6.9 Mẫu thiết kế PROXY

6.9.1 Mở đầu

Trong bài này chúng ta sẽ bàn luận về một mẫu cấu trúc được gọi là mẫu Proxy. Mẫu Proxy cung cấp một sự thỏa thuận hay một chỗ để cho một đối tượng khác kiểm soát truy cập đến nó.

Mẫu Proxy có nhiều phương án khác nhau. Một số phương án quan trọng là: Remote Proxy, Virtual Proxy và Protection Proxy. Trong bài này chúng ta sẽ biết nhiều hơn về các biến thể này và chúng ta sẽ cài đặt mỗi một biến thể đó trong Java. Nhưng trước khi làm điều đó, chúng ta sẽ xem xét kỹ hơn về mẫu Proxy nói chung.

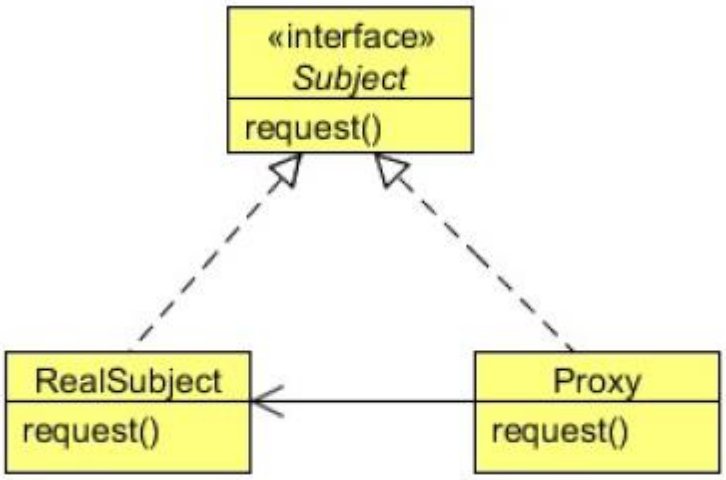
6.9.2 Mẫu Proxy là gì

Mẫu Proxy được sử dụng để tạo một đối tượng đại diện mà kiểm soát truy cập đến một đối tượng khác, mà có thể ở xa, tốn kém khi tạo hoặc cần phải được an toàn.

Một lý do để kiểm soát truy cập là trì hoãn trả giá đầy đủ để tạo và khởi tạo ra nó cho đến khi chúng ta thực sự cần sử dụng nó. Lý do khác có thể là hoạt động như một đại diện địa phương cho một đối tượng mà sống trong máy ảo Java khác. Proxy có thể rất hữu ích trong việc kiểm soát truy cập đến đối tượng gốc, đặc biệt khi các đối tượng này cần có các quyền truy cập khác nhau.

Trong mẫu Proxy, client không nói trực tiếp với đối tượng gốc, nó ủy quyền lời gọi cho đối tượng Proxy mà sẽ gọi các phương thức của đối tượng gốc. Điểm quan trọng là client không biết về proxy, proxy hoạt động như đối tượng gốc đối với client. Nhưng có nhiều biến thể cho cách tiếp cận này mà chúng ta sẽ thấy.

Bây giờ chúng ta xem cấu trúc của mẫu proxy và các thành phần quan trọng của nó.



- Proxy: 1a. Duy trì một tham chiếu mà cho phép proxy truy cập đến đối tượng thực. Proxy có thể tham chiếu đến một Subject, nếu các giao diện của RealSubject và Subject là như nhau. 1b. Cung cấp giao diện đồng nhất với Subject sao cho proxy có thể thay thế cho real subject. 1c, Kiểm soát truy cập đến real subject và có thể có trách nhiệm cho việc tạo và hủy nó.
- Subject: 2a. Xác định giao diện chung cho RealSubject và Proxy sao cho Proxy có thể được sử dụng ở bất cứ nơi nào mà RealSubject được cho phép.
- RealSubject: 3a. Xác định đối tượng thực tế mà proxy đại diện cho nó.

Có ba biến thể chính của mẫu Proxy:

- Remote Proxy cung cấp đại diện địa phương cho một đối tượng ở một không gian địa chỉ khác.
- Virtual Proxy tạo các đối tượng đắt giá theo yêu cầu.
- Protection Proxy kiểm soát truy cập đến đối tượng gốc. Protection Proxy là hữu ích khi các đối tượng có các quyền truy cập khác nhau.

Chúng ta sẽ bàn luận các proxy này trong từng mục sau.

6.9.3 Remote Proxy

Có một công ty Pizza, mà có một số cửa hàng ở các vị trí khác nhau. Người chủ công ty nhận báo cáo hàng ngày do các nhân viên của công ty từ các hàng gửi về. Ứng dụng hiện tại hỗ trợ công ty Pizza là ứng dụng trên desktop, chứ không phải ứng dụng Web. Vì vậy, chủ cửa hàng phải yêu cầu nhân viên của anh ta sinh ra báo cáo và gửi về. Nhưng bây giờ ông chủ muốn tự sinh và kiểm tra báo cáo, sao cho anh ta có thể sinh ra báo cáo bất cứ lúc nào mà không cần ai trợ giúp. Ông chủ muốn bạn phát triển ứng dụng đó cho ông.

Vấn đề ở đây là mọi ứng dụng đều chạy trên máy Java ảo (JVM) tương ứng của chúng. Các máy ảo Java và ứng dụng kiểm tra báo cáo (mà chúng ta sẽ thiết kế bây giờ) cần phải chạy trên hệ thống cục bộ của ông chủ. Đối tượng yêu cầu để sinh ra báo cáo không cần tồn tại trong JVM của hệ thống ông chủ và bạn không cần gọi trực tiếp cho đối tượng ở xa.

Remote Proxy được sử dụng để giải quyết vấn đề này. Chúng ta biết rằng báo cáo được sinh bởi người sử dụng, như vậy có đối tượng mà được yêu cầu để sinh báo cáo. Mọi điều mà chúng ta cần là liên hệ với đối tượng này mà nằm trên máy ở xa để nhận được kết quả mà ta mong muốn. Remote Proxy hoạt động như đại diện địa phương cho máy ở xa. Một đối tượng ở xa là đối tượng mà sống trên heap của máy JVM khác. Bạn gọi các phương thức đến đối tượng cục bộ và nó sẽ chuyển lời gọi đến các đối tượng ở xa.

Đối tượng client của bạn đưa ra lời gọi phương thức ở xa. Nhưng nó gọi phương thức của đối tượng proxy trên heap cục bộ mà sẽ quản lý mọi chi tiết mức thấp của truyền thông mạng.

Java hỗ trợ truyền thông giữa hai đối tượng ở trên hai JVM khác nhau sử dụng RMI. RMI là *Remote Method Invocation* – triệu gọi phương thức từ xa mà được sử dụng để xây dựng đối tượng trợ giúp cho client và dịch vụ, mà tạo đối tượng trợ giúp client với cùng phương thức như một dịch vụ từ xa. Sử dụng RMI bạn không cần tự bạn viết bất cứ điều gì về mạng hoặc code vào ra. Với client của bạn, bạn gọi phương thức ở xa như lời gọi phương thức bình thường trên đối tượng chạy trên JVM cục bộ của client.

RMI cũng cung cấp hạ tầng thực thi để làm cho nó hoạt động, bao gồm cả tìm kiếm dịch vụ mà client có thể sử dụng để tìm và truy cập đối tượng ở xa. Có một khác biệt giữa lời gọi RMI và lời gọi phương thức cục bộ. Trợ giúp của client gửi lời gọi phương thức thông qua mạng, như vậy có đường mạng và vào ra mà được sử dụng trong lời gọi RMI.

Bây giờ ta sẽ xem code. Chúng ta có giao diện *ReportGenerator* và cài đặt cụ thể của nó *ReportGeneratorImpl* đã chạy trên JVM của địa điểm khác. Trước hết để tạo dịch vụ từ xa chúng ta cần thay đổi code.

Giao diên *ReportGenerator* bây giờ trông như sau:

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ReportGenerator extends Remote{
```

```
        public String generateDailyReport() throws RemoteException;

}
```

Đây là giao diện *Remote* định nghĩa các phương thức mà *client* có thể gọi từ xa Đó là cái mà *client* sẽ sử dụng như kiểu lớp đối với dịch vụ của bạn. Cả hai *Stub* và dịch vụ thực tế sẽ cài đặt cái này. Phương thức trong giao diện trả về đối tượng kiểu String. Bạn có thể trả về bất cứ kiểu gì từ phương thức, đối tượng này sẽ được truyền qua mạng từ máy chủ dịch vụ ngược lại cho *client*, như vậy nó cần phải là *Serializable*. Cần lưu ý rằng mọi phương thức trong giao diện này cần đưa ra thông báo lỗi *RemoteException*.

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Naming;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class ReportGeneratorImpl extends UnicastRemoteObject implements ReportGenerator{

        private static final long serialVersionUID = 3107413009881629428L;

        protected ReportGeneratorImpl() throws RemoteException {
        }

        @Override
        public String generateDailyReport() throws RemoteException {
                StringBuilder sb = new StringBuilder();
                sb.append("********************Location X Daily Report********************" ↩
                    );
                sb.append("\\n Location ID: 012");
                sb.append("\\n Today's Date: "+new Date());
                sb.append("\\n Total Pizza's Sell: 112");
                sb.append("\\n Total Price: $2534");
                sb.append("\\n Net Profit: $1985");
                sb.append("\\n  ↩
                    ************************************************************");

                return sb.toString();
        }

        public static void main(String[] args) {

                try {
                        ReportGenerator reportGenerator = new ReportGeneratorImpl();
                        Naming.rebind("PizzaCoRemoteGenerator", reportGenerator);
                } catch (Exception e) {
                        e.printStackTrace();
                }

        }

}
```

Lớp trên là cài đặt từ xa mà sẽ làm việc trên thực tế. Nó là đối tượng mà *client* muốn gọi phương thức trên đó. Lớp này mở rộng *UnicastRemoteObject* để làm việc như đối tượng dịch vụ từ xa, đối tượng của bạn cần chức năng nào đó liên quan đến việc phải là *remote*. Một cách đơn giản nhất là mở rộng *UnicastRemoteObject* từ gói *Java.rmi.server* và cho phép lớp này làm việc cho bạn.

Hàm tạo của lớp *UnicastRemoteObject* đưa ra báo lỗi *RemoteException*, nên bạn cần viết hàm tạo không đối số mà khai báo *RemoteException*.

Để làm cho dịch vụ từ xa sẵn sàng cho *client*, bạn cần đăng ký dịch vụ này với *RMI registry*. Bạn làm điều đó bằng cách khởi tạo nó và đặt nó vào *RMI registry*. Khi bạn đăng ký đối tượng cài đặt, hệ thống RMI thực tế sẽ đặt nó vào *Stub* trong *registry*, đây là cái *client* cần, Phương thức *Naming.rebind* được sử dụng để đăng ký đối tượng, nó có hai tham số, cái đầu là *string* để đặt tên cho dịch vụ và tham số kia lấy đối tượng mà sẽ gắn vào để *client* sử dụng dịch vụ.

Bây giờ, để tạo *stub* bạn cần chạy *rmic* trên lớp cài đặt *remote*. Công cụ *rmic* đi cùng bộ phát triển phần mềm Java, lấy cài đặt dịch vụ và tạo nên *stub* mới. Bạn cần mở dấu nhắc lệnh của bạn (*cmd*) và chạy *rmic* từ thư mục ở nơi có cài đặt từ xa của bạn.

Bước tiếp theo là chạy *rmiregistry*, mang đến một *terminal* và bắt đầu *registry*, chỉ bằng gõ lệnh *rmiregistry*. Nhưng tin tưởng là bạn *start* nó từ thư mục mà truy cập đến các lớp của bạn.

Bước cuối cùng là s*tart* dịch vụ này bằng cách chạy cài đặt cụ thể của lớp ở xa, trong trường hợp này là *ReportGeneratorImpl*.

Khi đó bạn đã tạo và chạy dịch vụ. Bây giờ chúng ta xem *client* sẽ sử dụng nó như thế nào. Ứng dụng báo cáo cho ông chủ công ty Pizza sẽ sử dụng dịch vụ này để sinh ra và kiểm tra báo cáo. Chúng ta cần cung cấp giao diện *ReportGenerator* và *stub* cho *client* mà sẽ sử dụng dịch vụ này. Bạn đơn giản cần trao tay *stub* hoặc các lớp hoặc các giao diện mà cần trong dịch vụ này.

```
package com.javacodegeeks.patterns.proxypattern.remoteproxy;

import java.rmi.Naming;

public class ReportGeneratorClient {

        public static void main(String[] args) {
                new ReportGeneratorClient().generateReport();
        }

        public void generateReport(){
                try {
                        ReportGenerator reportGenerator = (ReportGenerator)Naming.lookup(" ↩
                            rmi://127.0.0.1/PizzaCoRemoteGenerator");
                        System.out.println(reportGenerator.generateDailyReport());
                } catch (Exception e) {
                        e.printStackTrace();
                }
        }

}
```

Chương trình trên cho ra kết quả đầu ra như sau:

```
********************Location X Daily Report********************
 Location ID: 012
 Today's Date: Sun Sep 14 00:11:23 IST 2014
 Total Pizza Sell: 112
 Total Sale: $2534
 Net Profit: $1985
 ***************************************************************
```

Lớp trên thực hiện tìm kiếm tên và truy vấn đối tượng mà được sử dụng để sinh ra báo cáo hàng ngày. Bạn cần cung cấp IP của máy chủ và tên được sử dụng để gắn kết với dịch vụ. Những điều còn lại trông như lời gọi phương thức bình thường.

Tóm lại, *Remote Proxy* hoạt động như đại diện địa phương cho đối tượng mà sống ở JVM khác. Một phương thức gọi trên *Proxy* sẽ cho kết quả lời gọi đó được truyền qua mạng, triệu gọi từ xa, và kết quả được trả lại *Proxy* và sau đó là *client*.

6.9.4 Virtual Proxy

Mẫu *Virtual Proxy* là kỹ thuật tiết kiệm bộ nhớ mà được đề xuất trì hoãn tạo đối tượng cho đến khi nó cần. Nó được sử dụng khi việc tạo một đối tượng là đắt theo nghĩa tốn bộ nhớ sử dụng và xử lý. Trong ứng dụng thông thường, các đối tượng khác nhau tạo nên các phần khác nhau của chức năng. Khi một ứng dụng bắt đầu, nó có thể không cần mọi đối tượng của nó phải sẵn sàng ngay tức thì. Trong trường hợp này, mẫu *Virtual Proxy* đề xuất trì hoãn tạo đối tượng cho đến khi nó cần bởi ứng dụng. Một đối tượng mà được tạo, khi lần đầu được tham chiếu trong ứng dụng và cùng khởi tạo đó được tái sử dụng từ thời điểm đó trở đi. Ưu điểm của cách tiếp cận này là thời gian khởi tạo ứng dụng nhanh hơn, vì nó không yêu cầu được tạo và tải mọi đối tượng ứng dụng ngay.

Giả sử có đối tượng *Company* trong ứng dụng của bạn và đối tượng này chứa danh sách mọi nhân viên của Công ty trong đối tượng *ContactList*. Đây có thể là hàng vài ngàn nhân viên của công ty. Tải đối tượng *Company* từ cơ sở dữ liệu với danh sách các nhân viên của nó trong đối tượng *ContactList* có thể rất tốn thời gian. Trong một số trường hợp ngay cả bạn không cần đến danh sách nhân viên, nhưng bạn buộc phải chờ cho đến khi công ty và danh sách nhân viên của nó được tải vào bộ nhớ.

Một cách để tiết kiệm thời gian và bộ nhớ là tránh tải các đối tượng làm việc cho đến khi chúng được yêu cầu, và điều này được thực hiện sử dụng *Virtual Proxy*. Kỹ thuật này được biết đến là Tải Lười khi bạn đẩy dữ liệu vào chỉ khi chúng được yêu cầu.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

public class Company {

        private String companyName;
        private String companyAddress;
        private String companyContactNo;
        private ContactList contactList ;

        public Company(String companyName,String companyAddress, String companyContactNo, ↩
            ContactList contactList){
                this.companyName = companyName;
                this.companyAddress = companyAddress;
                this.companyContactNo = companyContactNo;
                this.contactList = contactList;

                System.out.println("Company object created...");
        }

        public String getCompanyName() {
                return companyName;
        }

        public String getCompanyAddress() {
                return companyAddress;
        }

        public String getCompanyContactNo() {
                return companyContactNo;
        }

        public ContactList getContactList(){
                return contactList;
        }

}
```

Lớp *Company* trên đây có tham chiếu đến giao diện *ContactList* mà đối tượng thực tế đó sẽ được tải khi có yêu cầu gọi phương thức *getContactList ()*.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public interface ContactList {

        public List<Employee> getEmployeeList();
}
```

Giao diện *ContactList* chỉ chứa một phương thức *getEmployeeList ()* mà được sử dụng để nhận danh sách các nhân viên của Công ty.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.ArrayList;
import java.util.List;

public class ContactListImpl implements ContactList{

        @Override
        public List<Employee> getEmployeeList() {
                return getEmpList();
        }

        private static List<Employee>getEmpList(){
                List<Employee> empList = new ArrayList<Employee>(5);
                empList.add(new Employee("Employee A", 2565.55, "SE"));
                empList.add(new Employee("Employee B", 22574, "Manager"));
                empList.add(new Employee("Employee C", 3256.77, "SSE"));
                empList.add(new Employee("Employee D", 4875.54, "SSE"));
                empList.add(new Employee("Employee E", 2847.01, "SE"));
                return empList;
        }

}
```

Lớp trên sẽ tạo ra đối tượng *ContactList* thực tế mà trả về danh sách các nhân viên của công ty. Đối tượng này sẽ được tải theo yêu cầu và chỉ khi nó được đòi hỏi.

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public class ContactListProxyImpl implements ContactList{

        private ContactList contactList;

        @Override
        public List<Employee> getEmployeeList() {
                if(contactList == null){
                        System.out.println("Creating contact list and fetching list of ↩
                            employees...");
                        contactList = new ContactListImpl();
                }
                return contactList.getEmployeeList();
        }

}
```

ContactListProxyImpl là lớp *proxy* mà cũng cài đặt *ContactList* và giữ tham chiếu đến đối tượng *ContactList* thực tế. Trong cài đặt của phương thức *getEmployeeList ()* nó sẽ kiểm tra nếu tham chiếu ContactList là null, thì nó sẽ tạo đối tượng *ContactList* thực tế và sau đó triệu gọi phương thức *getEmployeeList ()* trên nó để nhận được danh sách nhân viên.

Lớp *Employee* trông như sau:

```
package com.javacodegeeks.patterns.proxypattern.virtualproxy;

public class Employee {

        private String employeeName;
        private double employeeSalary;
        private String employeeDesignation;

        public Employee(String employeeName,double employeeSalary,String
            employeeDesignation){
                this.employeeName = employeeName;
                this.employeeSalary = employeeSalary;
                this.employeeDesignation = employeeDesignation;
        }

        public String getEmployeeName() {
                return employeeName;
        }

        public double getEmployeeSalary() {
                return employeeSalary;
        }

        public String getEmployeeDesignation() {
                return employeeDesignation;
        }

        public String toString(){
                return "Employee Name: "+employeeName+", EmployeeDesignation: "+
                    employeeDesignation+", Employee Salary: "+employeeSalary;
        }

}

package com.javacodegeeks.patterns.proxypattern.virtualproxy;

import java.util.List;

public class TestVirtualProxy {

        public static void main(String[] args) {
                ContactList contactList = new ContactListProxyImpl();
                Company company = new Company("ABC Company", "India", "+91-011-28458965",
                    contactList);

                System.out.println("Company Name: "+company.getCompanyName());
                System.out.println("Company Address: "+company.getCompanyAddress());
                System.out.println("Company Contact No.: "+company.getCompanyContactNo());

                System.out.println("Requesting for contact list");

                contactList = company.getContactList();

                List<Employee>empList = contactList.getEmployeeList();
                for(Employee emp : empList){
                        System.out.println(emp);
                }
        }

}
```

Chương trình trên sẽ cho ra kết quả:

```
Company object created...
Company Name: ABC Company
Company Address: India
Company Contact No.: +91-011-28458965
```

```
Requesting for contact list
Creating contact list and fetching list of employees...
Employee Name: Employee A, EmployeeDesignation: SE, Employee Salary: 2565.55
Employee Name: Employee B, EmployeeDesignation: Manager, Employee Salary: 22574.0
Employee Name: Employee C, EmployeeDesignation: SSE, Employee Salary: 3256.77
Employee Name: Employee D, EmployeeDesignation: SSE, Employee Salary: 4875.54
Employee Name: Employee E, EmployeeDesignation: SE, Employee Salary: 2847.01
```

Như bạn nhìn thấy trong đầu ra được sinh bởi *TestVirtualProxy*, trước hết chúng ta tạo ra đối tượng *Company* với đối tượng *proxy ContactList*. Tại thời điểm này, đối tượng *Company* giữ tham chiếu *proxy*, không phải tham chiếu của đối tượng *ContactList* thực tế, như vậy không có danh sách nhân viên được tải vào bộ nhớ. Chúng ta thực hiện mấy lời gọi đối tượng *Company* và sau đó yêu cầu danh sách nhân viên từ đối tượng *contact list proxy* sử dụng phương thức *getEmployeeList ()*. Trong lời gọi phương thức này, đối tượng *proxy* tạo đối tượng *ContactList* thực tế và cung cấp danh sách các nhân viên.

6.9.5 Protection Proxy

Nhìn chung, các đối tượng trong một ứng dụng tương tác với nhau để thực hiện chức năng ứng dụng tổng thể. Hầu hết đối tượng ứng dụng là được truy cập chung cho mọi đối tượng khác trong ứng dụng. Đôi khi, nó có thể được yêu cầu hạn chế truy cập đối tượng cho tập hạn chế các đối tượng client dựa trên quyền truy cập của chúng. Khi một đối tượng thử truy cập đến đối tượng như vậy, *client* chỉ được truy cập đến dịch vụ được cung cấp bởi đối tượng, nếu client có thể cung cấp giấy xác thực đúng. Trong các trường hợp như vậy, một đối tượng tách biệt có thể được chỉ định với trách nhiệm kiểm chứng quyền truy cập của các đối tượng client khác nhau khi họ truy cập đến các đối tượng thực tế. Nói cách khác, mỗi client cần xác thực thành công với đối tượng chỉ định đó để được truy cập đến chức năng đối tượng thực tế. Đối tượng như vậy, mà với nó client cần để xác thực được truy cập đến đối tượng thực tế, có thể được tham chiếu như xác thực đối tượng mà được cài đặt sử dụng *Protection Proxy*.

Trở lại ứng dụng *ReportGenerator* mà chúng ta đã phát triển cho Công ty Pizza, ông chủ bây giờ yêu cầu chỉ có ông có thể sinh ra báo cáo hàng ngày. Không nhân viên nào khác có thể làm như vậy được.

Để cài đặt đặc tính an ninh đó, chúng ta sử dụng *Protection Proxy* mà kiểm tra nếu đối tượng mà thử sinh ra báo cáo có phải là ông chủ không, nếu đúng, thì báo cáo sẽ sinh ra, ngược lại thì không.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public interface Staff {

        public boolean isOwner();
        public void setReportGenerator(ReportGeneratorProxy reportGenerator);
}
```

Giao diện *Staff* được sử dụng bởi các lớp *Owner* và *Employee* và giao diện này có hai phương thức: *isOwner ()* trả về biến Bool kiểm tra đối tượng gọi là owner hay không. Phương thức khác được sử dụng để đặt *ReportGeneratorProxy* mà được *Protection proxy* sử dụng để sinh ra báo cáo nếu *isOwner ()* trả về *true.*

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class Employee implements Staff{

        private ReportGeneratorProxy reportGenerator;

        @Override
        public void setReportGenerator(ReportGeneratorProxy reportGenerator){
                this.reportGenerator = reportGenerator;
        }

        @Override
        public boolean isOwner() {
                return false;
        }
```

```
        public String generateDailyReport(){
                try {
                        return reportGenerator.generateDailyReport();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return "";
        }

}
```

Lớp *Employee* cài đặt giao diện *Staff*, vì phương thức *isOwner* của *Employee* trả về *false.* Phương thức *generateDallyReport ()* yêu cầu *ReportGenerator* sinh ra báo cáo hàng ngày.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class Owner implements Staff {

        private boolean isOwner=true;
        private ReportGeneratorProxy reportGenerator;

        @Override
        public void setReportGenerator(ReportGeneratorProxy reportGenerator){
                this.reportGenerator = reportGenerator;
        }

        @Override
        public boolean isOwner(){
                return isOwner;
        }

        public String generateDailyReport(){
                try {
                        return reportGenerator.generateDailyReport();
                } catch (Exception e) {
                        e.printStackTrace();
                }
                return "";
        }

}
```

Lớp *Owner* trông giống lớp *Employee*, nhưng sự khác biệt chỉ ở chỗ phương thức *isOwner* trả về true.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public interface ReportGeneratorProxy {

        public String generateDailyReport();
}
```

ReportGeneratorProxy hoạt động như một *Protection Proxy* mà chỉ có một phương thức *generateDailyReport ()* mà được sử dụng để sinh báo cáo.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

import java.rmi.Naming;

import com.javacodegeeks.patterns.proxypattern.remoteproxy.ReportGenerator;

public class ReportGeneratorProtectionProxy implements ReportGeneratorProxy{
```

```
        ReportGenerator reportGenerator;
        Staff staff;

        public ReportGeneratorProtectionProxy(Staff staff){
                this.staff = staff;
        }

        @Override
        public String generateDailyReport() {
                if(staff.isOwner()){
                        ReportGenerator reportGenerator = null;
                        try {
                                reportGenerator = (ReportGenerator)Naming.lookup("rmi ↩
                                    ://127.0.0.1/PizzaCoRemoteGenerator");
                                return reportGenerator.generateDailyReport();
                        } catch (Exception e) {
                                e.printStackTrace();
                        }
                        return "";
                }
                else{
                        return "Not Authorized to view report.";
                }
        }
}
```

Lớp trên là cài đặt cụ thể của *ReportGeneratorProxy*, mà giữ tham chiếu đến giao diện *ReportGenerator* mà là *proxy* từ xa.Trong phương thức *generateDailyReport ().* Nó kiểm tra nếu *Staff* được tham chiếu đến *Owner*, thì yêu cầu *remote proxy* sinh ra báo cáo, ngược lại thì trả về xâu với thông báo *"Not Authorized to view report"*.

```
package com.javacodegeeks.patterns.proxypattern.protectionproxy;

public class TestProtectionProxy {

        public static void main(String[] args) {

                Owner owner = new Owner();
                ReportGeneratorProxy reportGenerator = new ReportGeneratorProtectionProxy( ↩
                    owner);
                owner.setReportGenerator(reportGenerator);

                Employee employee = new Employee();
                reportGenerator = new ReportGeneratorProtectionProxy(employee);
                employee.setReportGenerator(reportGenerator);
                System.out.println("For owner:");
                System.out.println(owner.generateDailyReport());
                System.out.println("For employee:");
                System.out.println(employee.generateDailyReport());

        }

}
```

Chương trình trên cho ra kết quả:

```
For owner:
********************Location X Daily Report********************
 Location ID: 012
 Today's Date: Sun Sep 14 13:28:12 IST 2014
 Total Pizza Sell: 112
 Total Sale: $2534

 Net Profit: $1985
 ************************************************************
For employee:
Not Authorized to view report.
```

Đầu ra trên đây chỉ rõ rằng ông chủ có thể sinh ra báo cáo, nhân viên thì không. *Protection Proxy* bảo vệ truy cập sinh ra báo cáo và chỉ cho phép đối tượng có chủ quyền mới sinh được báo cáo.

6.9.6 Khi nào sử dụng mẫu Proxy

Proxy được áp dụng khi có nhu cầu một tham chiếu hay thay đổi hay triết lý hơn đến một đối tượng so với bình thường. Ở đây có một số tình huống chung ở đó mẫu *proxy* được áp dụng:

- *Remote Proxy* cung cấp đại diện địa phương cho một đối tượng ở không gian địa chỉ khác
- *Virtual Proxy* tạo các đối tượng đắt theo nhu cầu.
- *Protetion proxy* kiểm soát truy cập đến đối tượng gốc. *Protection proxy* là hữu ích khi các đối tượng có các quyền truy cập khác nhau.