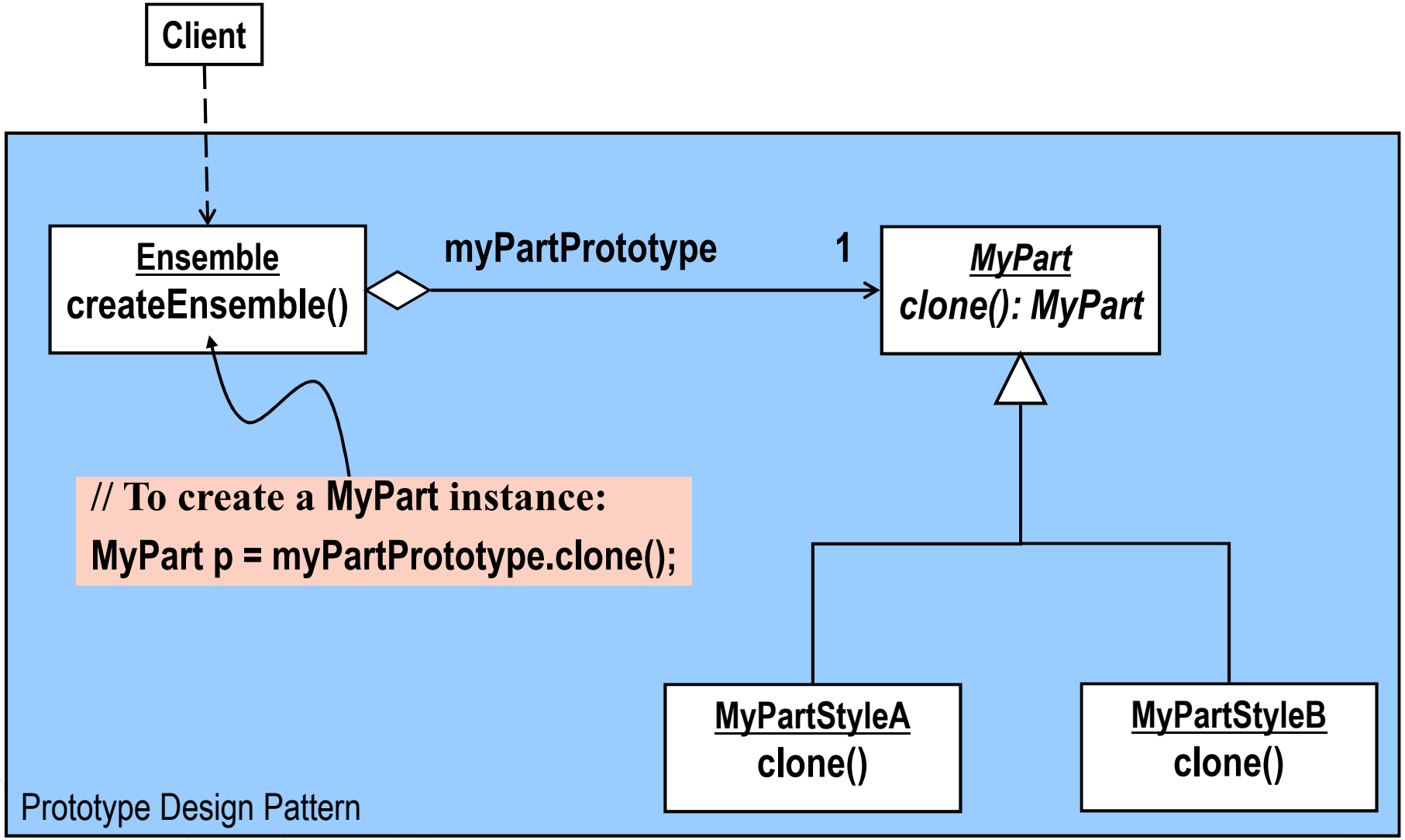


프로토타입 패턴

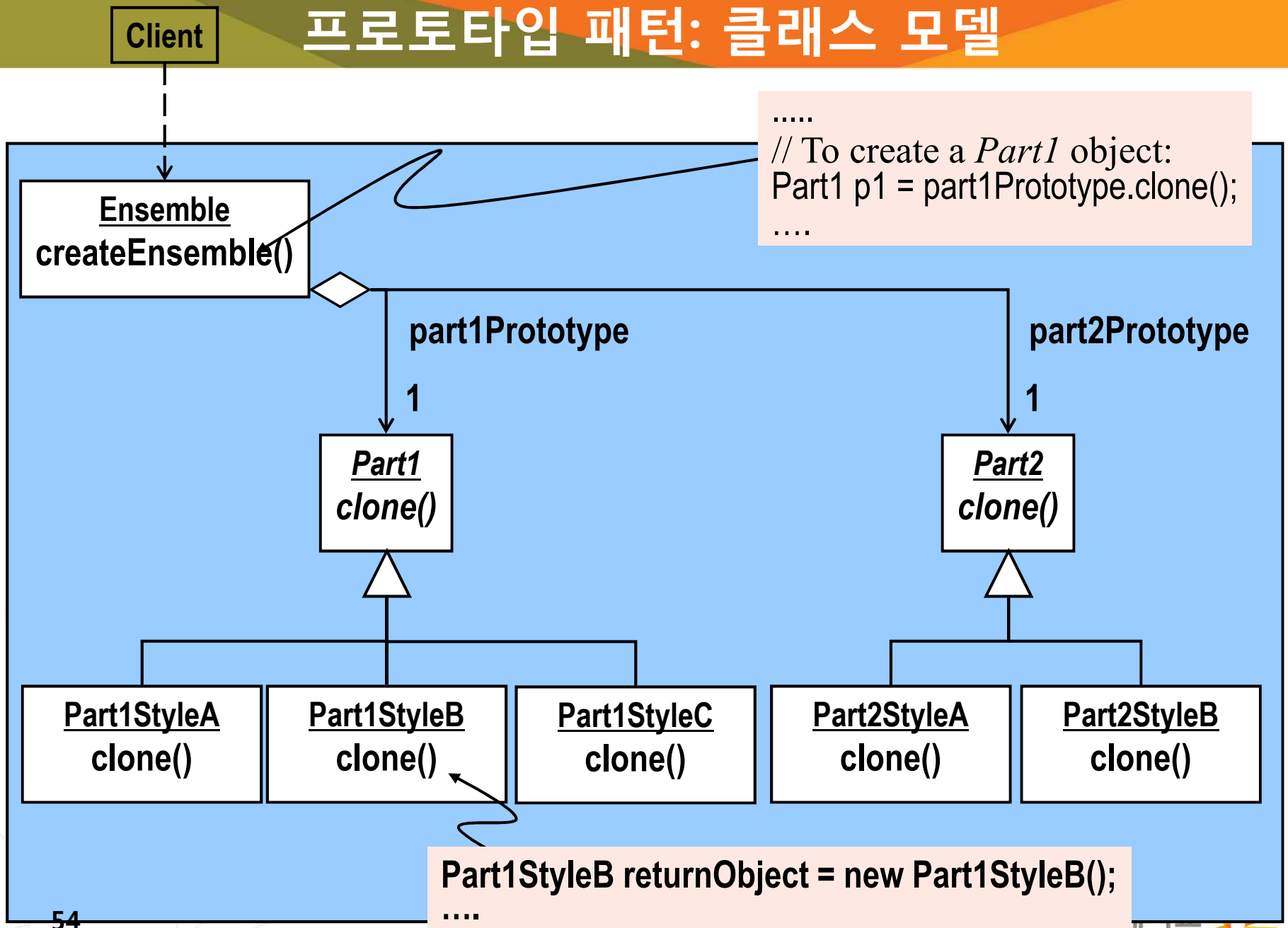



```
MyPart  anotherMyPart = myPartPrototype.clone();
MyPart  yetAnotherMyPart = MyPartPrototype.clone();
```

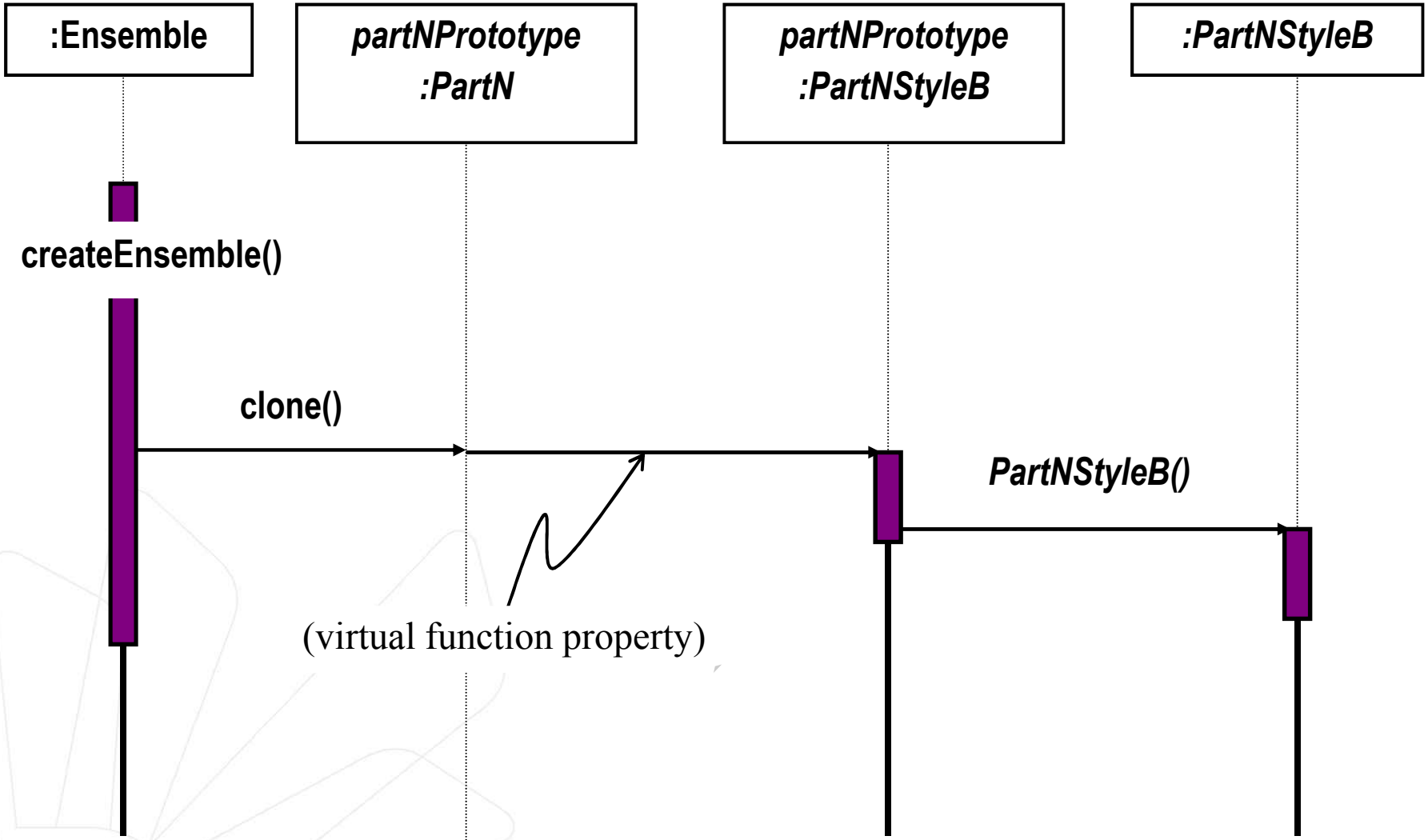
프로토타입 패턴



프로토타입 패턴: 클래스 모델

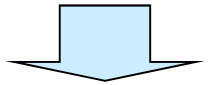


프로토타입 패턴: 동적 다이어그램



프토토타입 패턴 사례 - 미로게임

- **Sample Code**
  - **Abstract Factory pattern의 Sample Code 에서**
    - 미로 구축에 사용되는 product 객체들(room, wall, door) 을 생성하기 위하여, factory 를 사용하였다.
    - 다른 family의 product 객체를 생성하기 위하여 각 family 별로 factory 를 작성하였다. (EnchantedMazeFactory, BombedMazeFactory)



만일 Factory를 하나만 두고, 미리 지정된 prototype 객체를 이용하여 Room, Wall, Door 객체를 생성하려면?

- Factory 생성시 각 product 객체의 prototype 객체를 지정
- 각 product 객체에 Clone()함수를 구현하고, 필요시 Initialize() 함수를 구현

```
class MazePrototypeFactory : public MazeFactory {
public:
    MazePrototypeFactory(Maze*, Wall*, Room*, Door*) {
      _prototypeMaze = m;
      _prototypeWall = w;
      _prototypeRoom = r;
      _prototypeDoor = d;
    }

    virtual Maze* MakeMaze() const {
          return _prototypeMaze->Clone(); }
    virtual Room* MakeRoom(int roomNo) const {
          Room *room = _prototypeRoom->Clone();
          room->Initialize(roomNo);
          return room; }
    virtual Wall* MakeWall() const {
         return _ptototypeWall->Clone();   }
    virtual Door* MakeDoor(Room*, Room*) const {
          Door *door = _prototypeDoor->Clone();
          door->Initialize(r1,r2);
          return door;

private:
    Maze* _prototypeMaze;
    Room* _prototypeRoom;
    Wall* _prototypeWall;
    Door* _prototypeDoor;
};
```

객체 생성을 위한 prototype 객체 지정

- prototype 객체를 이용하여 Maze, Room, Wall, Door 객체 생성.
- Room & Door 의 경우 Initialize 함수를 호출하여 초기값지정

객체 생성에 사용될 prototype 객체

```
class Door : public MapSite {
public:
    Door();
    Door(const Door&){
        _room1 = other._room1;
        _room2 = other._room2;
    }
    virtual void Initialize(Room*, Room*){
        _room1 = r1;
        _room2 = r2;
    }
    virtual Door* Clone() const{
        return new Door(*this);
    }
    virtual void Enter();
    Room* OtherSideFrom(Room*);
private:
    Room* _room1;
    Room* _room2;
};

class Wall : public MapSite {
   …
};

class Room: public MapSite {
   …
};
```

clone() 후 초기값 설정함수

객체 복제를 위한 clone()함수

```
// Client Code

MazeGame game;
MazePrototypeFactory simpleMazeFactory(
    new Maze, new Wall, new Room, new Door
);

Maze* maze = game.CreateMaze(simpleMazeFactory);
```

factory에 각 객체 생성을 위한 prototype 객체 지정

프토토타입 패턴 사례 - 미로게임

```
class BombedWall : public Wall {
public:
    BombedWall();
    BombedWall(const BombedWall&): Wall(other) {
        _bomb = other._bomb;
    }
    virtual Wall* Clone() const {
        return new BombedWall(*this);
    }
    bool HasBomb() { return _bomb; }
private:
    bool _bomb;
};

class RoomWithABomb : public MapSite {
   …
};
```

```
// Client Code
MazeGame game;
MazePrototypeFactory bombedMazeFactory(
    new Maze, new BombedWall,
    new RoomWithABomb, new Door
);

Maze* maze = game.CreateMaze(bombedMazeFactory);
```

Questions?

dongguk UNIVERSITY