Software Architecture
Course's Code: CSE 483
Architectural Software Patterns in Java EE (JEE) (Chapter 3)

Chapter 3

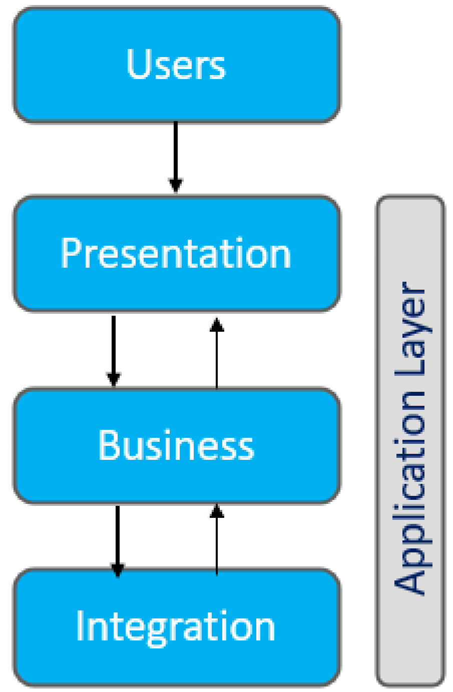
Chapter 3. Architectural Software Patterns in Java EE

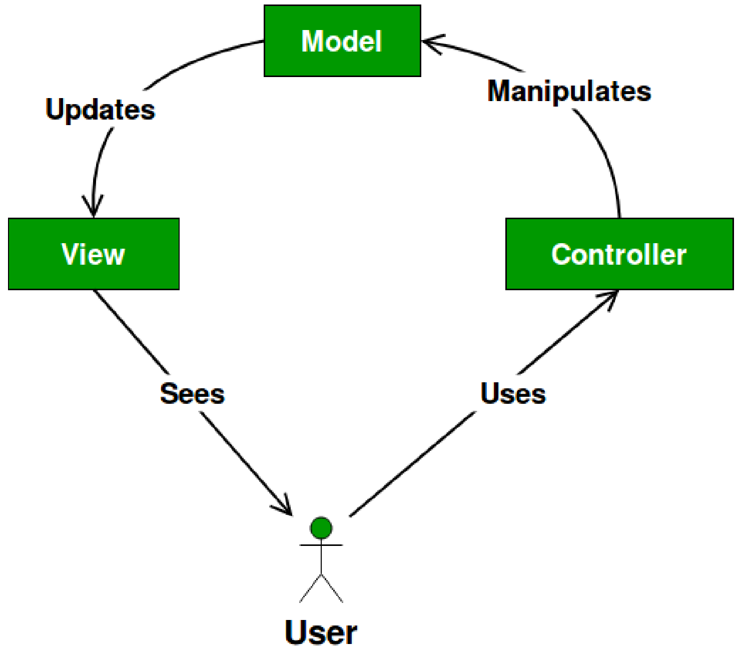
What are enterprise software patterns?

- Solutions to the enterprise system problems

- Model-View-Controller (MVC) pattern

- Used in web applications



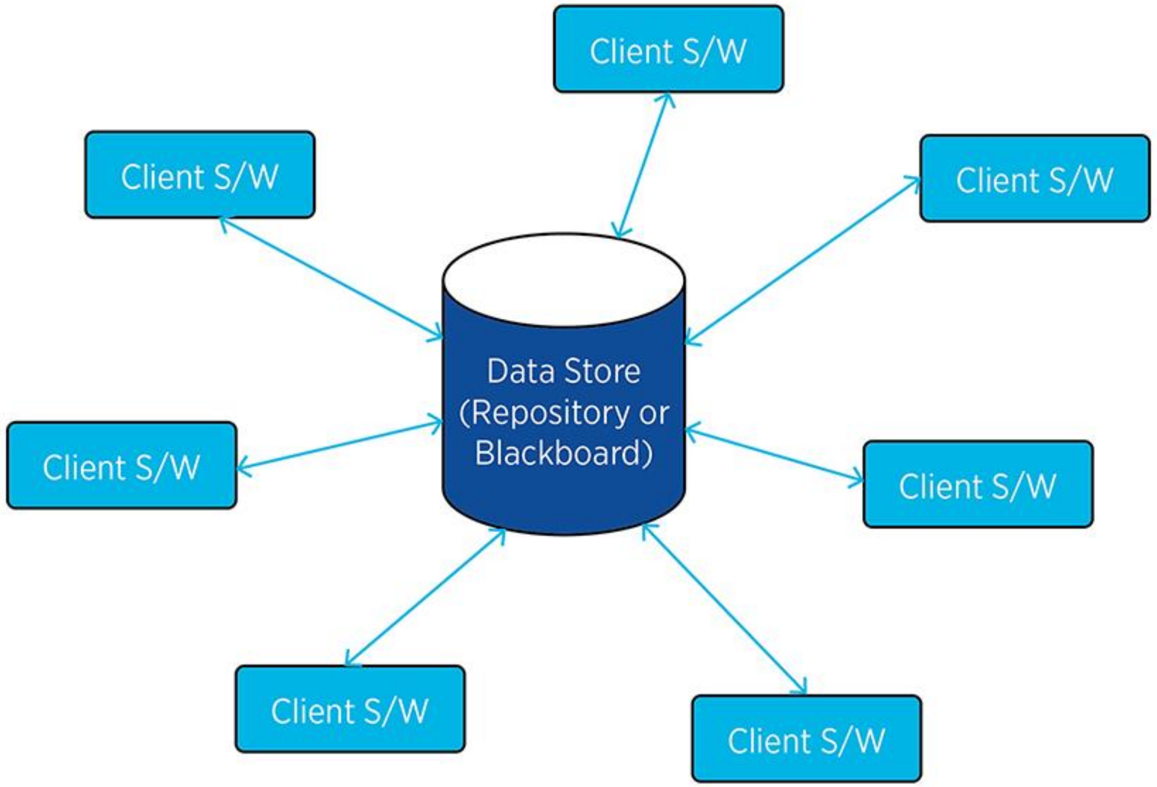


What are enterprise software patterns?

- Solutions to the enterprise system problems

- Structure of the application

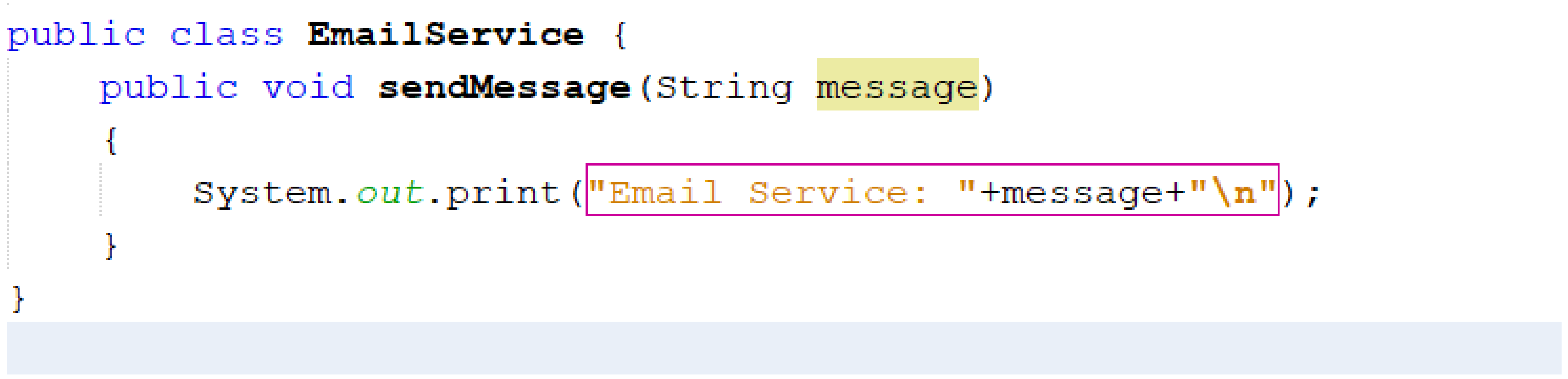
- Database-centric



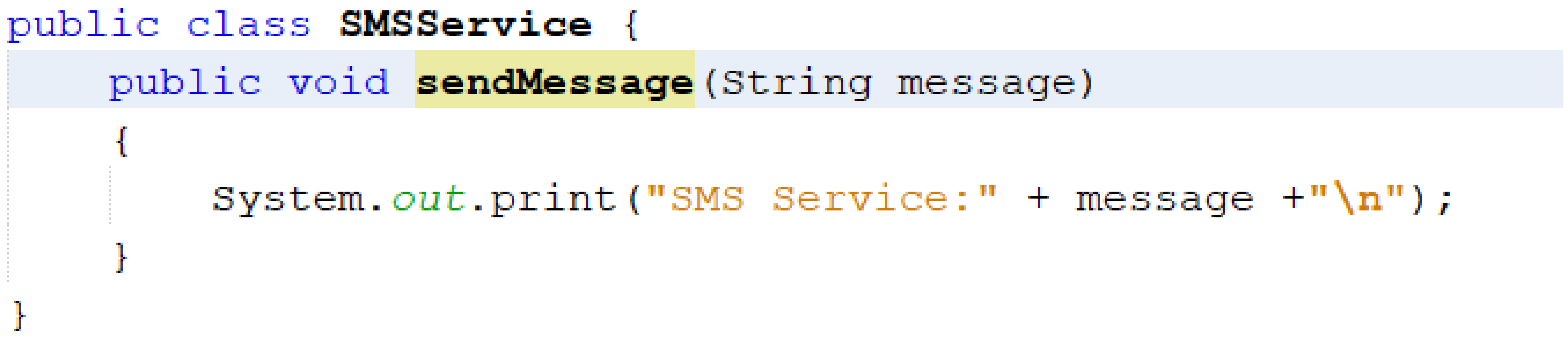
Chapter 3

Chapter 3. Architectural Software Patterns in Java EE

Creating a Messaging App using Facade pattern

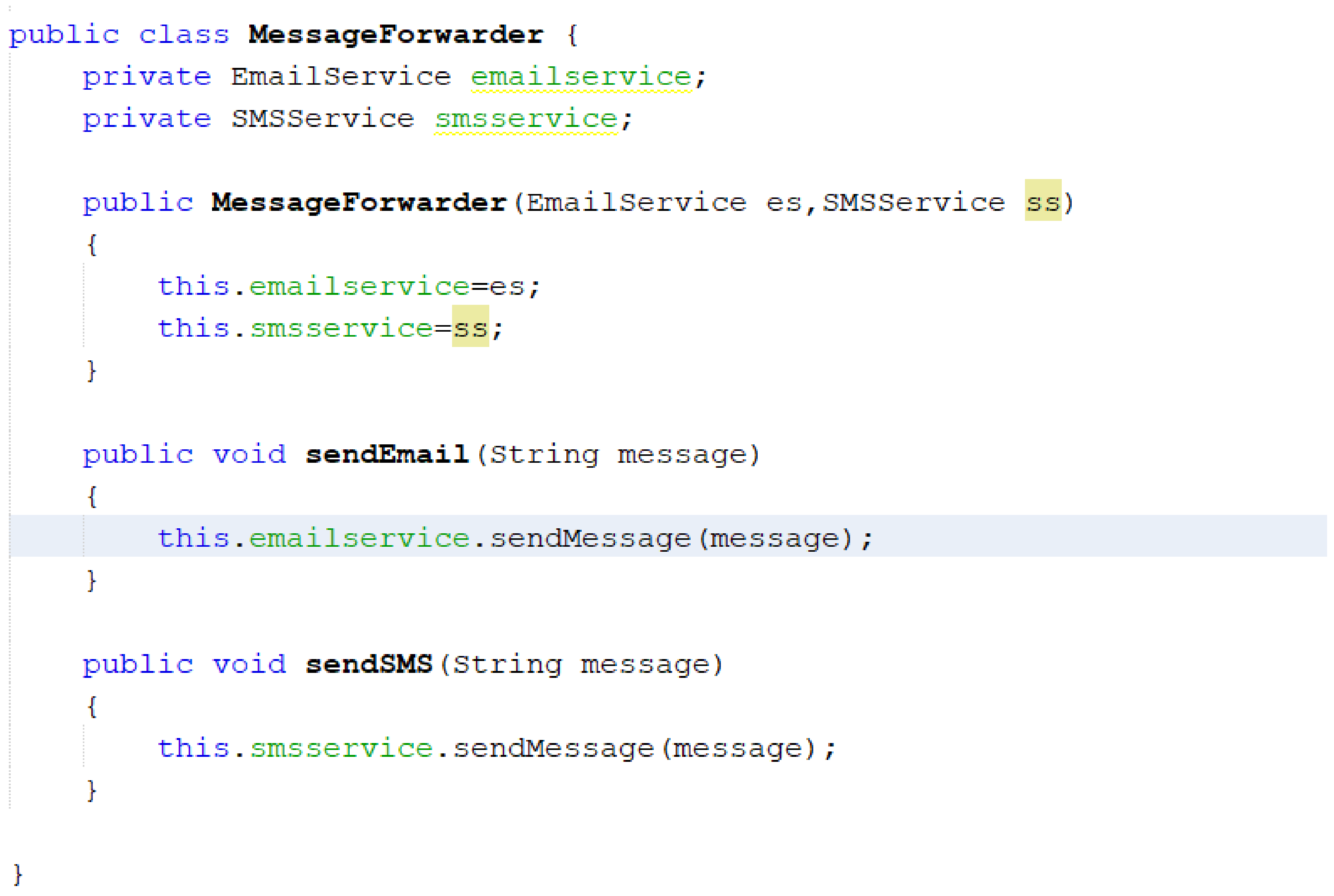


```java
public class EmailService {
     public void sendMessage(String message)
     {
          System.out.print("Email Service: "+message+"\n");
     }
}
```



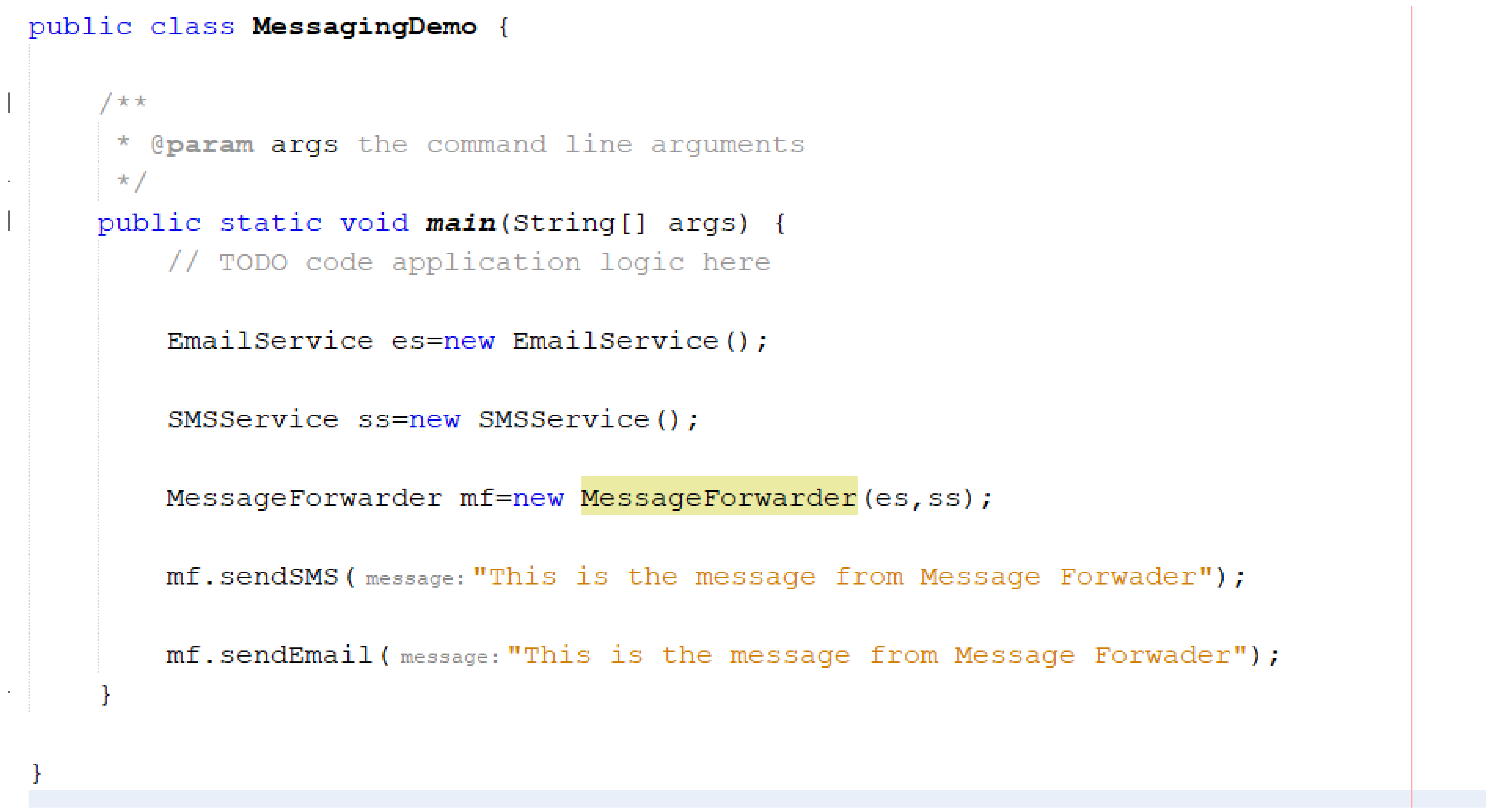
```java
public class SMSService {
     public void sendMessage(String message)
     {
          System.out.print("SMS Service:" + message +"\n");
     }
}
```

Creating a Messaging App using Facade pattern



```java
public class MessageForwarder {
    private EmailService emailservice;
    private SMSService smsservice;

    public MessageForwarder(EmailService es,SMSService ss)
    {
        this.emailservice=es;
        this.smsservice=ss;
    }

    public void sendEmail(String message)
    {
        this.emailservice.sendMessage(message);
    }

    public void sendSMS(String message)
    {
        this.smsservice.sendMessage(message);
    }

}
```

Creating a Messaging App using Facade pattern



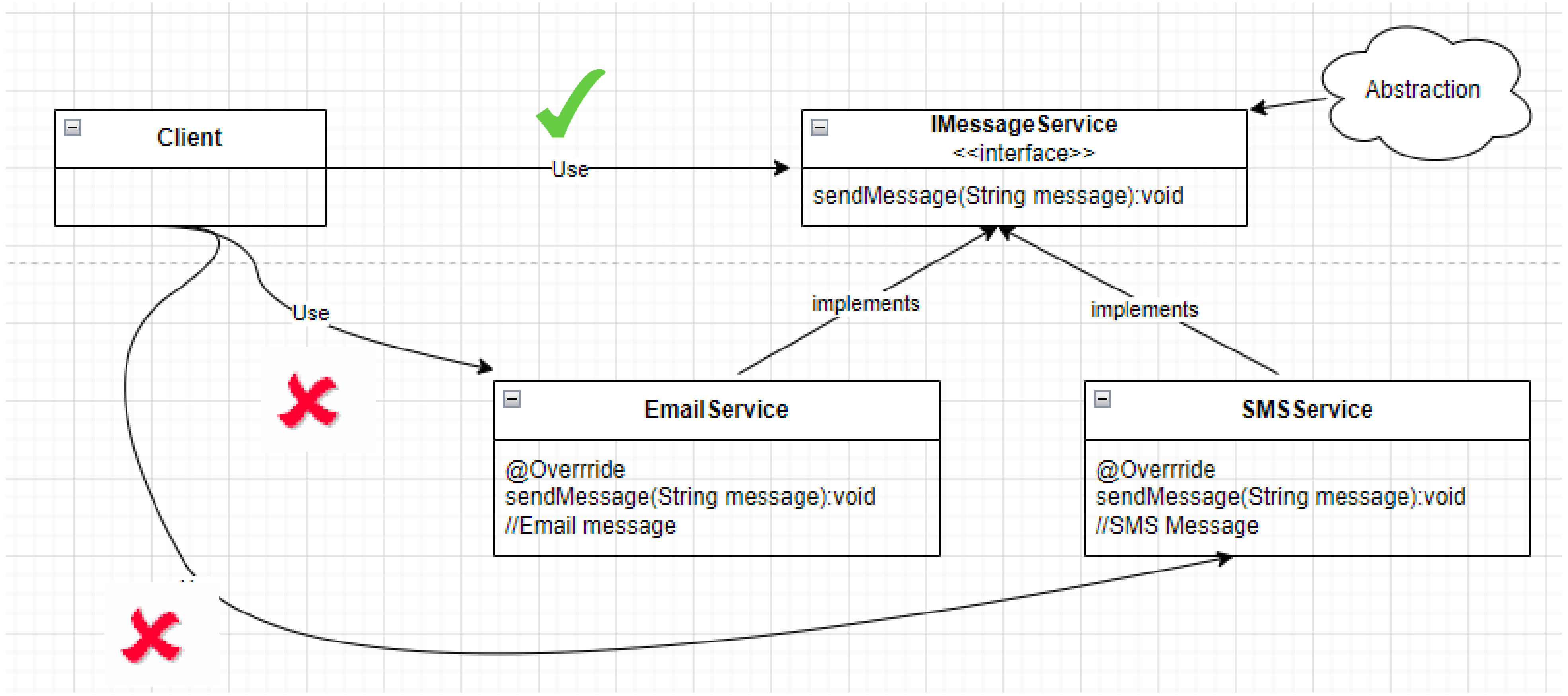
```java
public class MessagingDemo {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        EmailService es=new EmailService();

        SMSService ss=new SMSService();

        MessageForwarder mf=new MessageForwarder(es,ss);

        mf.sendSMS( message: "This is the message from Message Forwader");

        mf.sendEmail( message: "This is the message from Message Forwader");
    }

}
```

SOLID – The 5 Principles of Object Oriented Design

- SOLID stands for:
  - S - Single-responsiblity Principle
    - A class should have one and only one reason to change, meaning that a class should have only one job.
  - O - Open-closed Principle
    - Objects or entities should be open for extension but closed for modification
  - L - Liskov Substitution Principle
    - states that objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program
  - I - Interface Segregation Principle
    - A client should never be forced to implement an interface that it doesn't use, or clients shouldn't be forced to depend on methods they do not use.
  - D - Dependency Inversion Principle
    - Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

Dependency Inversion Principle

- Entities must depend on abstractions, not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.



Dependency Injection Pattern

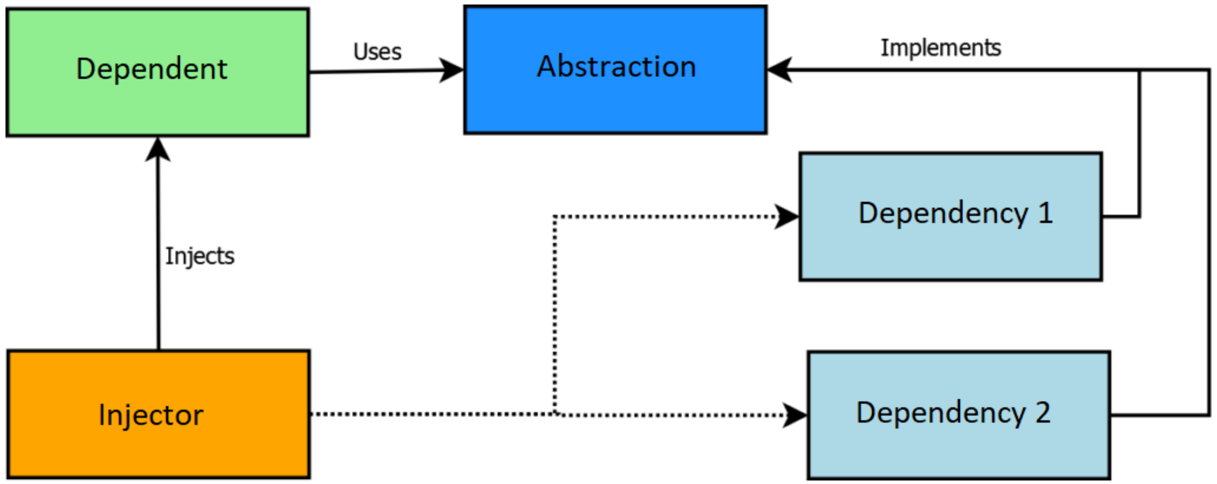
The Dependency Injection (DI) design pattern is a technique that focuses on separating the creation and management of dependencies from the classes that use them. It enables components to be loosely coupled by having their dependencies injected from external sources rather than creating them internally.

- Well-known but not a GOF pattern
- Implementation Independent
- Loosely Coupled
- Change where objects are created

Breakdown of components

The Dependency Injection Design Pattern involves 3 types of classes:

- **The Client Class:** (dependent class) is a class that depends on the Service Class. That means the Client Class wants to use the Services (Methods) of the Service Class.
- **Service Class:** The Service Class (dependency) is a class that provides the actual services to the client class.
- **Injector Class:** The Injector Class is a class that injects the Service Class object into the Client Class.



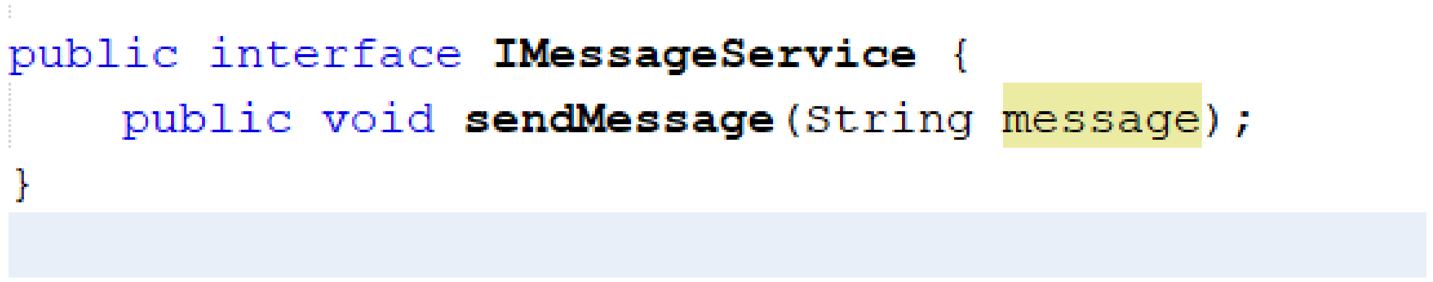
Types of Dependency Injection

The following are the types of dependency injections that could be injected into your application:

- **Constructor-based dependency injection**
  - supplies the dependency through the dependent class constructor.
- **Setter-based dependency injection**
  - supplies the dependency through any method (SETTER) of the dependent class

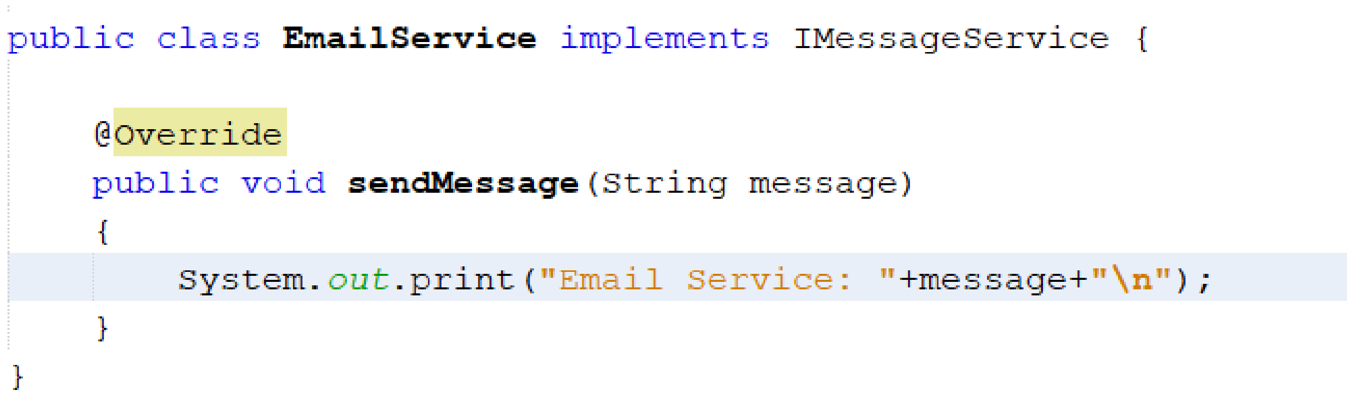
Creating a Messaging App using Dependency Injection

- 1. Create an interface for dependencies

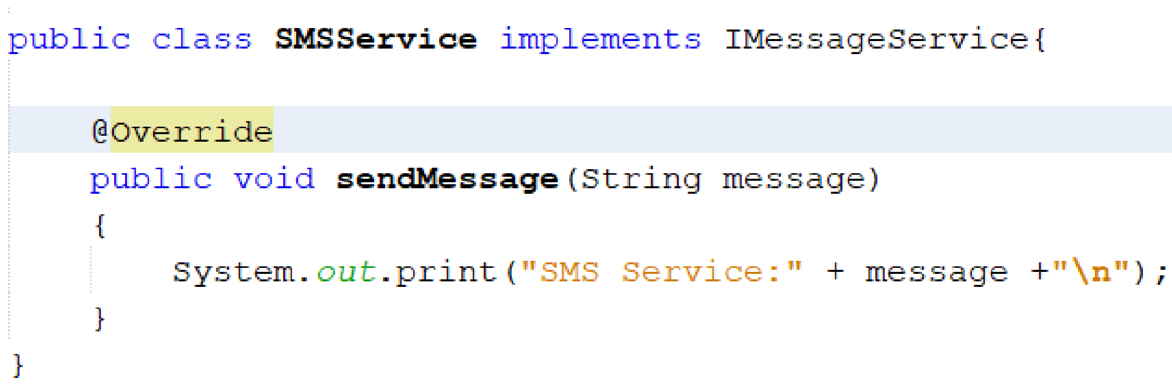


```
public interface IMessageService {
    public void sendMessage(String message);
}
```

- 2. Create an implementations of dependencies



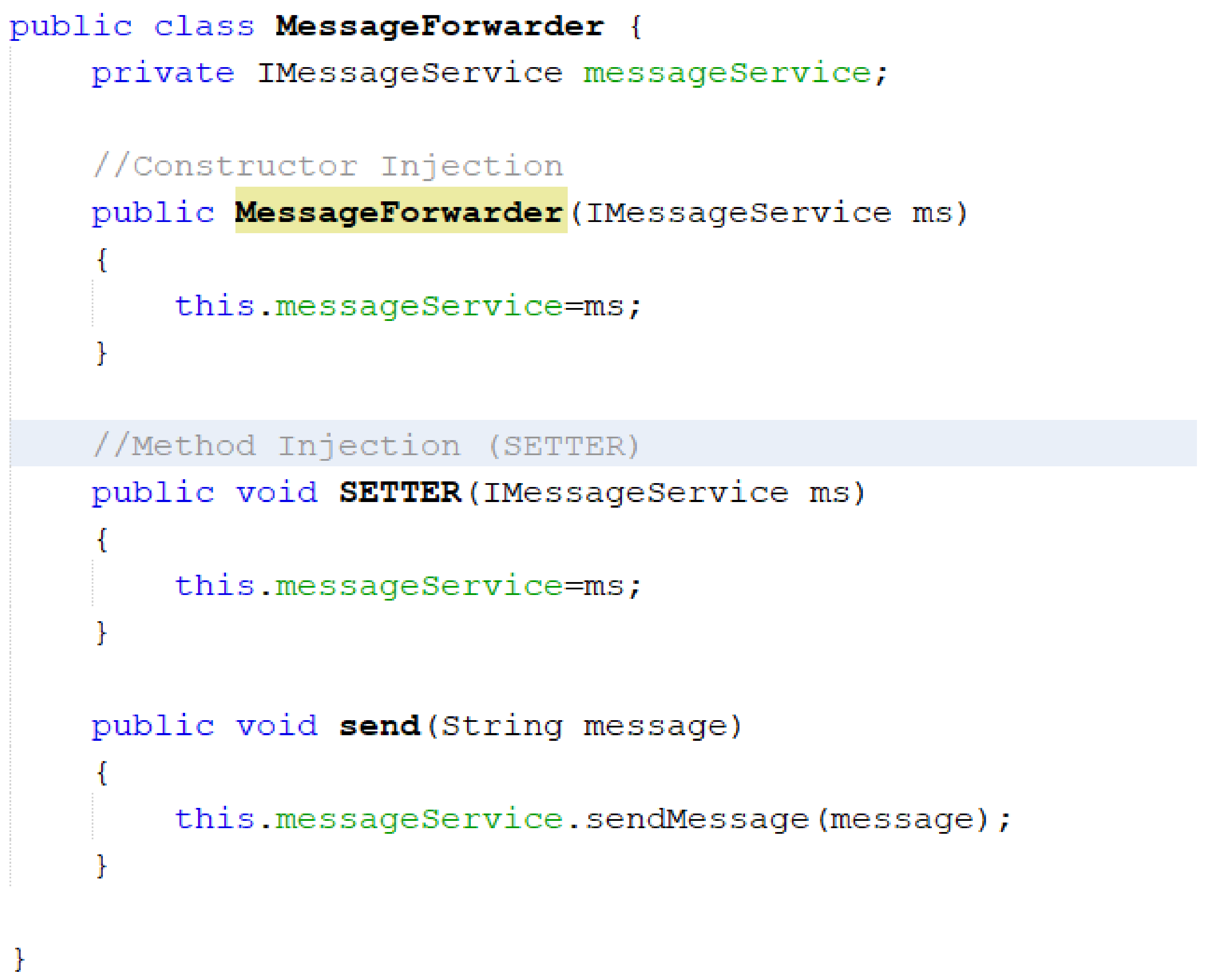
```
public class EmailService implements IMessageService {

    @Override
    public void sendMessage(String message)
    {
        System.out.print("Email Service: "+message+"\n");
    }
}
```



```
public class SMSService implements IMessageService{

    @Override
    public void sendMessage(String message)
    {
        System.out.print("SMS Service:" + message +"\n");
    }
}
```

Creating a Messaging App using Dependency Injection

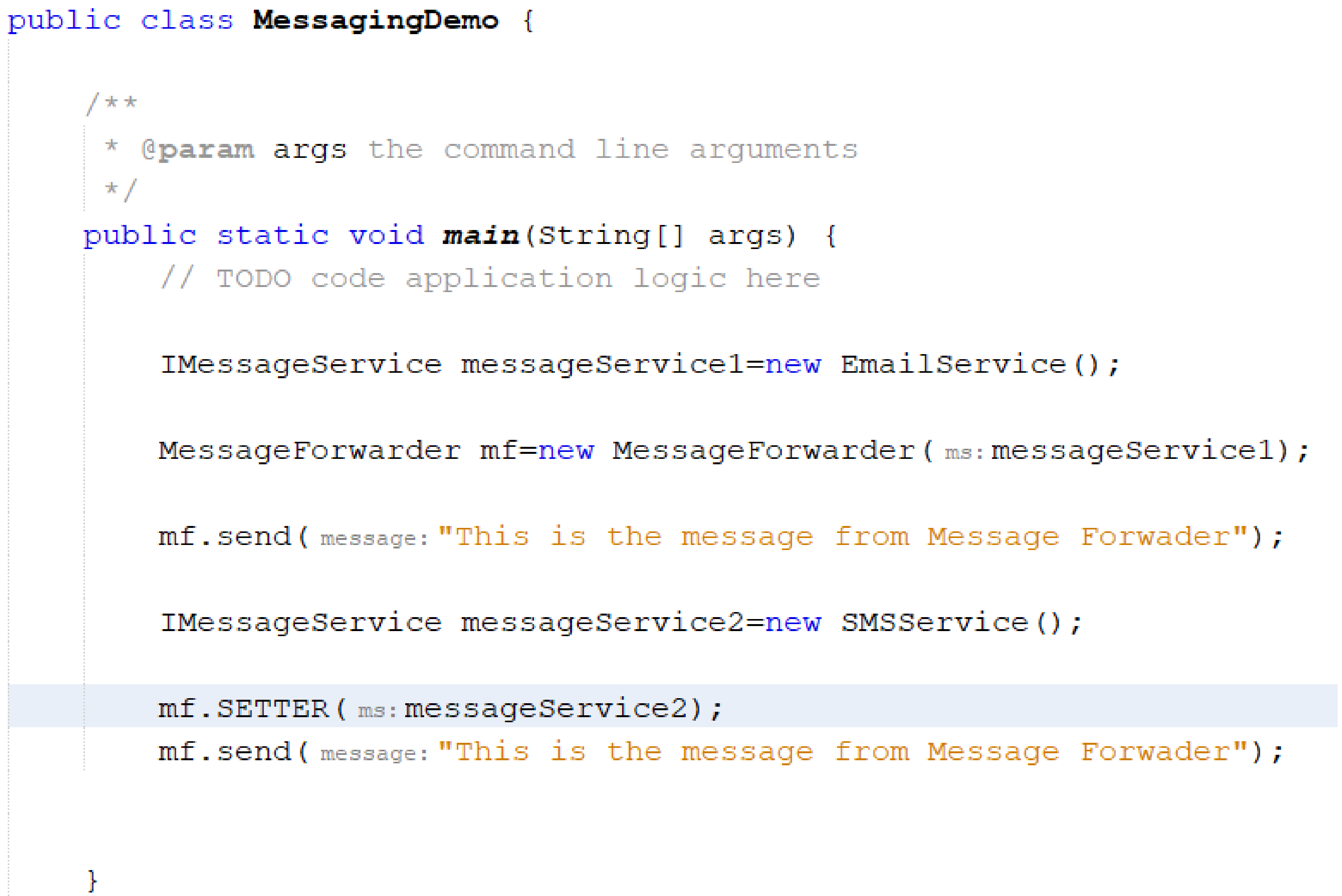
3. Create a dependent class



```
public class MessageForwarder {
    private IMessageService messageService;

    //Constructor Injection
    public MessageForwarder(IMessageService ms)
    {
        this.messageService=ms;
    }

    //Method Injection (SETTER)
    public void SETTER(IMessageService ms)
    {
        this.messageService=ms;
    }

    public void send(String message)
    {
        this.messageService.sendMessage(message);
    }
}
```

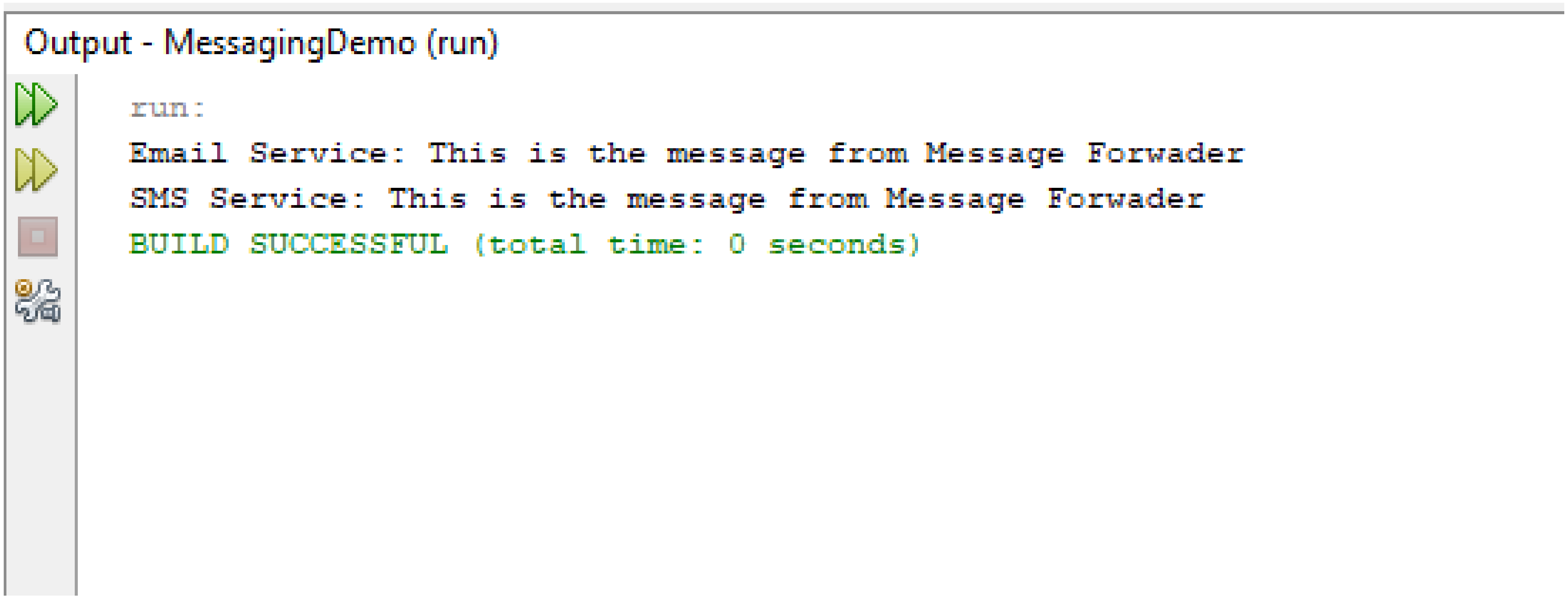
Creating a Messaging App using Dependency Injection

4. Create a injector



```
public class MessagingDemo {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here

        IMessageService messageService1=new EmailService();

        MessageForwarder mf=new MessageForwarder( ms: messageService1);

        mf.send( message: "This is the message from Message Forwader");

        IMessageService messageService2=new SMSService();

        mf.SETTER( ms: messageService2);
        mf.send( message: "This is the message from Message Forwader");


    }
```

Creating a Messaging App using Dependency Injection



Output - MessagingDemo (run)

```
run:
Email Service: This is the message from Message Forwader
SMS Service: This is the message from Message Forwader
BUILD SUCCESSFUL (total time: 0 seconds)
```

Benefits of Dependency Injection

1. **Loose Coupling**: DI promotes loose coupling between classes, reducing interdependencies and making the codebase more maintainable and flexible to changes.

2. **Testability**: By allowing dependencies to be easily replaced or mocked during testing, DI makes unit testing more manageable and reliable.

3. **Scalability**: DI facilitates the development of scalable applications by enabling easy swapping of implementations and promoting modular design.

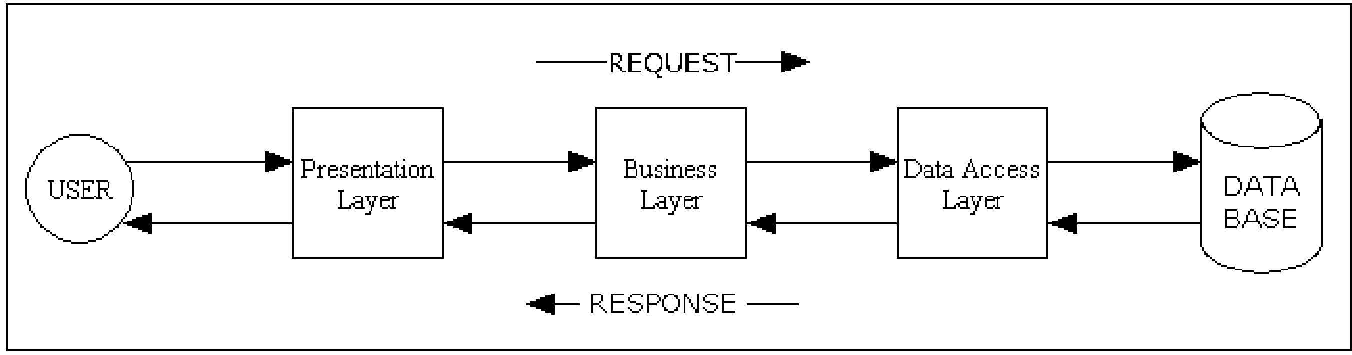
4. **Reusability**: DI encourages the use of interfaces and abstractions, leading to more reusable and extensible code.

Chapter 3

Chapter 3. Architectural Software Patterns in Java EE

Context and Problem

- The presentation layer receives different types of requests, some requires varied types of processing.

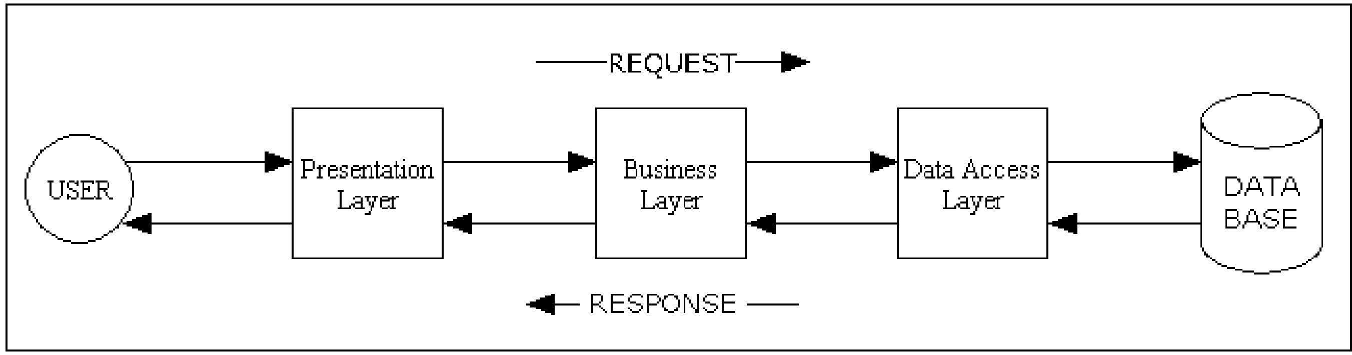


- A Web request and response: Preprocessing and post-processing

- A request enters a Web application often pass several entrance tests:
  - + Client authenticated?
  - + Client having valid session?
  - + Client’s IP trusted?
  - + Browser type is supported ?
  - ….

Context and Problem

- Each test needs Yes or No to continue to next test
- Any failed check will abort the request



- Need a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

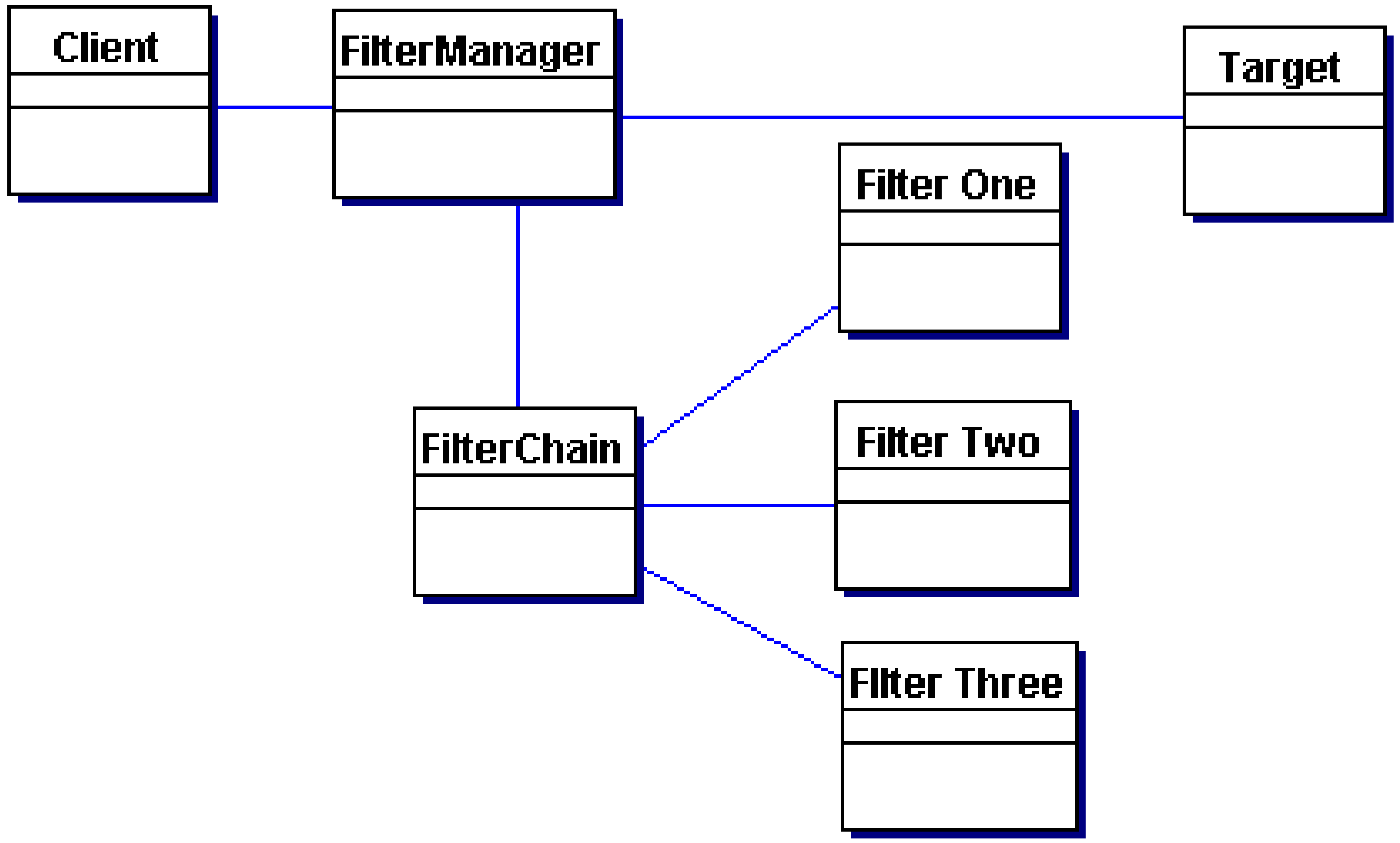
Solution

Requirements
Services should be easy to add or remove unobtrusively without affecting existing components:
- **Logging and authentication**
- Debugging and transformation of output for a specific client
- Uncompressing and converting encoding scheme of input

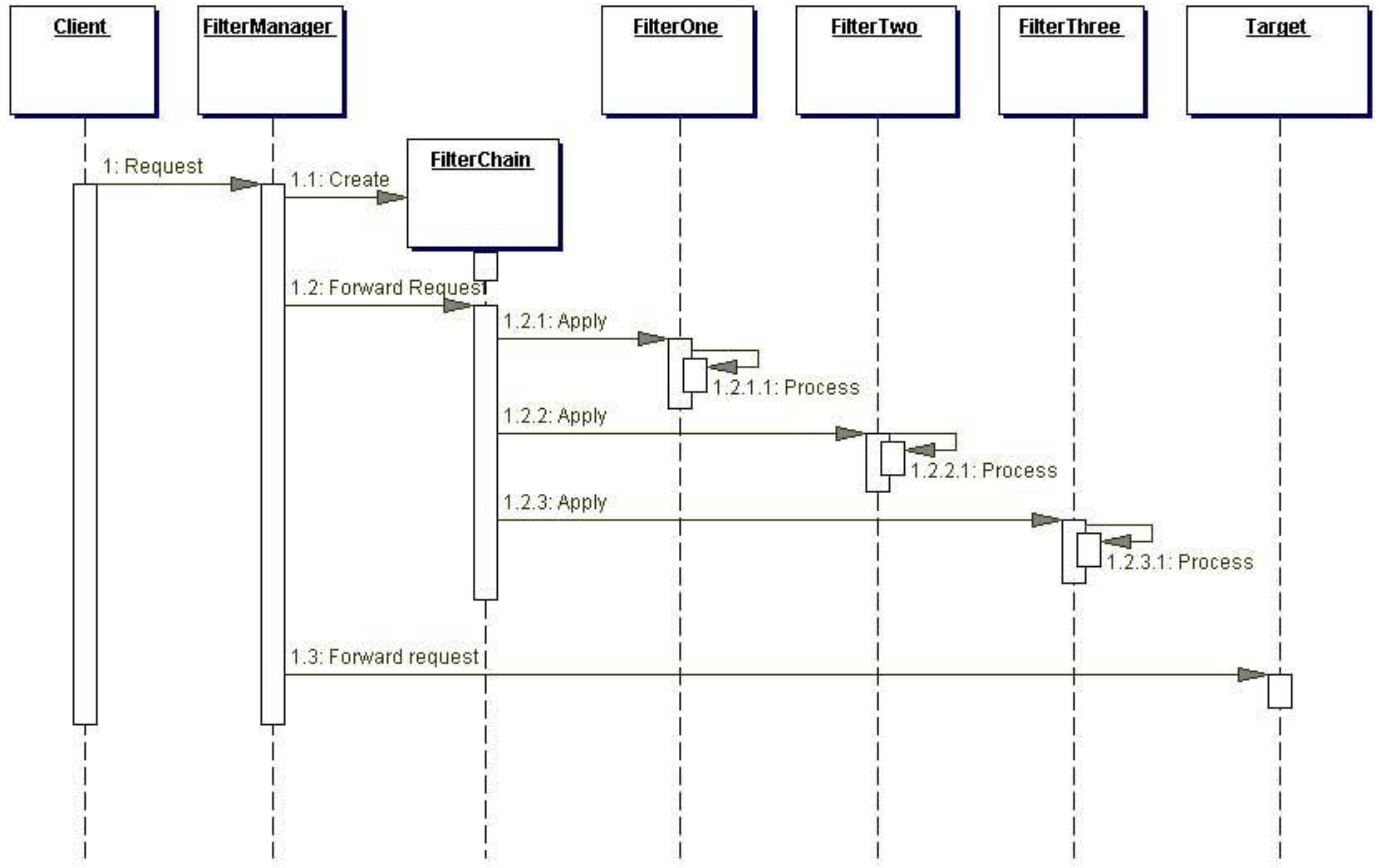
Solution!

Create pluggable filters to process common services in a standard manner without requiring changes to core request processing code. The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing. We are able to add and remove these filters unobtrusively, without requiring changes to our existing code.

Structure of the Filter Design Pattern



Participants and Responsibilities



Breakdown of components

Filter

- Filter which will performs certain task prior or after execution of request by request handler.

Filter Chain

- Filter Chain carries multiple filters and help to execute them in defined order on target.

Target

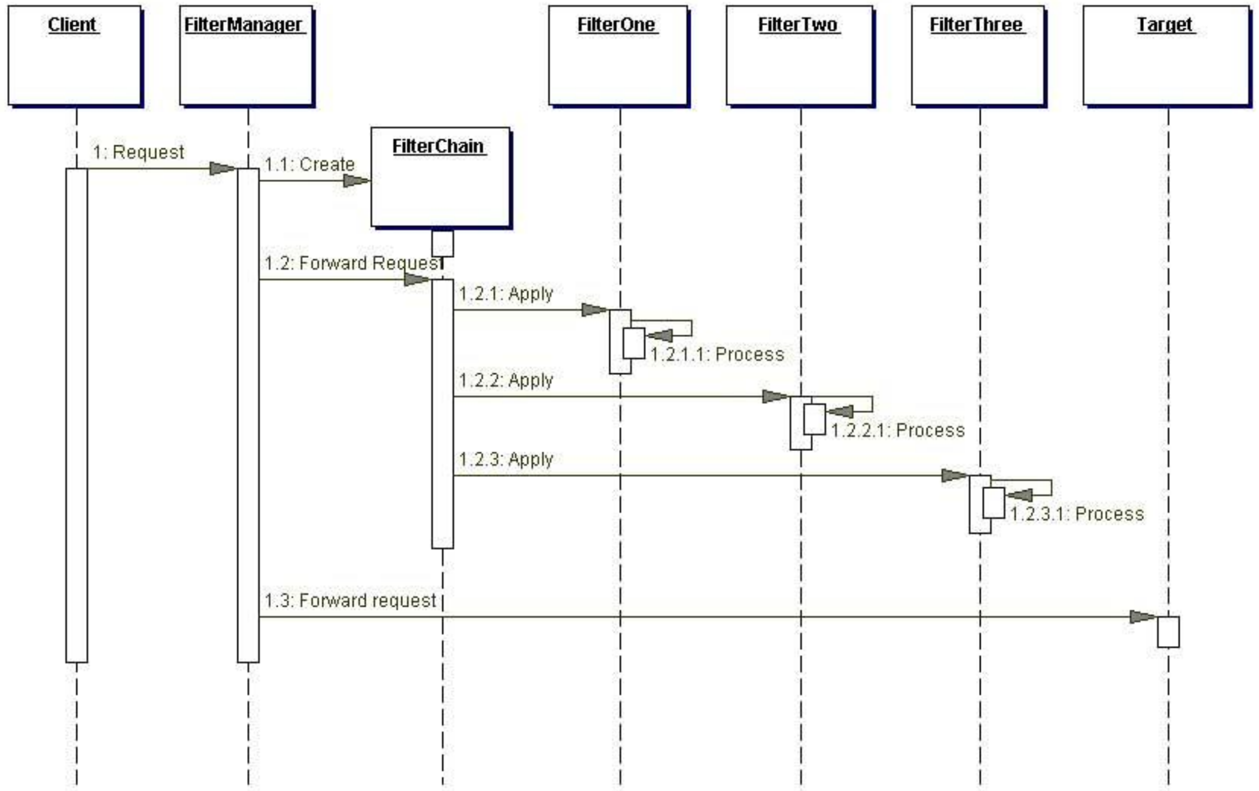
- Target object is the request handler

Filter Manager

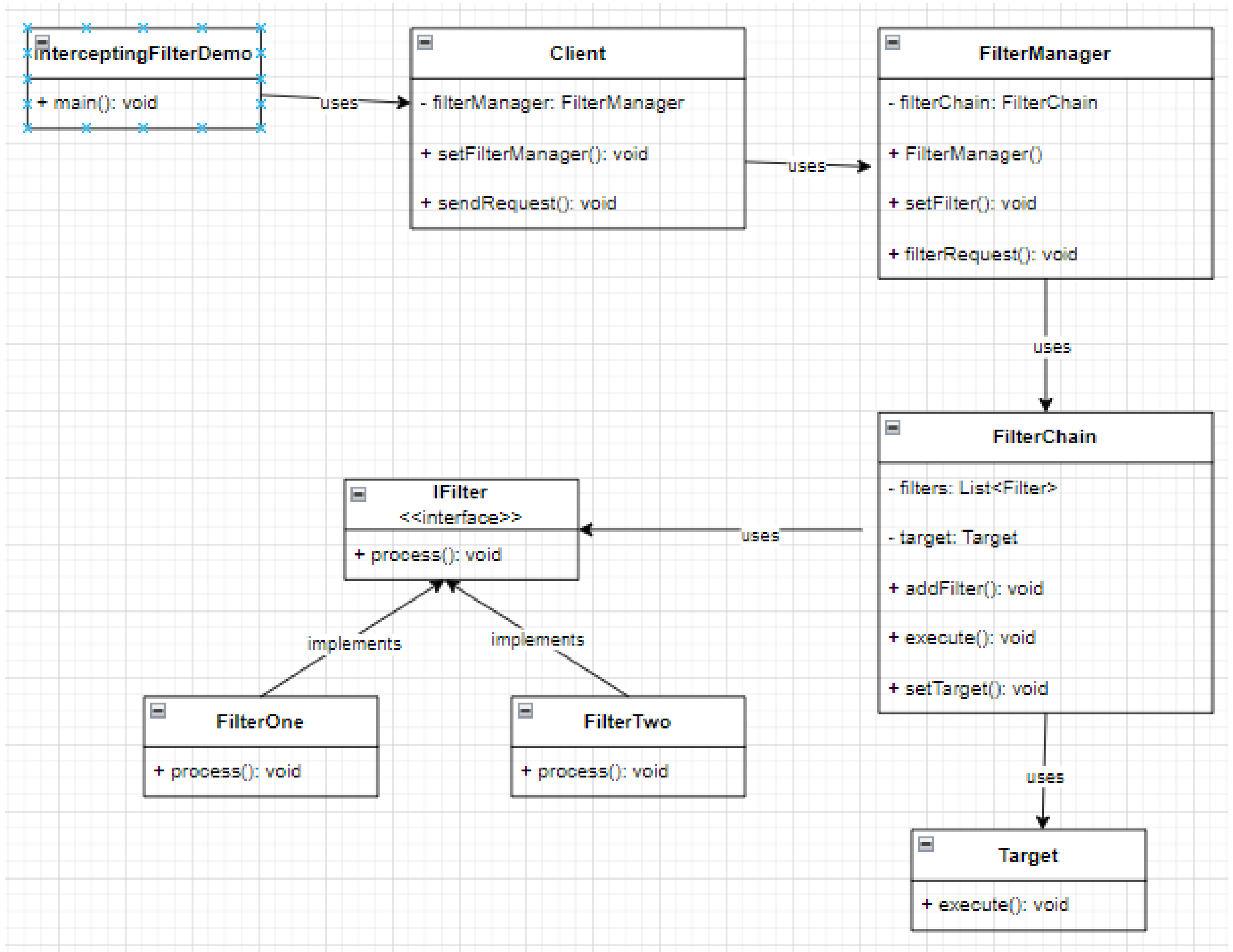
- Filter Manager manages the filters and Filter Chain.

Client

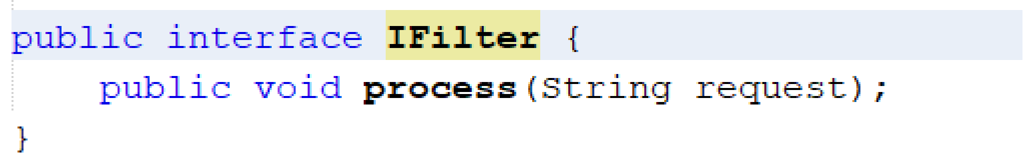
- Client is the object who sends request to the Target object.



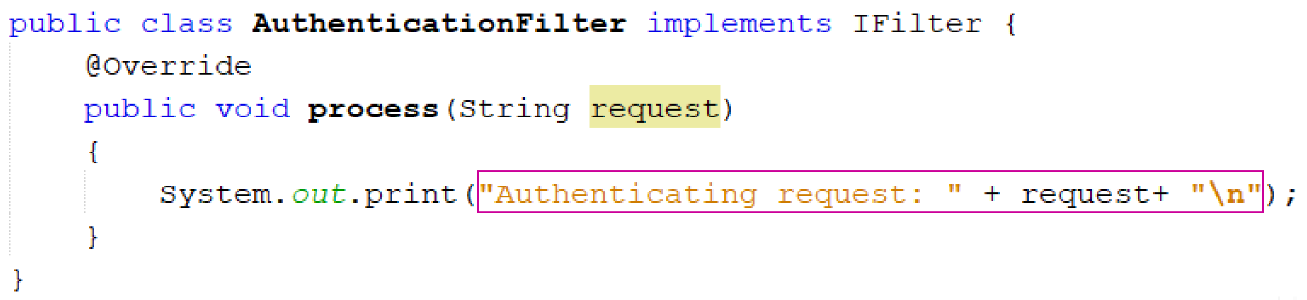
Implementation



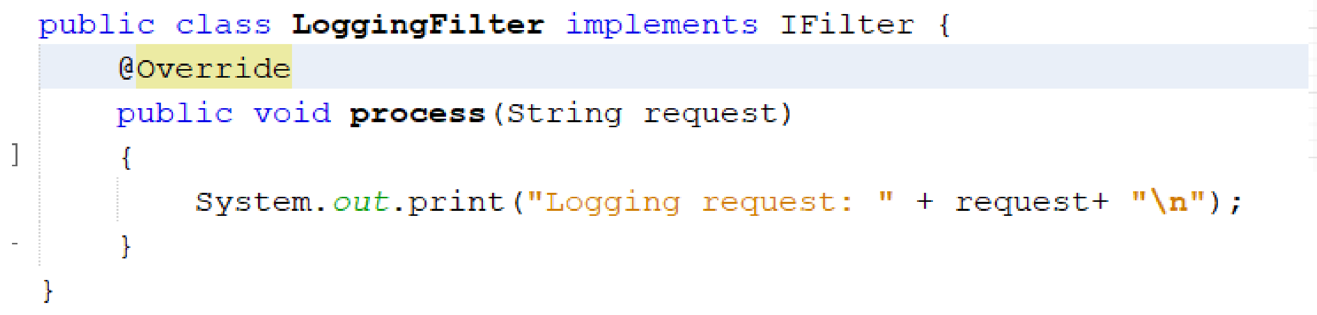
Implementation



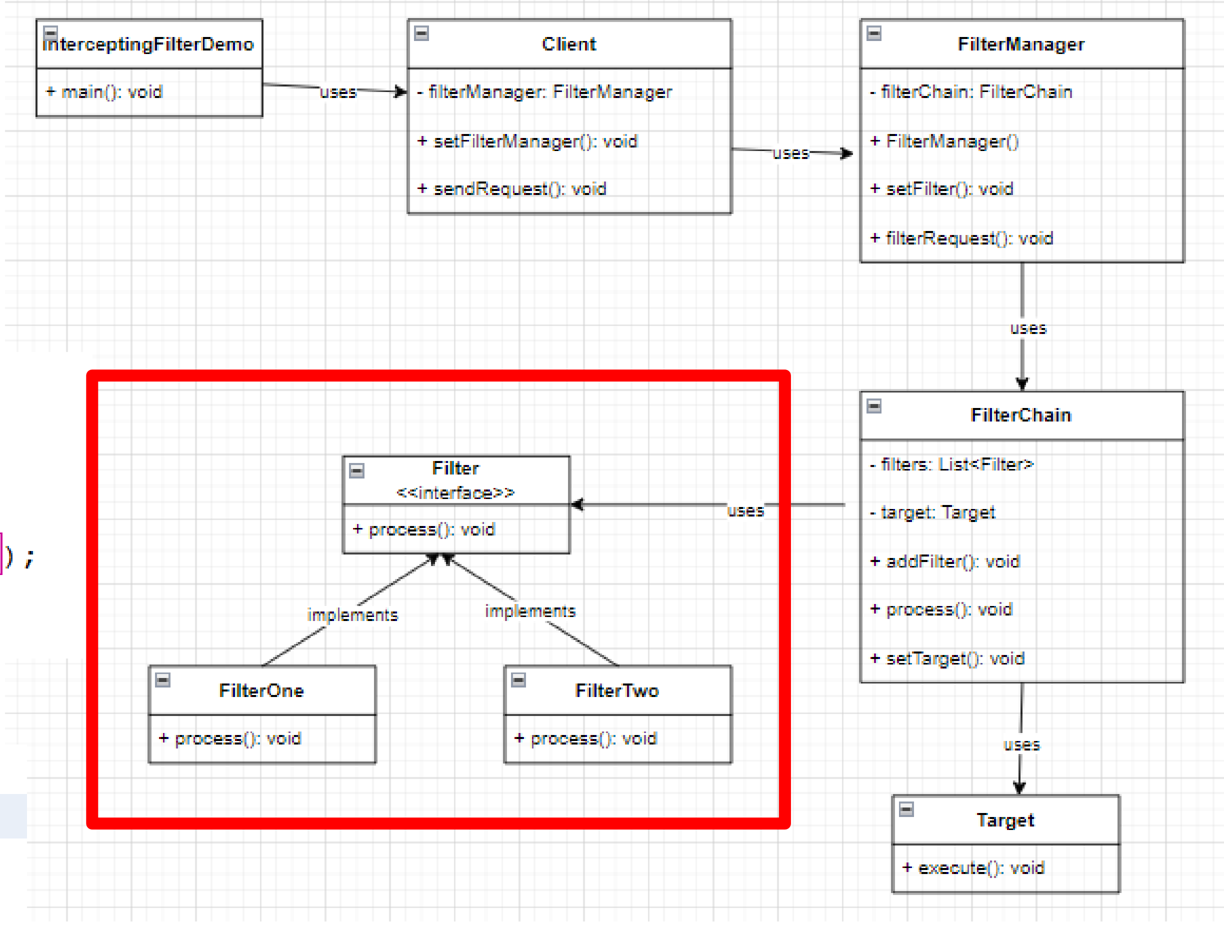
```
public interface IFilter {
    public void process(String request);
}
```



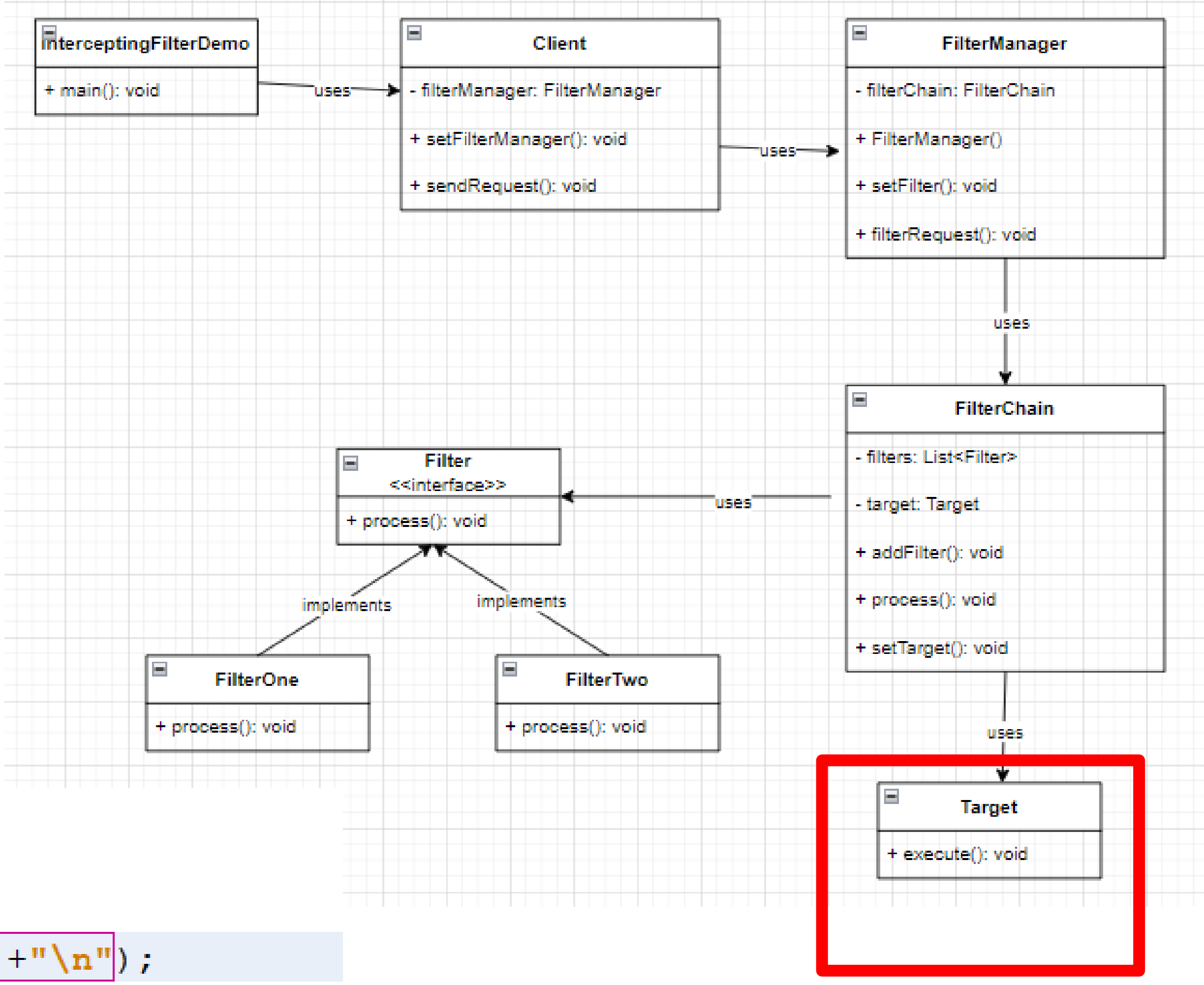
```
public class AuthenticationFilter implements IFilter {
    @Override
    public void process(String request)
    {
        System.out.print("Authenticating request: " + request+ "\n");
    }
}
```



```
public class LoggingFilter implements IFilter {
    @Override
    public void process(String request)
    {
        System.out.print("Logging request: " + request+ "\n");
    }
}
```

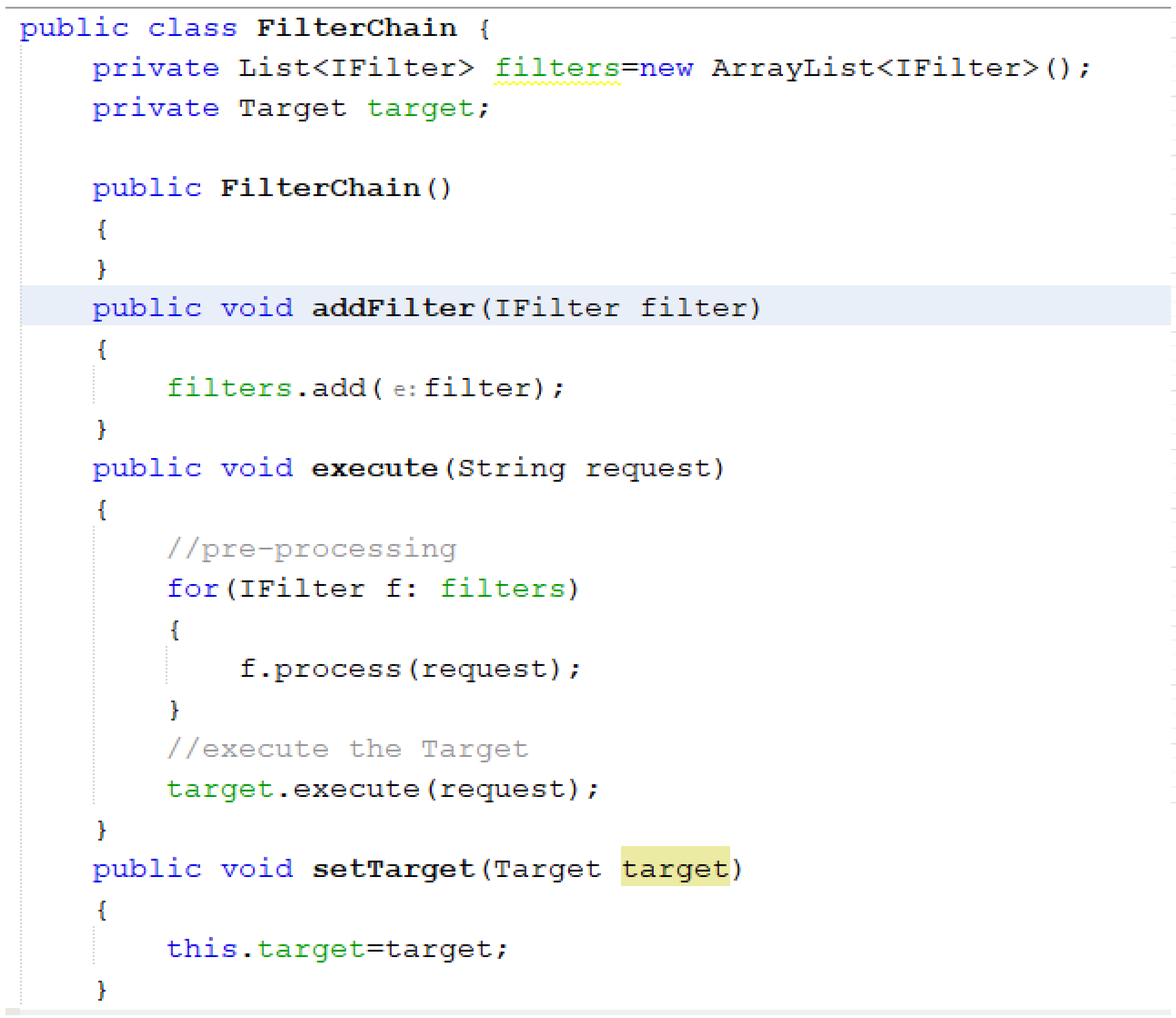


Implementation

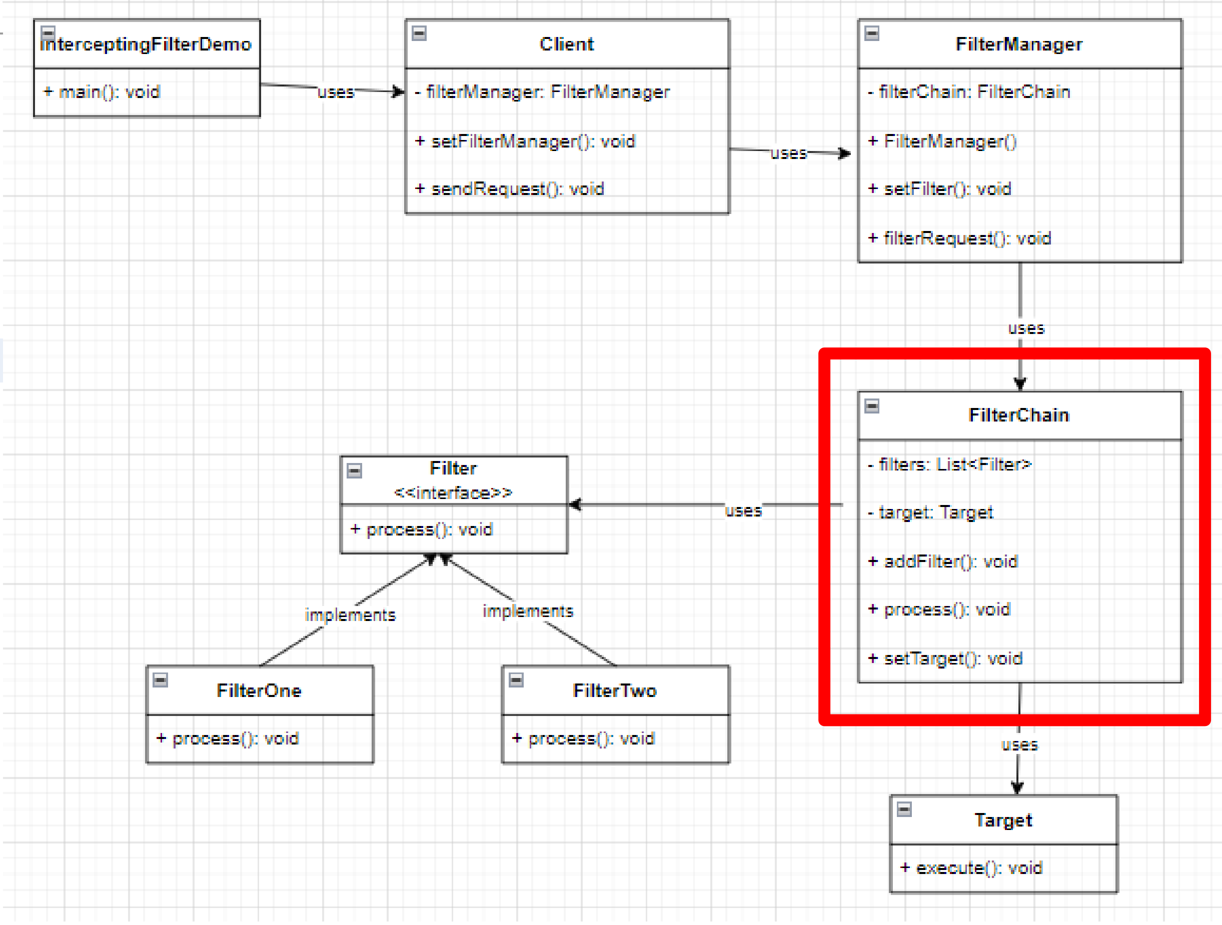


```java
public class Target {
    public void execute(String request)
    {
        System.out.print("Executing request:  "+ request +"\n");
    }
}
```

Implementation

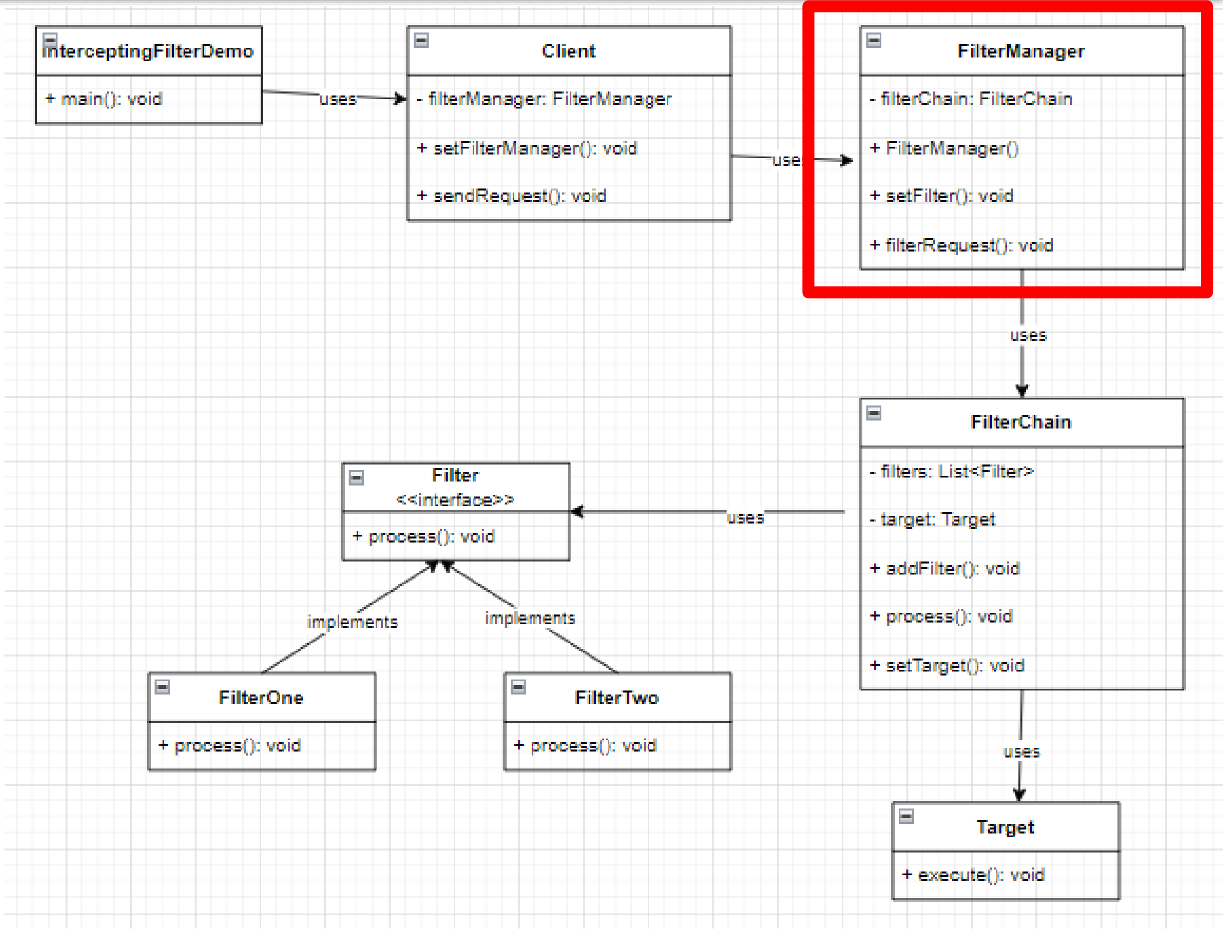


```
public class FilterChain {
    private List<IFilter> filters=new ArrayList<IFilter>();
    private Target target;

    public FilterChain()
    {
    }
    public void addFilter(IFilter filter)
    {
        filters.add( e:filter);
    }
    public void execute(String request)
    {
        //pre-processing
        for(IFilter f: filters)
        {
            f.process(request);
        }
        //execute the Target
        target.execute(request);
    }
    public void setTarget(Target target)
    {
        this.target=target;
    }
```



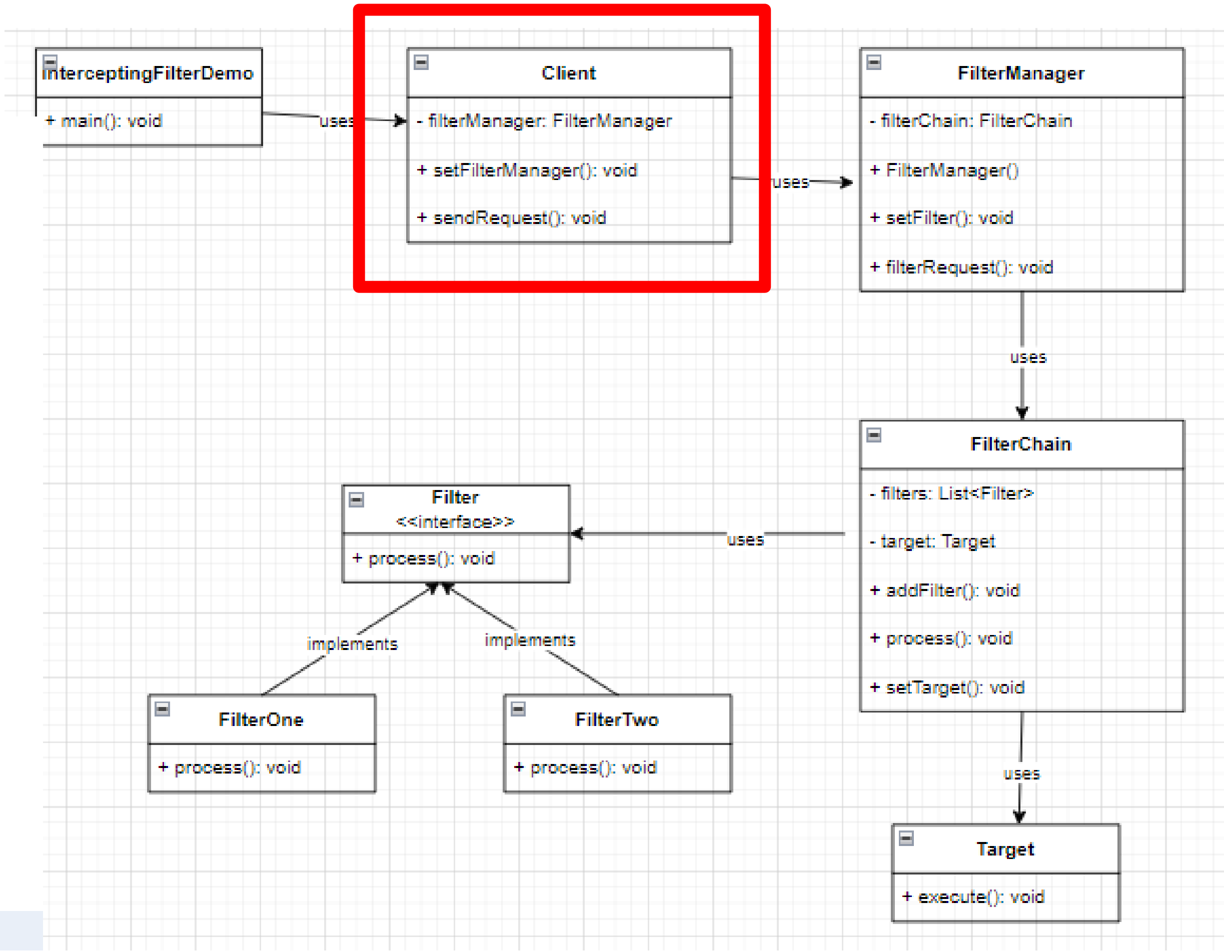
Implementation

```java
public class FilterManager {
    private FilterChain filterChain;

    public FilterManager(Target target)
    {
        filterChain=new FilterChain();
        filterChain.setTarget(target);
    }

    public void setFilter(IFilter filter)
    {
        filterChain.addFilter(filter);
    }

    public void filterRequest(String request)
    {
        filterChain.execute(request);
    }
}
```

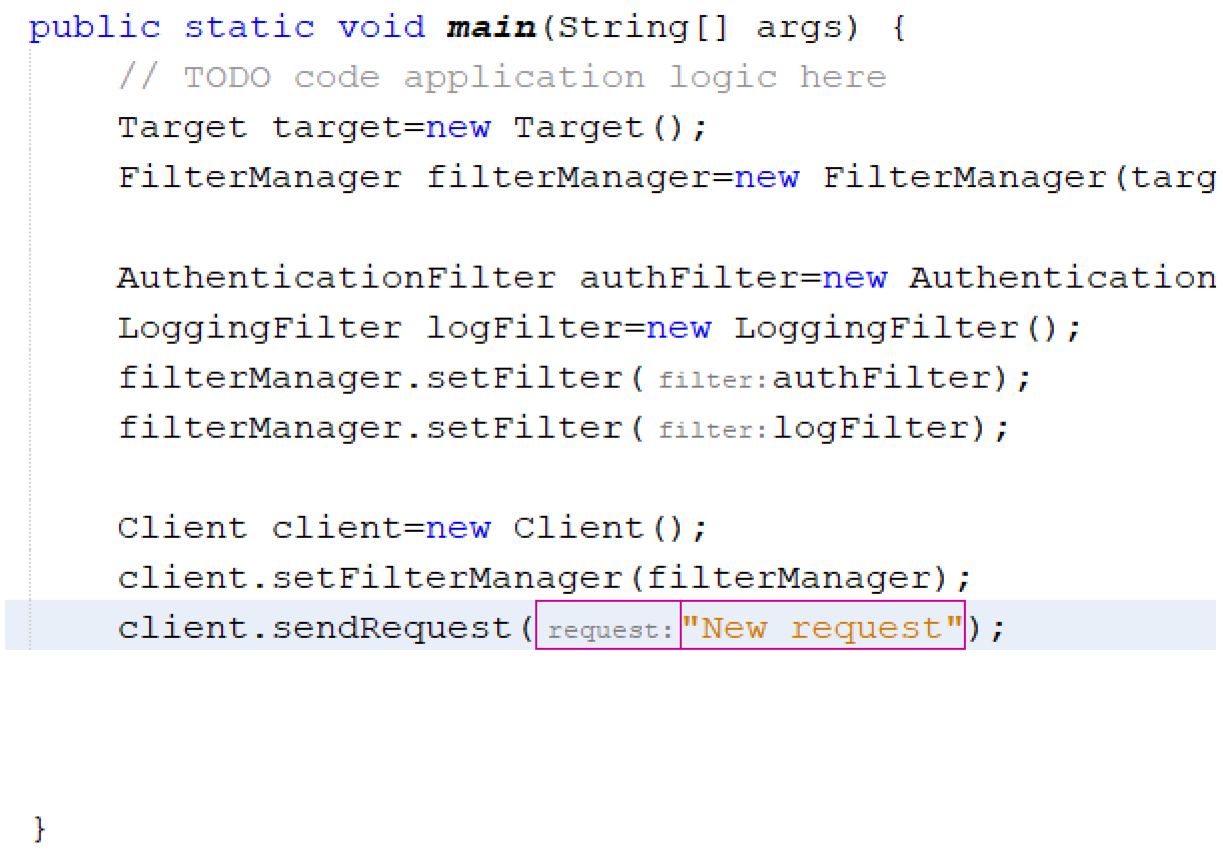


Implementation

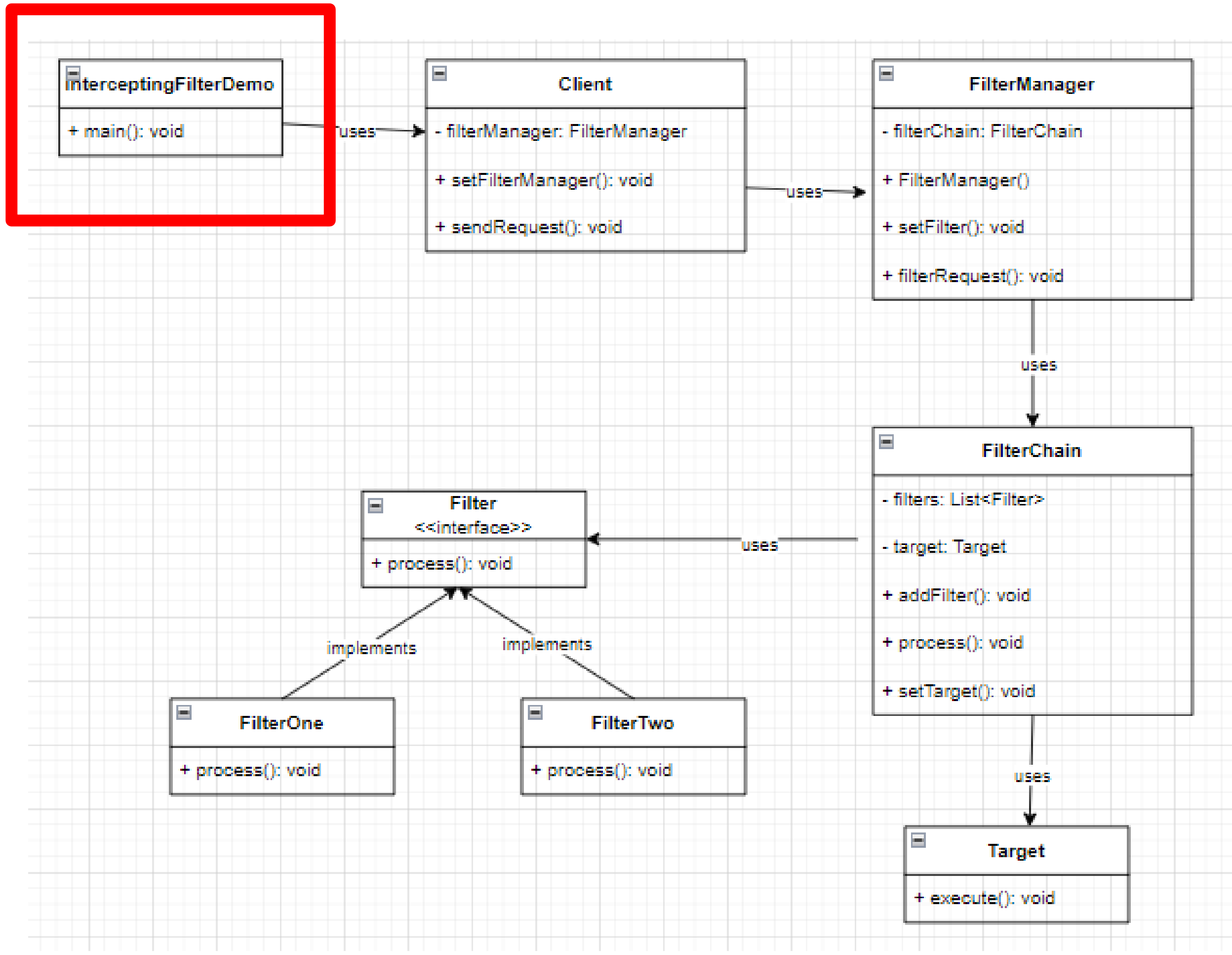
```java
public class Client {
    private FilterManager filterManager;

    public Client()
    {
    }

    public void setFilterManager(FilterManager filterManager)
    {
        this.filterManager=filterManager;
    }
    public void sendRequest(String request)
    {
        this.filterManager.filterRequest(request);
    }

}
```



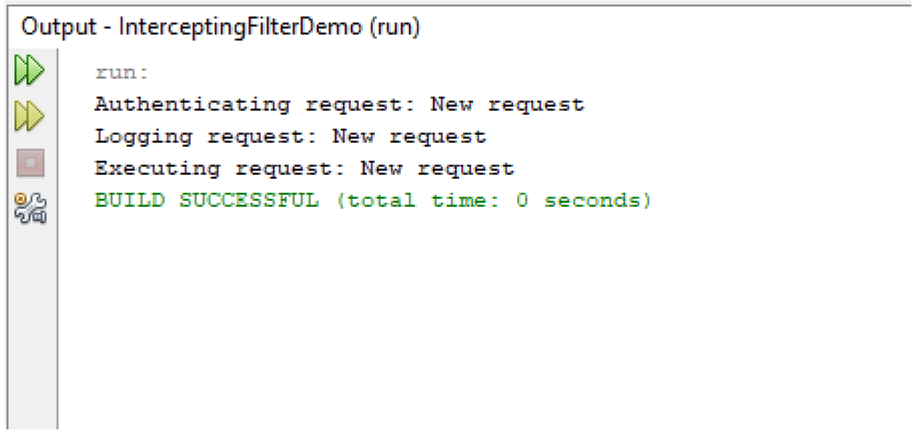
Implementation



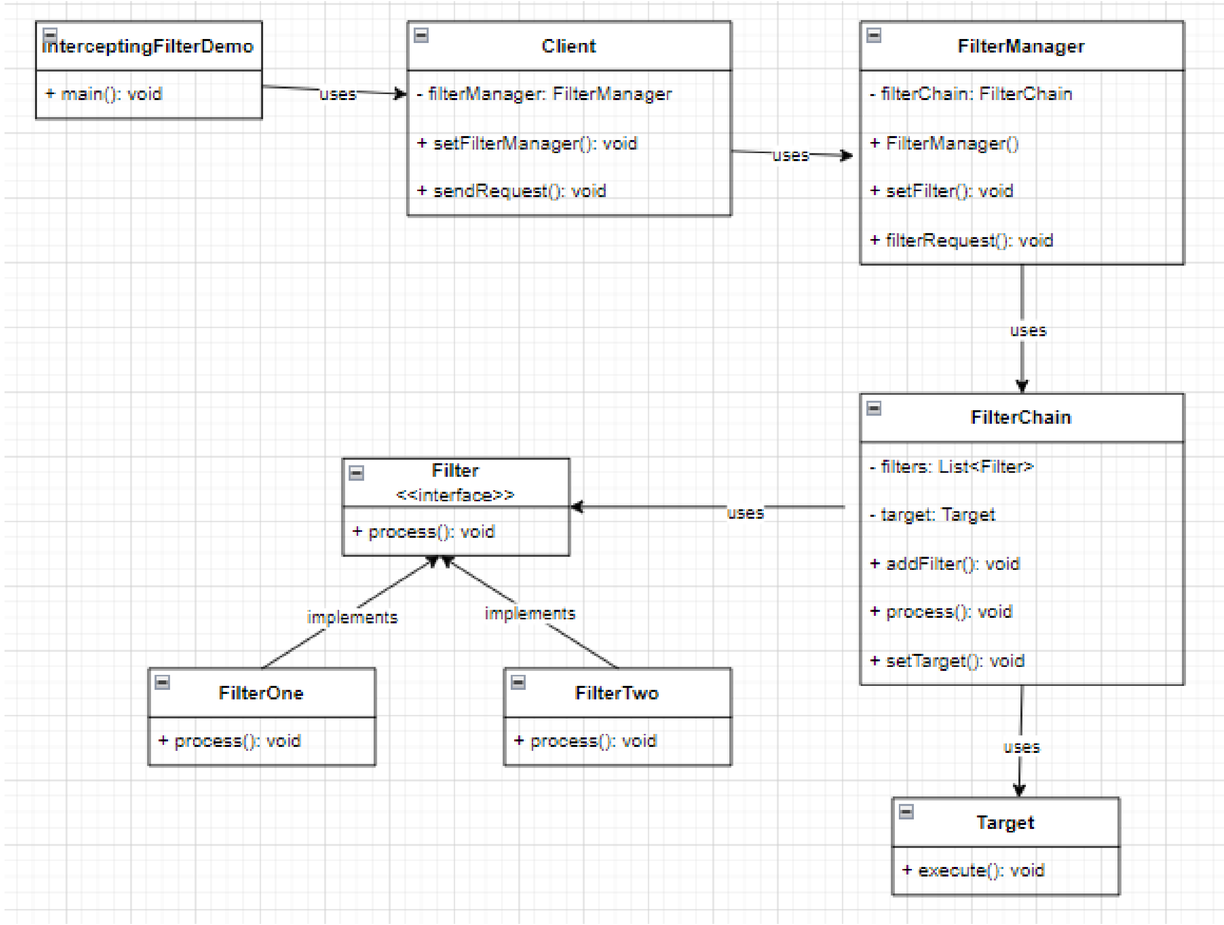
```
public static void main(String[] args) {
    // TODO code application logic here
    Target target=new Target();
    FilterManager filterManager=new FilterManager(targ

    AuthenticationFilter authFilter=new Authentication
    LoggingFilter logFilter=new LoggingFilter();
    filterManager.setFilter( filter:authFilter);
    filterManager.setFilter( filter:logFilter);

    Client client=new Client();
    client.setFilterManager(filterManager);
    client.sendRequest( request:"New request");


}
```



Implementation


Output - InterceptingFilterDemo (run)

```
run:
Authenticating request: New request
Logging request: New request
Executing request: New request
BUILD SUCCESSFUL (total time: 0 seconds)
```



Benefits of the Filter Design Pattern

Separation of Concerns: The Intercepting Filter pattern promotes the separation of concerns by encapsulating cross-cutting functionality within individual filters. Each filter focuses on a specific concern, such as authentication or logging, allowing for modular and maintainable code.

Flexibility and Extensibility: The Intercepting Filter pattern offers flexibility and extensibility. New filters can be easily added to the filter chain to introduce new functionalities or modify existing ones. The modular nature of filters allows for easy customization and adaptation to changing requirements.

Code Decoupling: By intercepting requests and responses, the Intercepting Filter pattern decouples the core processing logic from the cross-cutting concerns. The core logic does not need to be aware of the specific filters being applied, promoting loose coupling and improving code maintainability.

Chapter 3

Chapter 3. Architectural Software Patterns in Java EE

The Interceptor Pattern (AOP)

- Aspect-oriented Programming (AOP)

- Cross-cutting concerns

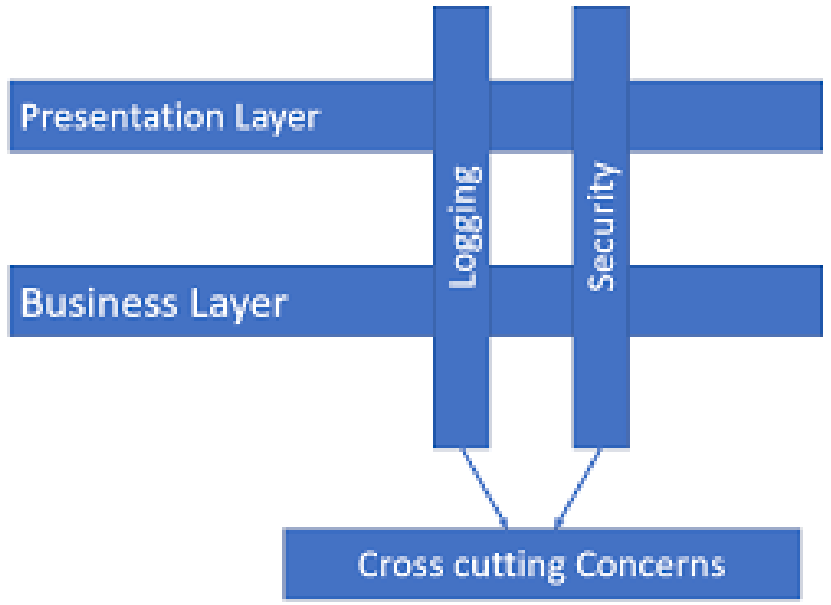
- Separate business and non-business concerns

- AOP uses code injection

- Intercepts method calls

Cross-cutting concerns

- **Some functionality that needed in many places but not related to application’s business logic**
- **Example:** role-based security before every business method in your application



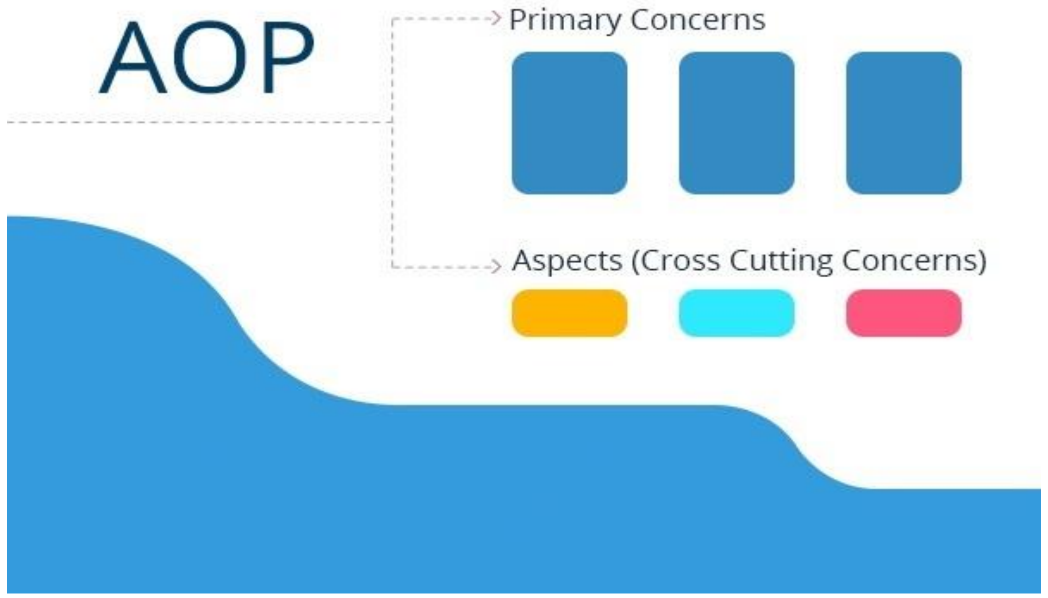
Cross-cutting concerns

- **Required for any application but not necessary from the business point of view.**
- **The following are examples:**

  - Logging and tracing
  - Transaction management
  - Security
  - Caching
  - Error handling
  - Performance monitoring

What is Aspect-Oriented Programming?

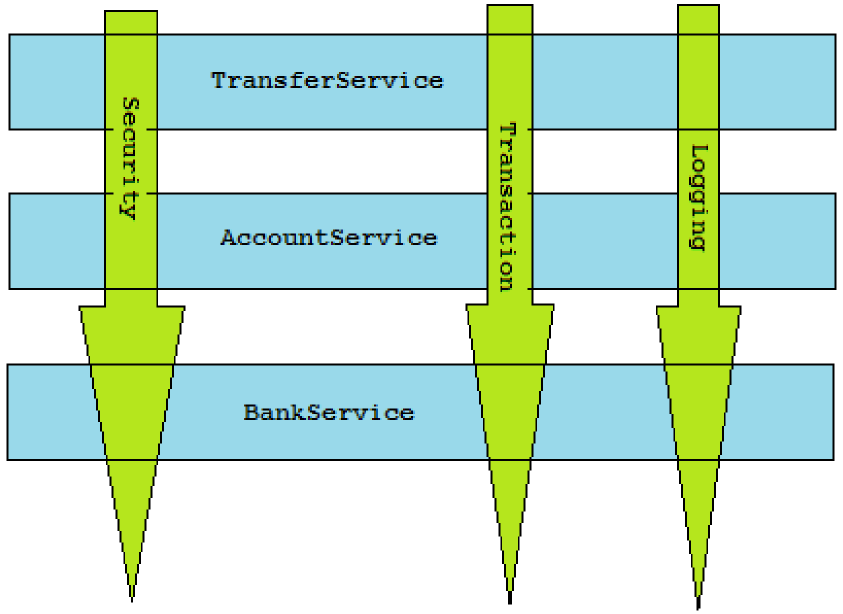
- AOP enables modularization of cross-cutting concerns
- While Object-oriented programming (OOP) has class and object as key element, Aspect-Oriented Programming (AOP) has aspect as key element.
- Aspect modularizes some functionalities across application at multiple places



What is Aspect-Oriented Programming?

- **For examples:**
  - Security is one of cross-cutting concerns, and have to apply at multiple places.

  - Logging and transaction are also cross-cutting concerns.



Problems solved by Aspect-Oriented Programming?

- Failing to modularize cross-cutting concerns lead to 2 main problems:

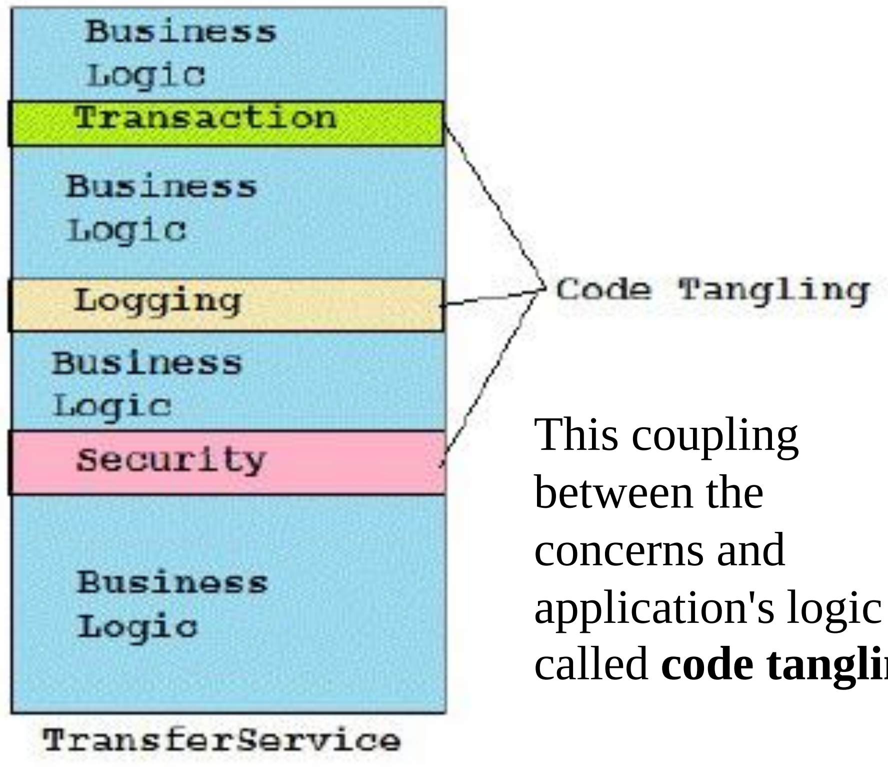
  A. Code tangling

  B. Code scattering

Code tangling

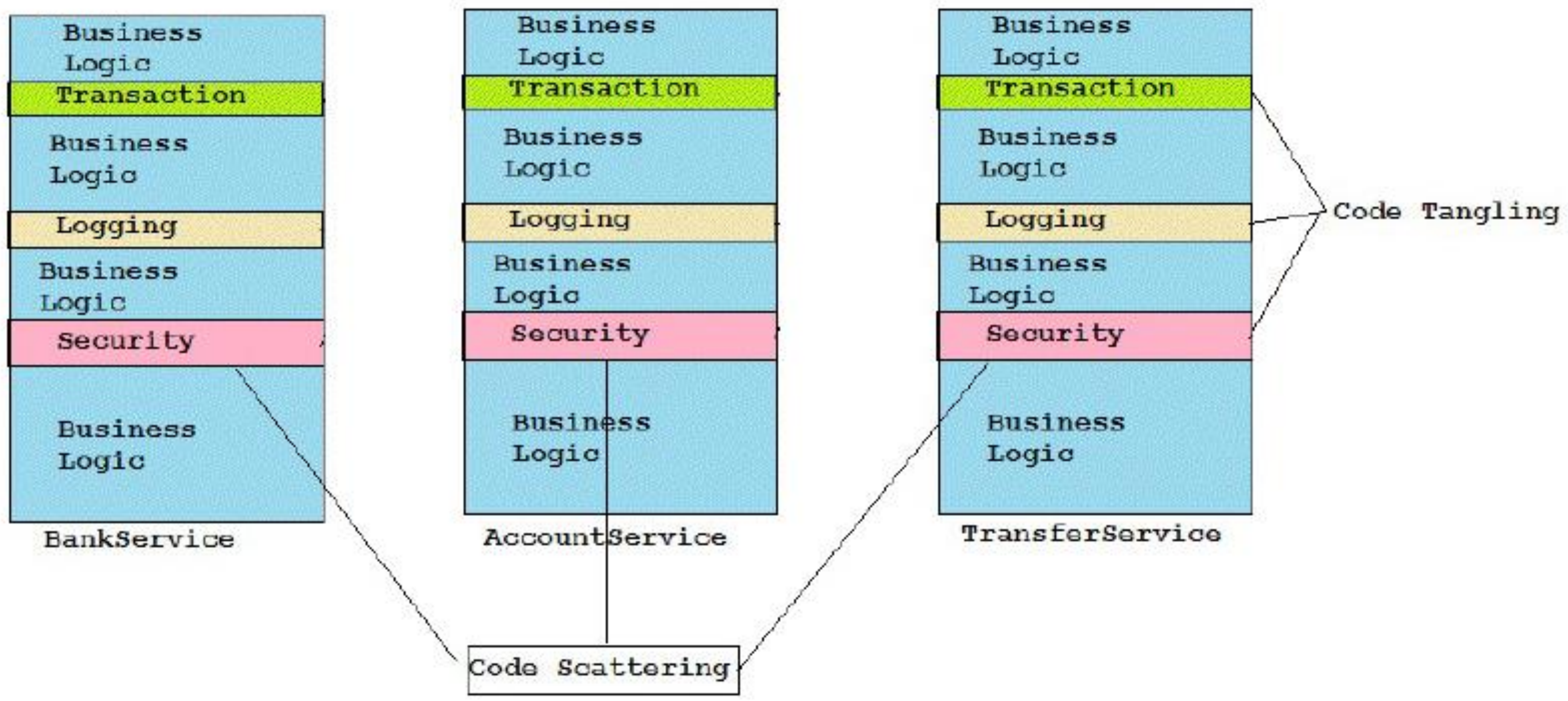
```
public class TransferServiceImpl implements TransferService {
  public void transfer(Account a, Account b, Double amount) {
    //Security concern start here
    if (!hasPermission(SecurityContext.getPrincipal()) {
      throw new AccessDeniedException();
    }
    //Security concern end here

    //Business logic start here
    Account aAct = accountRepository.findByAccountId(a);
    Account bAct = accountRepository.findByAccountId(b);
    accountRepository.transferAmount(aAct, bAct, amount);
    ...
  }
}
```

security concern code (highlighted) is mixing with application's business logic code



This coupling between the concerns and application's logic is called **code tangling**

Code scattering

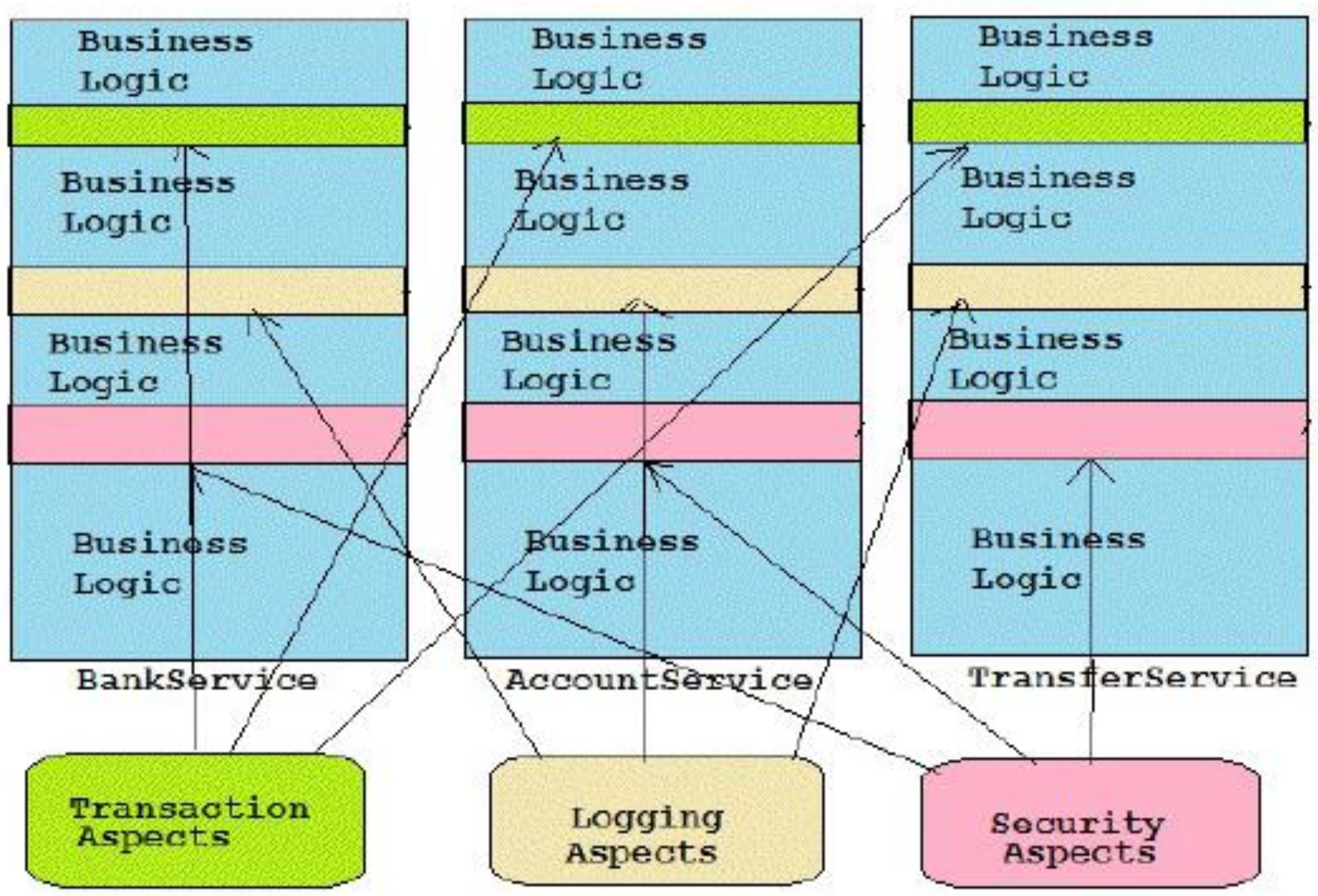


How AOP Works to solve problems

AOP allows you to keep cross-cutting concern logic separate from the mainline application logic.

- write aspects to implement your cross-cutting concerns.

- and then add these aspects that is, cross-cutting behaviors to the right places into your application



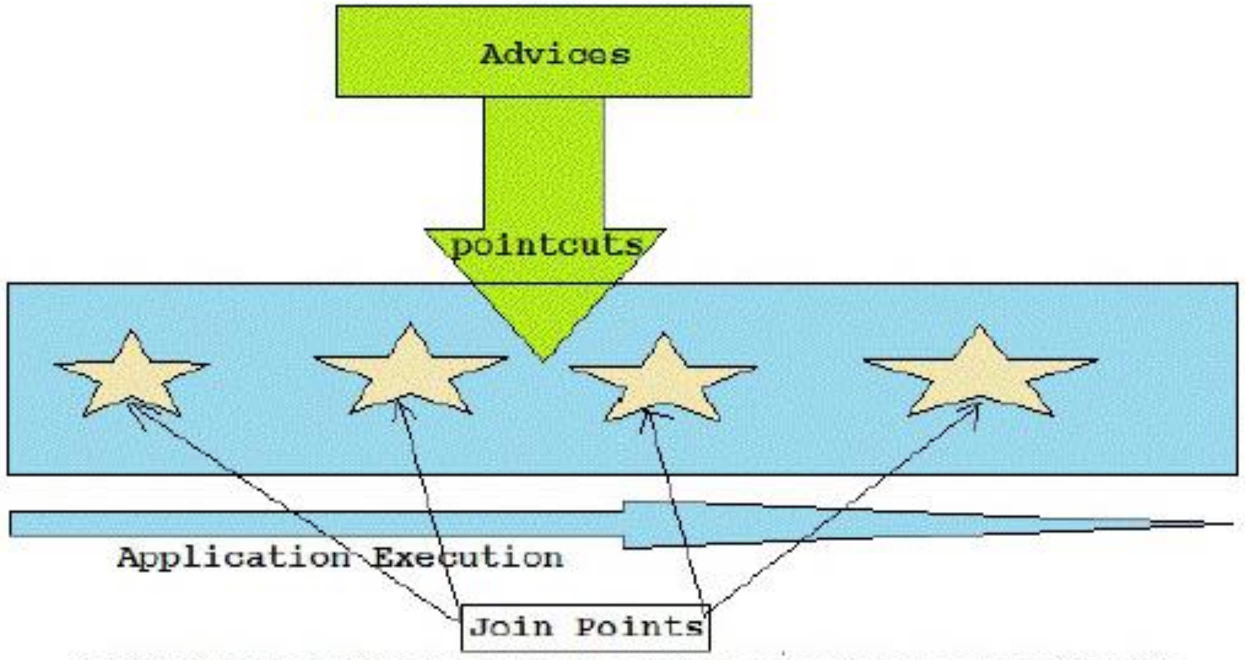
AOP Terminology and Concepts

1. **Advice:** cross-cutting concern
   - **Before**: advice's job executed before business method
   - **After**: advice's job executed after business meth
   - **After-returning**: advice's job executed after business method complete without exception
   - **After- throwing**: advice's job executed after business method exits with exception
   - **Around**: surround business method, advice's jol invoked before and after business method.

2. **Joinpoint:** places in application to apply advice

3. **Pointcut:** an expression to select one or more Joinpoints in the application

4. **Aspect:** a module to encapsulate pointcuts and advice



An AOP's functionality (advice) is woven into a program's execution at one or more join points.

Enabling Service Integration (Forces)

- ➢ A framework should allow the integration of additional services without requiring modifications to its core architecture
- ➢ The integration of application-specific services into a framework should not ...
  - affect existing framework components
  - require changes to the design or implementation of existing applications
- ➢ Applications that use a framework may need to monitor and control its behavior

Enabling Service Integration (Solution)

- Allow applications to **extend a framework transparently** by registering “out-of-band” services with the framework via predefined interfaces (*interceptor callback interfaces*)

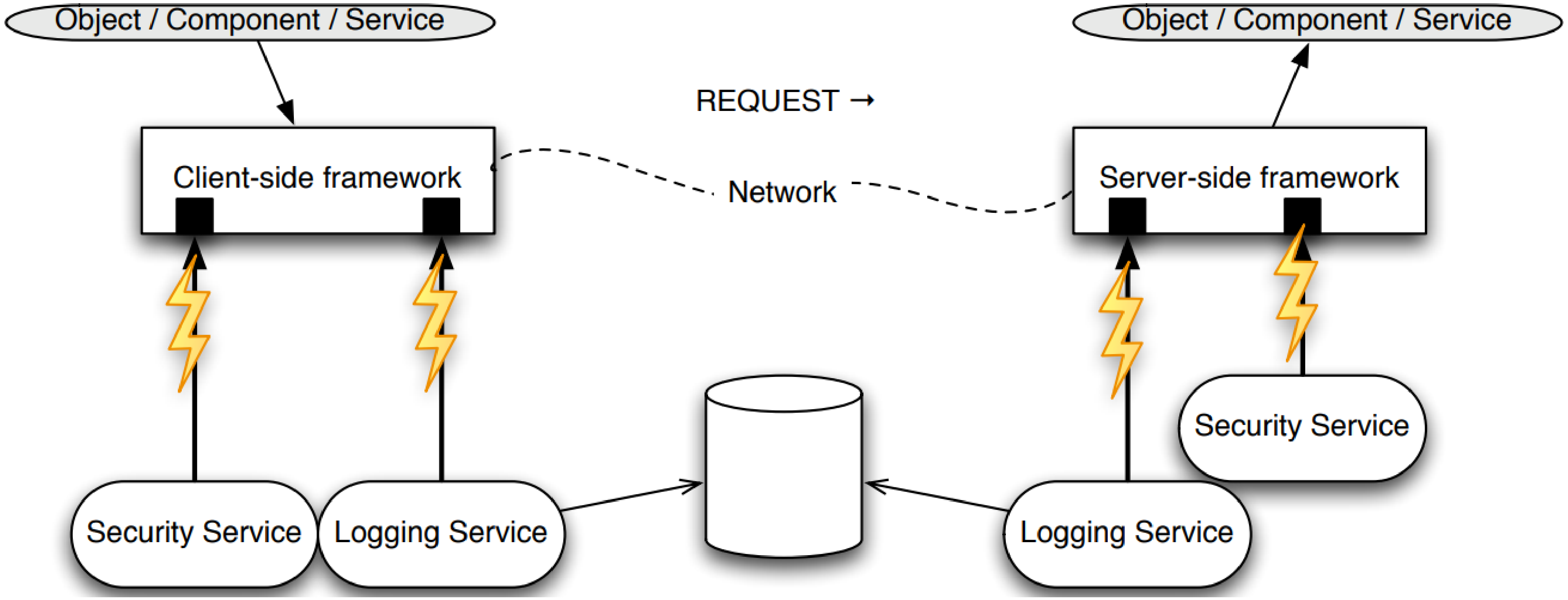
- Trigger these services when “certain” events occur

(... i.e., when application relevant events occur)

The Interceptor Pattern (AOP)

The interceptor architectural pattern allows services to be added transparently to a framework and triggered automatically when certain events occur.

Examples: logging, security, load balancing…



The Interceptor Pattern (AOP)

Logging example

- Intercept every method call

- Log out method name and parameter values

- Log method execution time

The Interceptor Pattern (AOP)

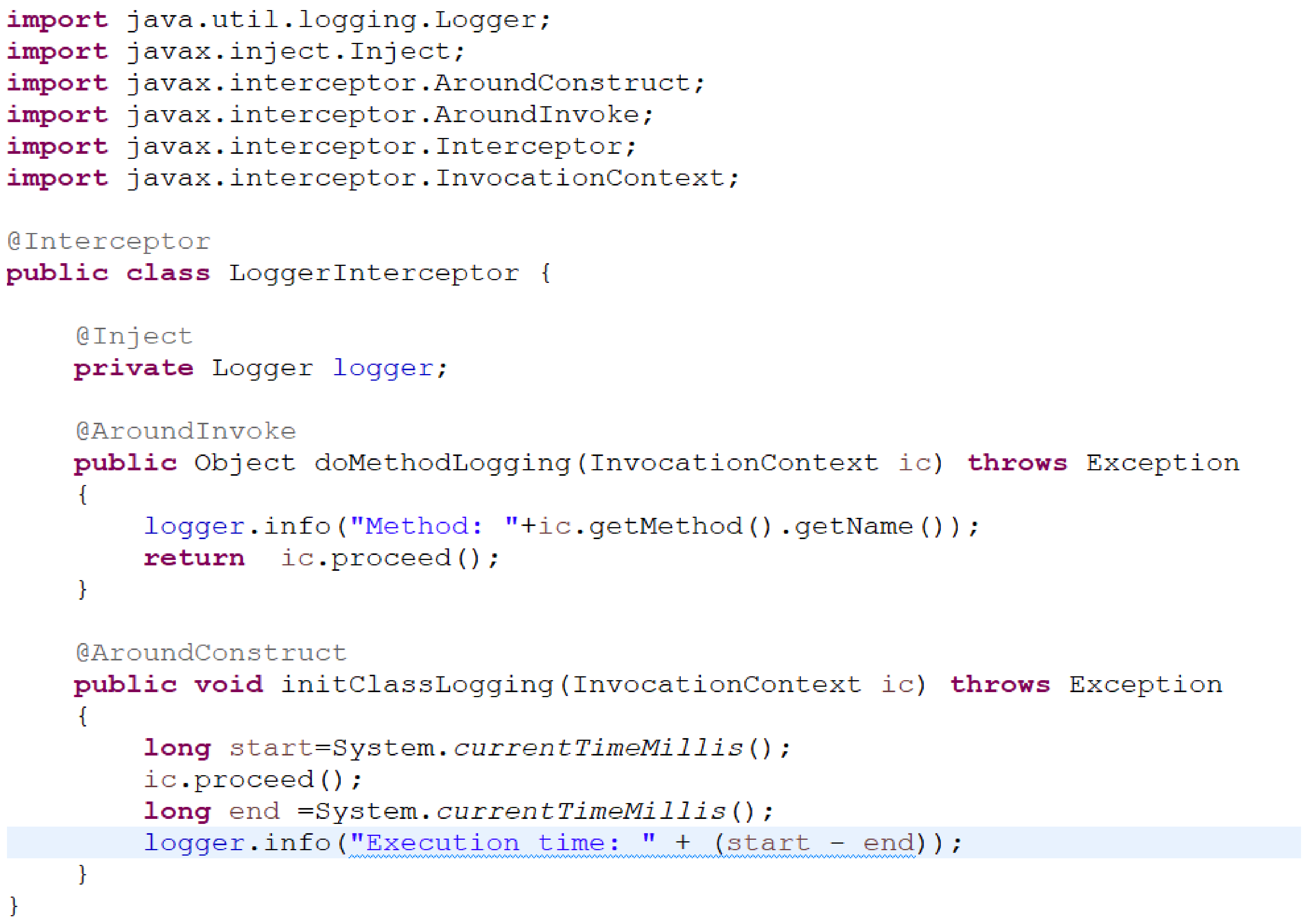
- Mixing of concerns resulting in bloat

- Interceptor separates out code

The Interceptor Pattern (AOP)

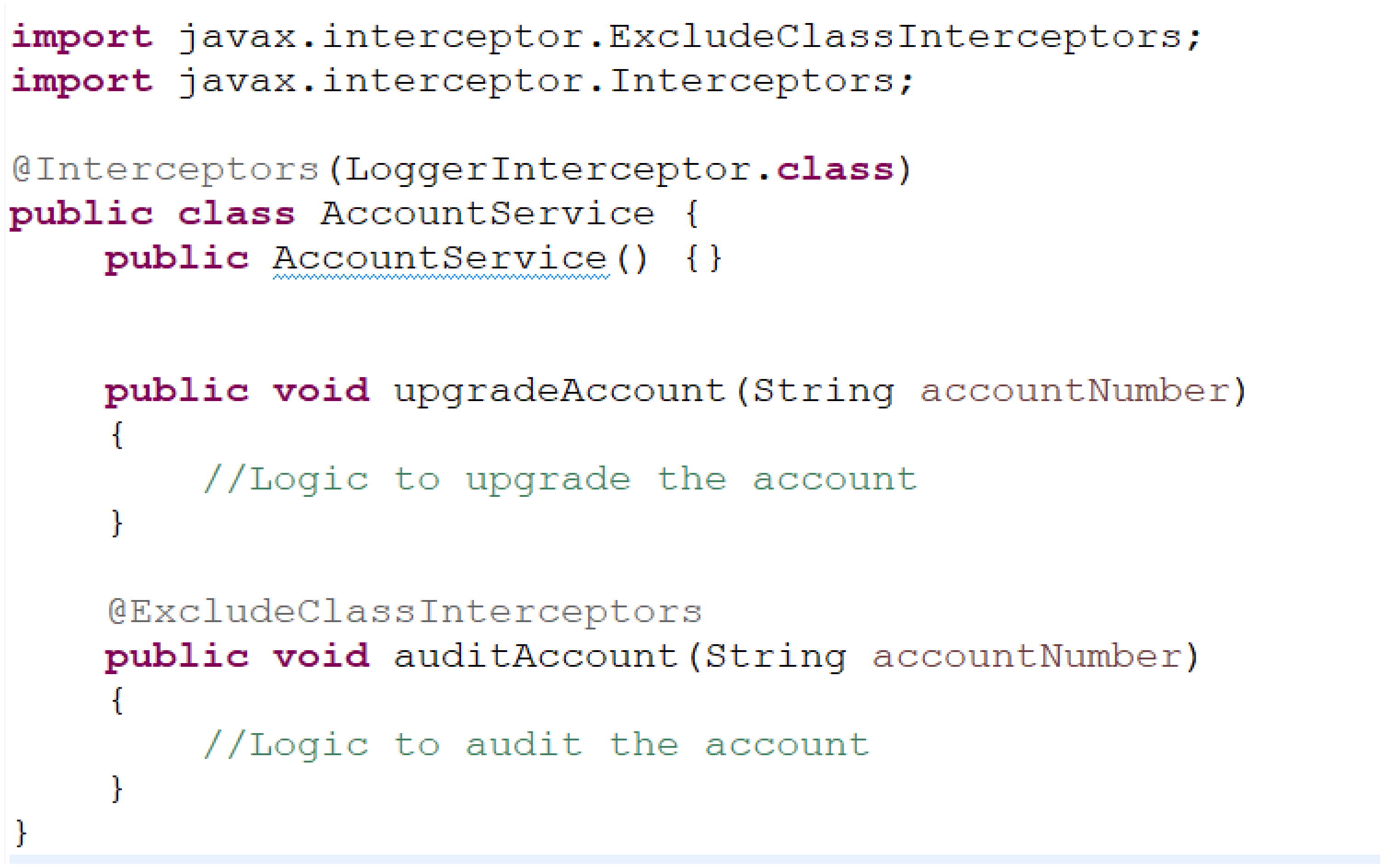
```
@Interceptor
Public class LoggerInterceptor {

        @AroundInvoke
        Private Object doMethodLogging (InvocationContext ic) throws Exception {
                //Before method call logic
                Object returnValue=ic.proceed();
                //After method returns logic
                return returnValue;
        }

}
```

Implementation of The Interceptor Pattern



```java
import java.util.logging.Logger;
import javax.inject.Inject;
import javax.interceptor.AroundConstruct;
import javax.interceptor.AroundInvoke;
import javax.interceptor.Interceptor;
import javax.interceptor.InvocationContext;

@Interceptor
public class LoggerInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    public Object doMethodLogging(InvocationContext ic) throws Exception
    {
        logger.info("Method: "+ic.getMethod().getName());
        return  ic.proceed();
    }

    @AroundConstruct
    public void initClassLogging(InvocationContext ic) throws Exception
    {
        long start=System.currentTimeMillis();
        ic.proceed();
        long end =System.currentTimeMillis();
        logger.info("Execution time: " + (start - end));
    }
}
```

Implementation of The Interceptor Pattern



```java
import javax.interceptor.ExcludeClassInterceptors;
import javax.interceptor.Interceptors;

@Interceptors(LoggerInterceptor.class)
public class AccountService {
    public AccountService() {}


    public void upgradeAccount(String accountNumber)
    {
        //Logic to upgrade the account
    }

    @ExcludeClassInterceptors
    public void auditAccount(String accountNumber)
    {
        //Logic to audit the account
    }
}
```

Chapter 3

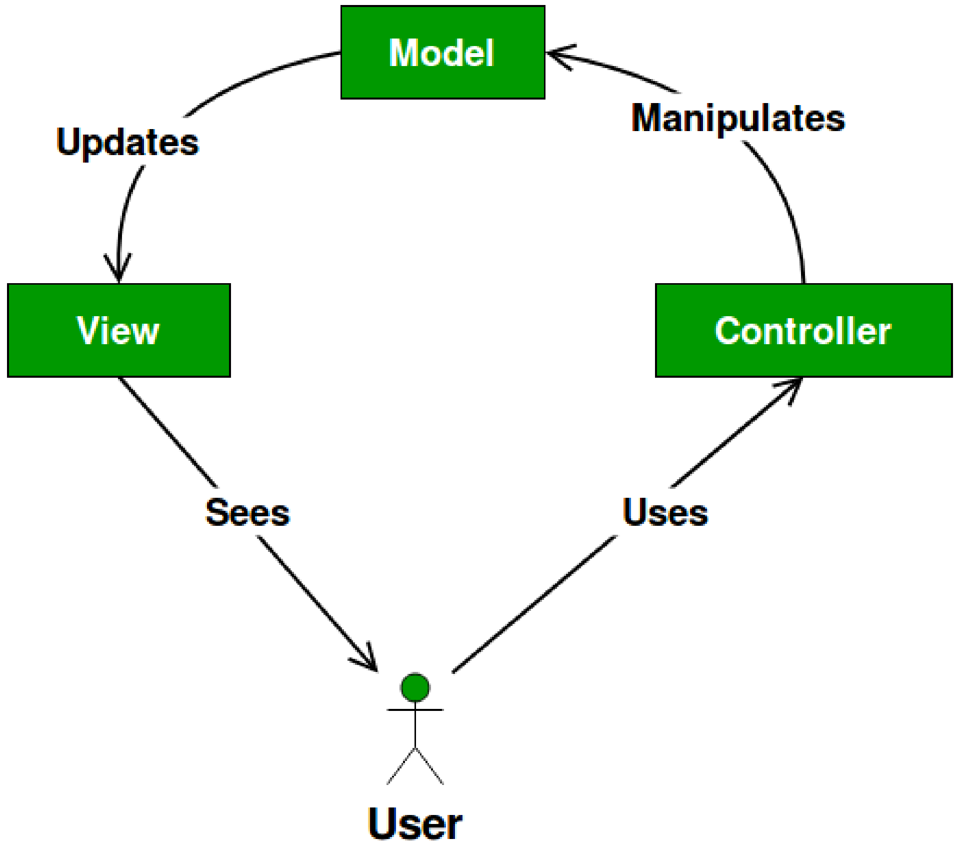
Chapter 3. Architectural Software Patterns in Java EE

MVC - Description

- the most-used in web app development

- Built on the philosophy of **separations of concerns**

- Separate the code into three distinct parts: the **model**, the **view**, and the **controller**



What is Model-View-Controller?

Model-view-controller is an architectural pattern that helps you separate the code in your web applications into three main components:

- **Model**

  The model represents the data in your application

- **View**

  The view is the user interface or what the user sees and interacts with

- **Controller**

  The controller manages the communication between the view and the model. It takes the data from the model and communicates it to the view for display.

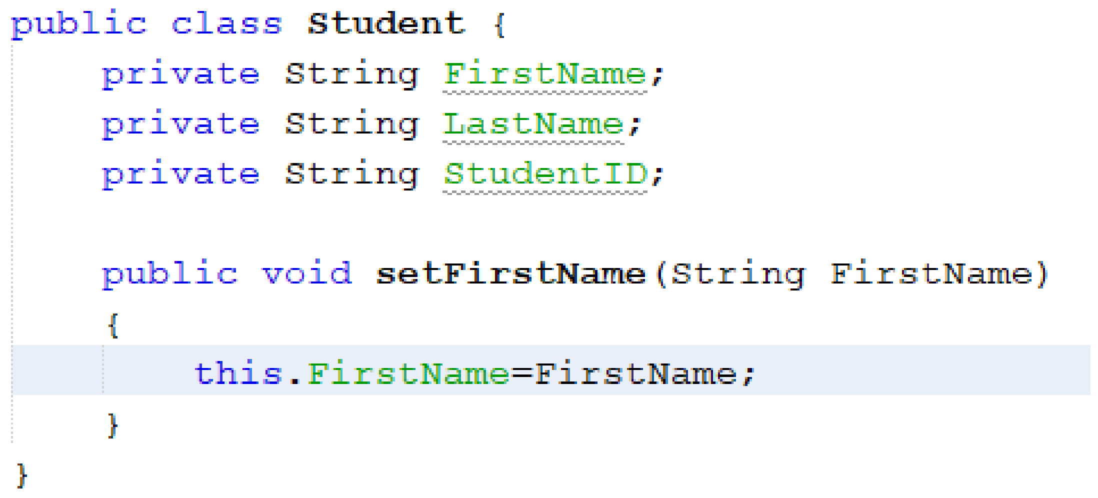
  In the same vein, the controller also takes the changed data (due to user interaction or something else) and communicates it back to the model.

The Model

The model represents the data in your application

The model contains all the data-related logic that the user works with, like the schemas and interfaces of a project, the databases, and their fields

For example: - a student class
- to carry data of student
- to retrieve the student information from the database, manipulate or update their record in the database, or use it to render data.



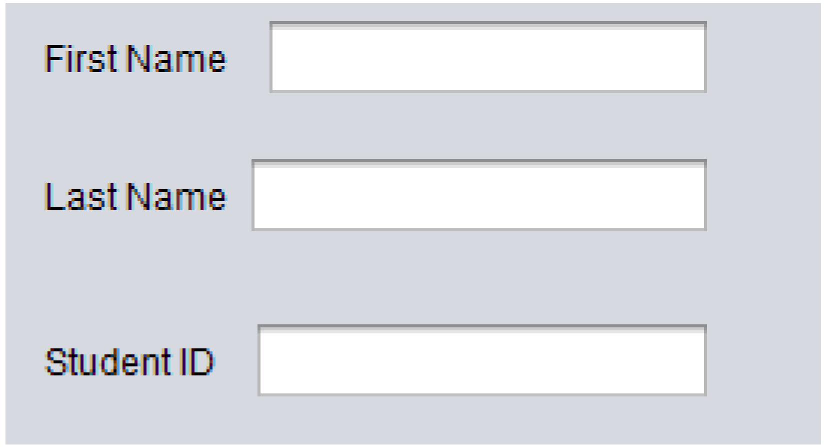
```
public class Student {
    private String FirstName;
    private String LastName;
    private String StudentID;

    public void setFirstName(String FirstName)
    {
        this.FirstName=FirstName;
    }
}
```

The View

The view is the user interface or what the user sees and interacts with

The view contains the UI and the presentation of an application.

For example: the student view will include all the UI components such as text boxes, dropdowns, and other things that the user interacts with.



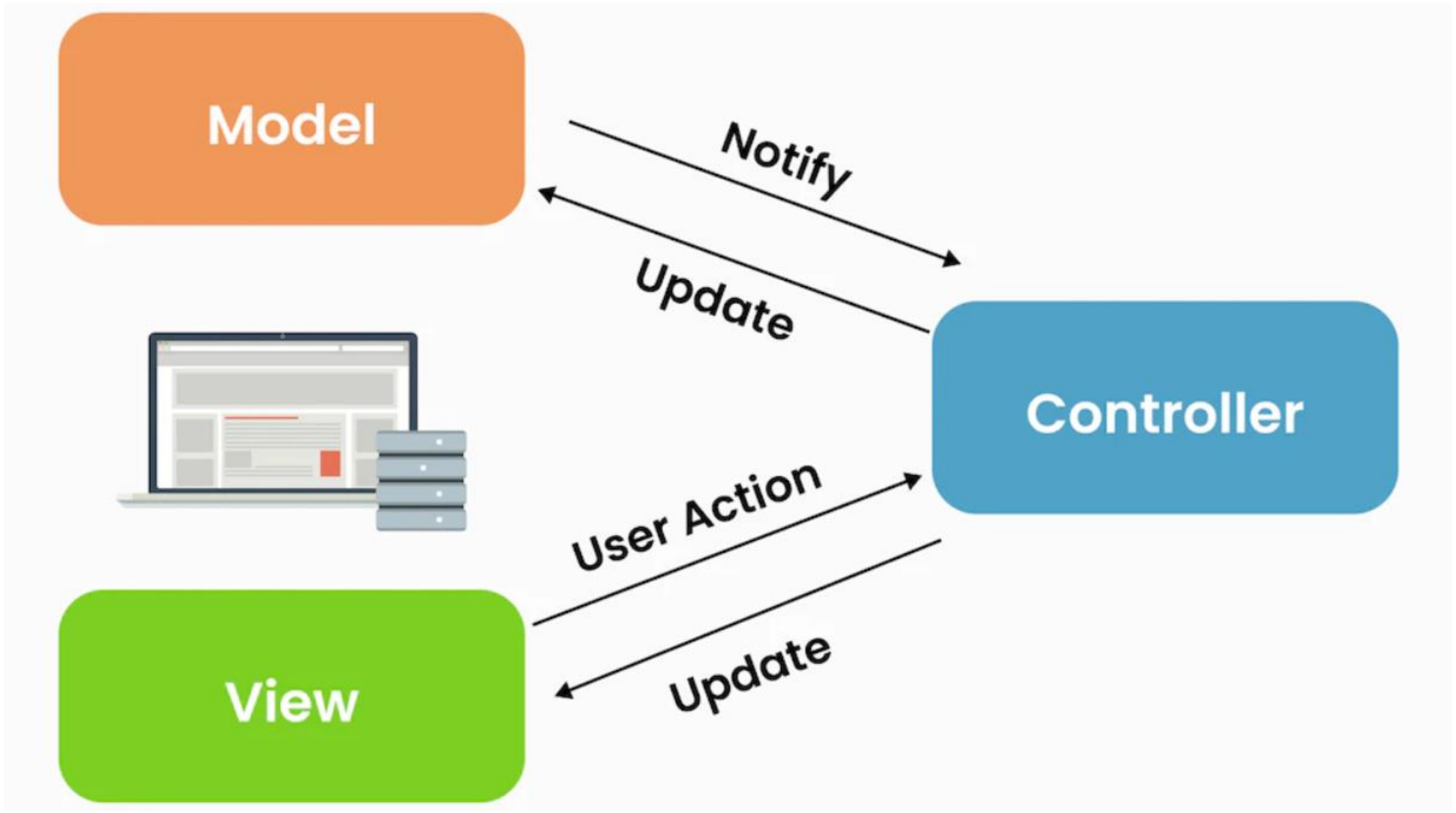
The Controller

And finally, the controller contains all the business-related logic and handles incoming requests. It is the interface between the Model and the View.

For example: the student controller will handle all the interactions and inputs from the student view and update the database using the customer model. The same controller will be used to view the student data.

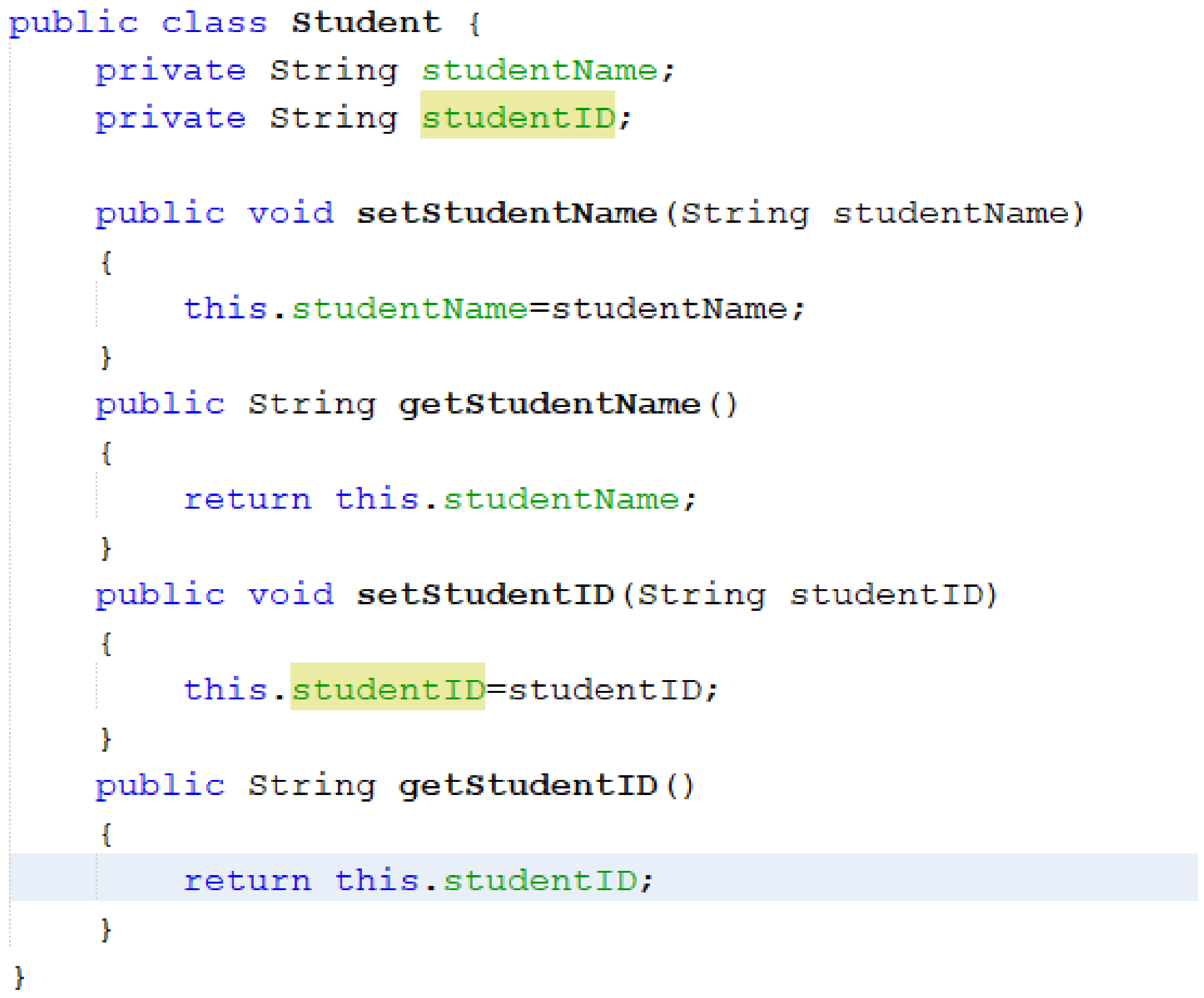
Flow Diagram

Model And **View** Never Talk To Each Other



Implementation of MVC

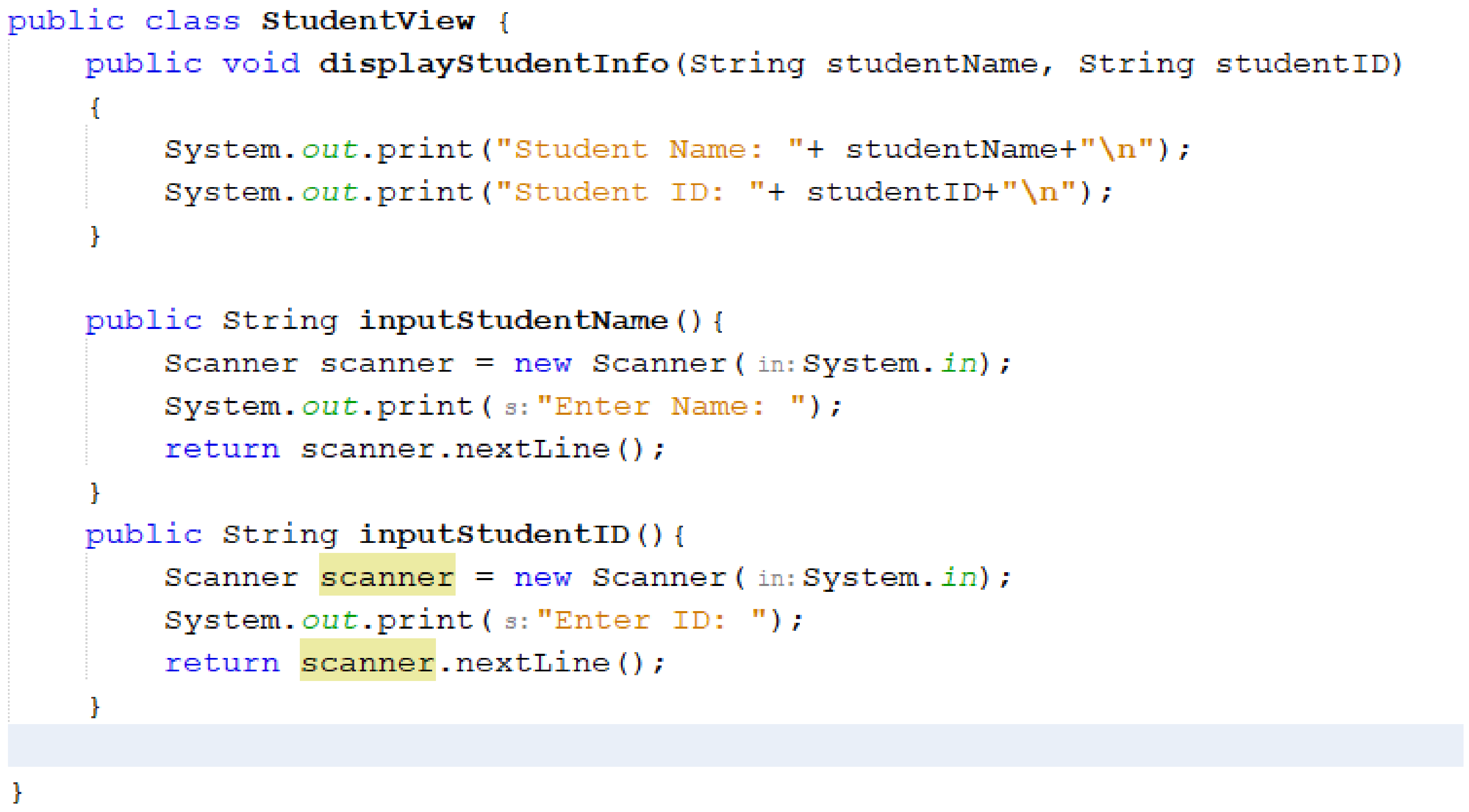
1. Model



```
public class Student {
    private String studentName;
    private String studentID;

    public void setStudentName(String studentName)
    {
        this.studentName=studentName;
    }
    public String getStudentName()
    {
        return this.studentName;
    }
    public void setStudentID(String studentID)
    {
        this.studentID=studentID;
    }
    public String getStudentID()
    {
        return this.studentID;
    }
}
```

Implementation of MVC

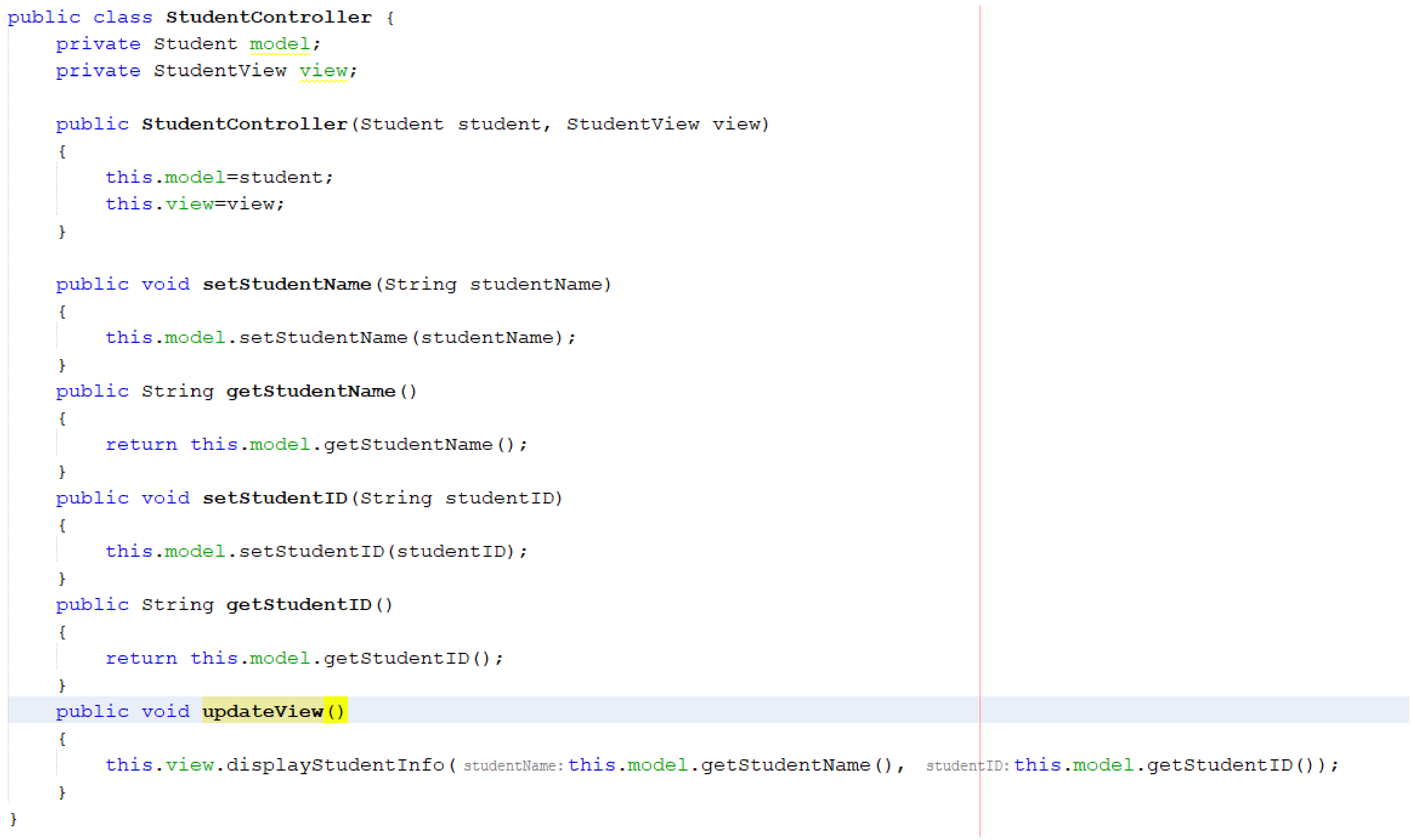
2. View



```
public class StudentView {
    public void displayStudentInfo(String studentName, String studentID)
    {
        System.out.print("Student Name: "+ studentName+"\n");
        System.out.print("Student ID: "+ studentID+"\n");
    }

    public String inputStudentName(){
        Scanner scanner = new Scanner( in: System.in);
        System.out.print( s: "Enter Name: ");
        return scanner.nextLine();
    }
    public String inputStudentID(){
        Scanner scanner = new Scanner( in: System.in);
        System.out.print( s: "Enter ID: ");
        return scanner.nextLine();
    }

}
```

Implementation of MVC

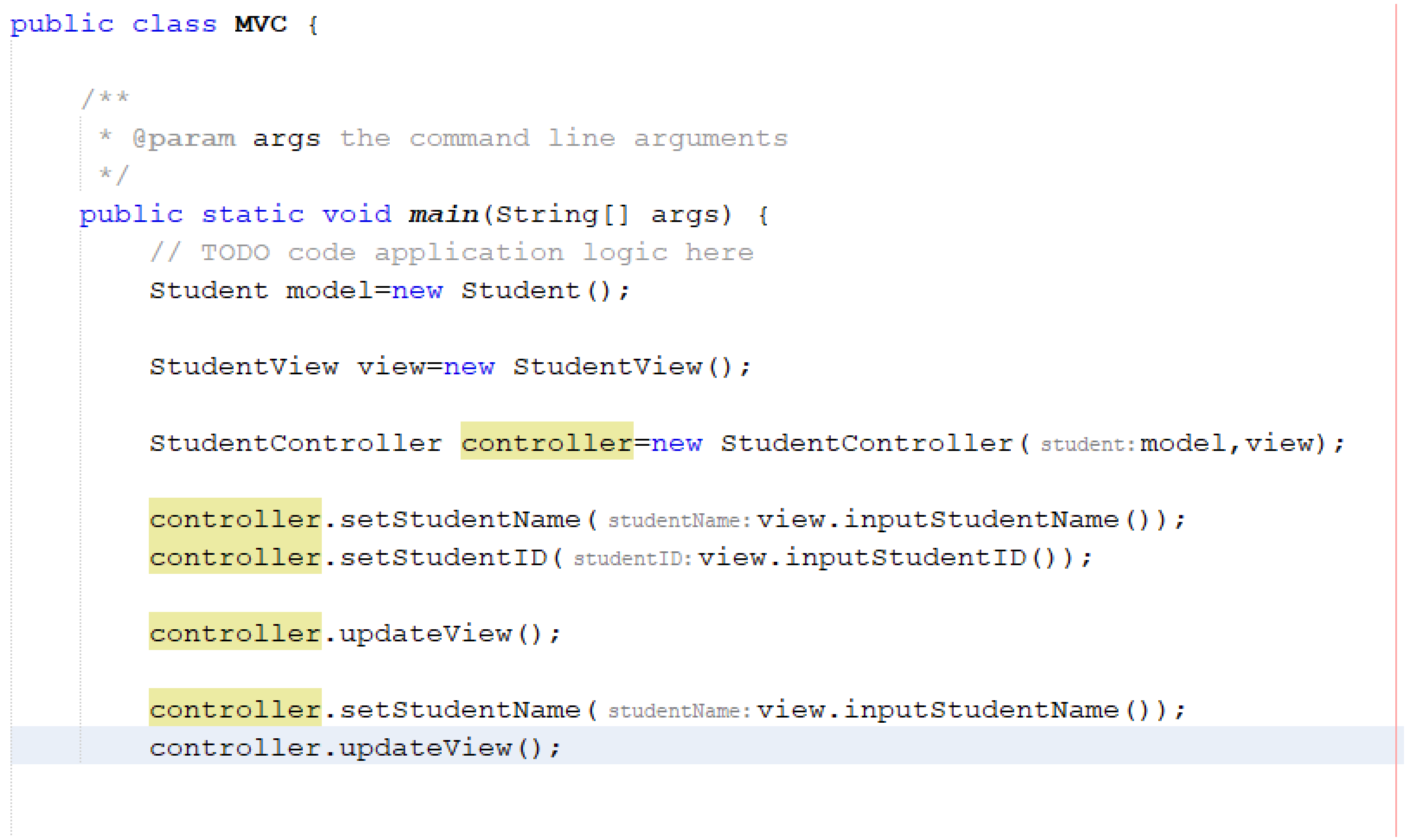
3. Controller



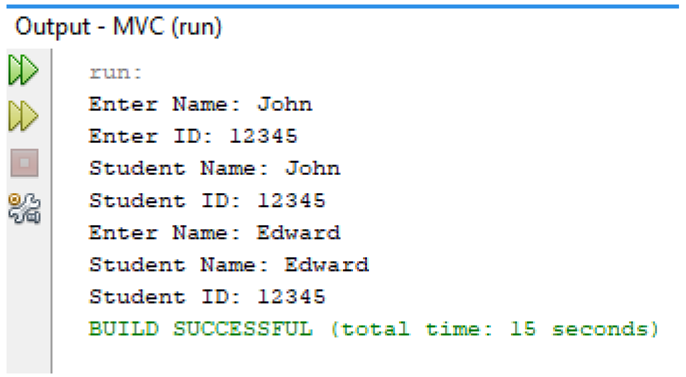
```
public class StudentController {
    private Student model;
    private StudentView view;

    public StudentController(Student student, StudentView view)
    {
        this.model=student;
        this.view=view;
    }

    public void setStudentName(String studentName)
    {
        this.model.setStudentName(studentName);
    }
    public String getStudentName()
    {
        return this.model.getStudentName();
    }
    public void setStudentID(String studentID)
    {
        this.model.setStudentID(studentID);
    }
    public String getStudentID()
    {
        return this.model.getStudentID();
    }
    public void updateView()
    {
        this.view.displayStudentInfo( studentName: this.model.getStudentName(),  studentID: this.model.getStudentID());
    }
}
```

Implementation of MVC

4. Demo



```
public class MVC {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Student model=new Student();

        StudentView view=new StudentView();

        StudentController controller=new StudentController( student: model,view);

        controller.setStudentName( studentName: view.inputStudentName());
        controller.setStudentID( studentID: view.inputStudentID());

        controller.updateView();

        controller.setStudentName( studentName: view.inputStudentName());
        controller.updateView();
```



Output - MVC (run)

```
run:
Enter Name: John
Enter ID: 12345
Student Name: John
Student ID: 12345
Enter Name: Edward
Student Name: Edward
Student ID: 12345
BUILD SUCCESSFUL (total time: 15 seconds)
```

Benefits of the MVC design pattern

1. Helps you organize your code
2. Makes code easy to modify
3. Supports faster development
4. Enables easy planning and maintenance
5. Supports TTD (test-driven development)
6. Organizes code for scaled web applications