الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Week 5

كلية الهندسة المعلوماتية

مقرر بنيان البرمجيات

Practical Concerns

د. رياض سنبل

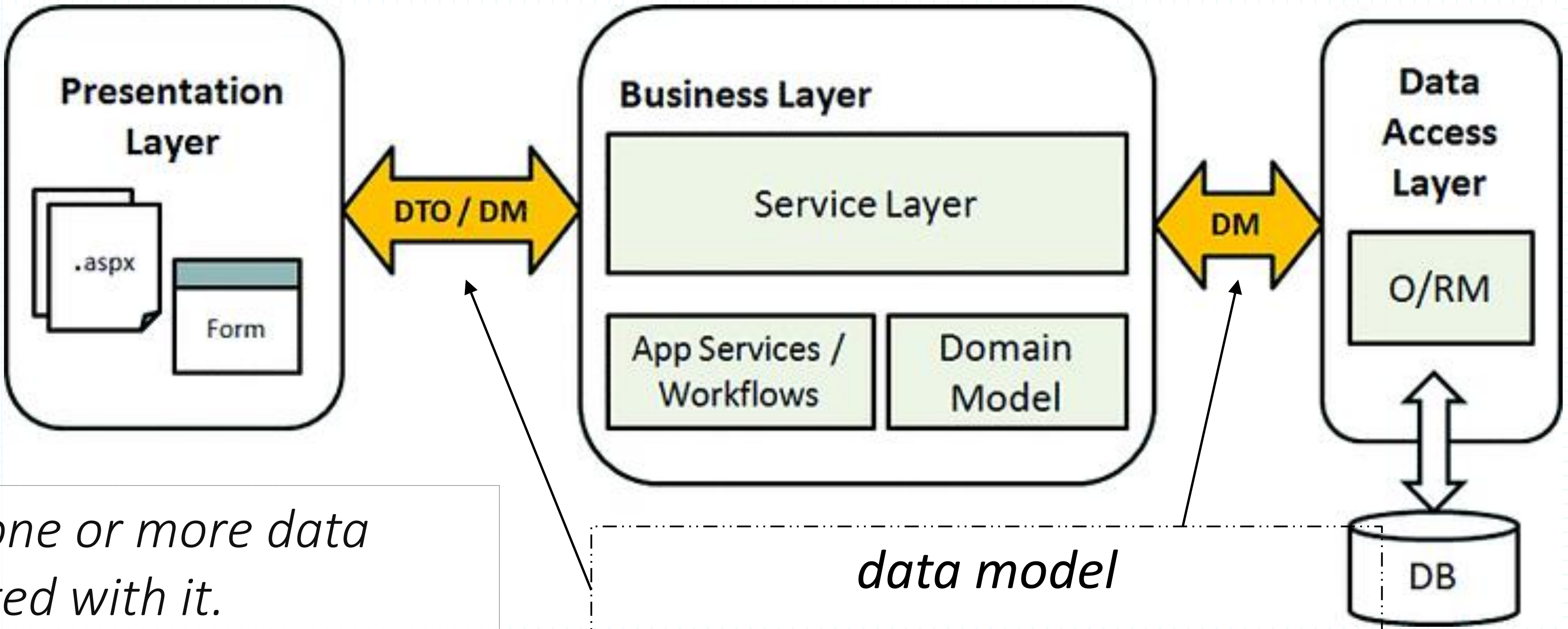
1st Practical Concern

Detailed Structure

HOW TO HANDLE THE COMMUNICATION BETWEEN LAYERS (HOW TO TRANSFER DATA?)

Application data is located in data models

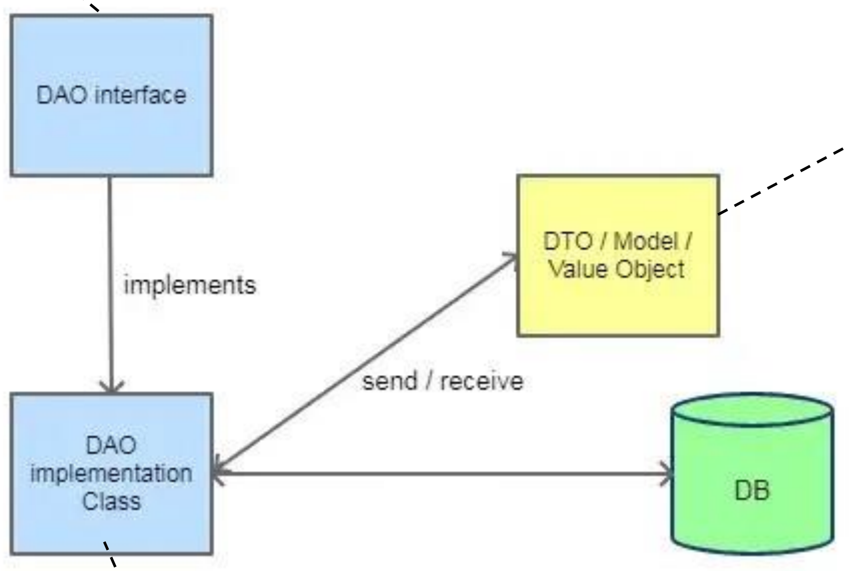
Data Transfer Object (DTO) is a pattern used to transfer data between software components



- *Each layer has one or more data models associated with it.*
- *The layer functionality can be implemented both in separate specialized classes and as methods in data model classes.*

data model
=
a collection of data structures
+
operations for processing them.

```java
interface DeveloperDao {
    public List<Developer> getAllDevelopers();
    public Developer getDeveloper(int DeveloperId);
    public void updateDeveloper(Developer Developer);
    public void deleteDeveloper(Developer Developer);
}
```



```java
// Implementing above defined interface
class DeveloperDaoImpl implements DeveloperDao {

….

}
```

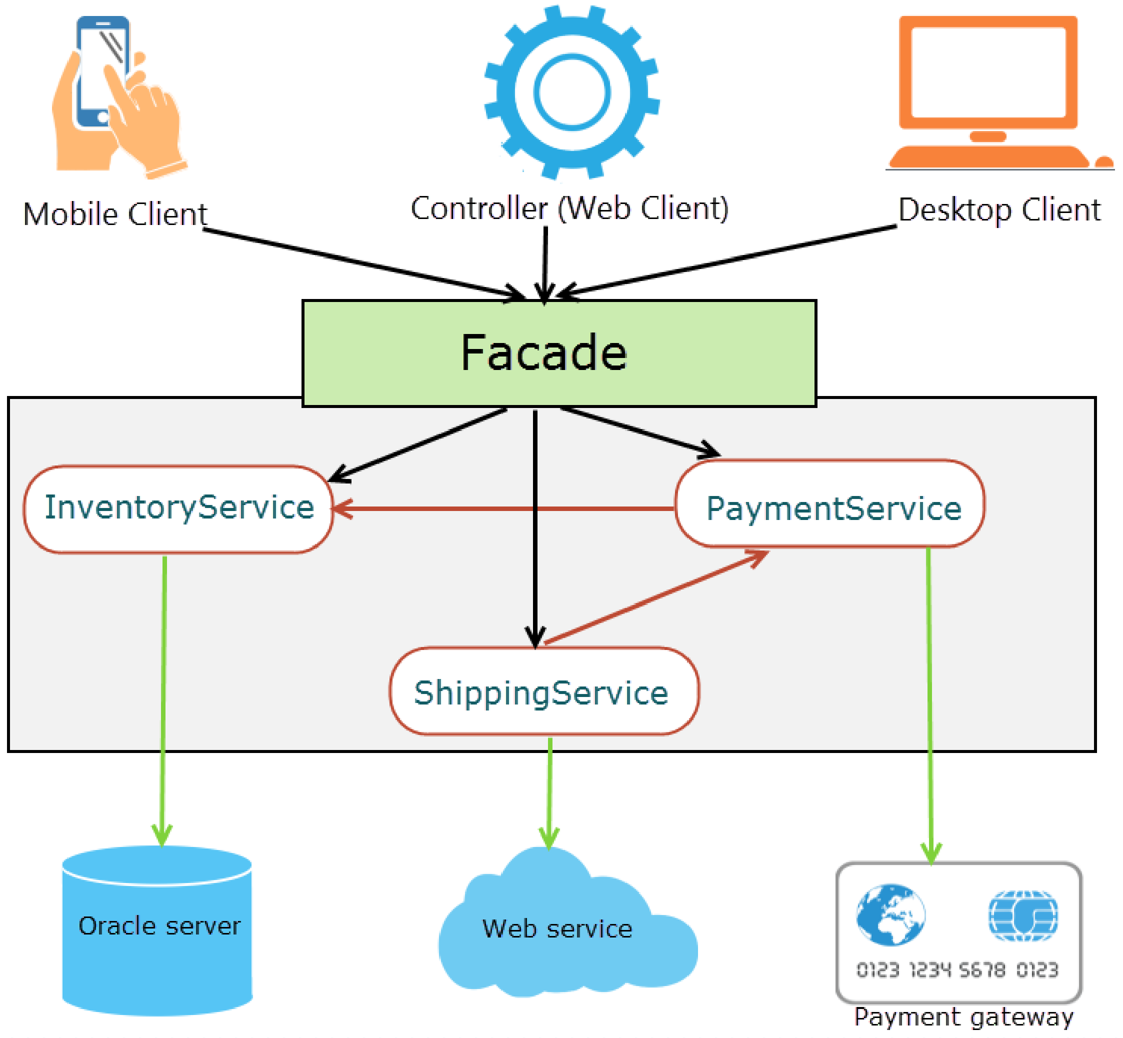
```java
class Developer {
    private String name;
    private int DeveloperId;
    // Constructor of Developer class
    Developer(String name, int DeveloperId)
    {
        this.name = name;
        this.DeveloperId = DeveloperId;
    }

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public int getDeveloperId() { return DeveloperId; }

    public void setDeveloperId(int DeveloperId)
    {
        this.DeveloperId = DeveloperId;
    }
}
```

2st Practical Concern

Detailed Structure

FAÇADE DESIGN PATTERN

Façade Design Pattern



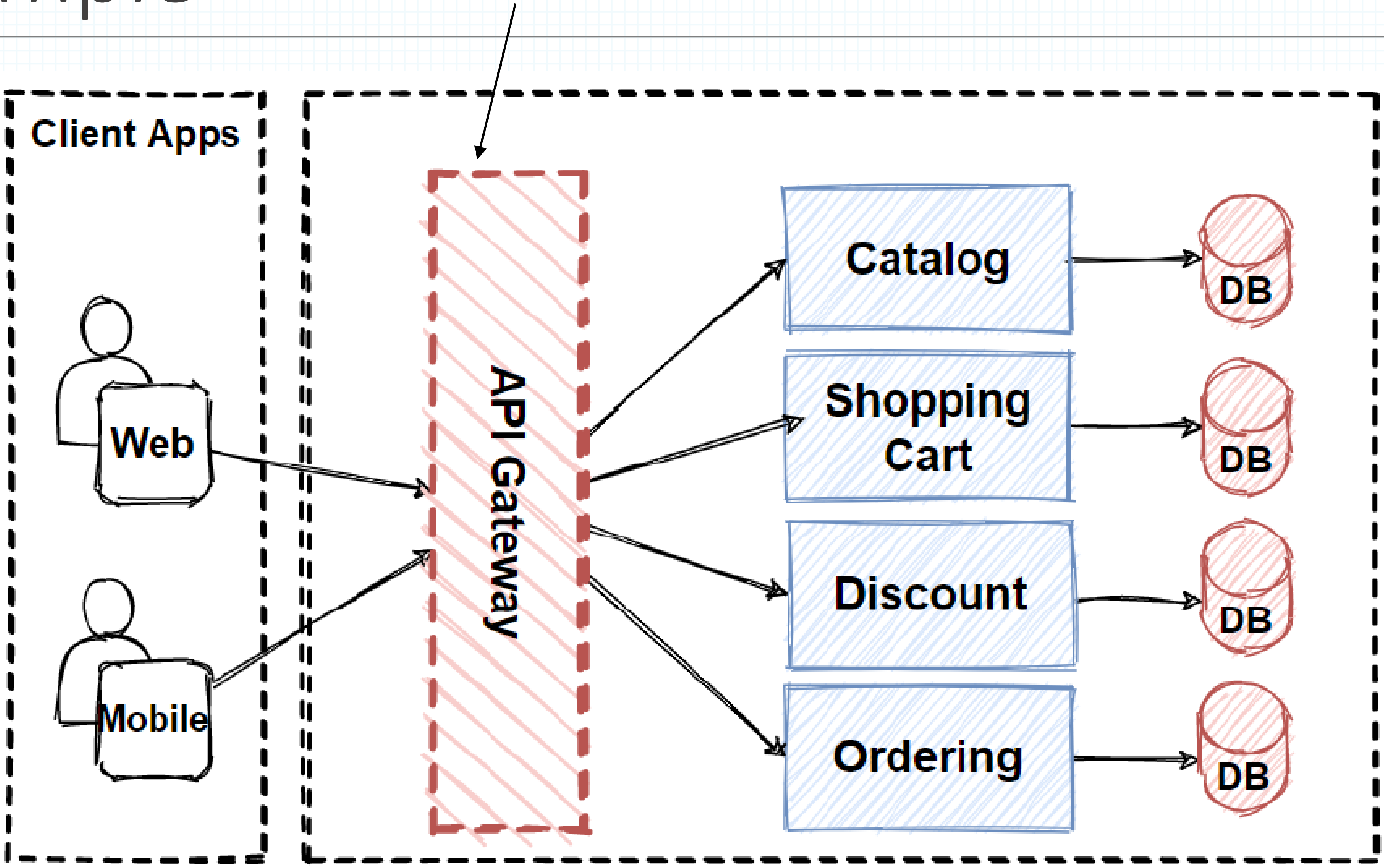
Façade sublayer encapsulates the functionality of the layer's functionality and provides high-level methods or operations that abstract away the complexity

Services implemented by subsystem classes

External Resources

Example

it provides a single entry point to the APIs with encapsulating the underlying system architecture.

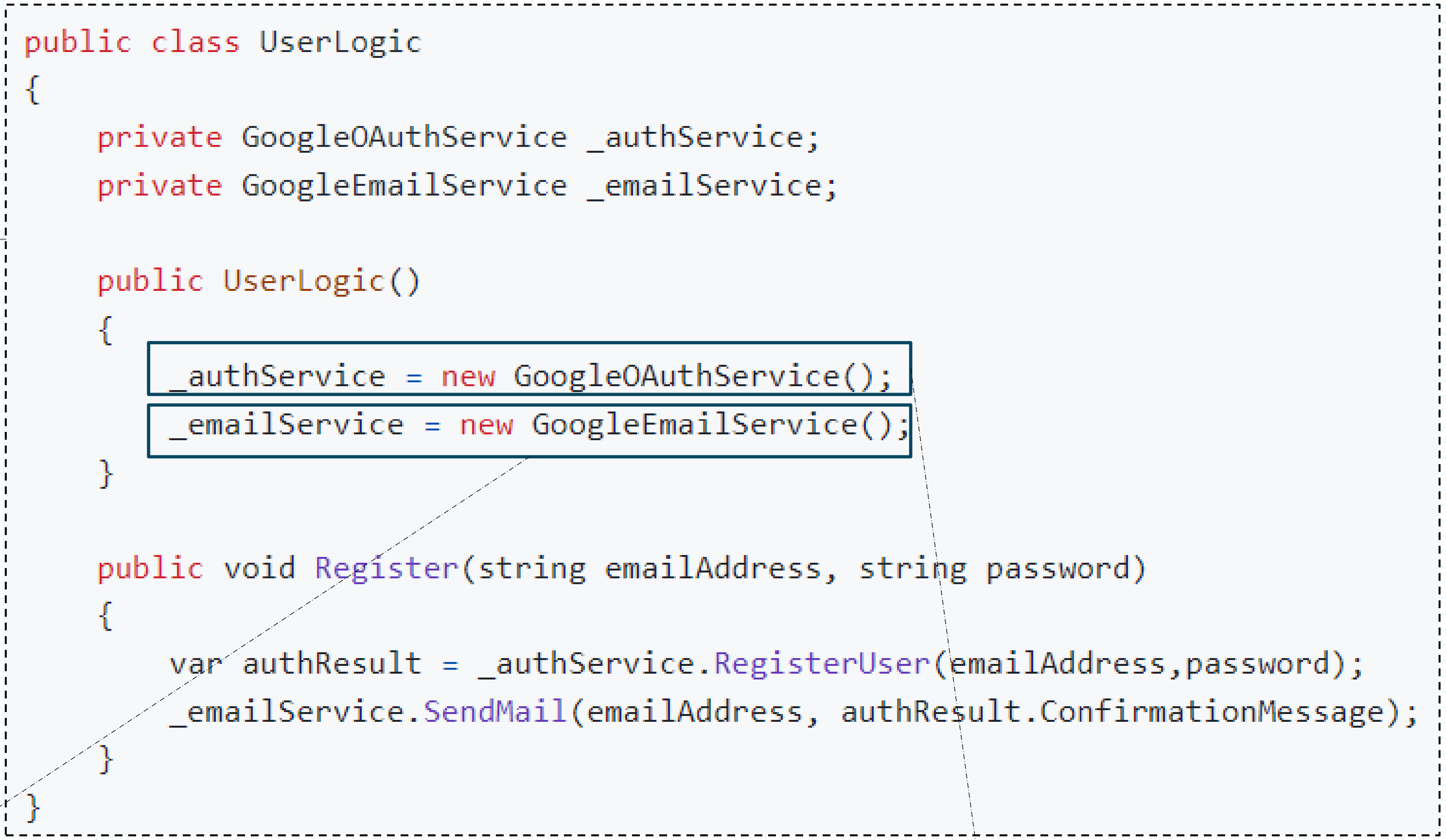


3rd Practical Concern

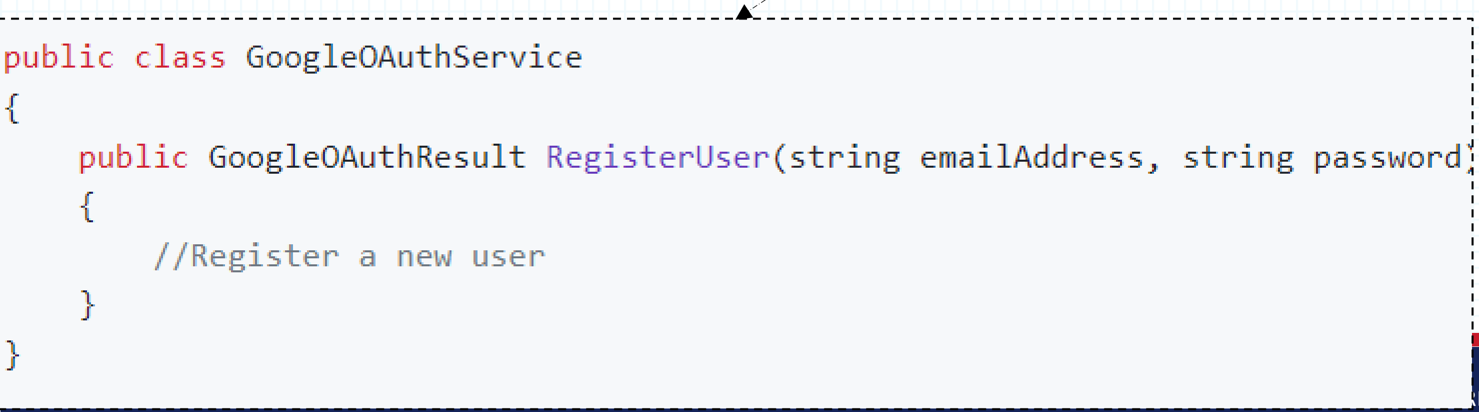
Dependency Injection

Changing a service would imply changing a lot of the codebase, especially if the service has been used in multiple parts of the project.

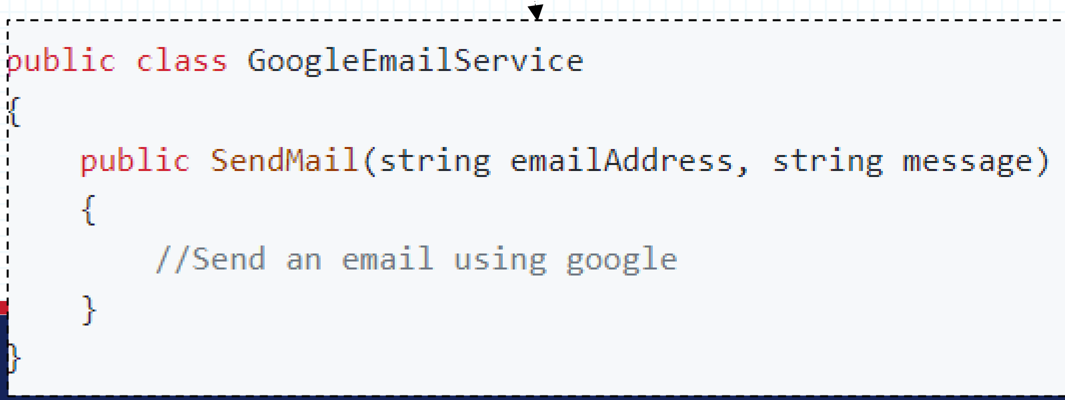
Ex: if the email service is replaced with a new one (OutlookEmailServie, etc)



```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private GoogleEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new GoogleEmailService();
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        _emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}
```



```
public class GoogleOAuthService
{
    public GoogleOAuthResult RegisterUser(string emailAddress, string password
    {
        //Register a new user
    }
}
```



```
public class GoogleEmailService
{
    public SendMail(string emailAddress, string message)
    {
        //Send an email using google
    }
}
```

High-level modules should depend on abstractions rather than concrete implementations

```
public interface IEmailService
{
    void SendMail(string emailAddress, string message)
}
```

As a Contract

BOTH depends on abstraction!

```
public class GoogleEmailService: IEmailService
{
    public SendMail(string emailAddress, string message)
    {
        //Send an email using google
    }
}

public class OutlookEmailService: IEmailService
{
    public void SendMail(string emailAddress, string message)
    {
        //Send an email using outlook
    }
}
```

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService() // or Google;
    }

    public void Register(string emailAddress, string password)
    {
        var authResult = _authService.RegisterUser(emailAddress,password
        _emailService.SendMail(emailAddress, authResult.ConfirmationMess
```

Better.. But Still Tightly Coupled

- Classes request dependencies instead of referencing directly
- Dependent object instances are injected

1. Constructor Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public UserLogic(IEmailSevice emailService)
    {
        _authService = new GoogleOAuthService();
        _emailService = emailService;
    }
    ...
}
```

```
...
    GoogleEmailService googleEmailService = new GoogleEmailService();
    UserLogic userLogic = new UserLogic(googleEmailService);
...
```

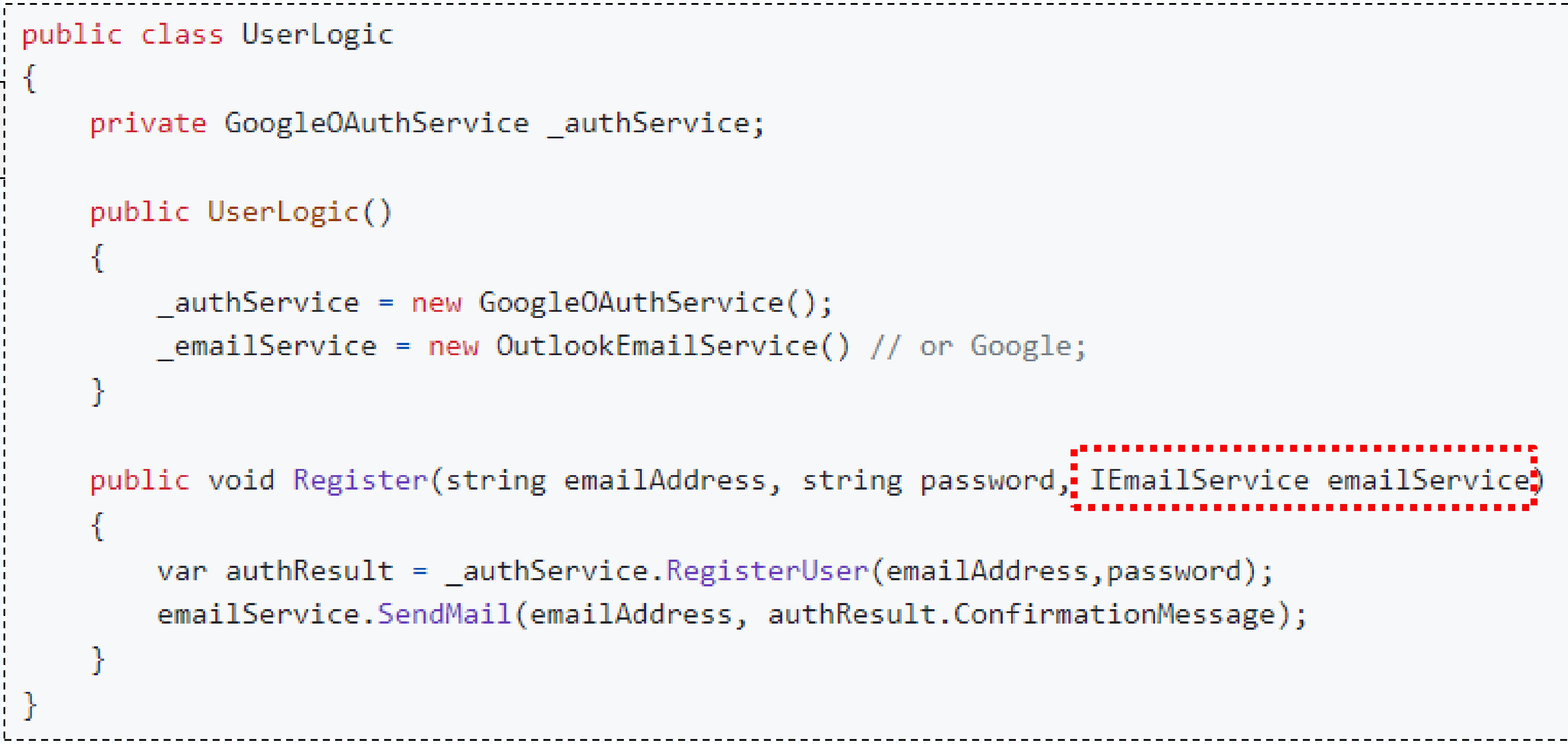
- Classes request dependencies instead of referencing directly
- Dependent object instances are injected

2. Setter Injection

```
public class UserLogic
{
    private GoogleOAuthService _authService;
    private IEmailService _emailService;

    public IEmailService EmailService
    {
        get
        {
            return _emailService;
        }
        set
        {
            _emailService = value;
        }
    }
```

- Classes request dependencies instead of referencing directly
- Dependent object instances are injected

2. Method Injection



```
public class UserLogic
{
    private GoogleOAuthService _authService;

    public UserLogic()
    {
        _authService = new GoogleOAuthService();
        _emailService = new OutlookEmailService() // or Google;
    }

    public void Register(string emailAddress, string password, IEmailService emailService)
    {
        var authResult = _authService.RegisterUser(emailAddress,password);
        emailService.SendMail(emailAddress, authResult.ConfirmationMessage);
    }
}
```