

Lesson 8



- L1: ASD Introduction
- L2: Strategy, Template method
- L3: Observer pattern
- L4: Composite pattern, iterator pattern
- L5: Command pattern
- L6: State pattern
- L7: Chain Of Responsibility pattern

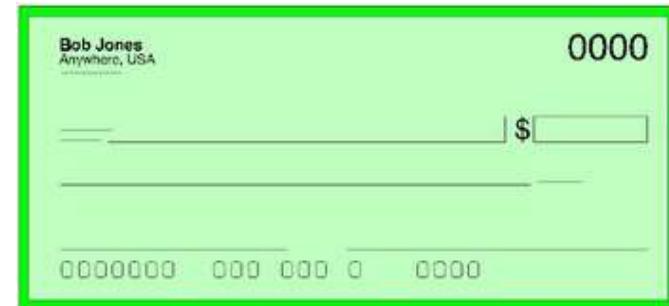
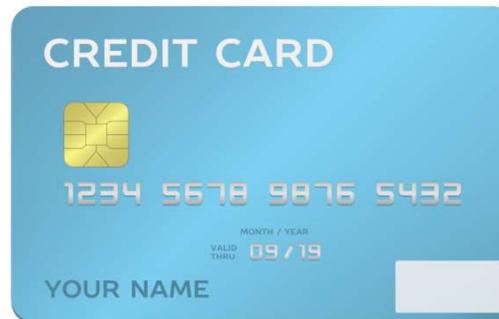
Midterm

- L8: Proxy, Adapter, Mediator
- L9: Factory, Builder, Decorator, Singleton
- L10: Framework design
- L11: Framework implementation
- L12: Framework example: Spring framework
- L13: Framework example: Spring framework

Final

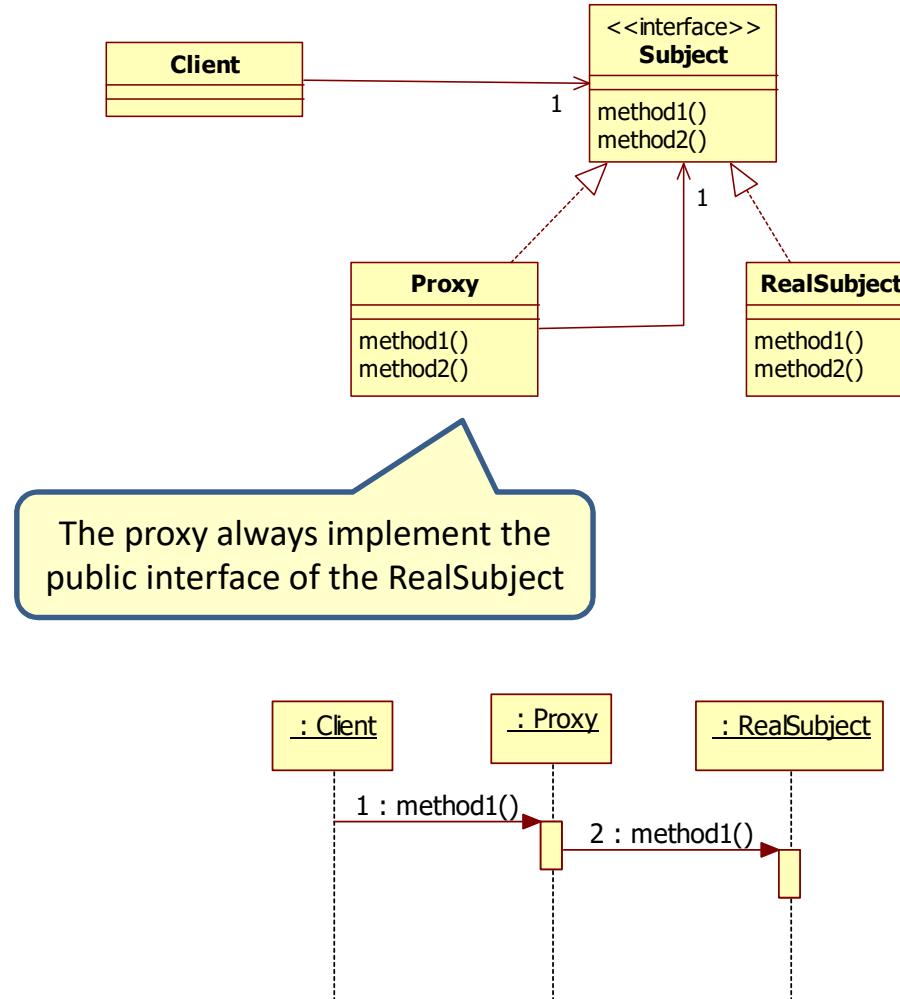
Proxy pattern

- Provides a surrogate or placeholder for another object.

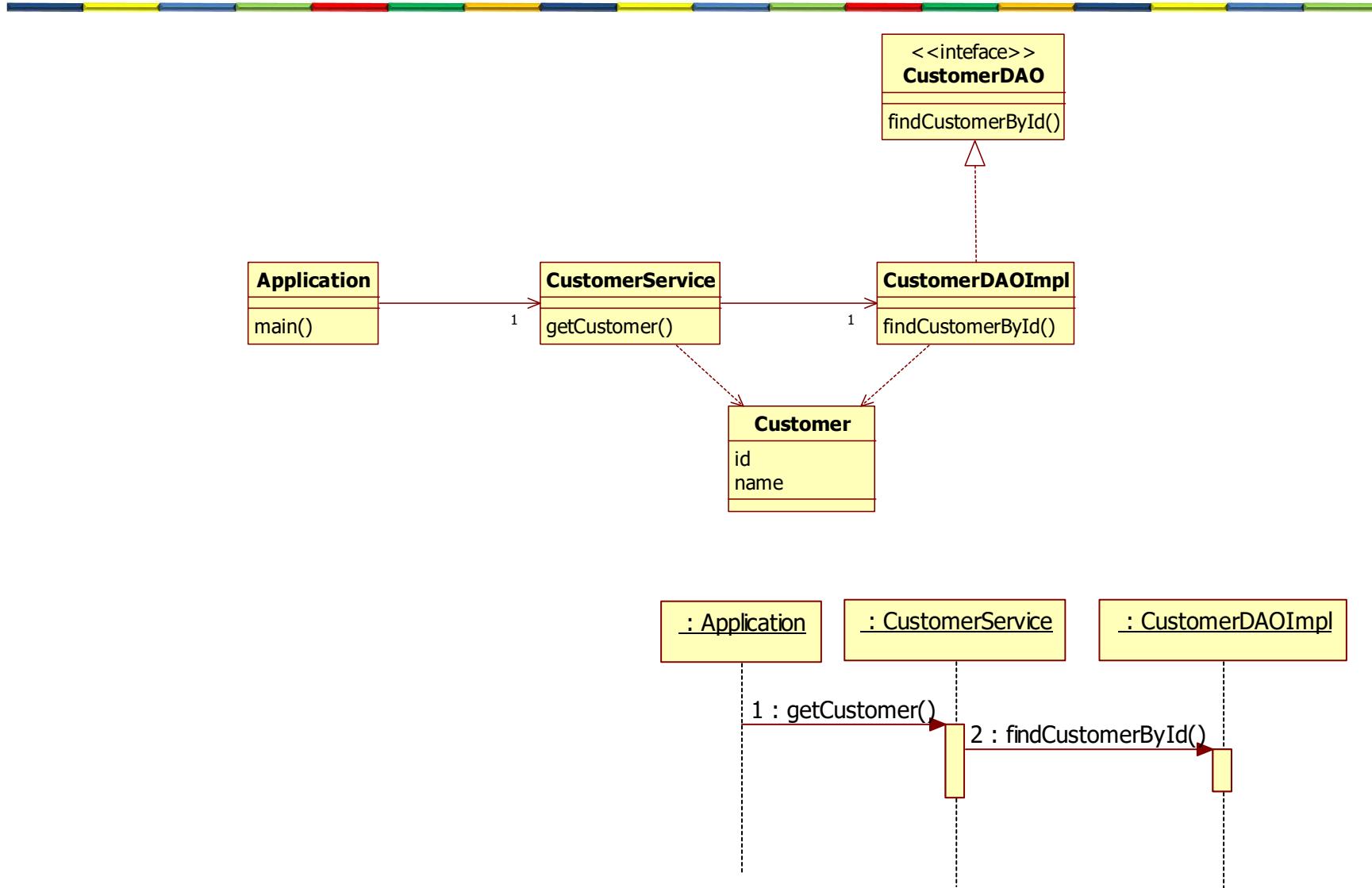


Proxy pattern

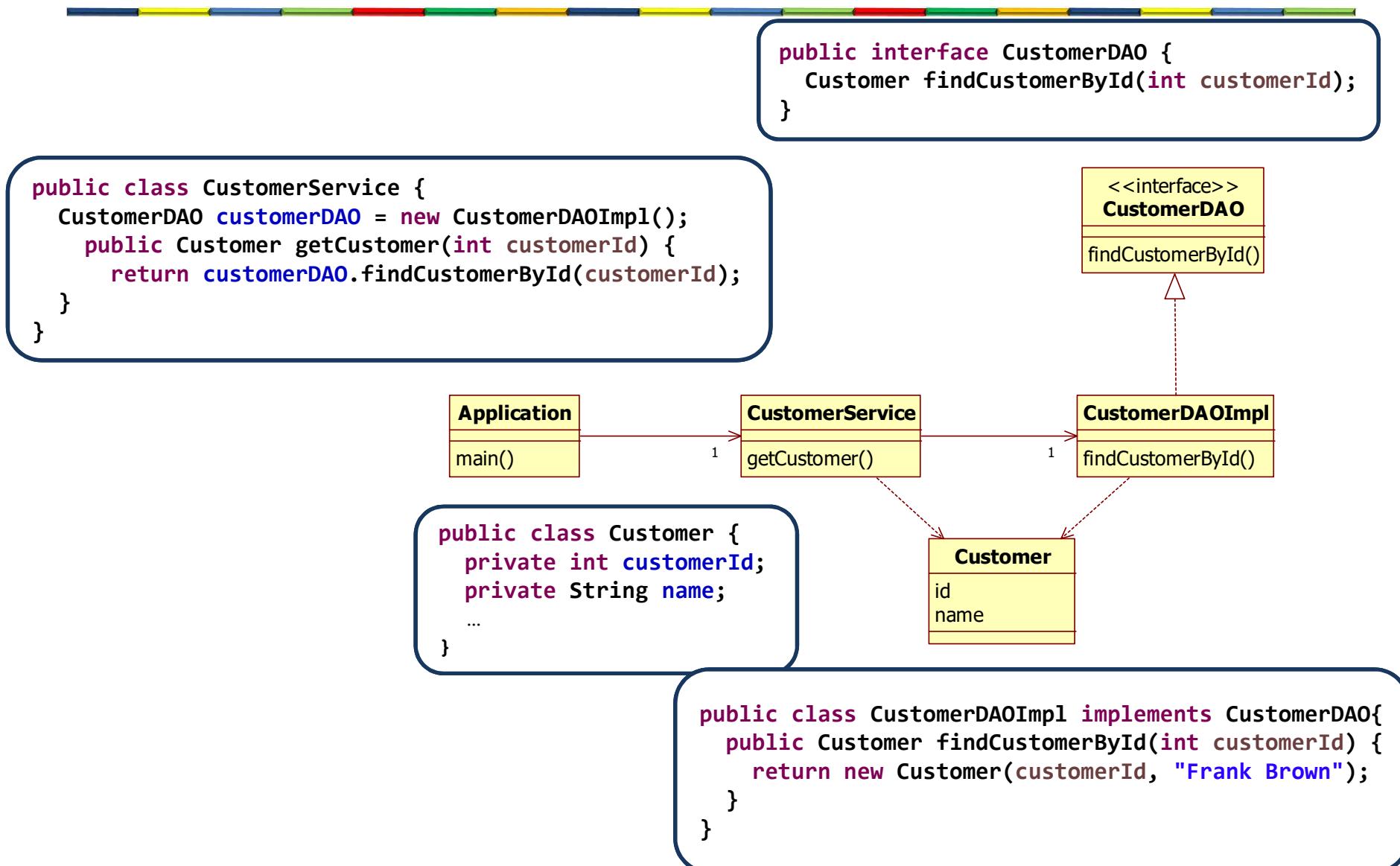
- Remote proxy
- Caching proxy
- Synchronization proxy
- Security proxy
- Transactional proxy
- Lazy load proxy
- Logging proxy
- ...



Example without proxy

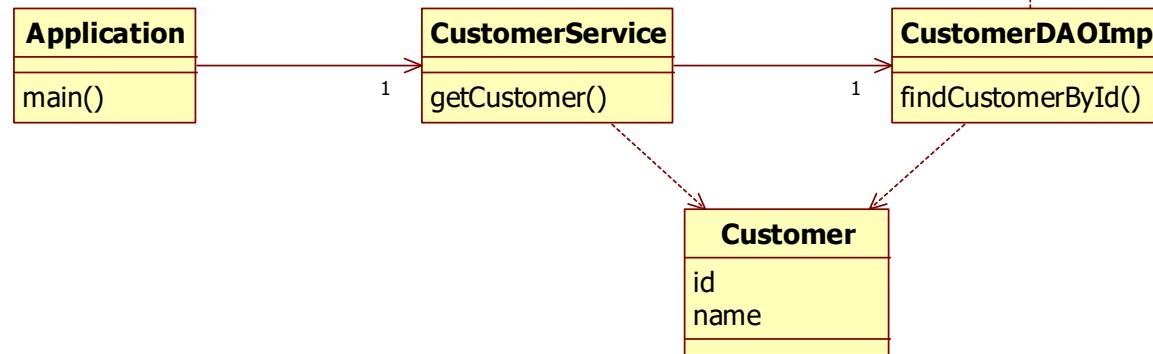
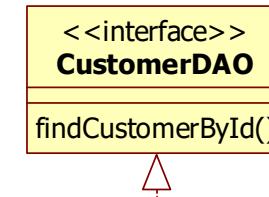


Example without proxy



Example without proxy: application

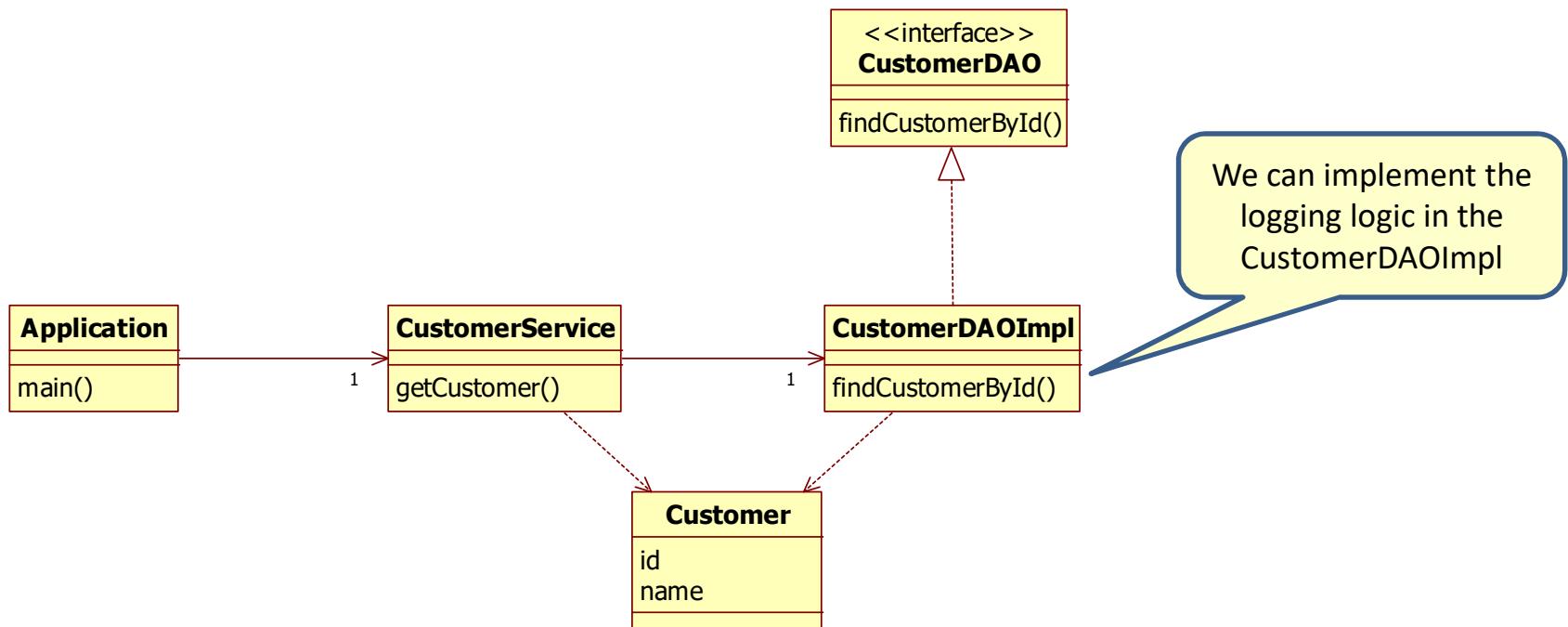
```
public class Application {  
    public static void main(String[] args) {  
        CustomerService customerService = new CustomerService();  
        Customer customer = customerService.getCustomer(1);  
        System.out.println(customer);  
    }  
}
```



Customer [customerId=1, name=Frank Brown]

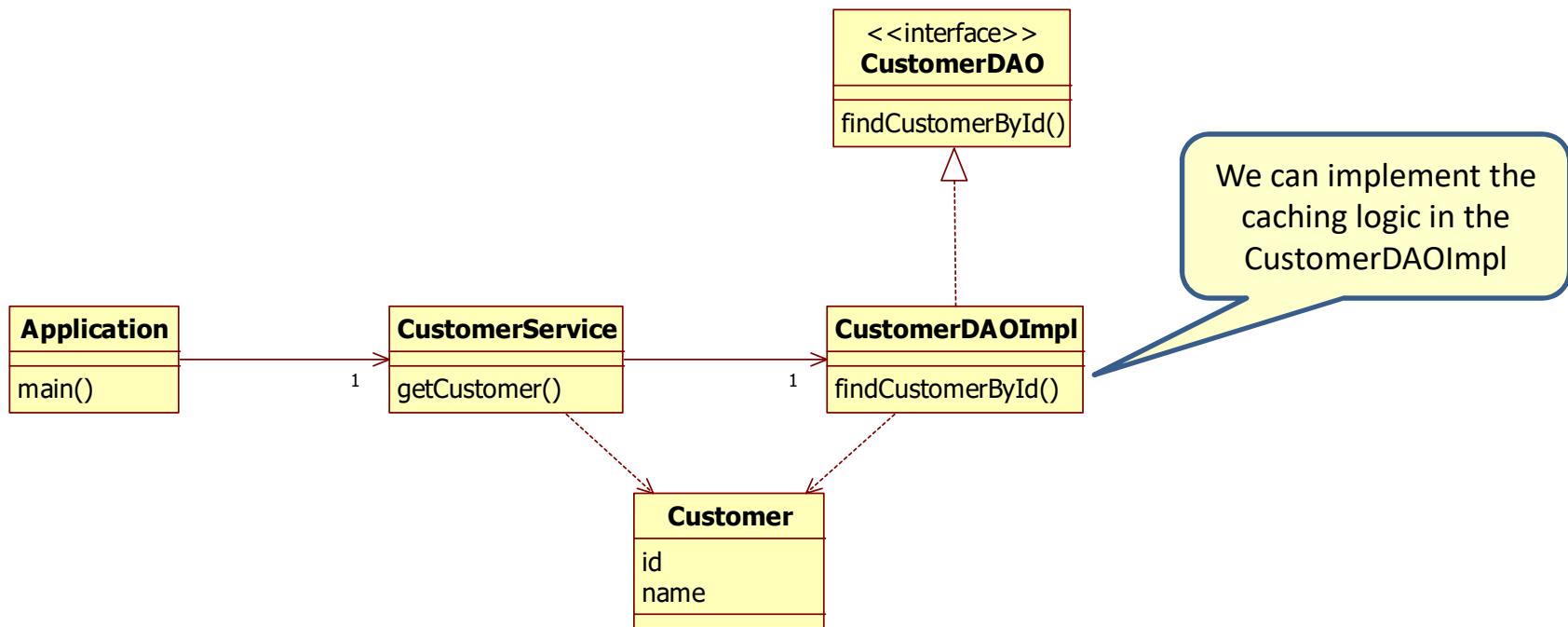
New requirement: logging

- For maintainability reasons we want to log every action on the database



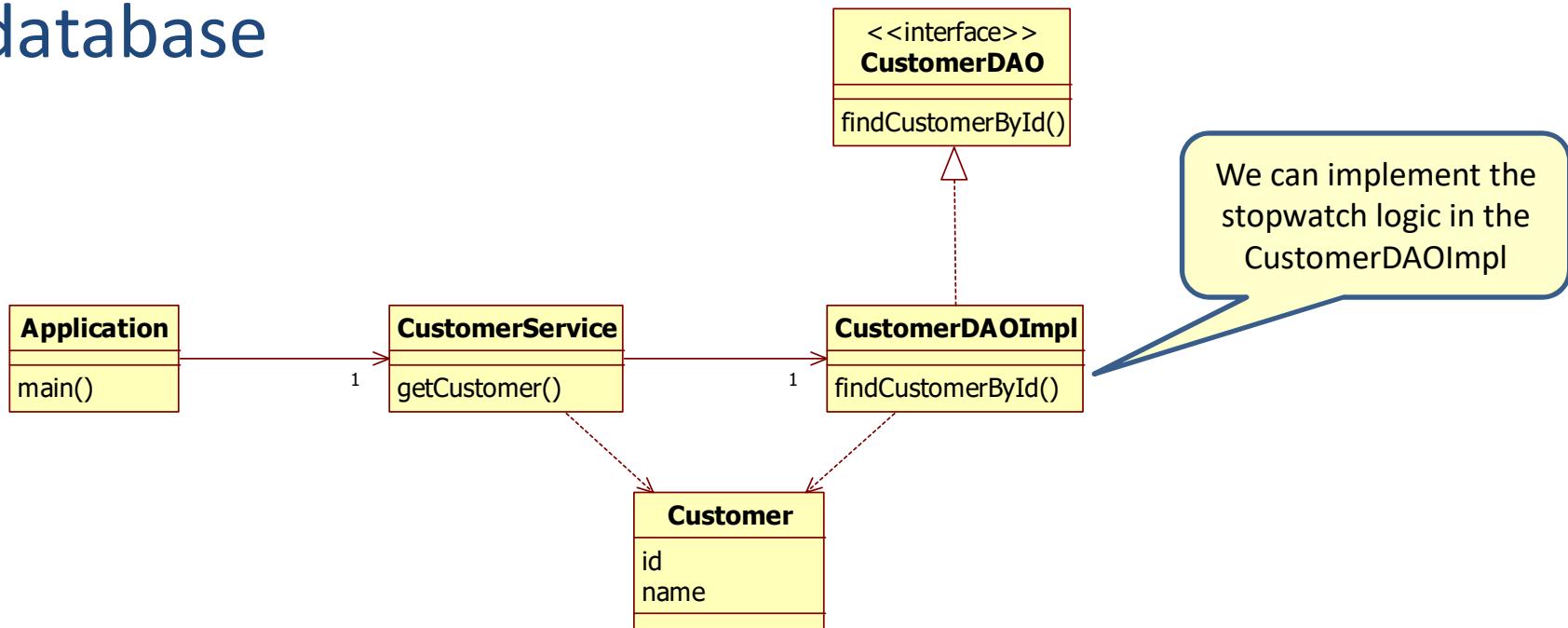
New requirement: caching

- For performance reasons we want to cache the Customers we retrieve from the database



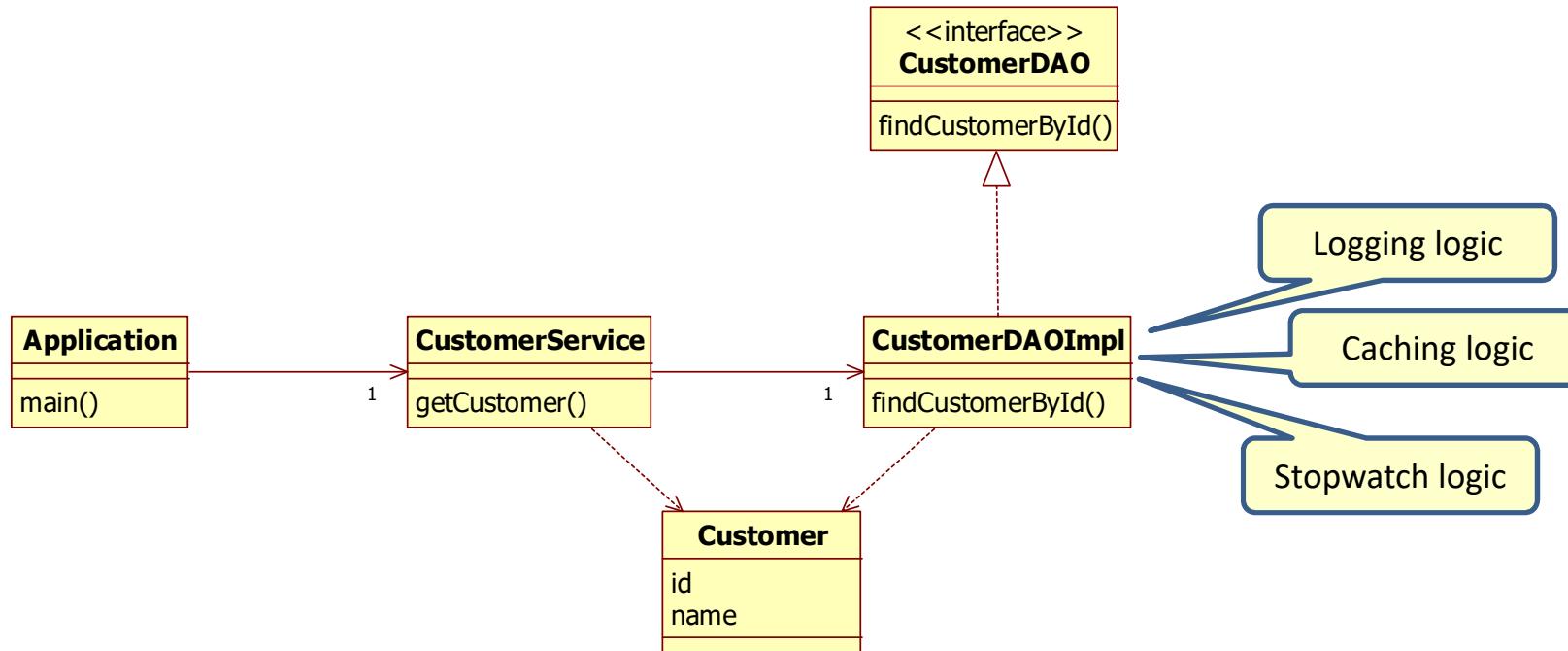
New requirement: time measurement

- For performance management reasons we want to measure the time of every call to the database



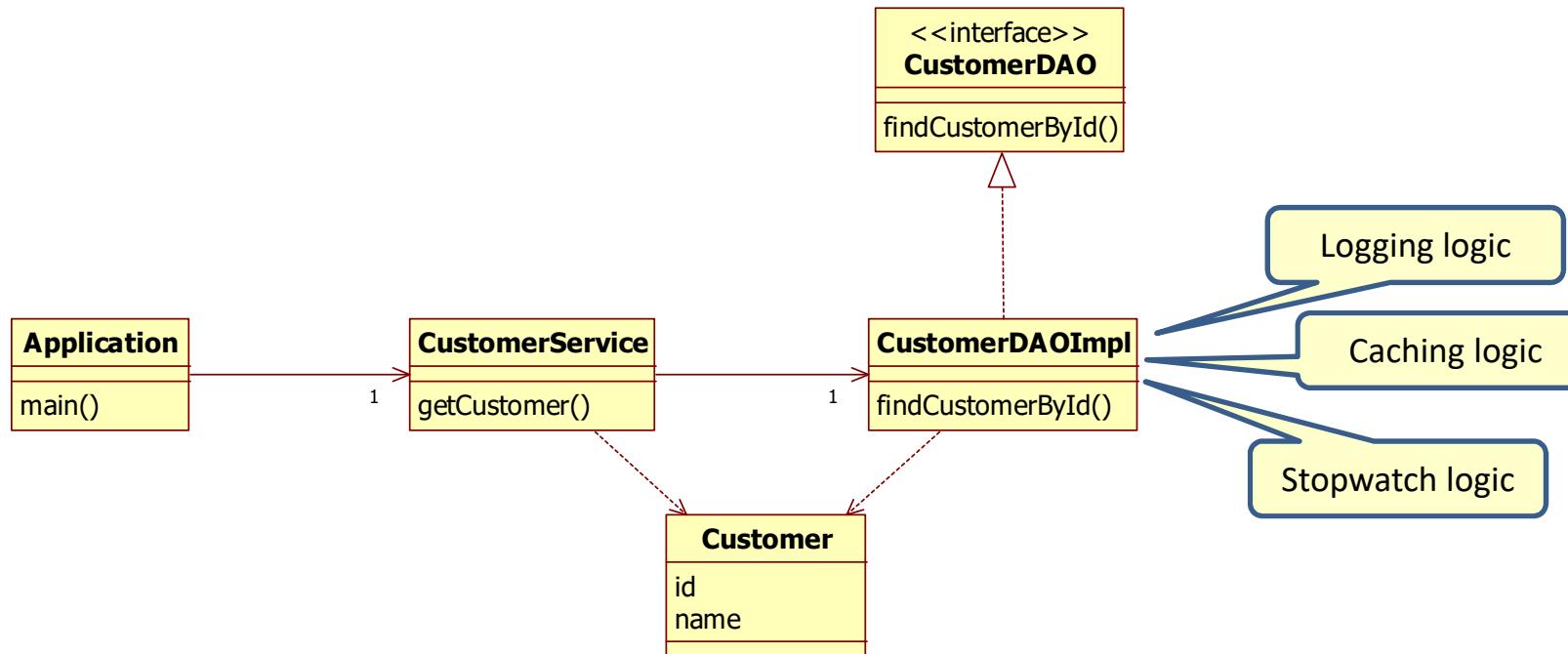
Single responsibility

- A class has one reason to change

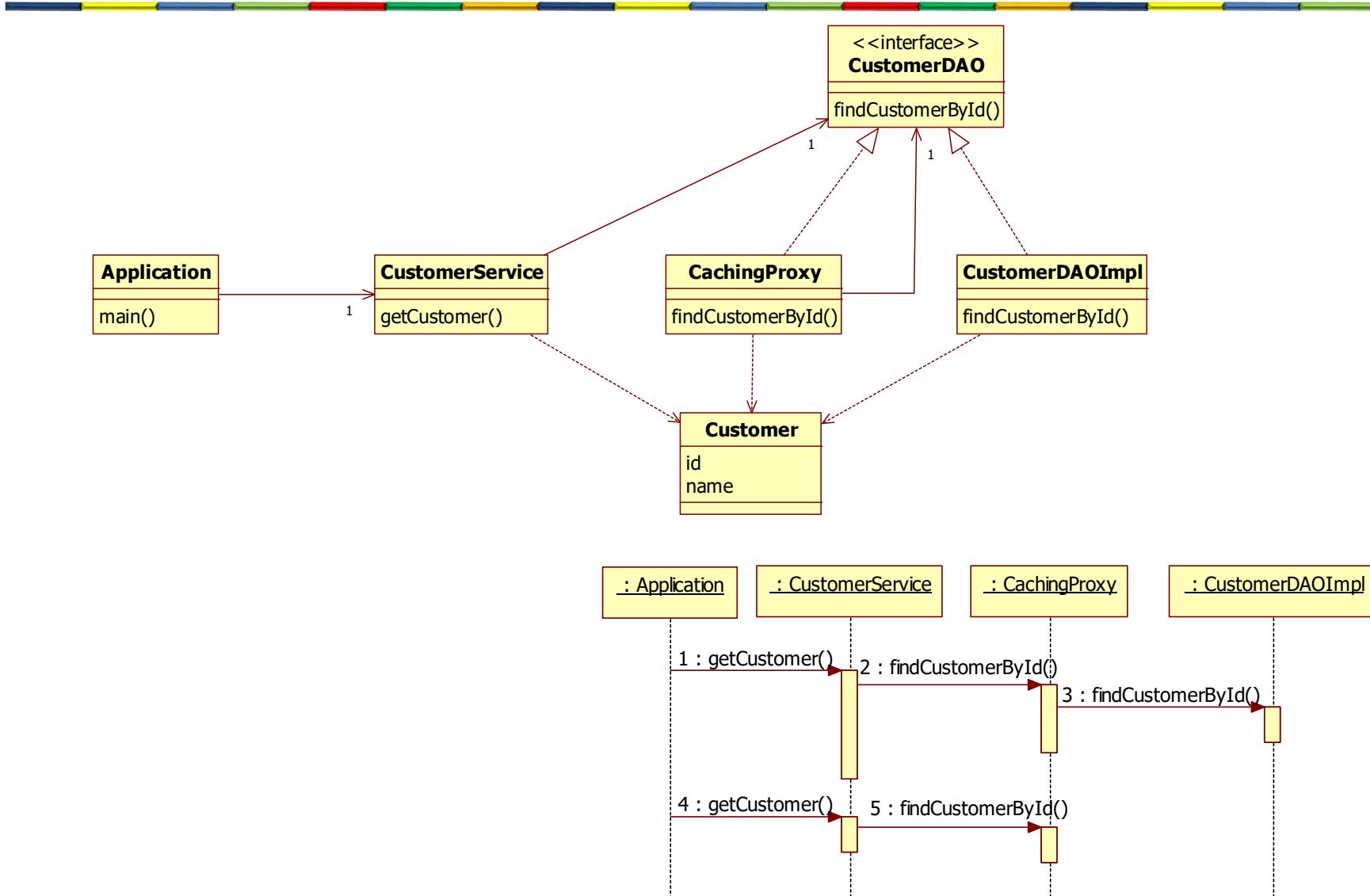


Open-closed principle

- Your design should be open for extension, but closed for change



Caching proxy

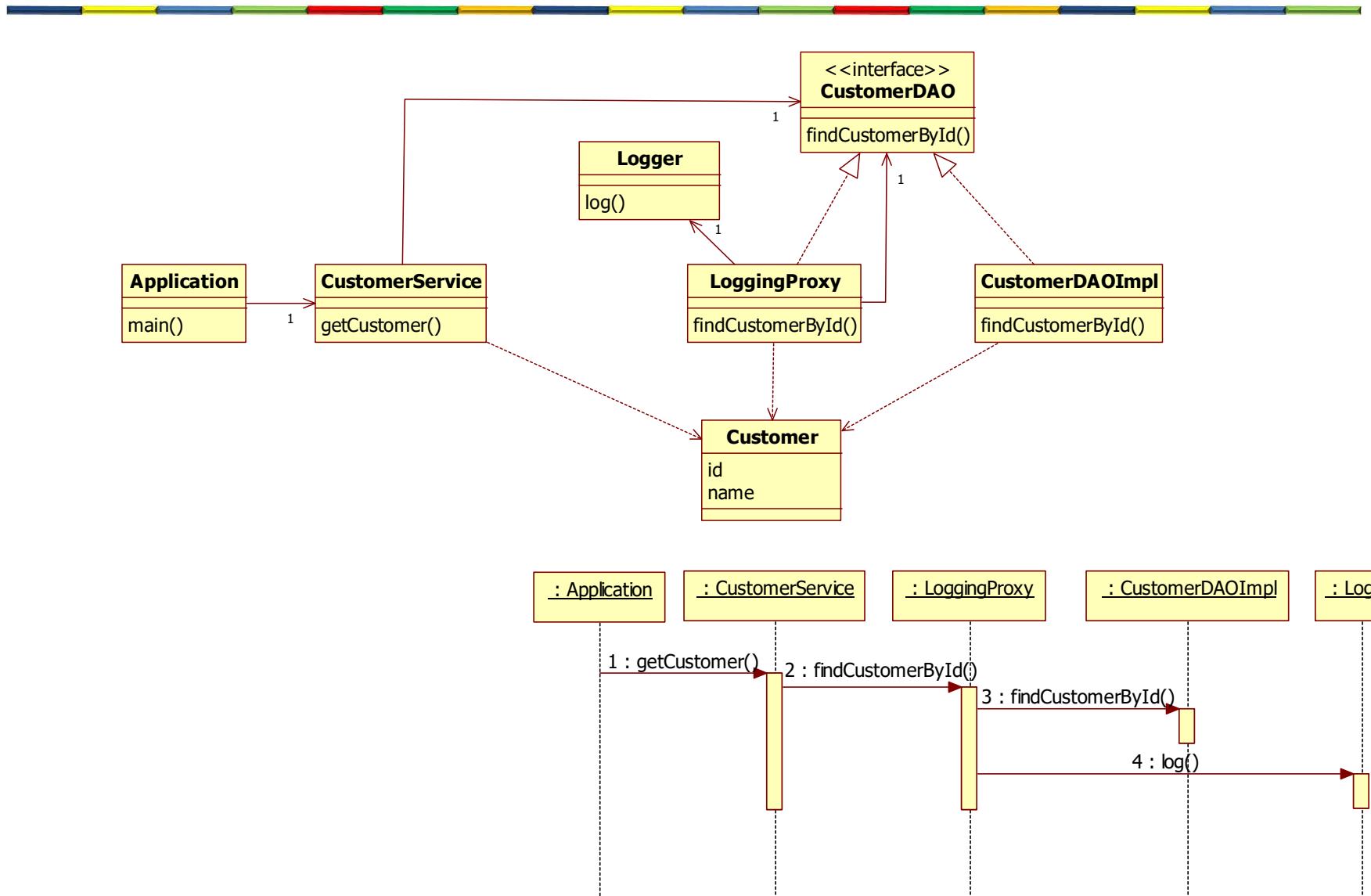


Caching proxy code

```
public class CustomerService {  
    CustomerDAO customerDAO = new CustomerDAOImpl();  
    CustomerDAO cachingProxy = new CachingProxy(customerDAO);  
  
    public Customer getCustomer(int customerId) {  
        return cachingProxy.findCustomerById(customerId);  
    }  
}
```

```
public class CachingProxy implements CustomerDAO{  
    CustomerDAO customerDAO;  
    Map<Integer, Customer> customerCache = new HashMap<Integer, Customer>();  
  
    public CachingProxy(CustomerDAO customerDAO) {  
        this.customerDAO = customerDAO;  
    }  
    public Customer findCustomerById(int customerId) {  
        Customer cachedCustomer = customerCache.get(customerId);  
        if (cachedCustomer == null) {  
            Customer customer = customerDAO.findCustomerById(customerId);  
            customerCache.put(customerId, customer);  
            return customer;  
        }  
        else  
            return cachedCustomer;  
    }  
}
```

Logging proxy



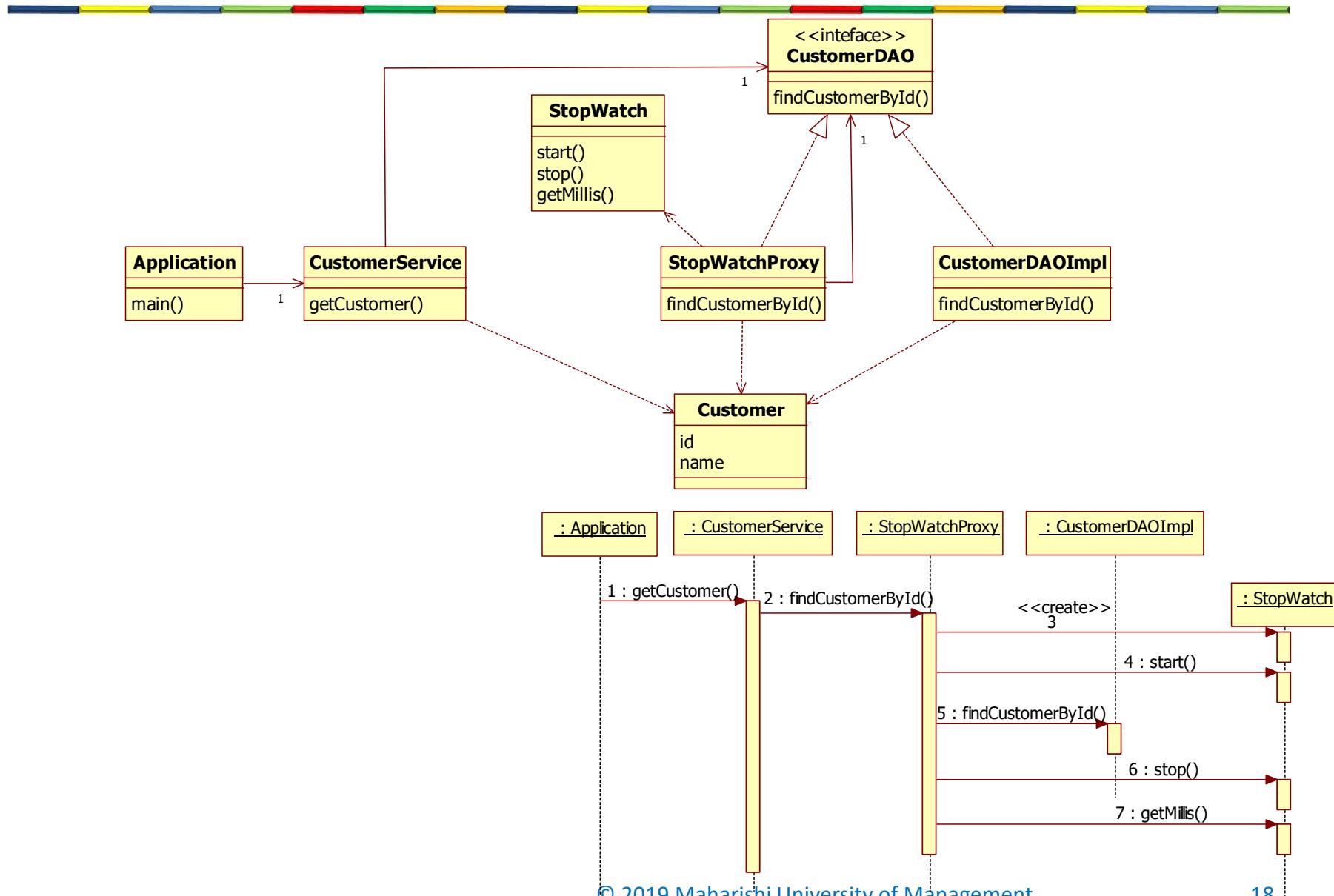
Logging proxy code

```
public class CustomerService {  
    CustomerDAO customerDAO = new CustomerDAOImpl();  
    CustomerDAO loggingProxy = new LoggingProxy(customerDAO);  
  
    public Customer getCustomer(int customerId) {  
        return loggingProxy.findCustomerById(customerId);  
    }  
}
```

```
public class LoggingProxy implements CustomerDAO {  
    CustomerDAO customerDAO;  
    Logger logger = new Logger();  
  
    public LoggingProxy(CustomerDAO customerDAO) {  
        this.customerDAO = customerDAO;  
    }  
  
    public Customer findCustomerById(int customerId) {  
        Customer customer = customerDAO.findCustomerById(customerId);  
        logger.log("getting customer with id= " + customerId);  
        return customer;  
    }  
}
```

```
public class Logger {  
    public void log(String message) {  
        System.out.println(message);  
    }  
}
```

Stopwatch proxy



StopWatch proxy code

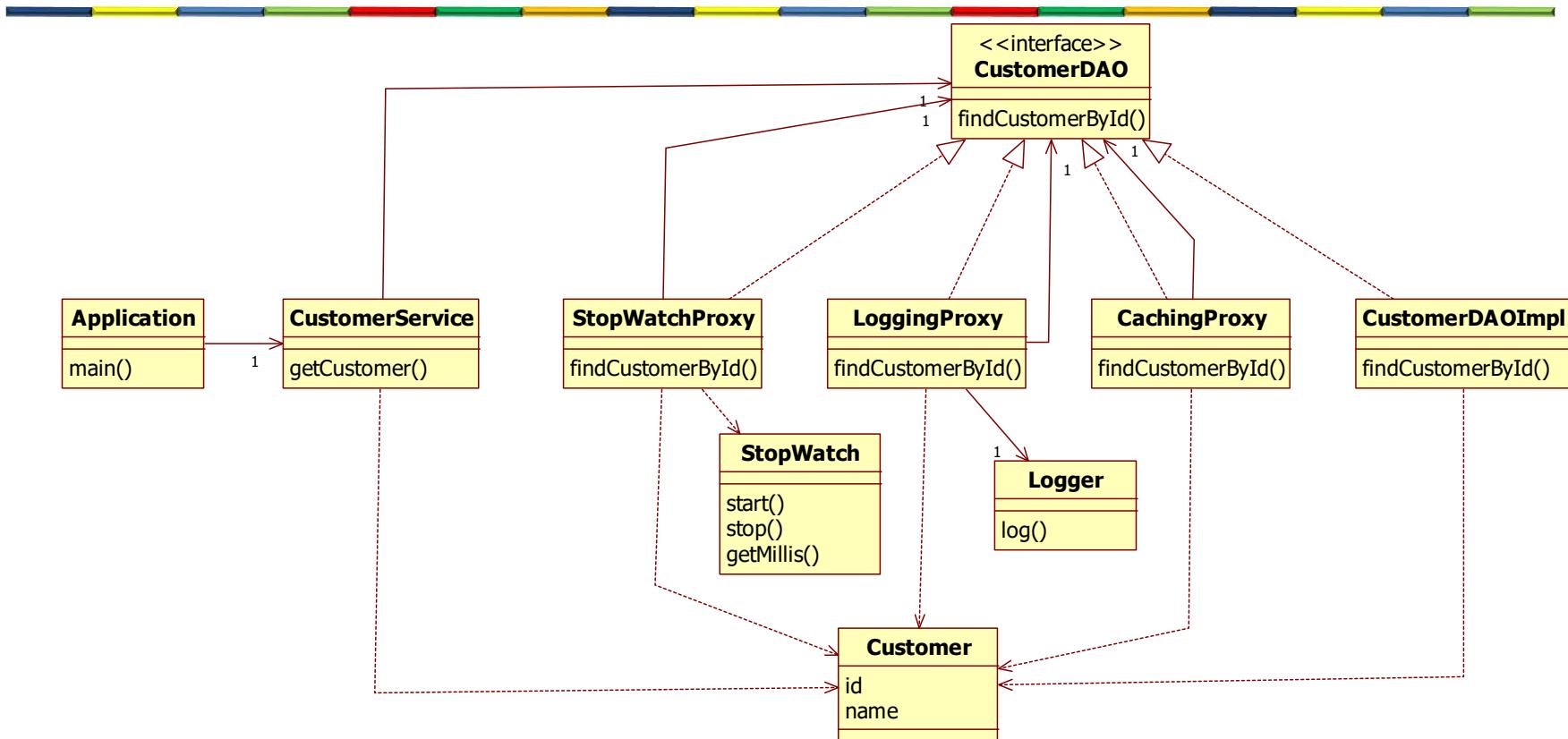
```
public class CustomerService {  
    CustomerDAO customerDAO = new CustomerDAOImpl();  
    CustomerDAO stopWatchProxy = new StopWatchProxy(customerDAO);  
  
    public Customer getCustomer(int customerId) {  
        return stopWatchProxy.findCustomerById(customerId);  
    }  
}
```

```
public class StopWatchProxy implements CustomerDAO {  
    CustomerDAO customerDAO;  
  
    public StopWatchProxy(CustomerDAO customerDAO) {  
        this.customerDAO = customerDAO;  
    }  
  
    public Customer findCustomerById(int customerId) {  
        Stopwatch stopwatch = new Stopwatch();  
        stopwatch.start();  
        Customer customer = customerDAO.findCustomerById(customerId);  
        stopwatch.stop();  
        System.out.println("The method CustomerDAO.getCustomer took  
                           "+stopwatch.getMillis()+" ms");  
        return customer;  
    }  
}
```

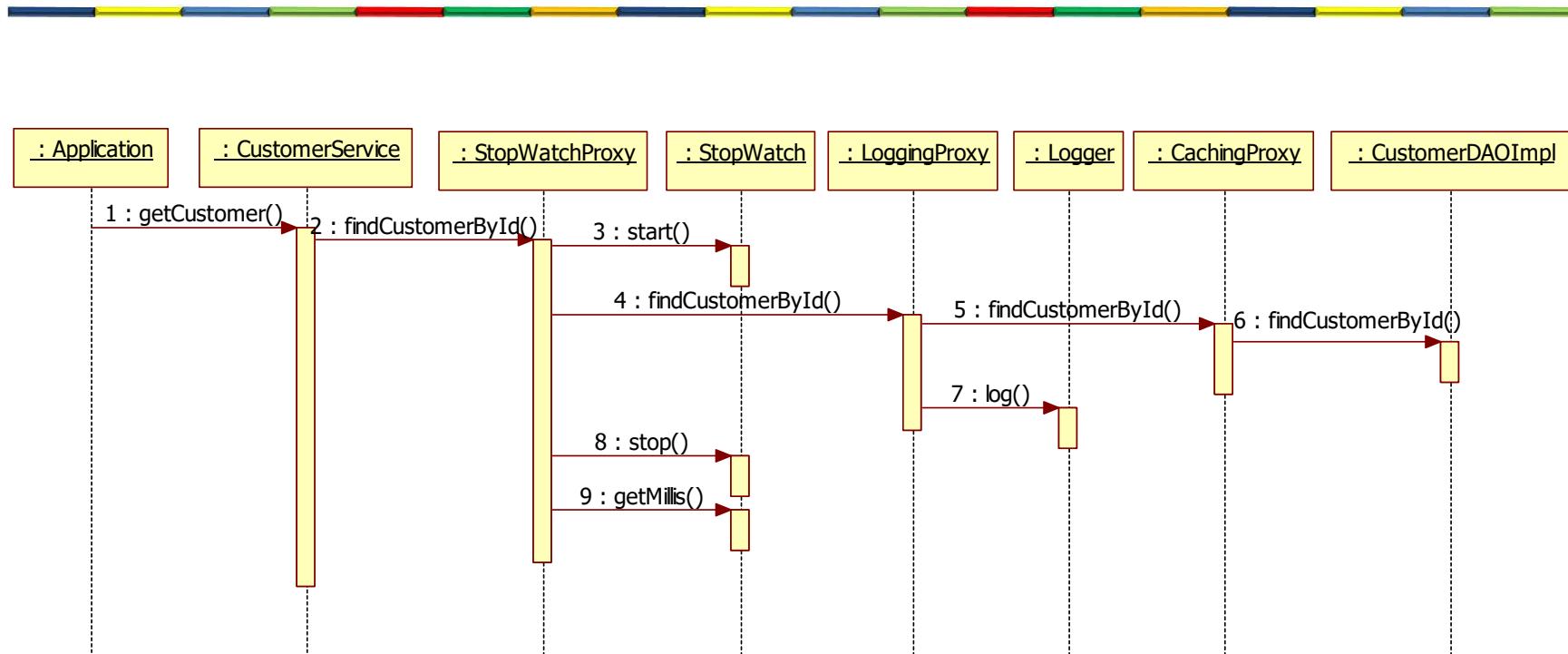
StopWatch code

```
public class Stopwatch {  
  
    private long start = 0;  
    private long finish = 0;  
    private long timeElapsed = 0;  
  
    public void start() {  
        start = System.currentTimeMillis();  
    }  
    public void stop() {  
        finish = System.currentTimeMillis();  
    }  
    public long getMillis() {  
        timeElapsed = finish - start;  
        return timeElapsed;  
    }  
}
```

Multiple proxies class diagram



Multiple proxies scenario



Nested proxies

```
public class CustomerService {  
    CustomerDAO customerDAO = new CustomerDAOImpl();  
    CustomerDAO cachingProxy = new CachingProxy(customerDAO);  
    CustomerDAO loggerProxy = new LoggingProxy(cachingProxy);  
    CustomerDAO stopWatchProxy = new StopWatchProxy(loggerProxy);  
  
    public Customer getCustomer(int customerId) {  
        return stopWatchProxy.findCustomerById(customerId);  
    }  
}
```

Creating a chain of nested proxies

Problem with simple proxy

```
public class LoggingProxy implements CustomerDAO {  
    CustomerDAO customerDAO;  
    Logger logger = new Logger();  
  
    public LoggingProxy(CustomerDAO customerDAO) {  
        this.customerDAO = customerDAO;  
    }  
  
    public Customer findCustomerById(int customerId) {  
        Customer customer = customerDAO.findCustomerById(customerId);  
        logger.log("getting customer with id= " + customerId);  
        return customer;  
    }  
}
```

This proxy can only be used in front of classes that implement the CustomerDAO interface

- We want a generic proxy that can be used in front of any class
 - Dynamic proxy

Dynamic stopwatch proxy

```
import java.lang.reflect.*;

public class StopWatchProxy implements InvocationHandler {
    private Object target;

    public StopWatchProxy(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.start();
        // invoke the method on the target
        Object returnValue = method.invoke(target, args);

        stopwatch.stop();
        System.out.println("The method " + method.getName() + " took " + stopwatch.getMillis() + " ms");
        return returnValue;
    }
}
```

Reflection: A technique to examine or modify the behavior of methods, classes, interfaces at runtime.

Invoking the dynamic proxy

```
import java.lang.reflect.Proxy;

public class CustomerService {
    CustomerDAO customerDAO = new CustomerDAOImpl();
    ClassLoader classLoader = CustomerDAO.class.getClassLoader();
    CustomerDAO stopWatchProxy = (CustomerDAO)
        Proxy.newProxyInstance(classLoader,
            new Class[] { CustomerDAO.class },
            new StopwatchProxy(customerDAO));

    public Customer getCustomer(int customerId) {
        return stopWatchProxy.findCustomerId(customerId);
    }
}
```

Create a Proxy

customerService

stopwatchProxy

customerDAO

getCustomer()

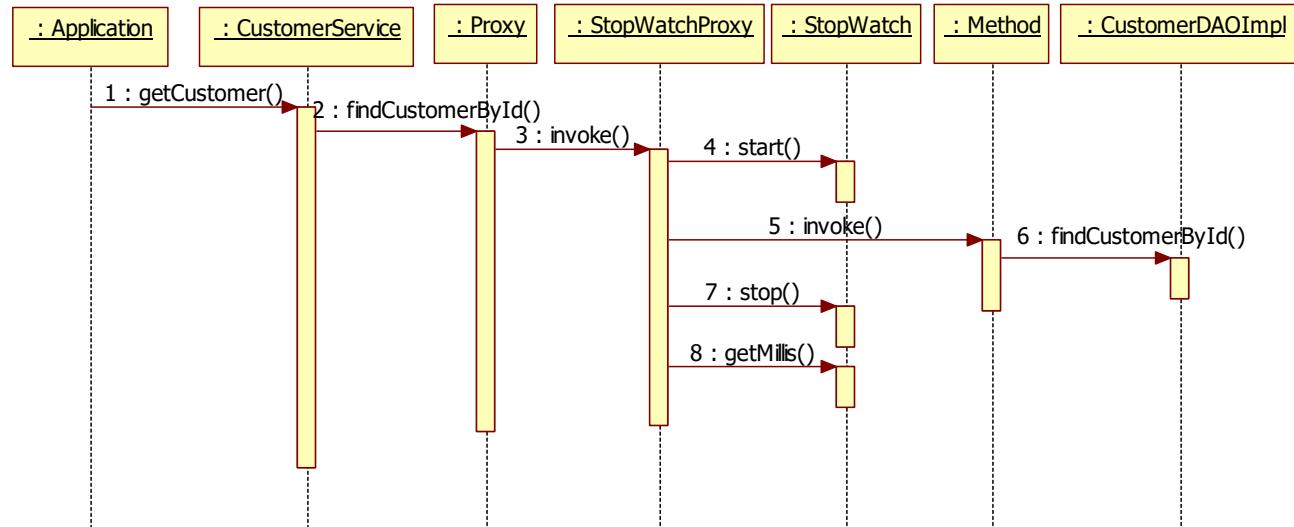
findCustomerId()

findCustomerId()

What really happens

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.start();
    // invoke the method on the target
    Object returnValue = method.invoke(target, args);

    stopwatch.stop();
    System.out.println("The method " + method.getName() + " took " + stopwatch.getMillis() + " ms");
    return returnValue;
}
```



Proxy vs. dynamic proxy

```
public class StopwatchProxy implements CustomerDAO {  
    CustomerDAO customerDAO;  
  
    public StopwatchProxy(CustomerDAO customerDAO) {  
        this.customerDAO = customerDAO;  
    }  
  
    public Customer findCustomerById(int customerId) {  
        Stopwatch stopwatch = new Stopwatch();  
        stopwatch.start();  
        Customer customer = customerDAO.findCustomerById(customerId);  
        stopwatch.stop();  
        System.out.println("The method CustomerDAO.getCustomer took  
                           "+stopwatch.getMillis()+" ms");  
        return customer;  
    }  
}
```

This proxy can only be used in front of classes that implement the CustomerDAO interface

You need to implement all interface methods

Is very specific in what you want to do on a CustomerDAO class

Proxy vs. dynamic proxy

```
public class StopWatchProxy implements InvocationHandler {  
    private Object target;  
  
    public StopWatchProxy(Object target) {  
        this.target = target;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        Stopwatch stopwatch = new Stopwatch();  
        stopwatch.start();  
        // invoke the method on the target  
        Object returnValue = method.invoke(target, args);  
  
        stopwatch.stop();  
        System.out.println("The method " + method.getName() + " took " + stopwatch.getMillis() + " ms");  
        return returnValue;  
    }  
}
```

This proxy can be used in front of every class

You only need to implement the invoke() method

Must be very generic if you want to apply this proxy for any other class

Dynamic logging proxy

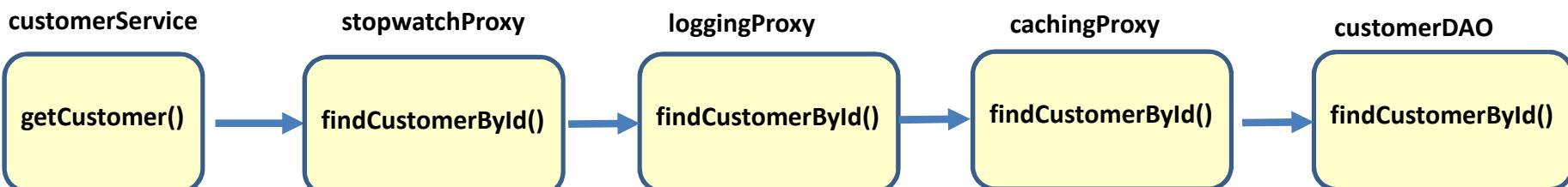
```
public class LoggingProxy implements InvocationHandler {  
    private Object target;  
    Logger logger = new Logger();  
  
    public LoggingProxy(Object target) {  
        this.target = target;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        // invoke the method on the target  
        Object returnValue = method.invoke(target, args);  
        logger.log("Calling method" + method.getName() + " with argument(s):");  
        for(int p=0; p<args.length;p++){  
            logger.log(" Param[" + p + "]: " + args[p].toString());  
        }  
        return returnValue;  
    }  
}
```

Dynamic caching proxy

```
public class CachingProxy implements InvocationHandler {  
    private Object target;  
    Map<String, Object> cache = new HashMap<String, Object>();  
  
    public CachingProxy(Object target) {  
        this.target = target;  
    }  
  
    @Override  
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {  
        String key = ""+args[0];  
        Object cachedObject = cache.get(key);  
        if (cachedObject == null) {  
            Object result = method.invoke(target, args);  
            cache.put(key, result);  
            return result;  
        } else  
            return cachedObject;  
    }  
}
```

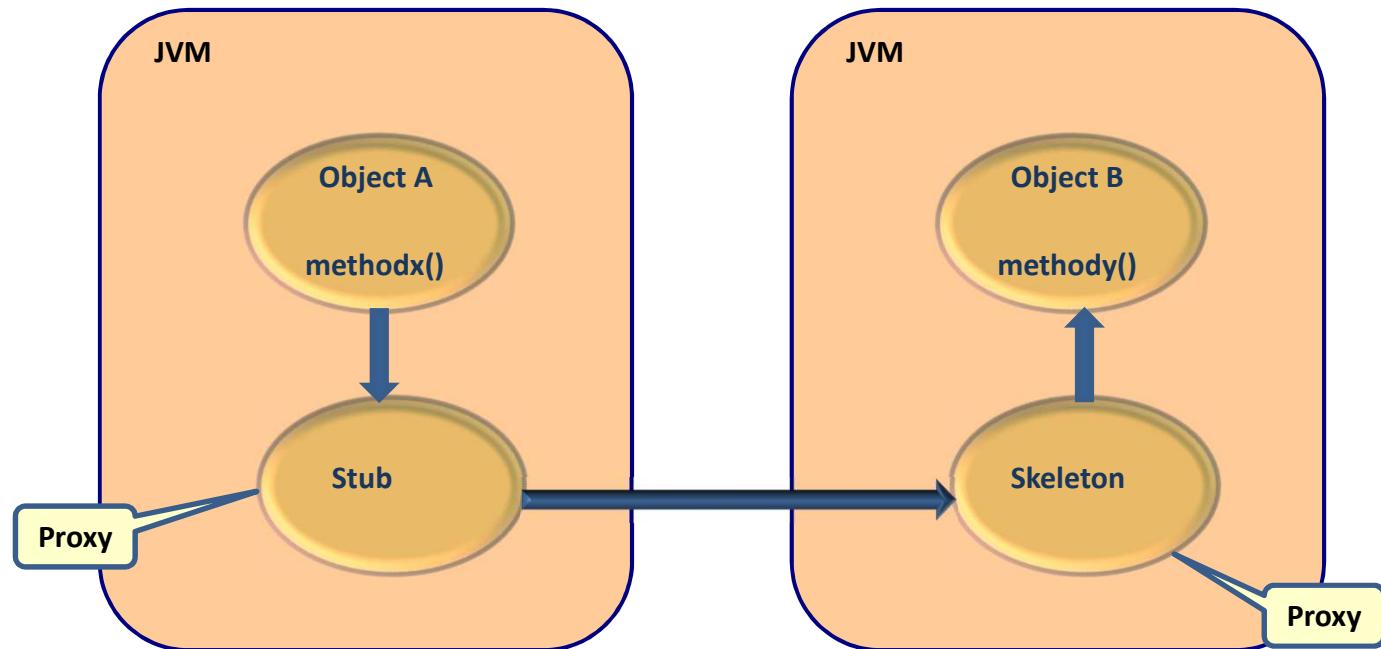
Nested dynamic proxies

```
public class CustomerService {  
    CustomerDAO customerDAO = new CustomerDAOImpl();  
    ClassLoader classLoader = CustomerDAO.class.getClassLoader();  
    CustomerDAO cachingProxy =  
        (CustomerDAO) Proxy.newProxyInstance(classLoader,  
            new Class[] { CustomerDAO.class },  
            new CachingProxy(customerDAO));  
    CustomerDAO loggingProxy =  
        (CustomerDAO) Proxy.newProxyInstance(classLoader,  
            new Class[] { CustomerDAO.class },  
            new LoggingProxy(cachingProxy));  
    CustomerDAO stopwatchProxy =  
        (CustomerDAO) Proxy.newProxyInstance(classLoader,  
            new Class[] { CustomerDAO.class },  
            new StopwatchProxy(loggingProxy));  
  
    public Customer getCustomer(int customerId) {  
        return stopwatchProxy.findCustomerById(customerId);  
    }  
}
```



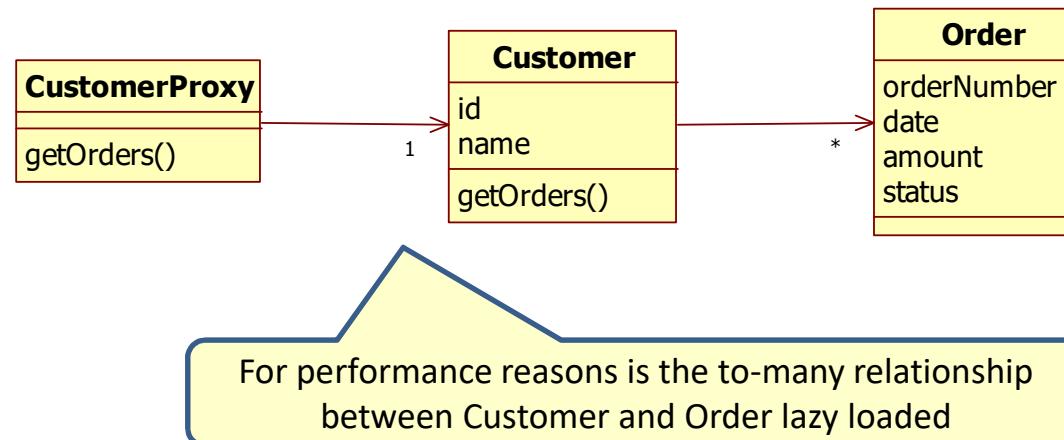
Where are proxies used: RPC

- Remote Procedure Calls
 - Remote Method Invocation (RMI)



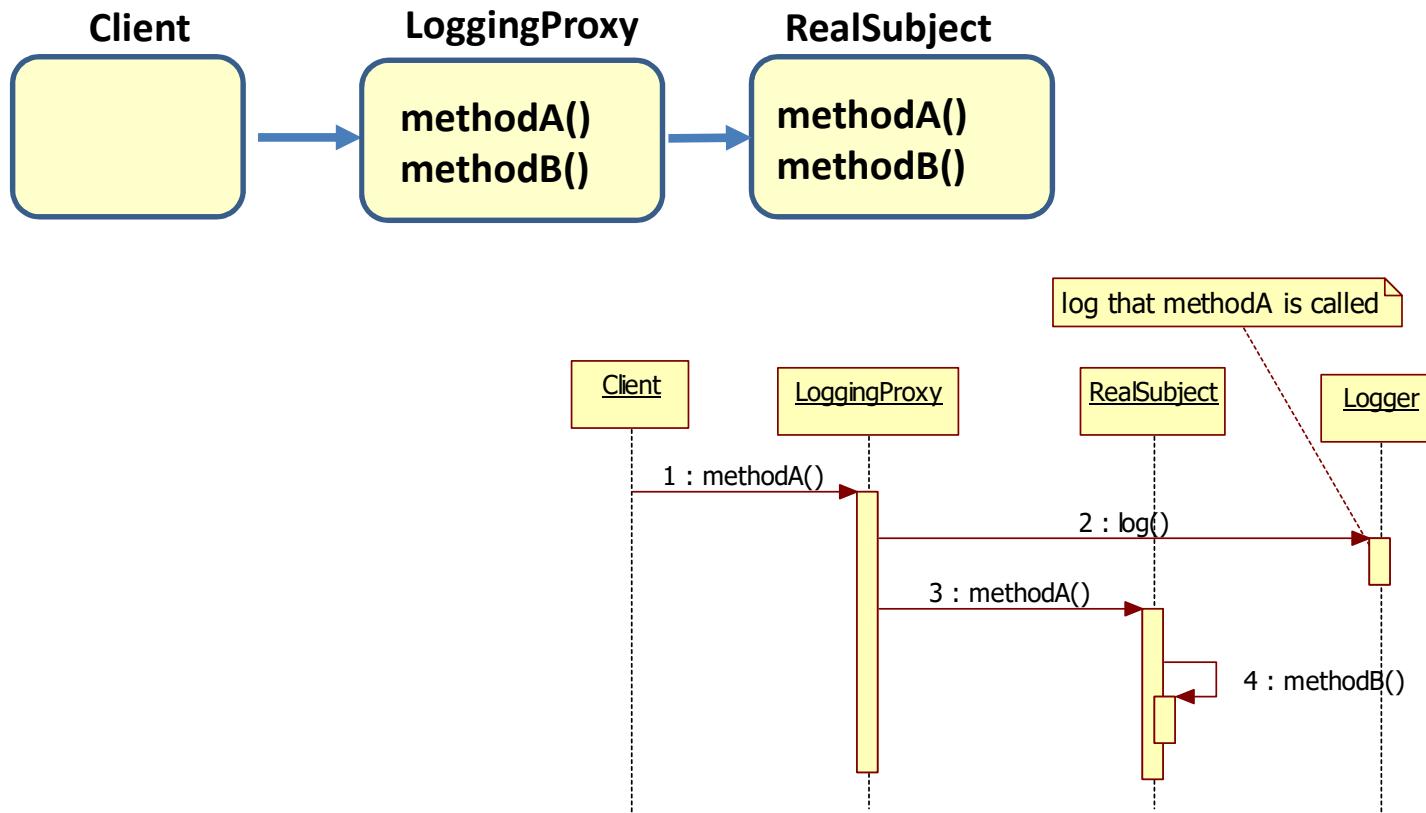
Where are proxies used: Hibernate

- Hibernate is an Object Relational Mapper (ORM) framework used for persisting objects
- It uses lazy loading using proxies



Issue with a proxy

- If a method of the real subject calls a method of itself, this will not go through the proxy.



Main point

- The Proxy pattern provides a surrogate or placeholder for another object to control access to it.
- In Unity Consciousness one realizes that every relative object you see around you, is just an expression of the same Pure Consciousness you experience within yourself.

Lesson 8 Adapter pattern



L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

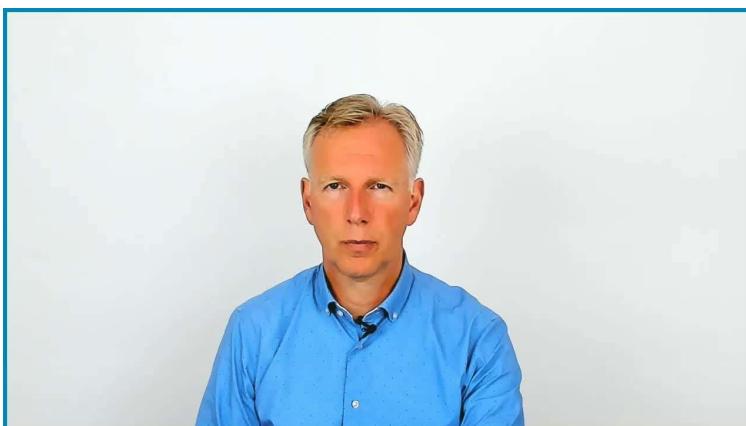
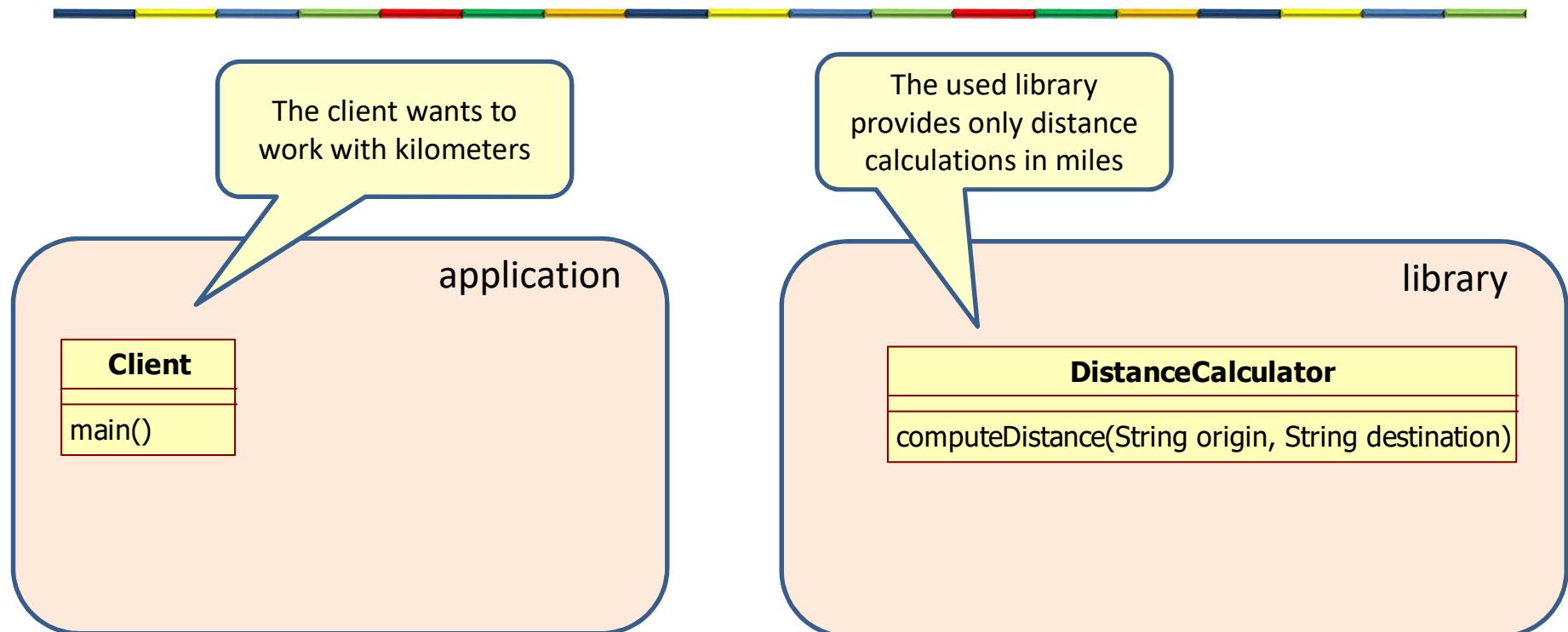
Final

Adapter pattern

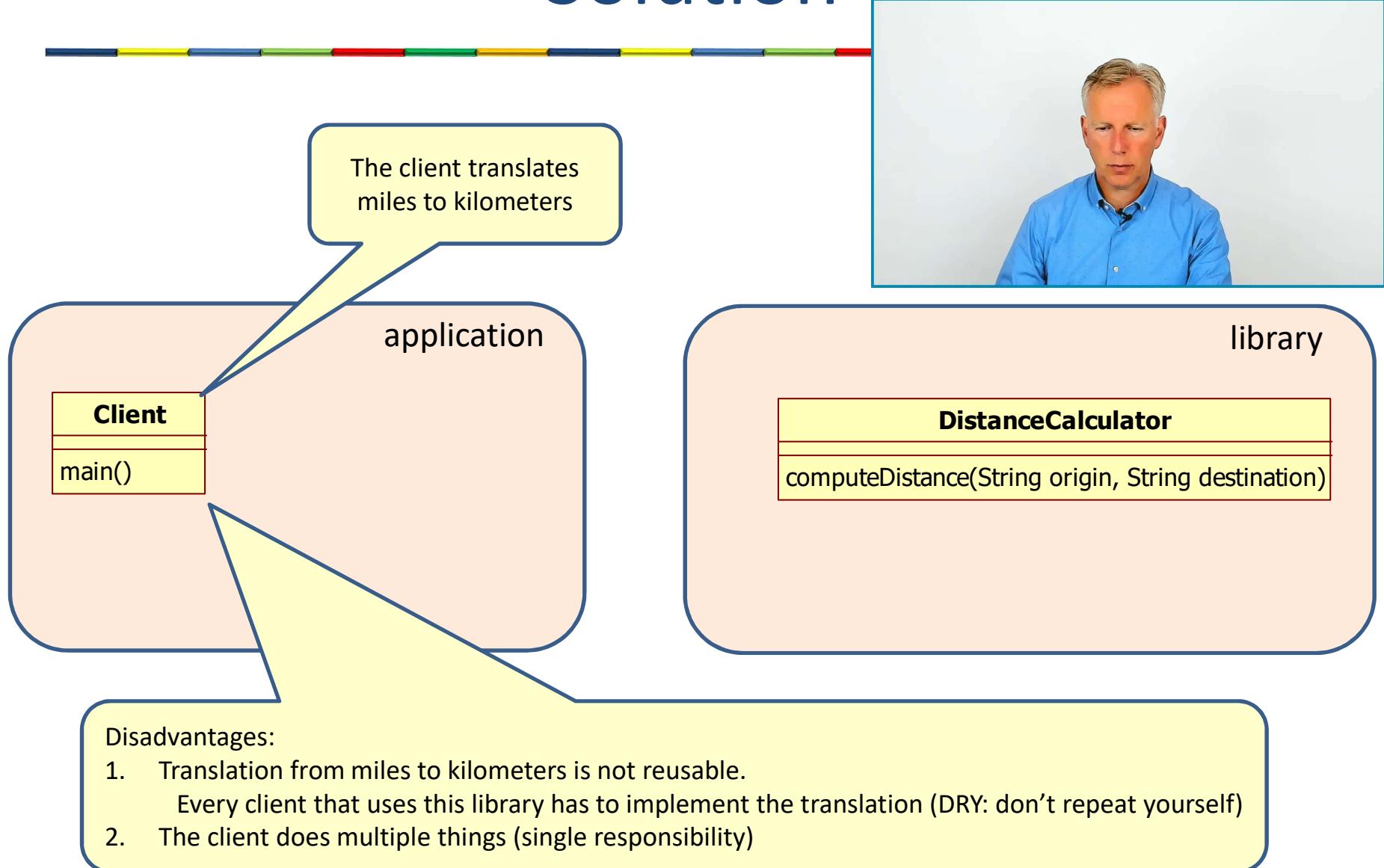
- Translates the existing interface of a class into an interface that the client requires.
 - (Reusable) wrapper



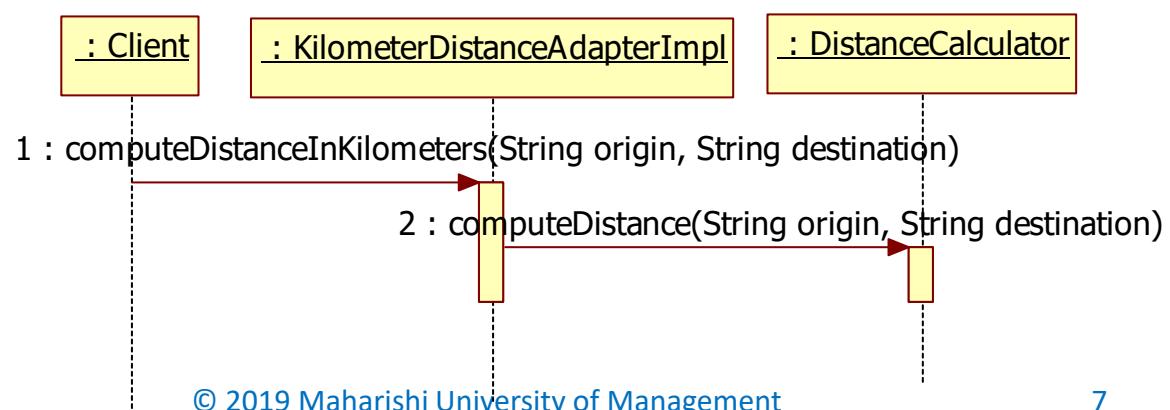
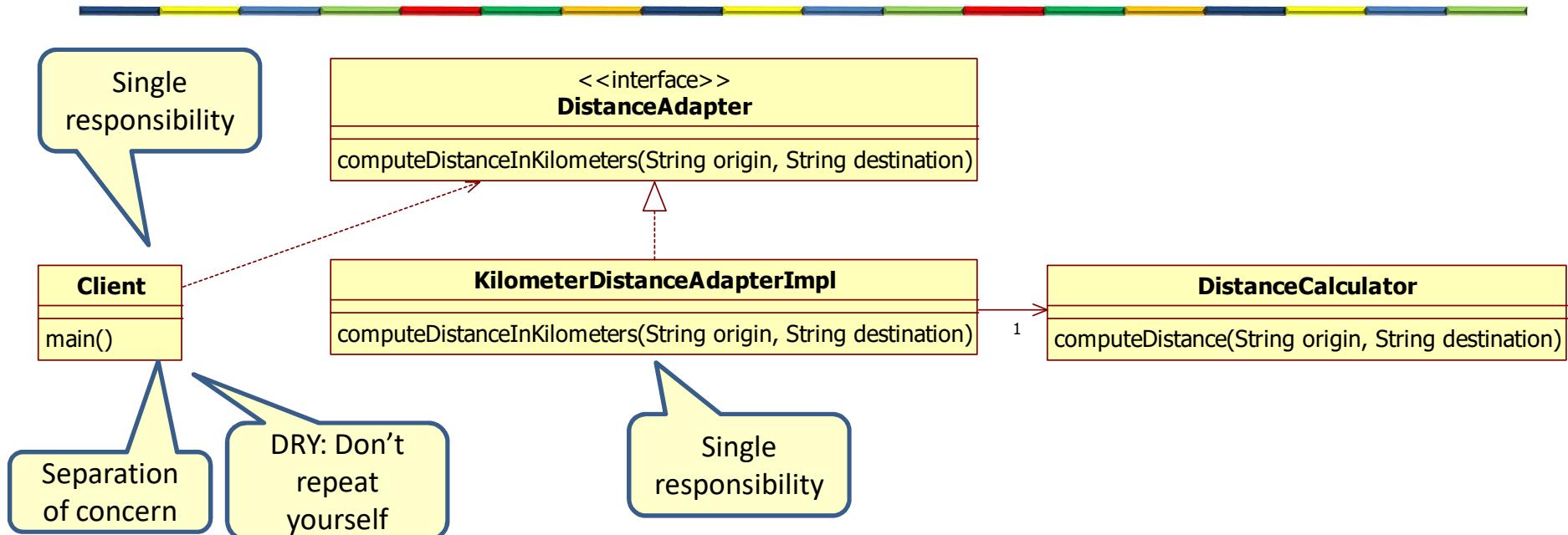
Design problem



Solution



Better solution: Adapter



Adapter + Adaptee

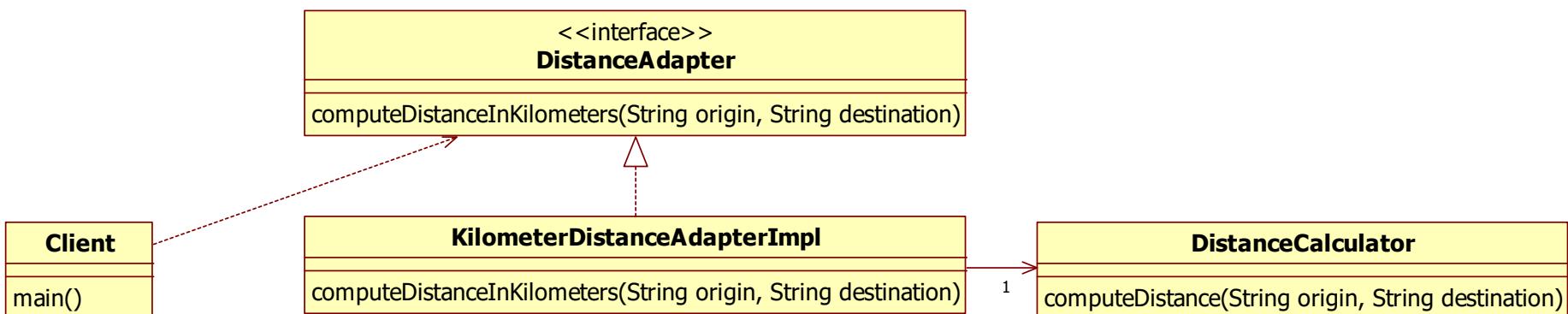
```
public class DistanceCalculator {  
    public double computeDistance(String origin, String destination) {  
        return (new Random()).nextInt(100);  
    }  
}
```

```
public interface DistanceAdapter {  
    double computeDistanceInKilometers(String origin, String destination);  
    void setDistanceCalculator(DistanceCalculator distanceCalculator);  
}
```

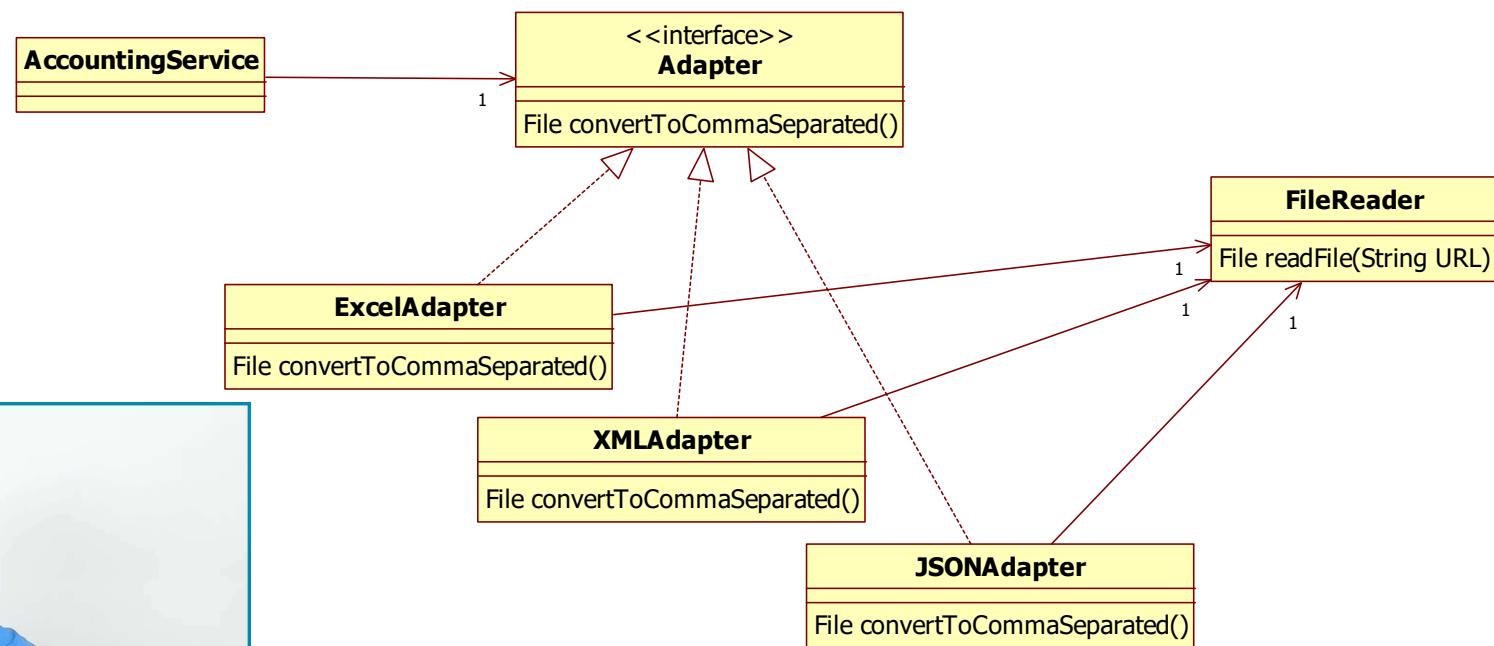
```
public class KilometerDistanceAdapterImpl implements DistanceAdapter {  
    private DistanceCalculator distanceCalculator;  
  
    public double computeDistanceInKilometers(String origin, String destination) {  
        double distanceInMiles = distanceCalculator.computeDistance(origin, destination);  
        return distanceInMiles * 1.609344;  
    }  
  
    public void setDistanceCalculator(DistanceCalculator distanceCalculator) {  
        this.distanceCalculator = distanceCalculator;  
    }  
}
```

Client

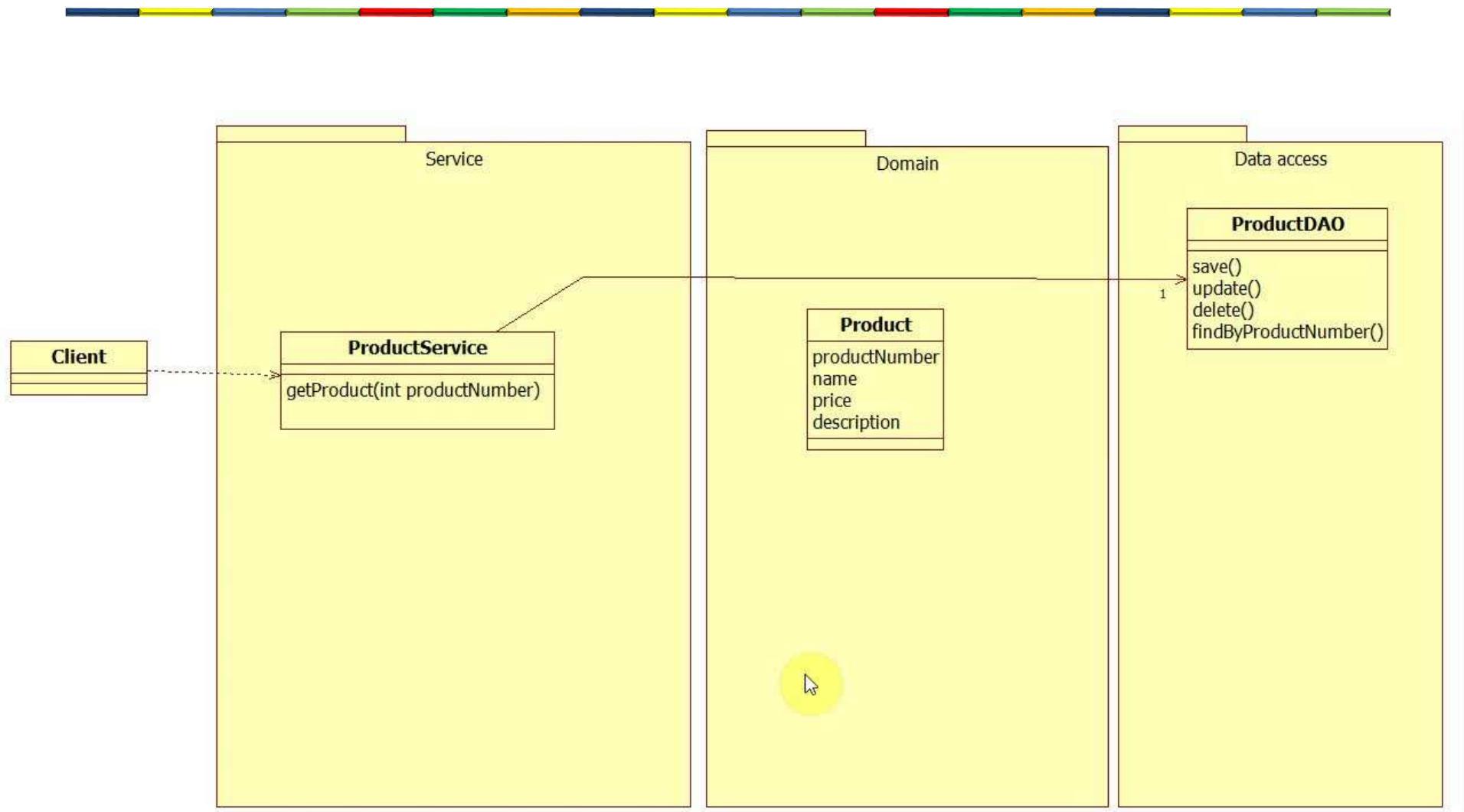
```
public class Client {  
  
    public static void main(String[] args) {  
        DistanceCalculator distanceCalculator = new DistanceCalculator();  
        double distanceInMiles = distanceCalculator.computeDistance("city1", "city2");  
        System.out.println("The distance between city1 and city2 =" + distanceInMiles + " miles");  
  
        DistanceAdapter distanceAdapter = new KilometerDistanceAdapterImpl();  
        distanceAdapter.setDistanceCalculator(distanceCalculator);  
  
        double distanceInKilometers = distanceAdapter.computeDistanceInKilometers("city3", "city4");  
        System.out.println("The distance between city3 and city4 =" + distanceInKilometers + " kilometers");  
    }  
}
```



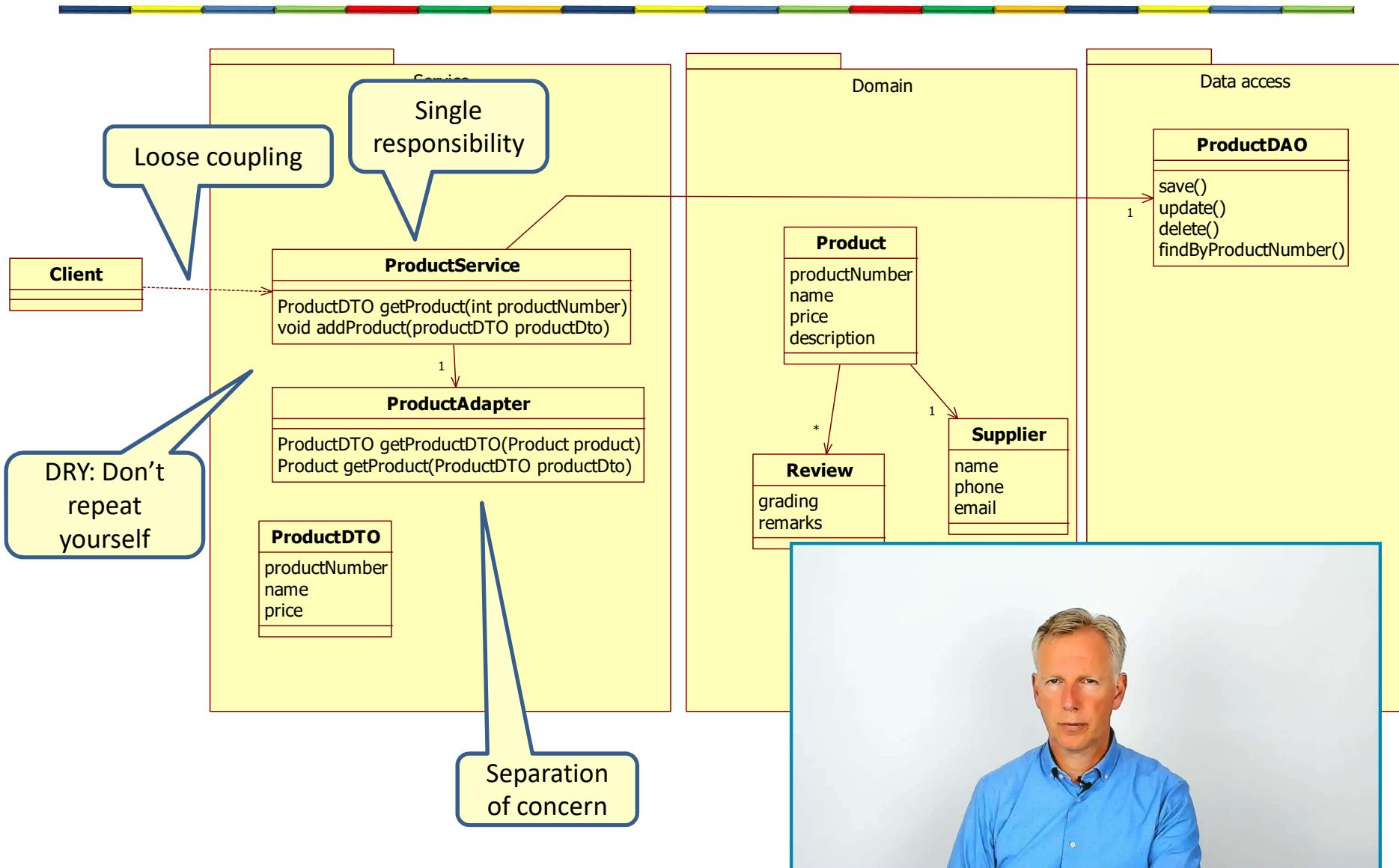
Where are adapters used



Where are adapters used

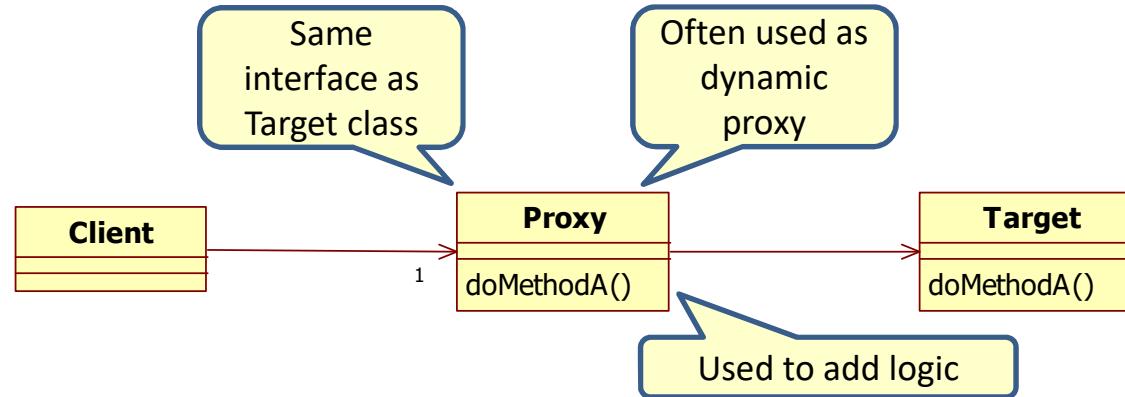


Where are adapters used

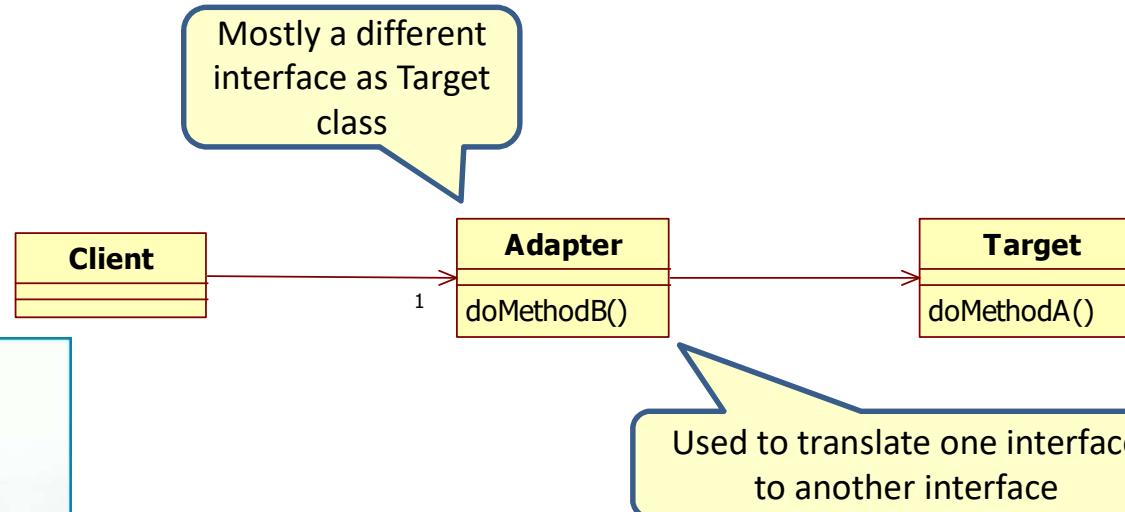


Adapter and Proxy: wrapper

■ Proxy



■ Adapter



Main point

- The Adapter translates an existing interface to a required interface.
- Life is found in layers, from the most abstract transcendental layer (Unified Field) to the most concrete layer.

Lesson 8 Mediator pattern

- L1: ASD Introduction
- L2: Strategy, Template method
- L3: Observer pattern
- L4: Composite pattern, iterator pattern
- L5: Command pattern
- L6: State pattern
- L7: Chain Of Responsibility pattern



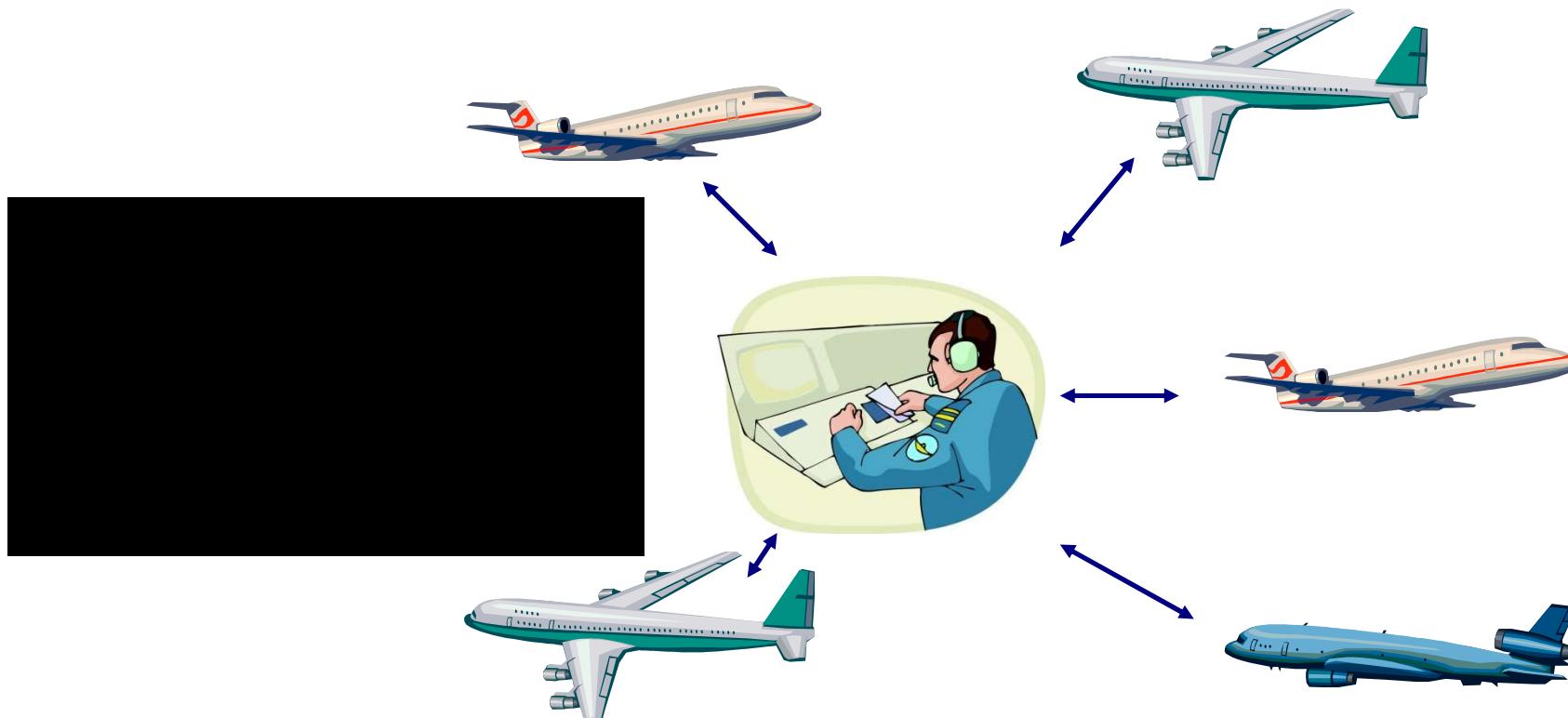
Midterm

- L8: Proxy, Adapter, Mediator
- L9: Factory, Builder, Decorator, Singleton
- L10: Framework design
- L11: Framework implementation
- L12: Framework example: Spring framework
- L13: Framework example: Spring framework

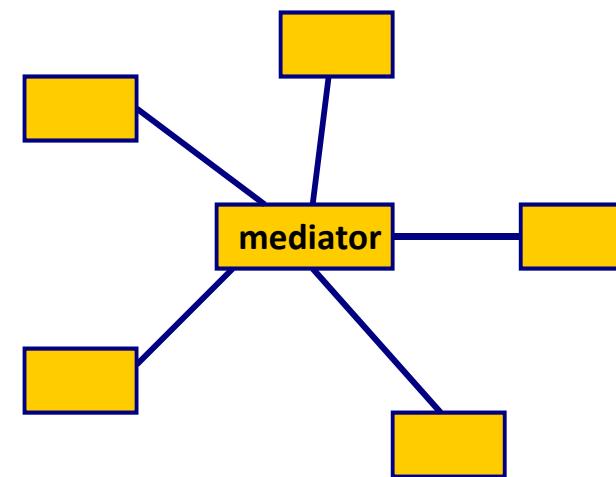
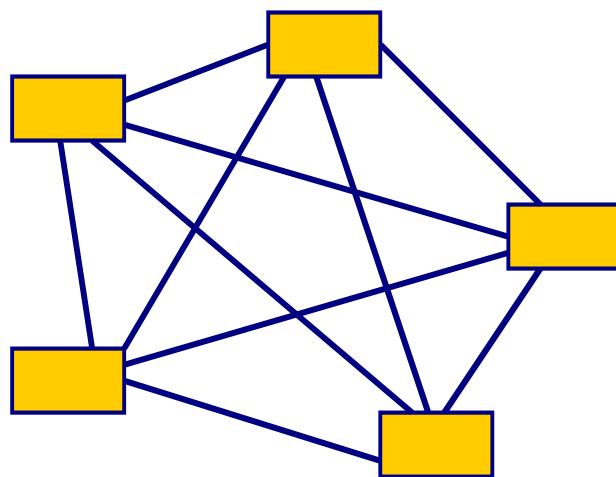
Final

Mediator pattern

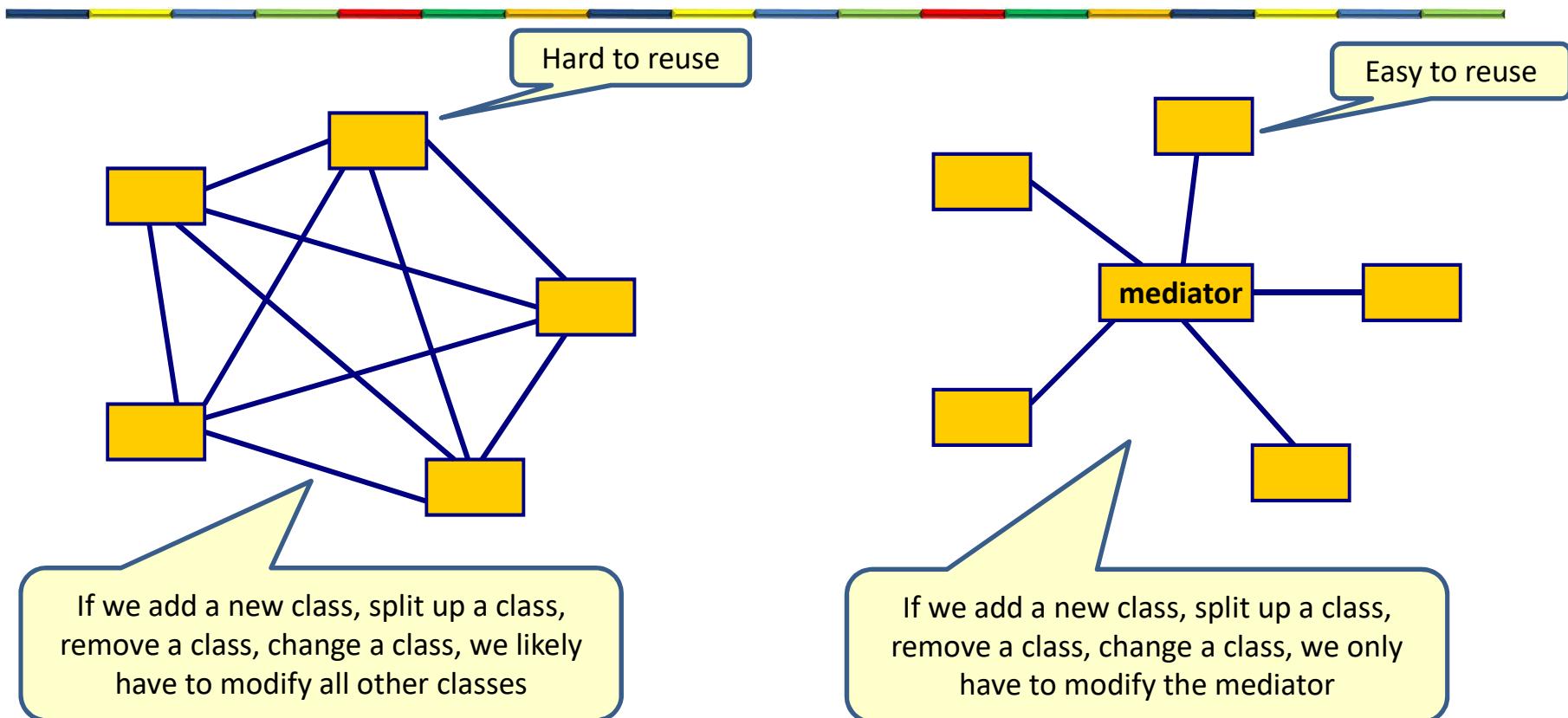
- Mediates between objects.
 - Encapsulates how different objects interact.



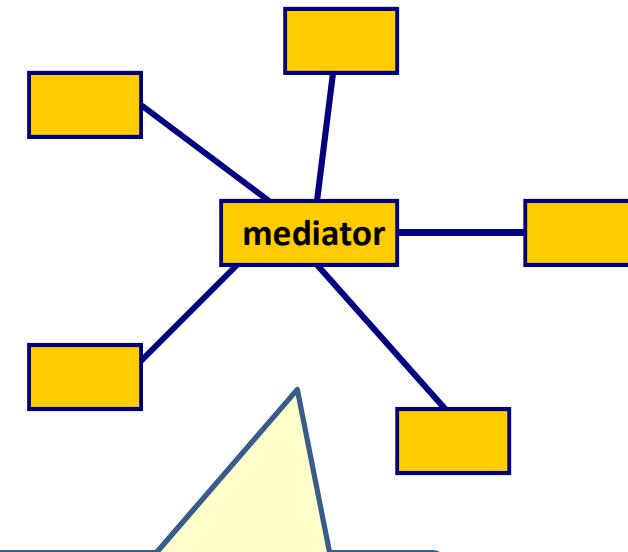
Mediator



Advantage Mediator: loose coupling

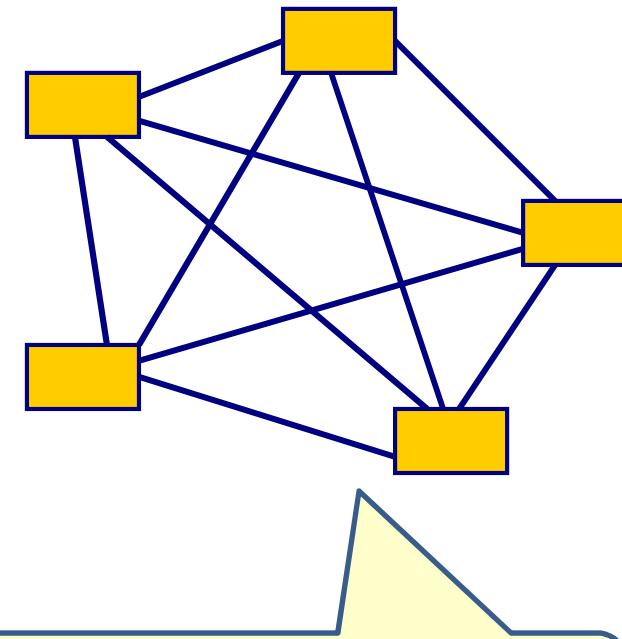


Disadvantage Mediator



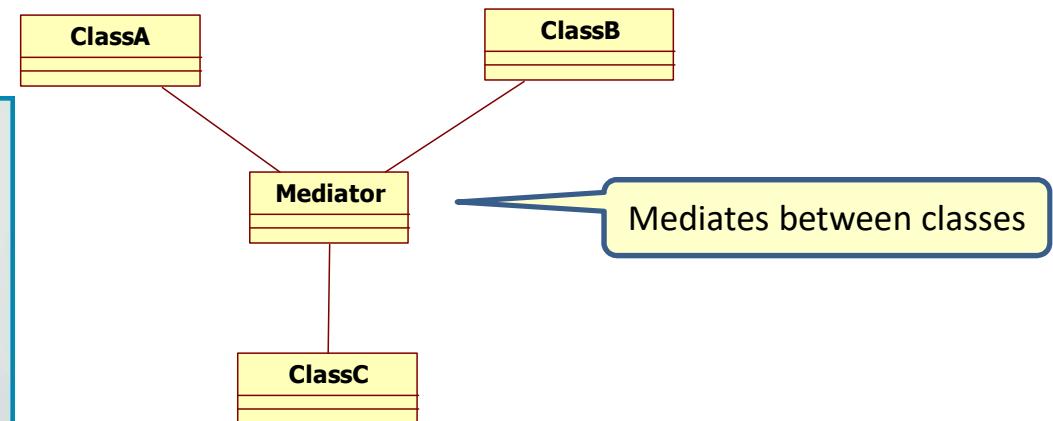
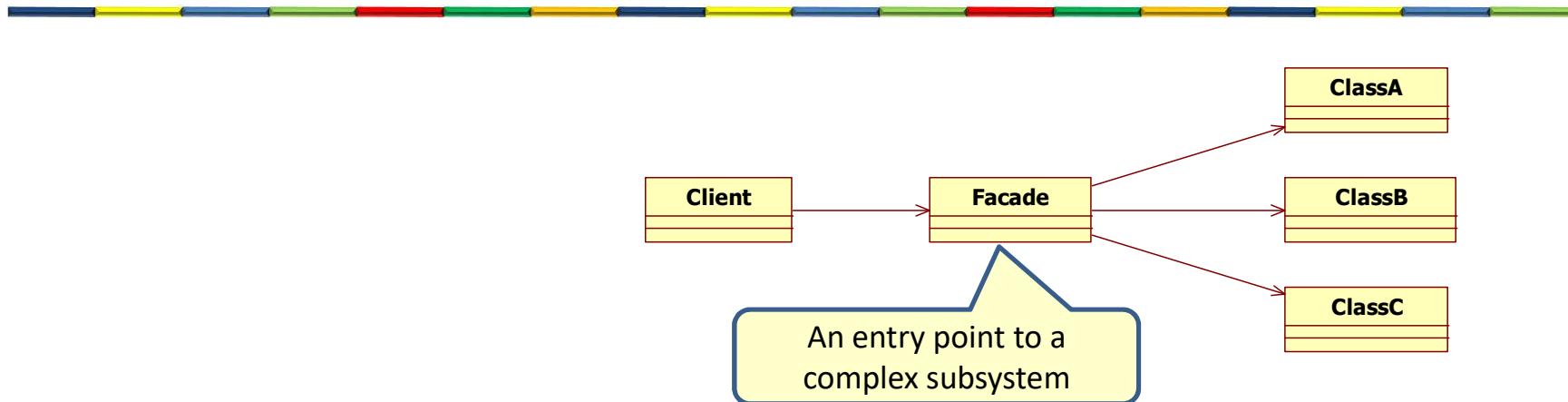
The mediator can become very complex

How did we get to this?

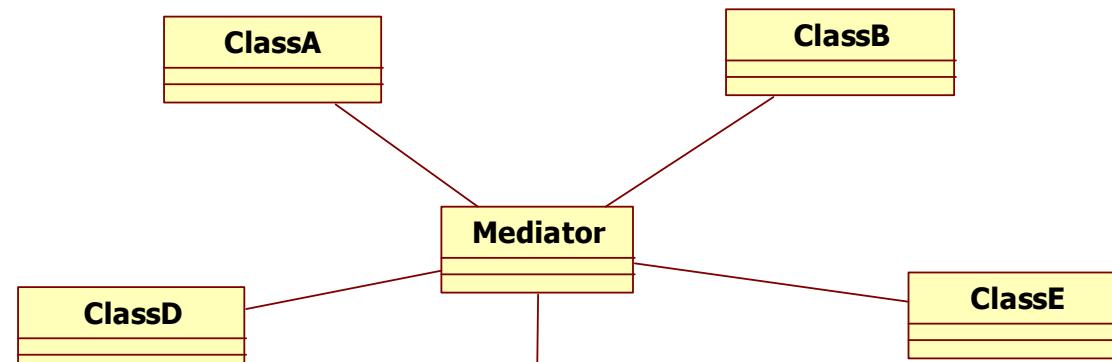


If you end up with a class diagram like this, something went wrong.
This is not good OO design!

Façade and Mediator



Examples of the mediator

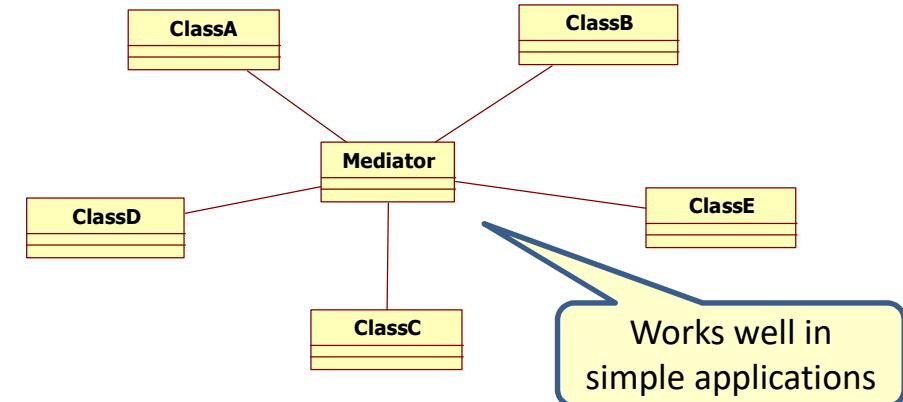


Do not use a mediator like this, unless

Orchestration vs. choreography

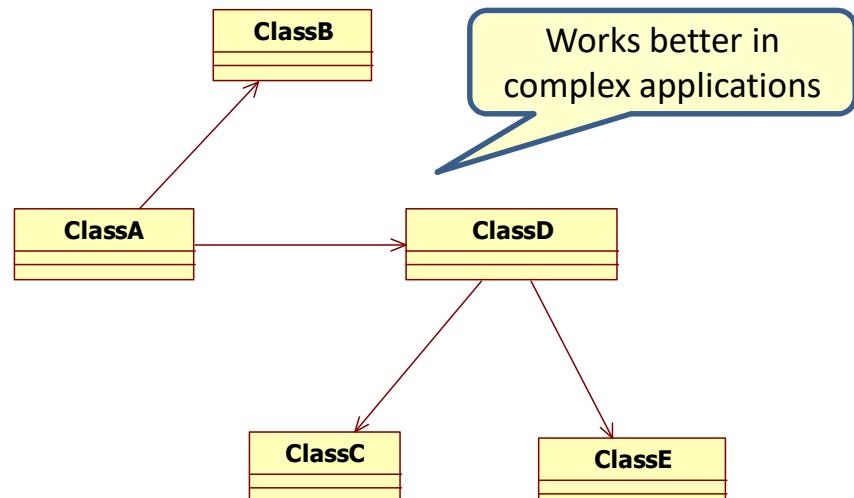
- **Orchestration**

- One central brain

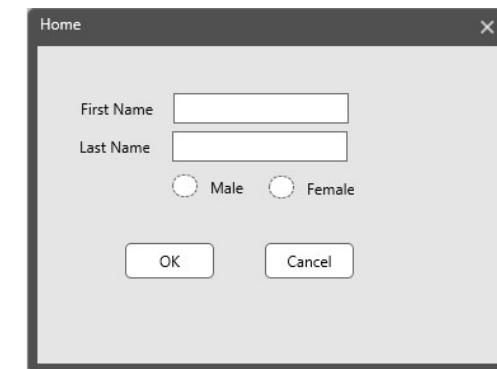
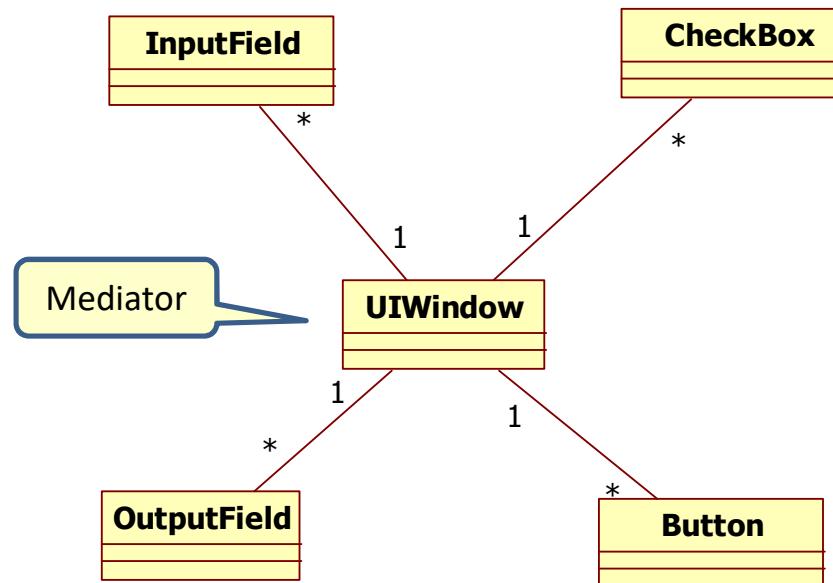


- **Choreography**

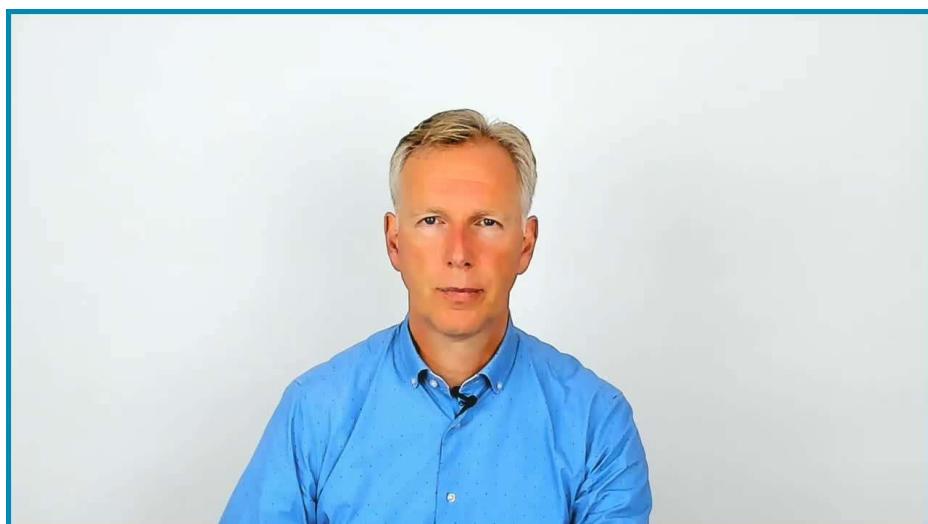
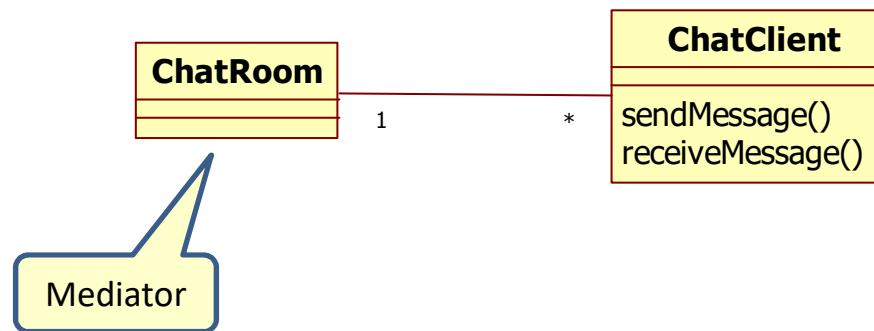
- No central brain



Examples of the mediator



Examples of the mediator



Main point

- The Mediator object is responsible for controlling and coordinating the interactions of a group of objects.
- The Unified Field is the source of creation that coordinates all interactions in the relative world.

Connecting the parts of knowledge with the wholeness of knowledge

1. The mediator pattern is an orchestrator between objects.
2. The proxy and the adapter are both a layer of indirection that solves a certain problem between the client and the target class

-
3. **Transcendental consciousness** is the natural experience of pure consciousness, the home of all the laws of nature.
 4. **Wholeness moving within itself:** In Unity Consciousness, one realizes that all relative objects are expressions of the field of Pure Intelligence.



Lesson 9 Factory pattern

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

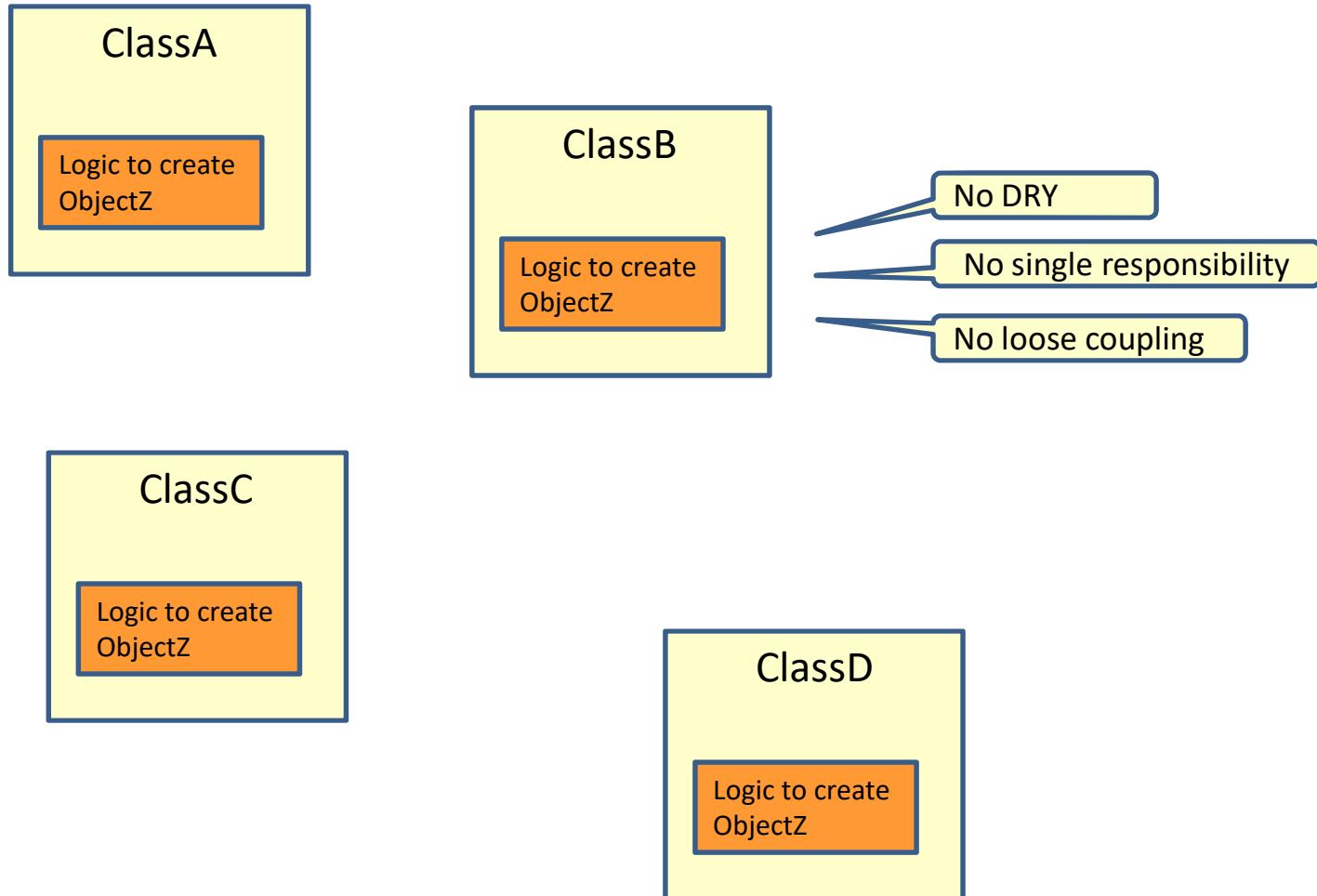
Final

Factory pattern

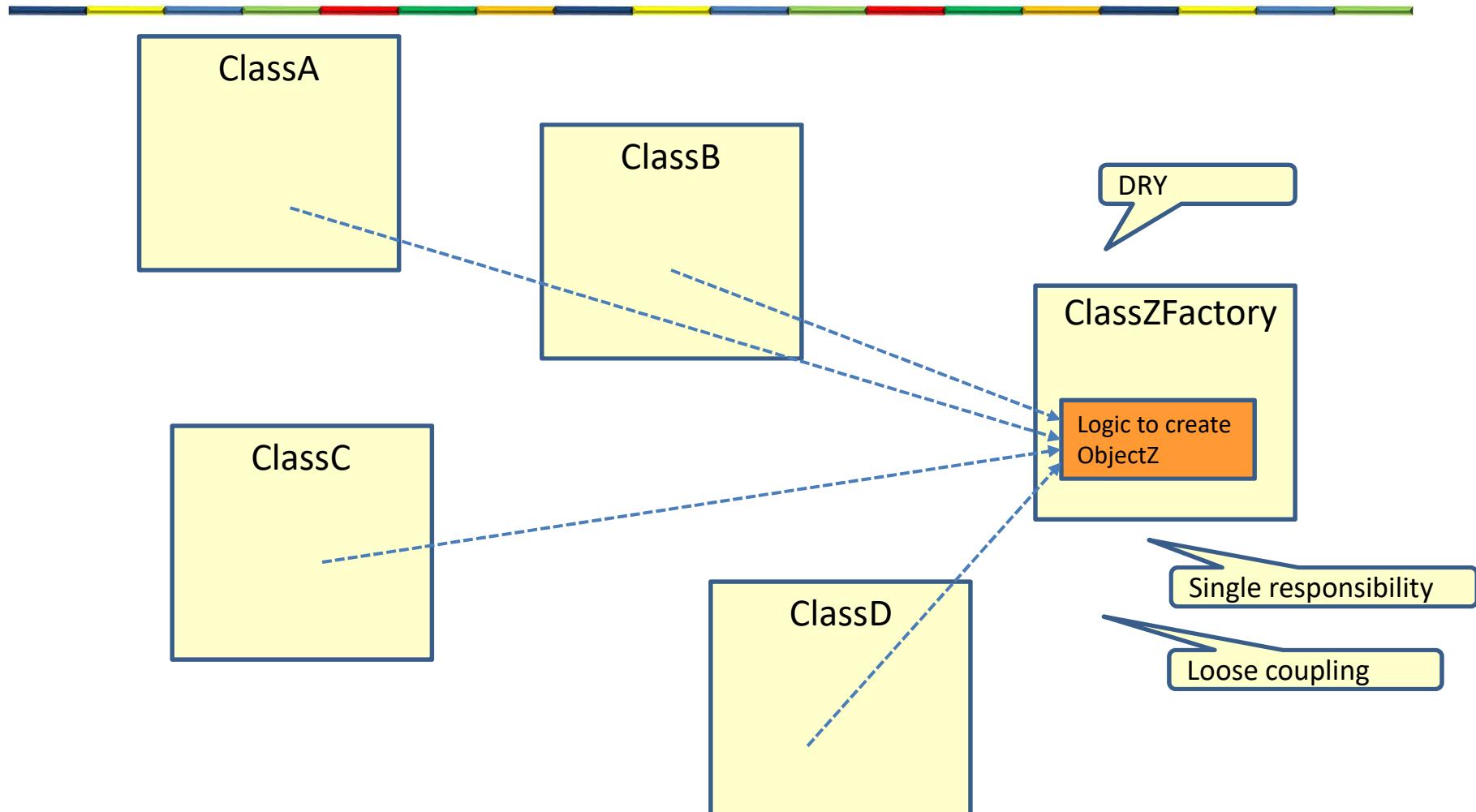
- A factory creates objects
 - Encapsulation of the logic to create objects



Without a factory



With a factory



Different types of factories

- Simple factory method
 - Static or not static
- Factory method pattern
- Abstract factory pattern

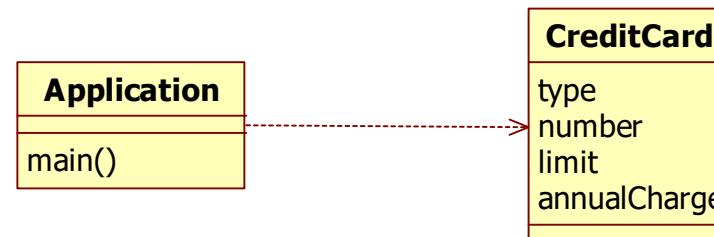


SIMPLE FACTORY METHOD

Using the constructor

```
public class CreditCard {  
    private String type;  
    private String number;  
    private double limit;  
    private double annualCharge;  
  
    public CreditCard(String type, String number, double limit, double annualCharge) {  
        this.type = type;  
        this.number = number;  
        this.limit = limit;  
        this.annualCharge = annualCharge;  
    }  
}
```

```
public class Application {  
  
    public static void main(String[] args) {  
        // with constructor  
        CreditCard creditCard = new CreditCard("visa", "1232786598763429", 2500.0, 10.0);  
    }  
}
```

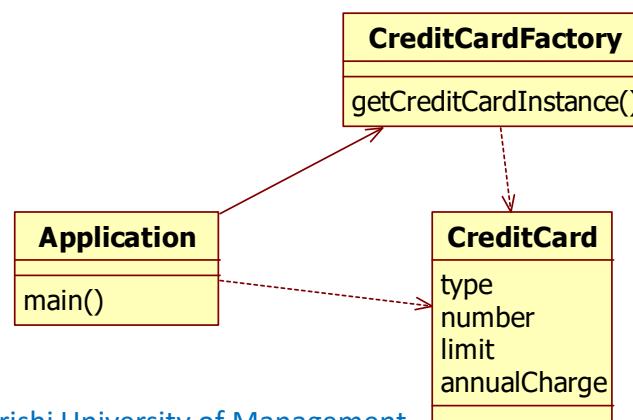


Using a static factory method

```
public class CreditCardFactory {  
    static CreditCard getCreditCardInstance(String type, String number, double limit,  
                                         double annualCharge) {  
        return new CreditCard(type, number, limit, annualCharge);  
    }  
}
```

Static factory
method

```
public class Application {  
  
    public static void main(String[] args) {  
        //with factory  
        CreditCard creditCard2 = CreditCardFactory.getCreditCardInstance("visa",  
                           "1232786598763429", 2500.0, 10);  
    }  
}
```



What is the difference?

```
public class Application {  
  
    public static void main(String[] args) {  
        // with constructor  
        CreditCard creditCard = new CreditCard("visa", "1232786598763429", 2500.0, 10);  
  
        //with factory  
        CreditCard creditCard2 = CreditCardFactory.getCreditCardInstance("visa",  
                            "1232786598763429", 2500.0, 10);  
    }  
}
```

- In this simple case: not much
 - But when creating objects get more complex, we can encapsulate this complexity in the factory method

Constructor

```
public class RandomIntGenerator {  
    private final int min;  
    private final int max;  
  
    public RandomIntGenerator(int min, int max) {  
        this.min = min;  
        this.max = max;  
    }  
  
    public RandomIntGenerator(int min) {  
        this.min = min;  
        this.max = Integer.MAX_VALUE;  
    }  
  
    public RandomIntGenerator(int max) {  
        this.max = max;  
        this.min = Integer.MIN_VALUE;  
    }  
  
    public int next() {...}  
}
```

Constructors do not have meaningful names

Constructors cannot return anything else:

- A subclass
- A cached class

Compilation error

```
RandomIntGenerator randomIntGenerator = new RandomIntGenerator(40, 100);
```

```
RandomIntGenerator randomIntGenerator = new RandomIntGenerator(50);
```

Static factory method

```
public class RandomIntGenerator {  
    private final int min;  
    private final int max;  
  
    private RandomIntGenerator(int min, int max) {  
        this.min = min;  
        this.max = max;  
    }  
  
    public static RandomIntGenerator between(int max, int min) {  
        return new RandomIntGenerator(min, max);  
    }  
  
    public static RandomIntGenerator biggerThan(int min) {  
        return new RandomIntGenerator(min, Integer.MAX_VALUE);  
    }  
  
    public static RandomIntGenerator smallerThan(int max) {  
        return new RandomIntGenerator(Integer.MIN_VALUE, max);  
    }  
  
    public int next() {...}  
}
```

Private !

Factory methods can return anything:

- A subclass
- A cached class

Meaningful names

We can have multiple factory methods with the same argument(s)

```
RandomIntGenerator randomIntGenerator = RandomIntGenerator.between(40, 100);  
RandomIntGenerator randomIntGenerator = RandomIntGenerator.smallerThan(50);  
RandomIntGenerator randomIntGenerator = RandomIntGenerator.biggerThan(50);
```

Prefer factory methods over constructors

```
// with constructor  
Range range = new Range( 0 , n-1);
```

```
//with factory  
Range range = RangeFactory.getUpto(n);
```

More descriptive

More flexible: Can return also
subclasses of Range

Testability: Can return also
MockRange which subclasses Range

Java 8 LocalTime

java.time

Class LocalTime

No constructors!

Static factory methods

static LocalTime now()
Obtains the current time from the system clock in the default time-zone.

static LocalTime now(Clock clock)
Obtains the current time from the specified clock.

static LocalTime now(ZoneId zone)
Obtains the current time from the system clock in the specified time-zone.

static LocalTime of(int hour, int minute)
Obtains an instance of LocalTime from an hour and minute.

static LocalTime of(int hour, int minute, int second)
Obtains an instance of LocalTime from an hour, minute and second.

static LocalTime of(int hour, int minute, int second, int nanoOfSecond)
Obtains an instance of LocalTime from an hour, minute, second and nanosecond.

static LocalTime ofNanoOfDay(long nanoOfDay)
Obtains an instance of LocalTime from a nanos-of-day value.

static LocalTime ofSecondOfDay(long secondOfDay)
Obtains an instance of LocalTime from a second-of-day value.

static LocalTime parse(CharSequence text)
Obtains an instance of LocalTime from a text string such as 10:15.

static LocalTime parse(CharSequence text, DateTimeFormatter formatter)
Obtains an instance of LocalTime from a text string using a specific formatter.

More descriptive

Logging static factory method

```
public class Application {  
    public static void main(String[] args) {  
        ProductService productService = new ProductService();  
        productService.addProduct();  
    }  
}
```

```
import java.util.logging.Logger;  
  
public class ProductService {  
    static Logger Logger = Logger.getLogger(ProductService.class.getName());  
  
    public void addProduct() {  
        Logger.info("Add a product");  
    }  
}
```

Static factory method

```
Aug 19, 2019 12:24:26 PM test.ProductService addProduct  
INFO: Add a product
```

Calendar static factory methods

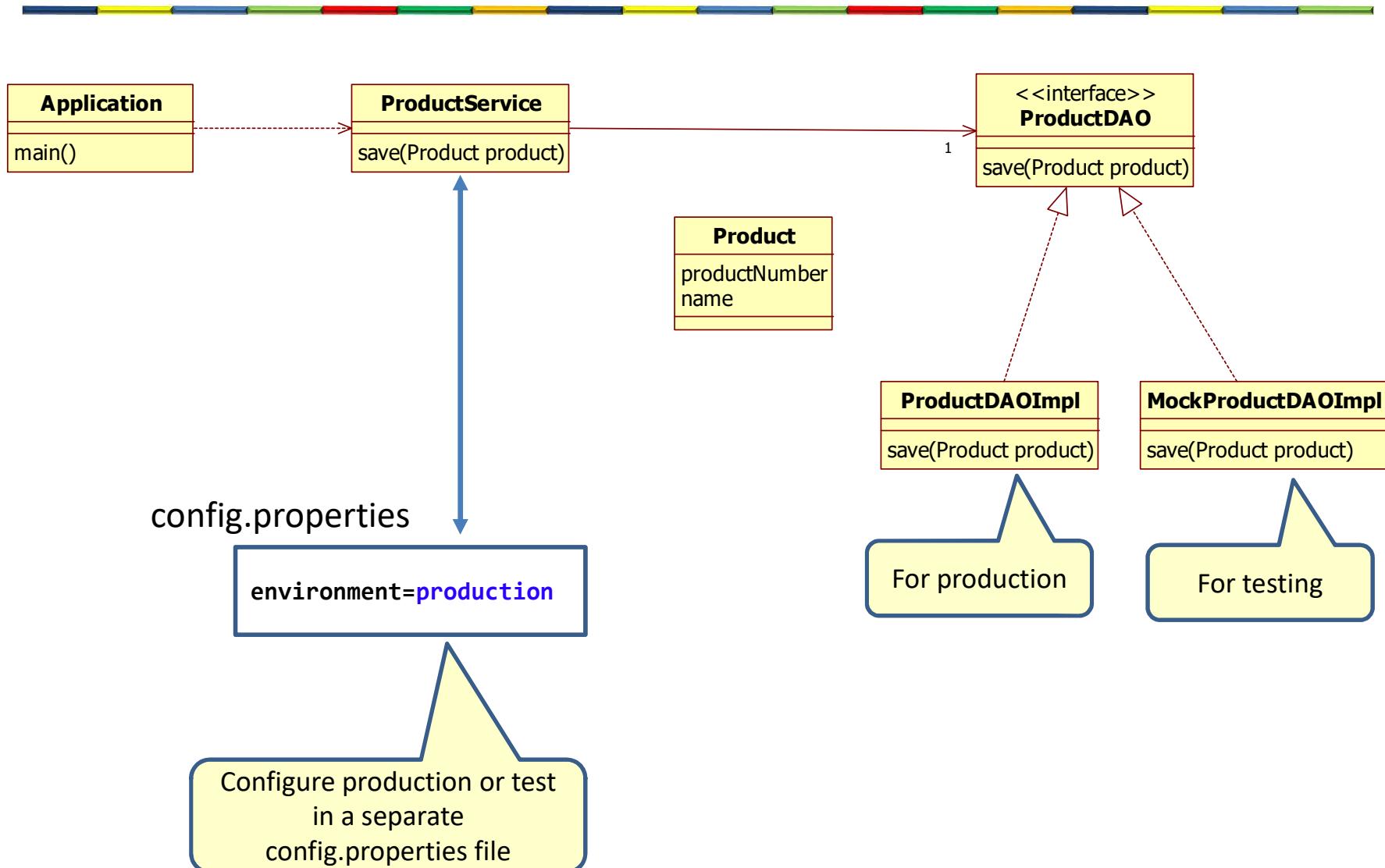
java.util

Class Calendar

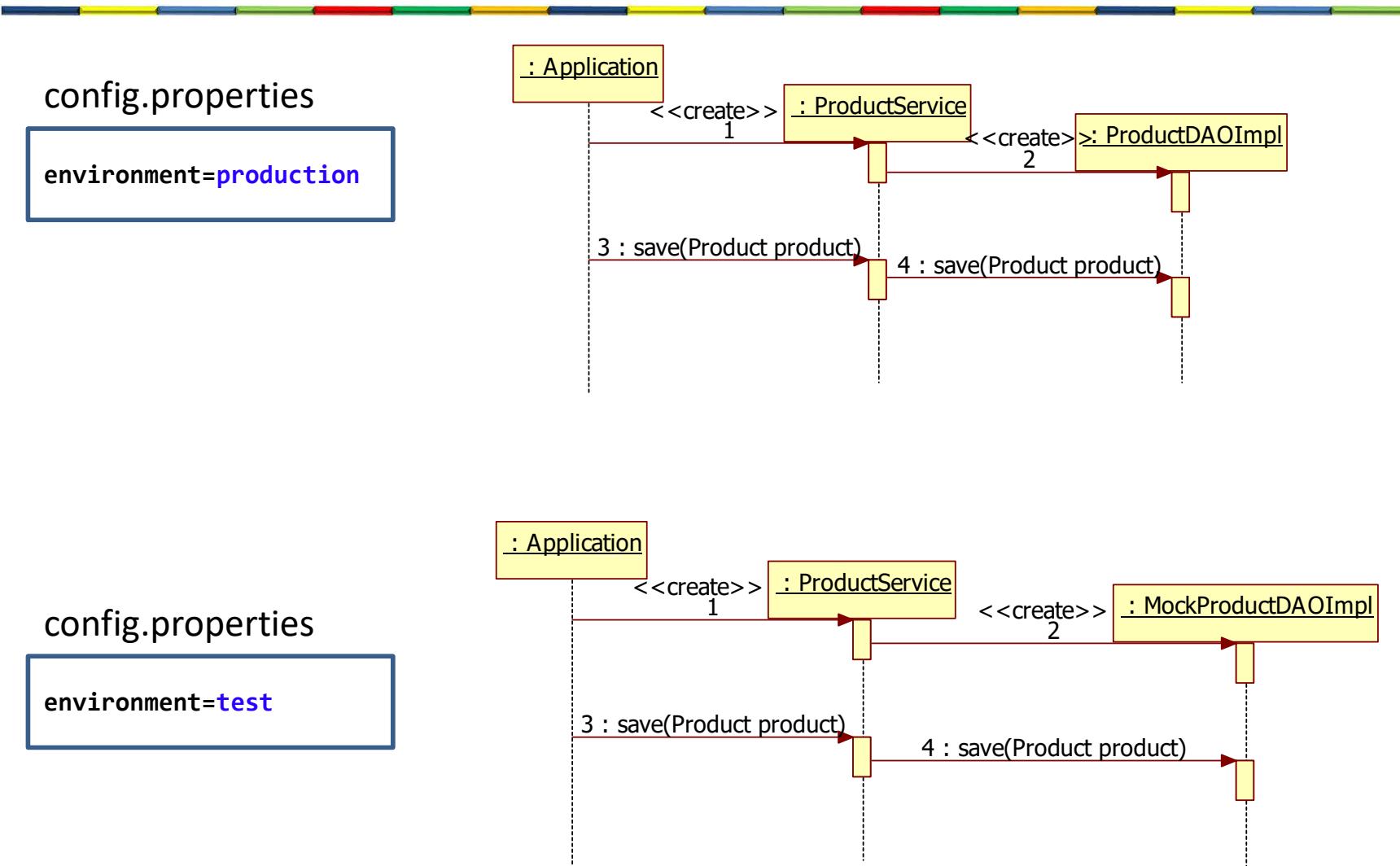
Static factory methods

static Calendar	getInstance()	Gets a calendar using the default time zone and locale.
static Calendar	getInstance(Locale aLocale)	Gets a calendar using the default time zone and specified locale.
static Calendar	getInstance(TimeZone zone)	Gets a calendar using the specified time zone and default locale.
static Calendar	getInstance(TimeZone zone, Locale aLocale)	Gets a calendar with the specified time zone and locale.

Example application



Example application



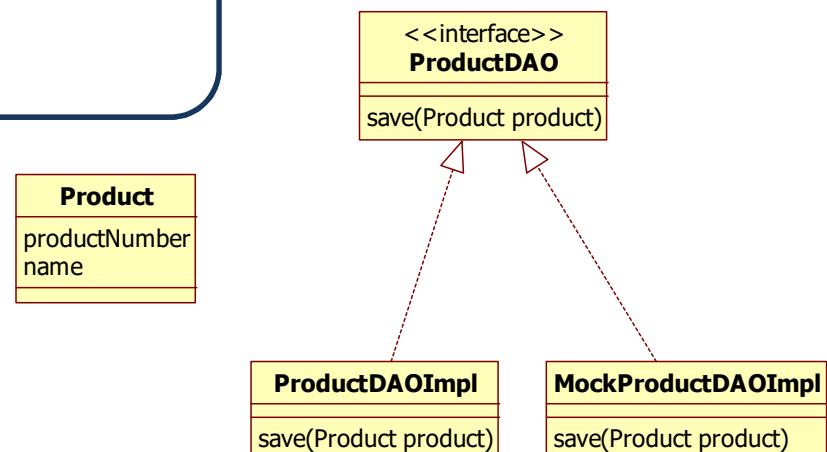
Product and DAO

```
public interface ProductDAO {  
    void save(Product product);  
}
```

```
public class ProductDAOImpl implements ProductDAO{  
  
    public void save(Product product) {  
        System.out.println("ProductDAOImpl saves product");  
    }  
}
```

```
public class MockProductDAOImpl implements ProductDAO{  
  
    public void save(Product product) {  
        System.out.println("MockProductDAOImpl saves product");  
    }  
}
```

```
public class Product {  
    private int productNumber;  
    private String name;  
  
    ....  
}
```



Product service

```
public class ProductService {  
    ProductDAO productDAO;  
  
    public ProductService() {  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            Properties prop = new Properties();  
            // load the properties file  
            prop.load(new FileInputStream(rootPath+"/config.properties"));  
            // get the property value  
            String environment= prop.getProperty("environment");  
  
            if (environment.equals("production")) {  
                productDAO = new ProductDAOImpl();  
            } else if (environment.equals("test")) {  
                productDAO = new MockProductDAOImpl();  
            } else {  
                System.out.println("environment property not set correctly");  
            }  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public void save(Product product) {  
        productDAO.save(product);  
    }  
}
```

Example application

```
public class Application {  
  
    public static void main(String[] args) {  
        Product product = new Product(3324, "DJI Mavic 2 Pro drone");  
  
        ProductService productService = new ProductService();  
        productService.save(product);  
    }  
}
```

config.properties

ProductDAOImpl saves product

environment=production

config.properties

MockProductDAOImpl saves product

environment=test

What is the problem?

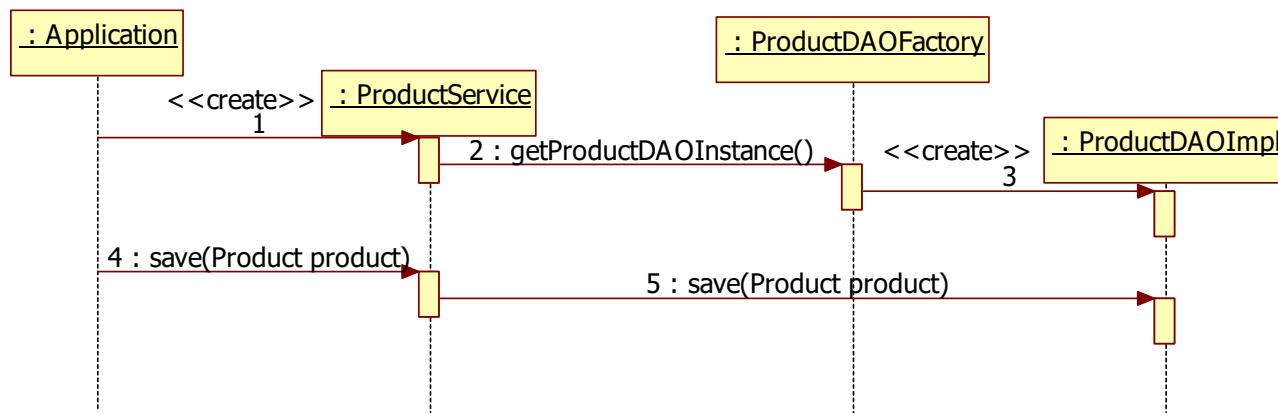
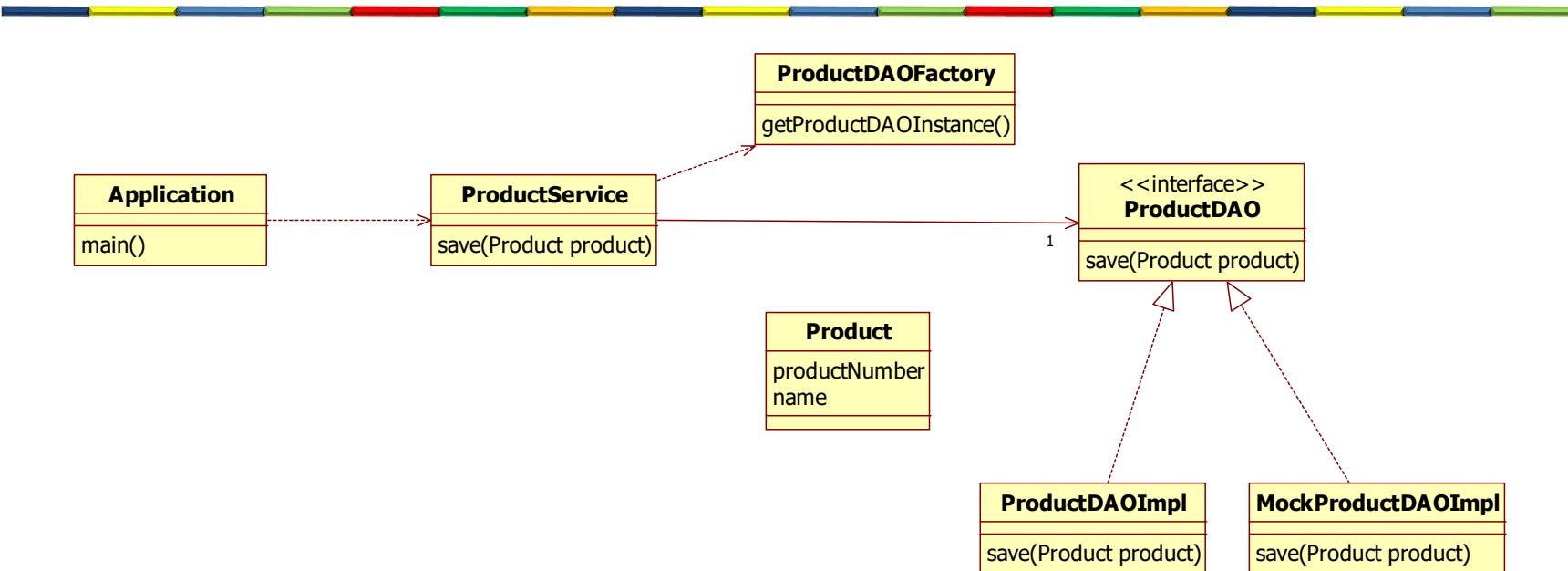
```
public class ProductService {  
    ProductDAO productDAO;  
  
    public ProductService() {  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            Properties prop = new Properties();  
            // load the properties file  
            prop.load(new FileInputStream(rootPath+"/config.properties"));  
            // get the property value  
            String environment= prop.getProperty("environment");  
  
            if (environment.equals("production")) {  
                productDAO = new ProductDAOImpl();  
            } else if (environment.equals("test")) {  
                productDAO = new MockProductDAOImpl();  
            } else {  
                System.out.println("environment property not set correctly");  
            }  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public void save(Product product) {  
        productDAO.save(product);  
    }  
}
```

ProductService contains complex logic about creating the ProductDAO

This code has to be copied to every class that needs the ProductDAO

Every service class that needs a DAO needs to have code like this

Solution: Factory method



Solution: Factory method

```
public class ProductDAOFactory {  
    static ProductDAO getProductDAOInstance() {  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            Properties prop = new Properties();  
            // load the properties file  
            prop.load(new FileInputStream(rootPath + "/config.properties"));  
            // get the property value  
            String environment = prop.getProperty("environment");  
  
            if (environment.equals("production")) {  
                return new ProductDAOImpl();  
            } else if (environment.equals("test")) {  
                return new MockProductDAOImpl();  
            } else {  
                System.out.println("environment property not set correctly");  
            }  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
        return null;  
    }  
}
```

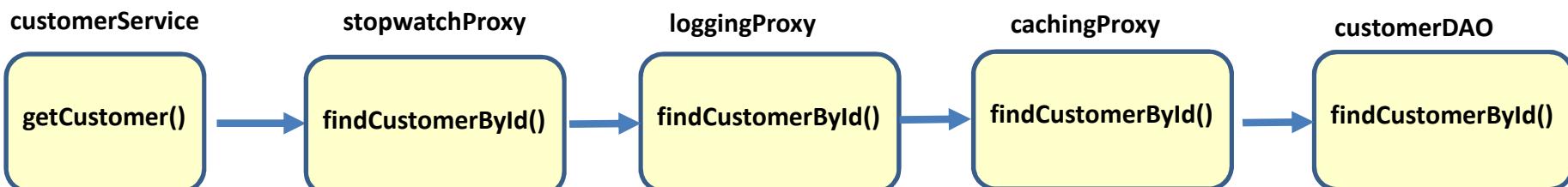
Encapsulate the logic
to create objects

```
public class ProductService {  
    ProductDAO productDAO;  
  
    public ProductService() {  
        productDAO=ProductDAOFactory.getProductDAOInstance();  
    }  
  
    public void save(Product product) {  
        productDAO.save(product);  
    }  
}
```

Creating a dynamic proxy

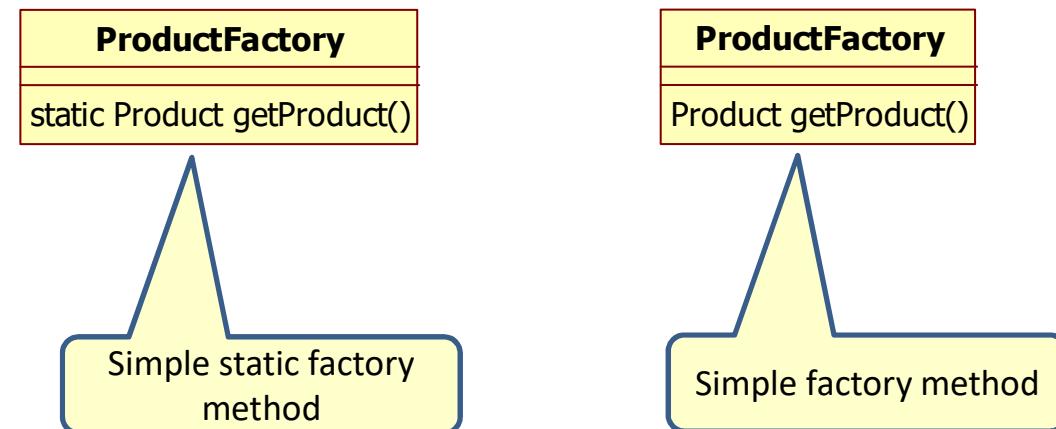
```
public class CustomerService {  
    CustomerDAO customerDAO = new CustomerDAOImpl();  
    ClassLoader classLoader = CustomerDAO.class.getClassLoader();  
    CustomerDAO cachingProxy =  
        (CustomerDAO) Proxy.newProxyInstance(classLoader,  
            new Class[] { CustomerDAO.class },  
            new CachingProxy(customerDAO));  
    CustomerDAO loggingProxy =  
        (CustomerDAO) Proxy.newProxyInstance(classLoader,  
            new Class[] { CustomerDAO.class },  
            new LoggingProxy(cachingProxy));  
    CustomerDAO stopwatchProxy =  
        (CustomerDAO) Proxy.newProxyInstance(classLoader,  
            new Class[] { CustomerDAO.class },  
            new StopwatchProxy(loggingProxy));  
  
    public Customer getCustomer(int customerId) {  
        return stopwatchProxy.findCustomerById(customerId);  
    }  
}
```

Move complex logic
for creating dynamic
proxies into a factory



Factory method that is not static

- Similar as static factory method, only now you instantiate the factory object, and then call the factory method.
 - Factory class needs state
 - Caching



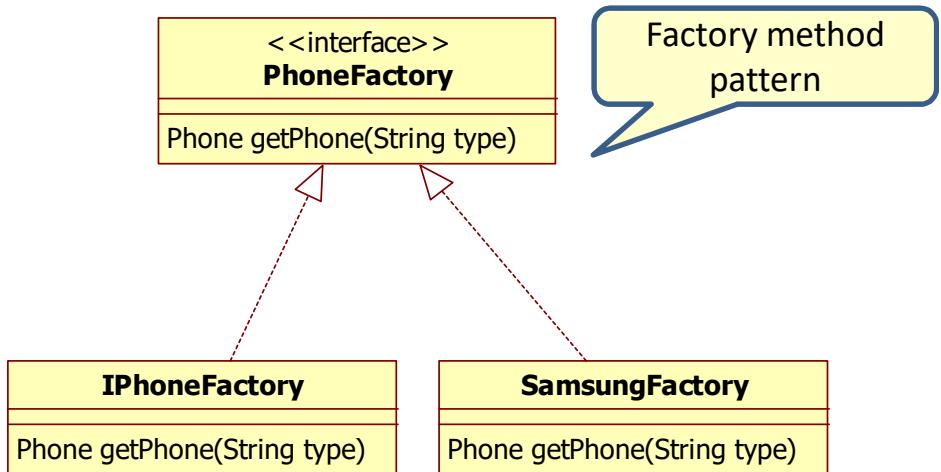
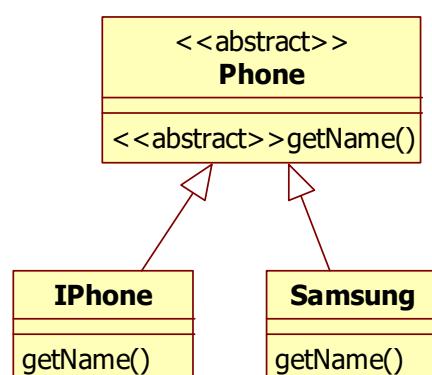
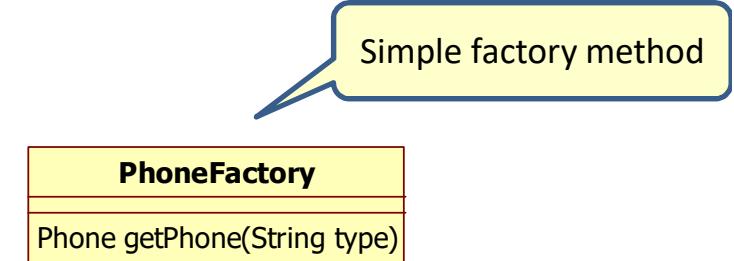
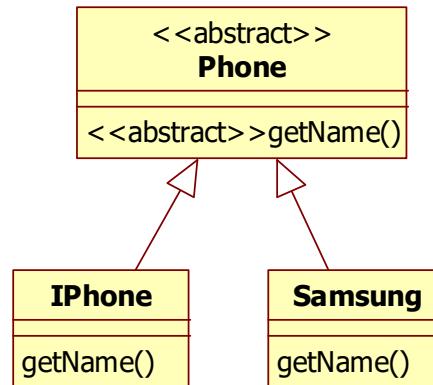
FACTORY METHOD PATTERN

Factory method pattern

- Defines an **interface** for creating an object, but leaves the choice of its type to the subclasses,
- Factory method lets the class creation being deferred at run-time.
 - Polymorphic factory



Simple factory method vs. Factory method pattern

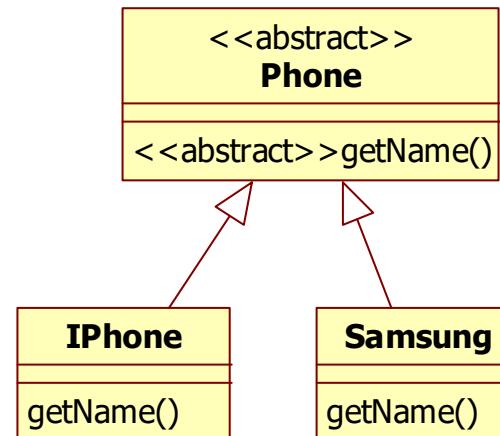


The phones

```
public abstract class Phone {  
    public abstract String getName();  
}
```

```
public class IPhone extends Phone{  
  
    @Override  
    public String getName() {  
        return "Iphone";  
    }  
}
```

```
public class Samsung extends Phone{  
  
    @Override  
    public String getName() {  
        return "Samsung phone";  
    }  
}
```

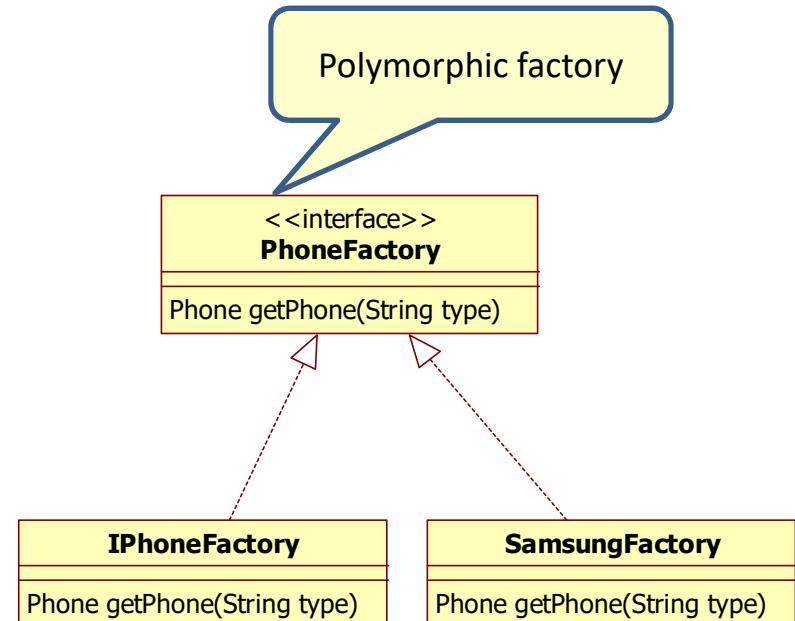


The phone factories

```
public interface PhoneFactory {  
    Phone getPhone();  
}
```

```
public class IPhoneFactory implements PhoneFactory{  
  
    @Override  
    public Phone getPhone() {  
        return new IPhone();  
    }  
}
```

```
public class SamsungFactory implements PhoneFactory{  
  
    @Override  
    public Phone getPhone() {  
        return new Samsung();  
    }  
}
```



The service and application

```
public class PhoneService {  
    private PhoneFactory phoneFactory;  
  
    public void setPhoneFactory(PhoneFactory phoneFactory) {  
        this.phoneFactory = phoneFactory;  
    }  
  
    public Phone getPhone() {  
        return phoneFactory.getPhone();  
    }  
}
```

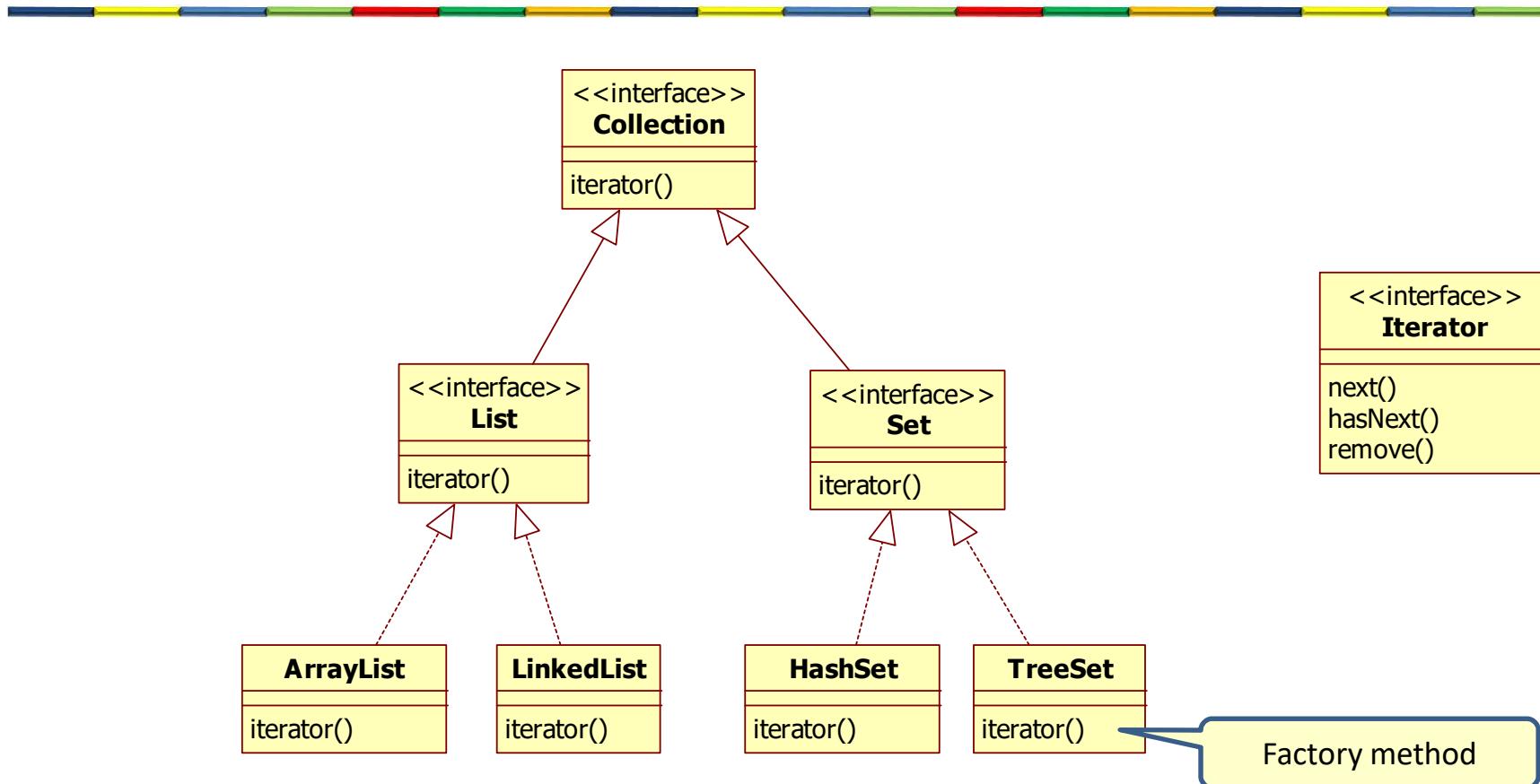
Flexibility: You can set (inject) any PhoneFactory

Testability: inject a MockPhoneFactory

```
public class Application {  
  
    public static void main(String[] args) {  
        PhoneService phoneService = new PhoneService();  
        phoneService.setPhoneFactory(new IPhoneFactory());  
        System.out.println(phoneService.getPhone().getName());  
  
        phoneService.setPhoneFactory(new SamsungFactory());  
        System.out.println(phoneService.getPhone().getName());  
    }  
}
```

Iphone
Samsung phone

iterator() factory method



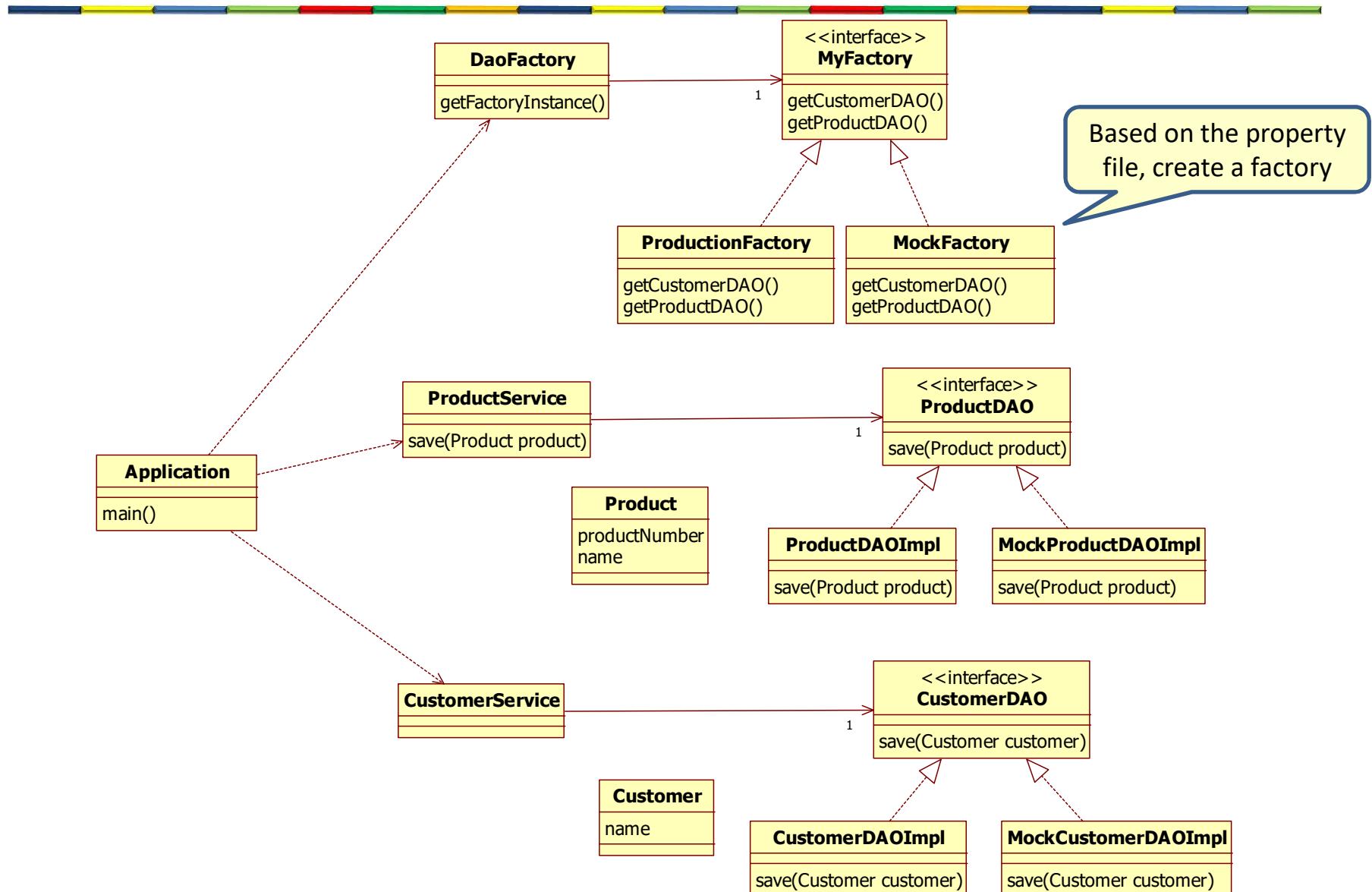
ABSTRACT FACTORY PATTERN

Abstract factory pattern

- Provides an interface for creating **families of related objects** without specifying their concrete classes.
 - Factory of factories



Abstract factory pattern example



Abstract factory example

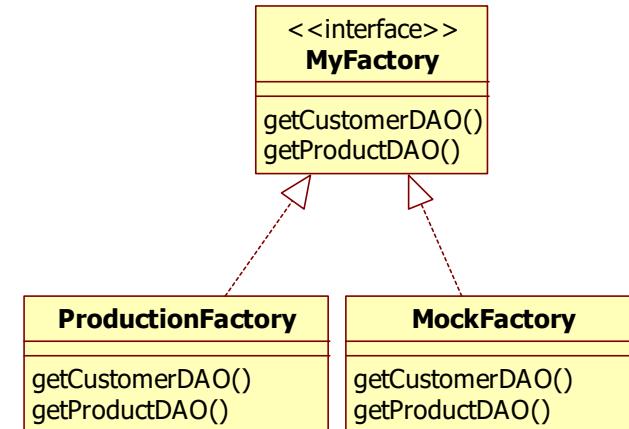
```
public class DaoFactory {  
    private MyFactory factory;  
  
    public DaoFactory() {  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            Properties prop = new Properties();  
            // load the properties file  
            prop.load(new FileInputStream(rootPath + "/config.properties"));  
            // get the property value  
            String environment = prop.getProperty("environment");  
  
            if (environment.equals("production")) {  
                factory= new ProductionFactory();  
            } else if (environment.equals("test")) {  
                factory= new MockFactory();  
            } else {  
                System.out.println("environment property not set correctly");  
            }  
        } catch (FileNotFoundException e) {  
            e.printStackTrace();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
  
    public MyFactory getFactoryInstance() {  
        return factory;  
    }  
}
```

Abstract factory example

```
public interface MyFactory {  
    public CustomerDAO getCustomerDAO();  
    public ProductDAO getProductDAO();  
}
```

```
public class ProductionFactory implements MyFactory{  
    public CustomerDAO getCustomerDAO() {  
        return new CustomerDAOImpl();  
    }  
  
    public ProductDAO getProductDAO() {  
        return new ProductDAOImpl();  
    }  
}
```

```
public class MockFactory implements MyFactory{  
    public CustomerDAO getCustomerDAO() {  
        return new MockCustomerDAOImpl();  
    }  
  
    public ProductDAO getProductDAO() {  
        return new MockProductDAOImpl();  
    }  
}
```



Product and DAO

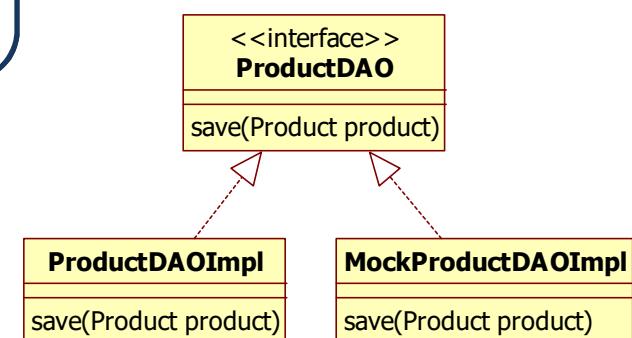
```
public interface ProductDAO {  
    void save(Product product);  
}
```

```
public class ProductDAOImpl implements ProductDAO{  
  
    public void save(Product product) {  
        System.out.println("ProductDAOImpl saves product");  
    }  
}
```

```
public class MockProductDAOImpl implements ProductDAO{  
  
    public void save(Product product) {  
        System.out.println("MockProductDAOImpl saves product");  
    }  
}
```

```
public class Product {  
    private int productNumber;  
    private String name;  
  
    ....  
}
```

Product
productNumber
name



Customer and DAO

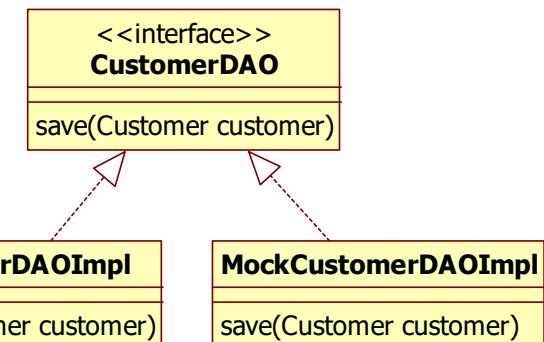
```
public interface CustomerDAO {  
    void save(Customer customer);  
}
```

```
public class CustomerDAOImpl implements CustomerDAO{  
  
    public void save(Customer customer) {  
        System.out.println("CustomerDAOImpl saves customer");  
    }  
}
```

```
public class MockCustomerDAOImpl implements CustomerDAO{  
  
    public void save(Customer customer) {  
        System.out.println("MockCustomerDAOImpl saves customer");  
    }  
}
```

```
public class Customer {  
    private String name;  
  
    ...  
}
```

Customer
name



Service classes

```
public class CustomerService {  
    private CustomerDAO customerDAO;  
  
    public CustomerService(CustomerDAO customerDAO) {  
        this.customerDAO= customerDAO;  
    }  
  
    public void save(Customer customer) {  
        customerDAO.save(customer);  
    }  
}
```

```
public class ProductService {  
    private ProductDAO productDAO;  
  
    public ProductService(ProductDAO productDAO) {  
        this.productDAO= productDAO;  
    }  
  
    public void save(Product product) {  
        productDAO.save(product);  
    }  
}
```

Application

```
public class Application {  
  
    public static void main(String[] args) {  
        Product product = new Product(3324, "DJI Mavic 2 Pro drone");  
        Customer customer = new Customer("Frank Brown");  
  
        DaoFactory mainfactory = new DaoFactory();  
        MyFactory factory = mainfactory.getFactoryInstance();  
  
        ProductDAO productDao = factory.getProductDAO();  
        CustomerDAO customerDao = factory.getCustomerDAO();  
  
        ProductService productService = new ProductService(productDao);  
        productService.save(product);  
        CustomerService customerService = new CustomerService(customerDao);  
        customerService.save(customer);  
    }  
}
```

Main point

- In the factory pattern, the logic of object creation is encapsulated in the factory.
- Whatever we put our attention on will grow stronger in our life.

Lesson 9 Builder pattern

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Builder

- Builds a complex object using a step by step approach



Immutable class

- Once created, an immutable object can never be changed

```
public class Money {  
    private BigDecimal value;
```

No setter methods

```
    public Money(BigDecimal value) {  
        this.value = value;  
    }
```

Mutation leads to the creation of new instances

```
    public Money add(Money money){  
        return new Money(value.add(money.getValue()));  
    }
```

```
    public Money subtract(Money money){  
        return new Money(value.subtract(money.getValue()));  
    }
```

```
    public BigDecimal getValue() {  
        return value;  
    }
```

Why immutable classes?

- Reasons to make a class immutable:
 - Less prone to errors
 - Easier to share
 - Thread safe
- Immutable classes in Java
 - `java.lang.String`
 - `java.io.File`
 - `java.util.Locale`
 - Almost all classes in `java.time`

Constructor with many parameters

```
public class Customer {  
    private String firstName;  
    private String lastname;  
    private String phone;  
    private String email;  
    private int age;  
    private int numberOfChildren;  
    private int shoesize;  
    private boolean isMarried;  
    private double yearlyIncome;  
    private double yearlyAmountSpendOnShoes;
```

```
public Customer(String firstName, String lastname, String phone, String email, int age, int  
    numberOfChildren, int shoesize, boolean isMarried, double yearlyIncome, double  
    yearlyAmountSpendOnShoes) {  
    this.firstName = firstName;  
    this.lastname = lastname;  
    this.phone = phone;  
    this.email = email;  
    this.age = age;  
    this.numberOfChildren = numberOfChildren;  
    this.shoesize = shoesize;  
    this.isMarried = isMarried;  
    this.yearlyIncome = yearlyIncome;  
    this.yearlyAmountSpendOnShoes = yearlyAmountSpendOnShoes;  
}
```

Constructor is not expressive

```
Customer customer = new Customer("Mary", "Jones", "0623416754",  
    "mjones@gmail.com", 34, 3, 8, true, 50000.0, 2000.0);
```

What do these parameters mean?

Easy to make mistakes

If you have optional parameters, you need many constructors

Class can be immutable

Using setters

```
public class ApplicationUsingSetters {  
    public static void main(String[] args) {  
        Customer customer = new Customer();  
        customer.setFirstName("Mary");  
        customer.setLastname("Jones");  
        customer.setPhone("0623416754");  
        customer.setEmail("mjones@gmail.com");  
        customer.setAge(34);  
        customer.setNumberOfChildren(3);  
        customer.setShoesize(8);  
        customer.setMarried(true);  
        customer.setYearlyIncome(50000.0);  
        customer.setYearlyAmountSpendOnShoes(2000.0);  
        System.out.println(customer);  
    }  
}
```

Clear what the parameters mean

Class is not immutable

What if we want

- Expressive code
- Immutable class
- Solution: Builder

Builder example

```
public class Customer {  
    private String firstName;  
    private String lastname;  
    private String phone;  
    private String email;  
    private int age;  
    private int numberOfChildren;  
    private int shoesize;  
    private boolean isMarried;  
    private double yearlyIncome;  
    private double yearlyAmountSpendOnShoes;  
  
    public static class Builder {  
  
        private String firstName="";  
        private String lastname="";  
        private String phone="";  
        private String email="";  
        private int age = 0;  
        private int numberOfChildren = 0;  
        private int shoesize = 0;  
        private boolean isMarried = false;  
        private double yearlyIncome = 0.0;  
        private double yearlyAmountSpendOnShoes = 0.0;  
  
        public Builder withFirstName(String firstName) {  
            this.firstName = firstName;  
            return this;  
        }  
    }  
}
```

Builder inner class

'Setter' method on the builder

Return 'this' for method chaining

Builder example

```
public Builder withLastname(String lastname) {  
    this.lastname = lastname;  
    return this;  
}  
public Builder withPhone(String phone) {  
    this.phone = phone;  
    return this;  
}  
public Builder withEmail(String email) {  
    this.email = email;  
    return this;  
}  
public Builder withAge(int age) {  
    this.age = age;  
    return this;  
}  
public Builder withNumberOfChildren(int numberOfChildren) {  
    this.numberOfChildren = numberOfChildren;  
    return this;  
}  
public Builder withShoesize(int shoesize) {  
    this.shoesize = shoesize;  
    return this;  
}  
public Builder isMarried() {  
    this.isMarried = true;  
    return this;  
}
```

Builder example

```
public Builder isNotMarried() {  
    this.isMarried = false;  
    return this;  
}  
public Builder withYearlyIncome(double yearlyIncome) {  
    this.yearlyIncome = yearlyIncome;  
    return this;  
}  
public Builder withYearlyAmountSpendOnShoes(double yearlyAmountSpendOnShoes) {  
    this.yearlyAmountSpendOnShoes = yearlyAmountSpendOnShoes;  
    return this;  
}  
public Customer build() {  
    return new Customer(this);  
}
```

The build() method does the
actual creation of the object

Builder example

```
private Customer(Builder builder) {  
    this.firstName = builder.firstName;  
    this.lastname = builder.lastname;  
    this.phone = builder.phone;  
    this.email = builder.email;  
    this.age = builder.age;  
    this.numberOfChildren = builder.numberOfChildren;  
    this.shoesize = builder.shoesize;  
    this.isMarried = builder.isMarried;  
    this.yearlyIncome = builder.yearlyIncome;  
    this.yearlyAmountSpendOnShoes = builder.yearlyAmountSpendOnShoes;  
}  
  
@Override  
public String toString() {  
    return "Customer [firstName=" + firstName + ", lastname=" + lastname + ", phone=" + phone + ",  
        email=" + email + ", age=" + age + ", numberOfChildren=" + numberOfChildren + ", shoesize=" +  
        + shoesize + ", isMarried=" + isMarried + ", yearlyIncome=" + yearlyIncome + ",  
        yearlyAmountSpendOnShoes=" + yearlyAmountSpendOnShoes + "]";  
}  
}
```

The constructor has a Builder as argument

The client code

```
public class Application {  
  
    public static void main(String[] args) {  
        Customer customer1 = new Customer.Builder()  
            .withFirstName("Mary")  
            .withLastname("Jones")  
            .withEmail("mjones@gmail.com")  
            .withAge(34)  
            .isMarried()  
            .withNumberOfChildren(3)  
            .withPhone("0623416754")  
            .withShoesize(8)  
            .withYearlyIncome(50000.0)  
            .withYearlyAmountSpendOnShoes(2000.0)  
            .build();  
        System.out.println(customer1);  
  
        Customer customer2 = new Customer.Builder()  
            .withFirstName("Lucy")  
            .withLastname("Jhonsen")  
            .isNotMarried()  
            .withPhone("0698345234")  
            .build();  
        System.out.println(customer2);  
    }  
}
```

Clear code

Customer is immutable

Builder used in Quartz

```
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = JobDetail("myJob", "group1", HelloJob.class);

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger("myTrigger", "group1", new Date(), null,
    SimpleTrigger.REPEAT_INDEFINITELY, 40)

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Quartz 1.0

```
SchedulerFactory schedFact = new StdSchedulerFactory();
Scheduler sched = schedFact.getScheduler();
sched.start();
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1")
    .build();

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Quartz 2.0

Main point

- The builder pattern is a great help if you want to create objects with many different parameters.
- All the intelligence of Nature is available at the level of the Unified Field

Lesson 9 Decorator pattern



L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Decorator pattern

- Allows to dynamically add new behavior to an existing object.

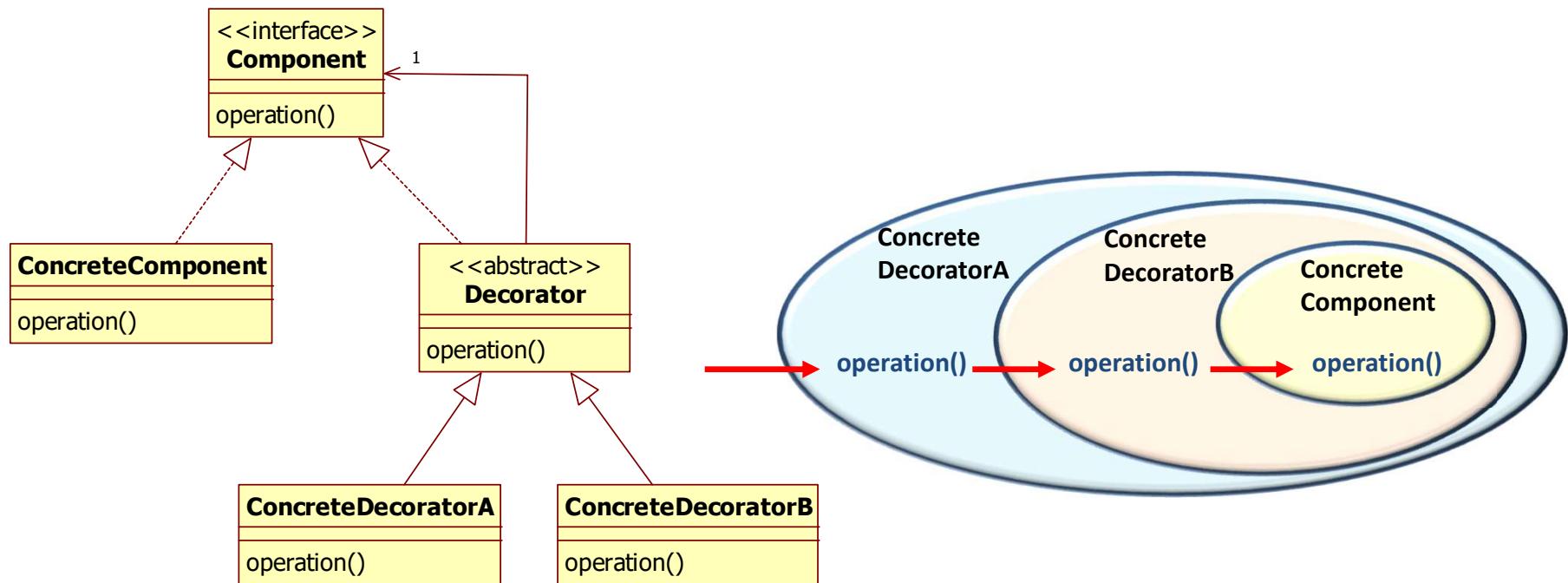
Plain pizza crust



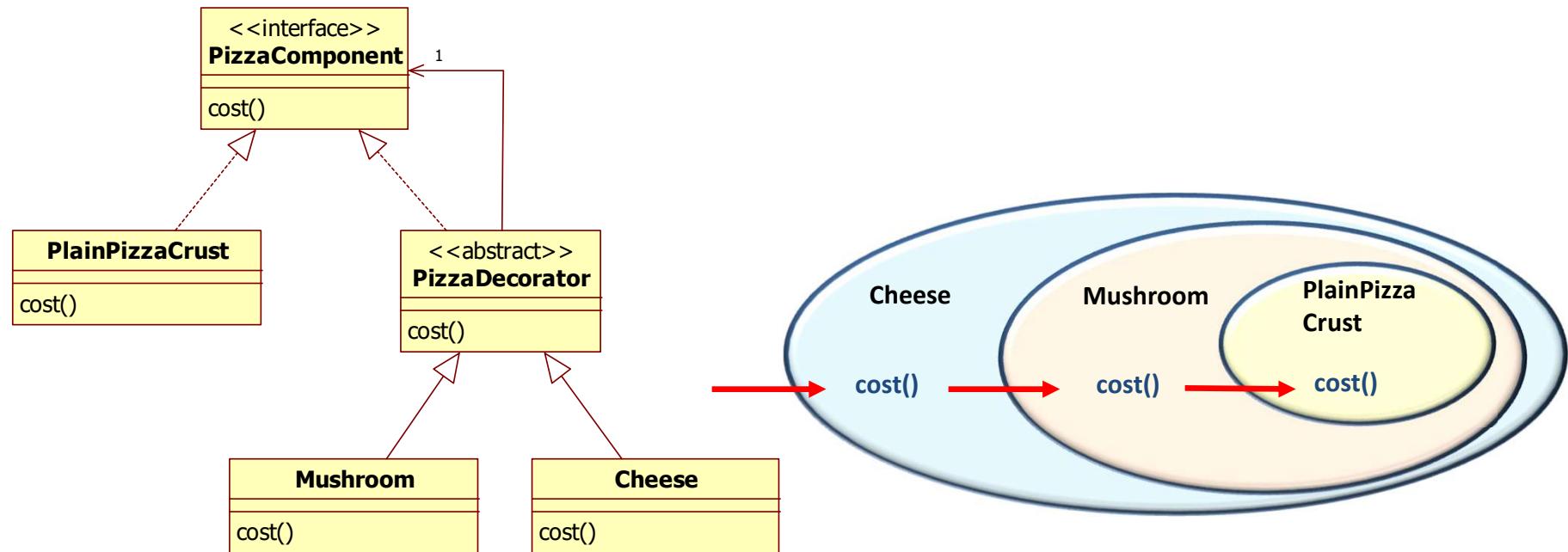
Pizza toppings



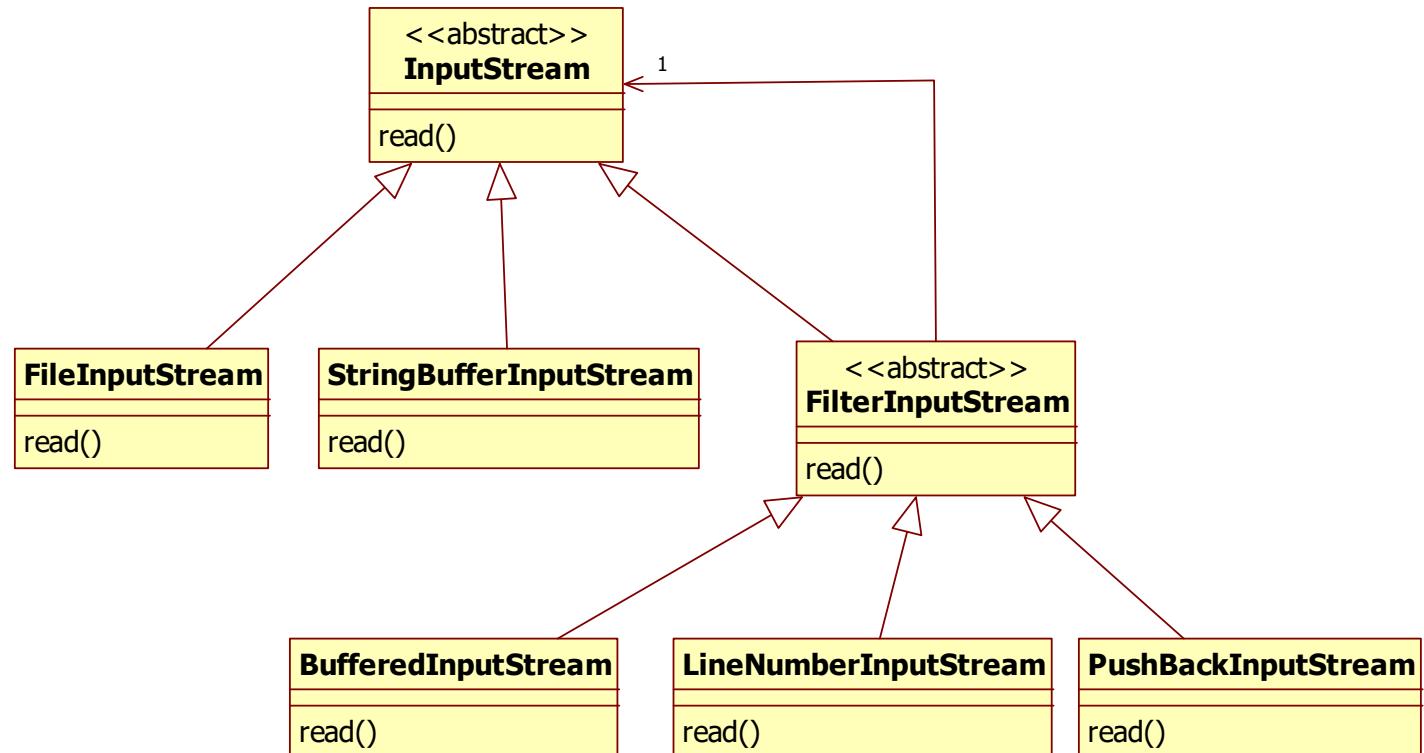
Decorator pattern



Decorating a pizza



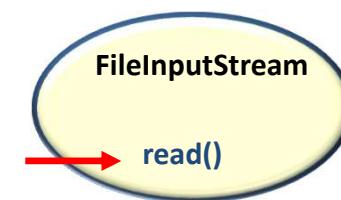
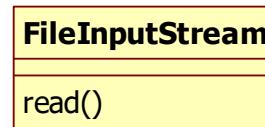
Java.io



FileInputStream

```
public class Application {  
  
    public static void main(String[] args) {  
        int c;  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            InputStream inputStream = new FileInputStream(rootPath + "/input.txt");  
  
            while ((c = inputStream.read()) >= 0) {  
                System.out.print((char) c);  
            }  
  
            inputStream.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

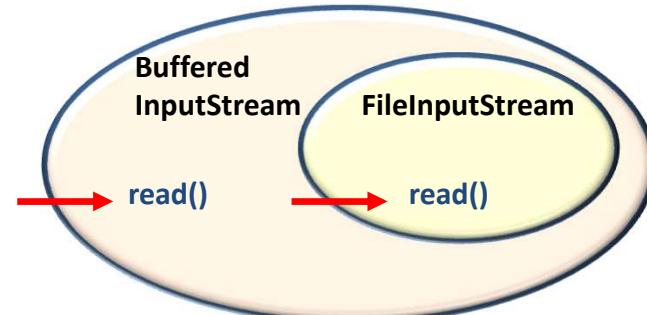
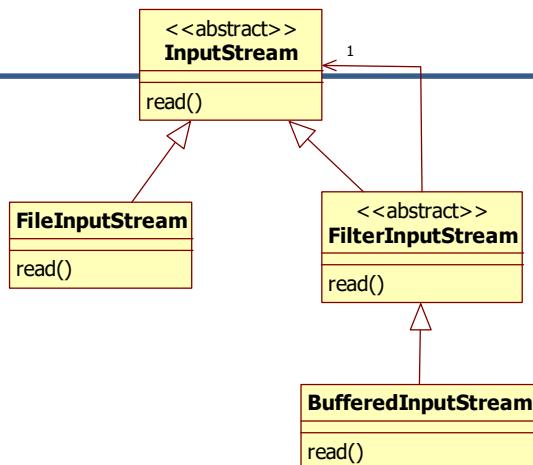
Reads a byte of data



BufferedInputStream

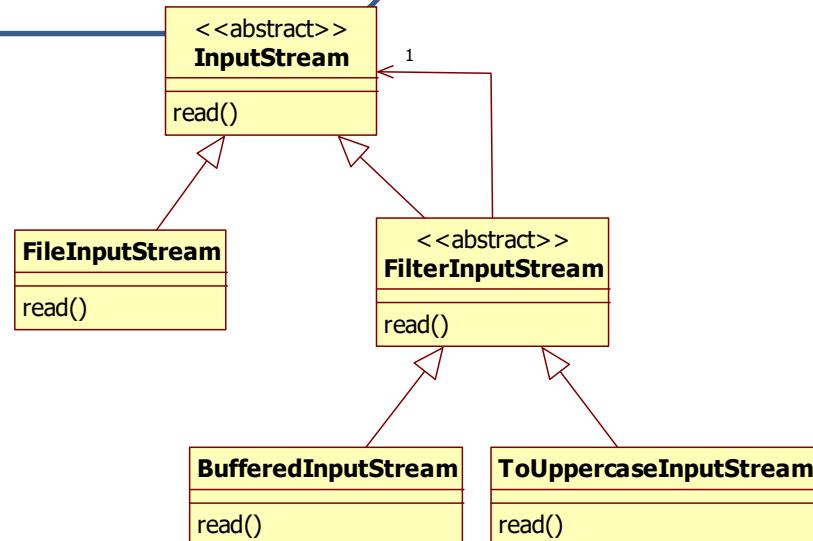
```
public class Application {  
  
    public static void main(String[] args) {  
        int c;  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            InputStream inputStream =  
                new BufferedInputStream(new FileInputStream(rootPath + "/input.txt"));  
  
            while ((c = inputStream.read()) >= 0) {  
                System.out.print((char) c);  
            }  
  
            inputStream.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Reads 8 kilobytes of data and buffers them



Write your own decorator

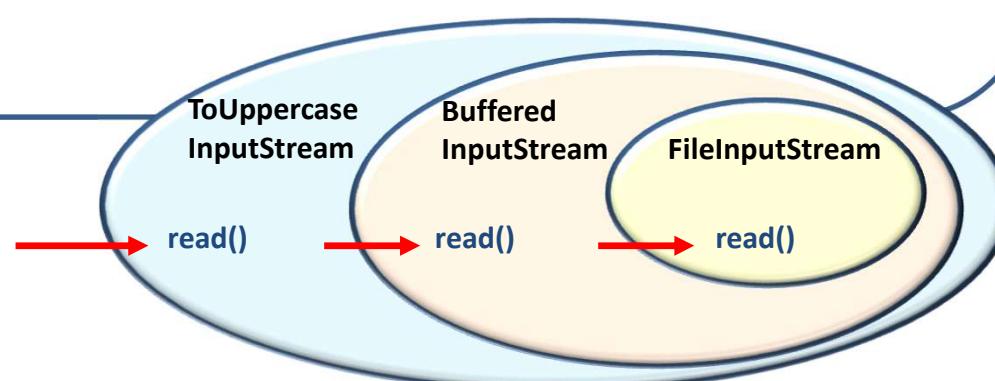
```
public class ToUppercaseInputStream extends FilterInputStream {  
  
    protected ToUppercaseInputStream(InputStream in) {  
        super(in);  
    }  
  
    @Override  
    public int read() throws IOException {  
        int c = super.read();  
        if (c != -1)  
            c = Character.toUpperCase((char)c);  
        return c;  
    }  
}
```



ToUppercaseInputStream

```
public class Application {  
  
    public static void main(String[] args) {  
        int c;  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            InputStream inputStream =  
                new ToUppercaseInputStream(new BufferedInputStream(  
                    new FileInputStream(rootPath + "/input.txt")));  
            while ((c = inputStream.read()) >= 0) {  
                System.out.print((char) c);  
            }  
  
            inputStream.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Add decorators to the
FileInputStream



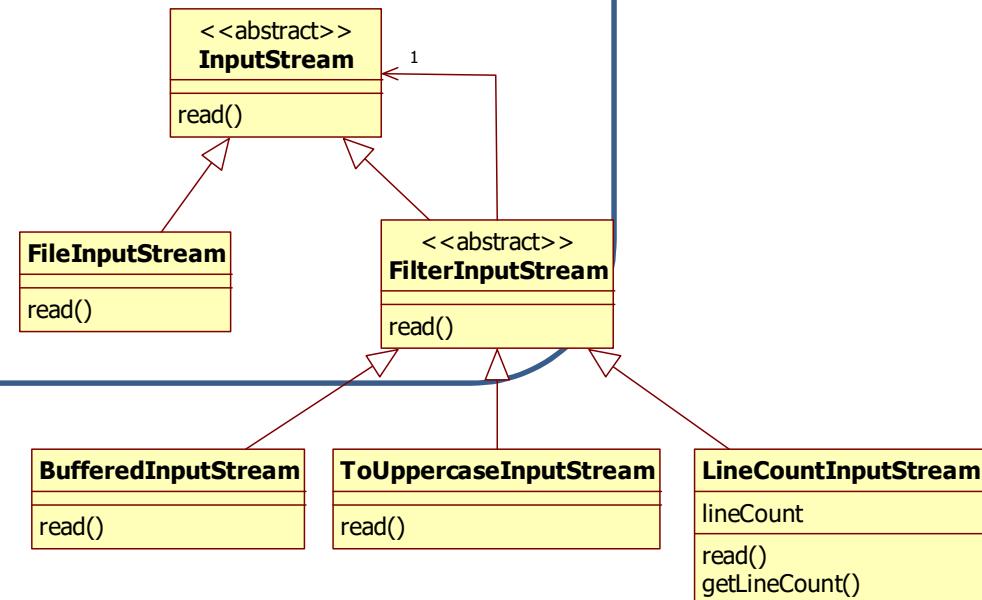
Write your own decorator

```
public class LineCountInputStream extends FilterInputStream {
    int lineCount = 0;

    protected LineCountInputStream(InputStream in) {
        super(in);
    }

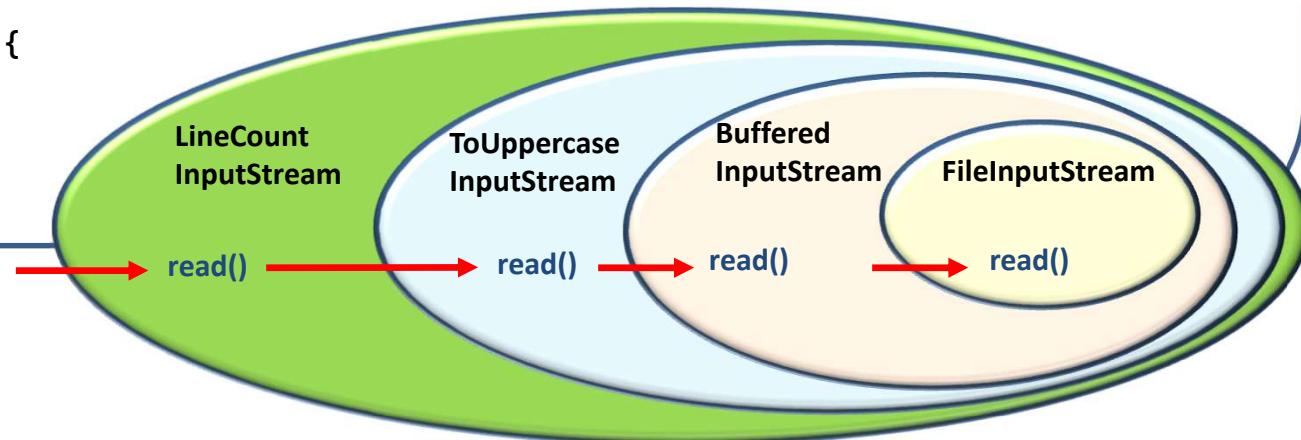
    @Override
    public int read() throws IOException {
        int c = super.read();
        if (c != -1 && c==10 ) //carriage return = 10
            lineCount++;
        return c;
    }

    public int getLineCount() {
        return lineCount;
    }
}
```

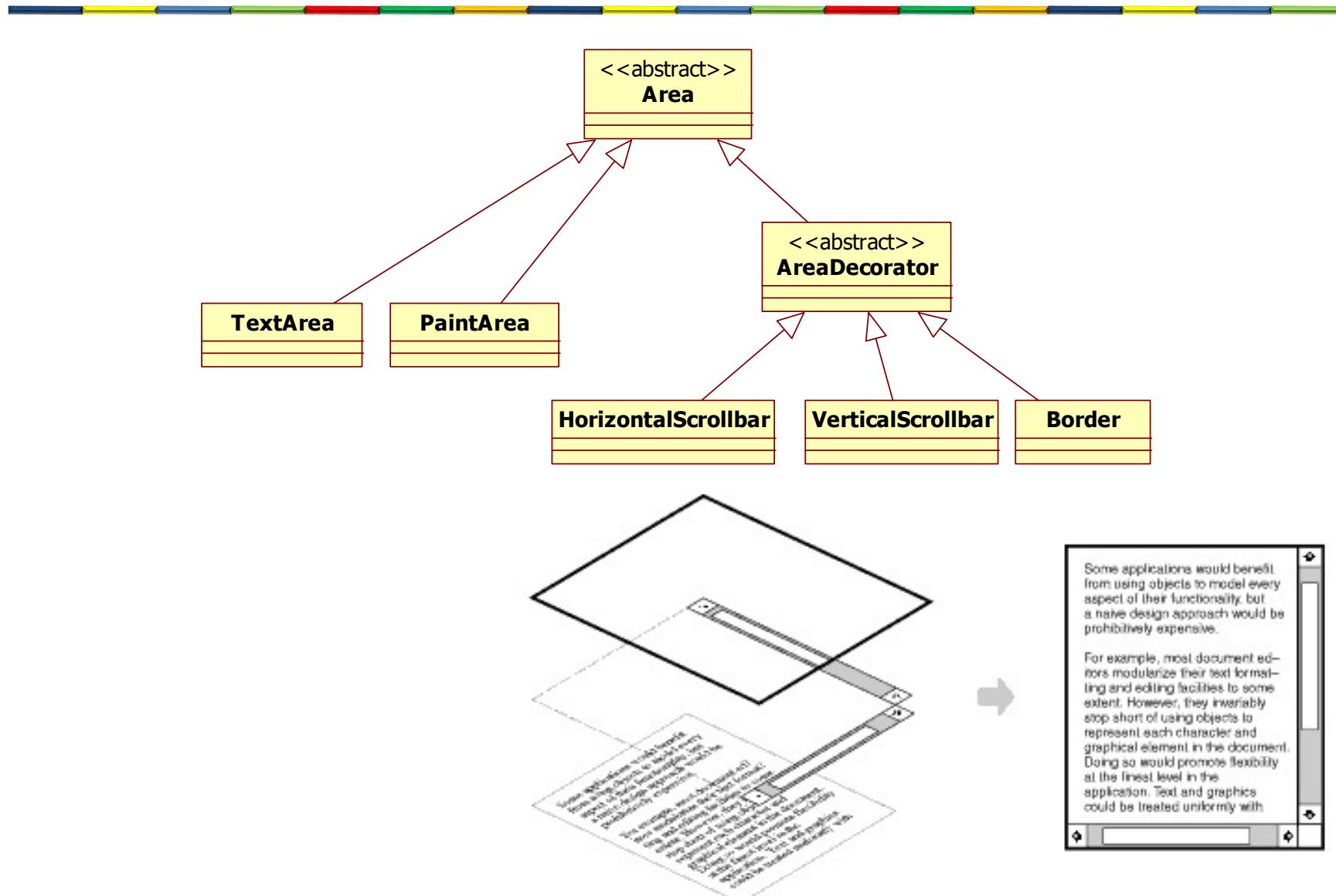


LineCountInputStream

```
public class Application {  
  
    public static void main(String[] args) {  
        int c;  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            LineCountInputStream inputStream =  
                new LineCountInputStream(new ToUppercaseInputStream(new BufferedInputStream(  
                    new FileInputStream(rootPath + "/input.txt"))));  
  
            while ((c = inputStream.read()) >= 0) {  
                System.out.print((char) c);  
            }  
            System.out.println("");  
            System.out.println("This file contains "+inputStream.getLineCount()+" Lines");  
            inputStream.close();  
        } catch (IOException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



Decorator example



Decorator in Java collections

```
public static <T> Collection<T> synchronizedCollection(Collection<T> c);
public static <T> Set<T> synchronizedSet(Set<T> s);
public static <T> List<T> synchronizedList(List<T> list);
```

Factory methods that return a decorated collection

Synchronization
decorator

List

```
public static <T> Collection<T> unmodifiableCollection(Collection<? extends T> c);
public static <T> Set<T> unmodifiableSet(Set<? extends T> s);
public static <T> List<T> unmodifiableList(List<? extends T> list);
```

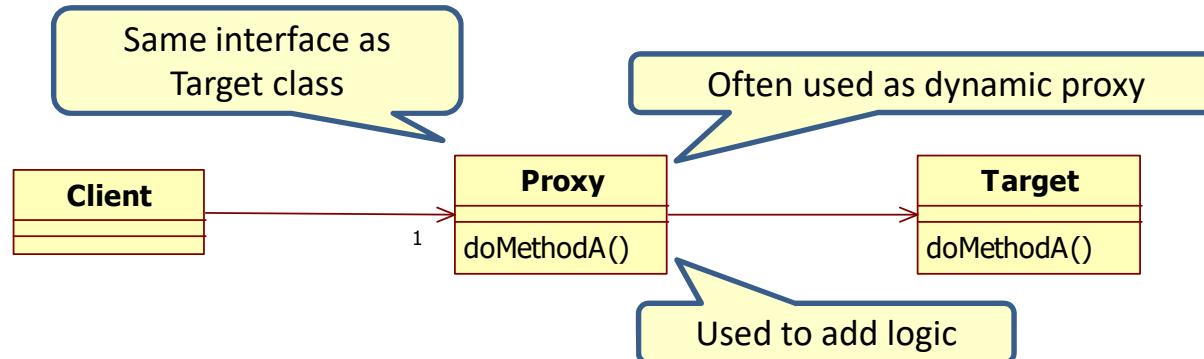
Factory methods that return an unmodifiable (immutable) collection

Unmodifiable
decorator

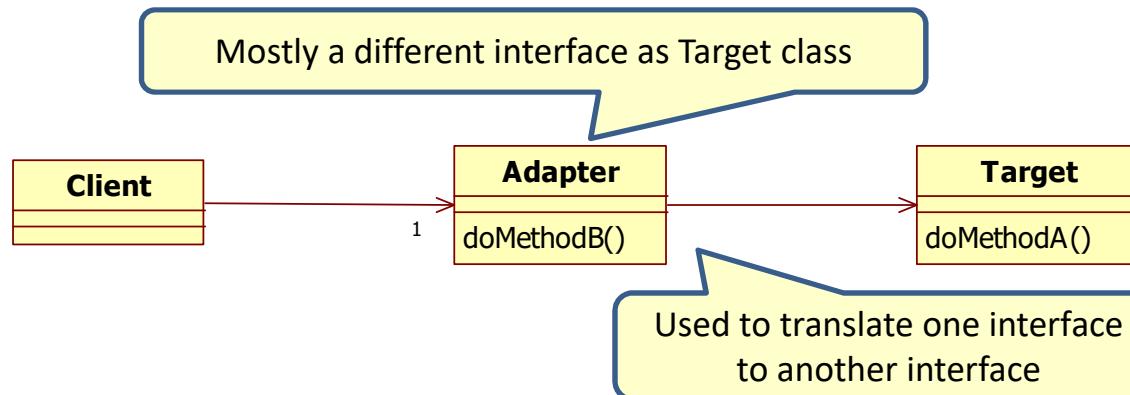
List

Wrappers

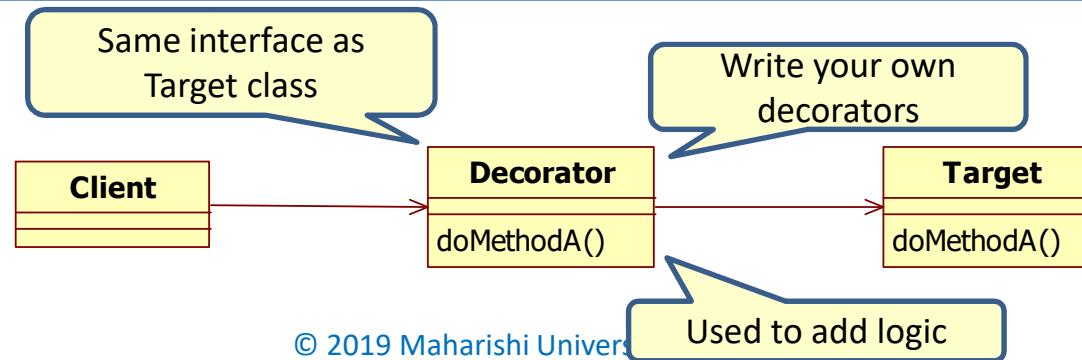
■ Proxy



■ Adapter



■ Decorator



Lesson 9 Singleton pattern

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Singleton

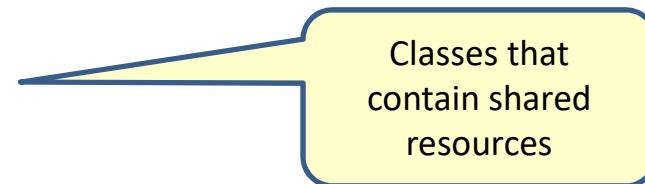
- A singleton class can have only one instance.



- The office of the President of the United States is a *Singleton*. The United States Constitution specifies that there can be at most one active president at any given time.

Examples of singleton classes

- ConnectionPool
- PrinterBuffer
- Cache
- Configuration from configuration file



Classes that contain shared resources

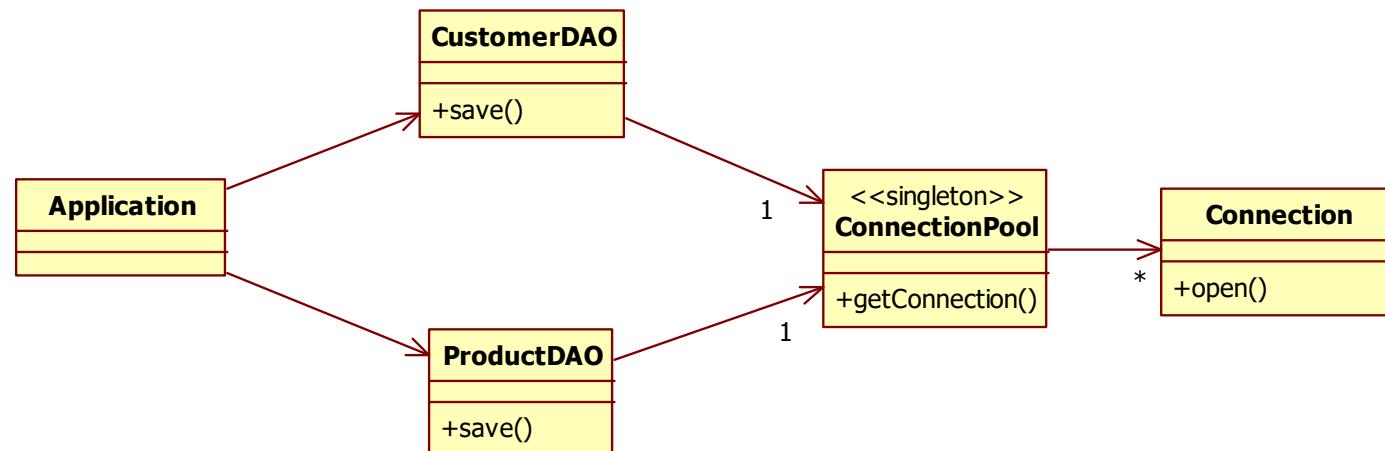
Singleton example

```
public class ConnectionPool {  
    private static ConnectionPool pool = new ConnectionPool();  
    //this is a pool with only 1 connection  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {}  
  
    public static ConnectionPool getPool() {  
        return pool;  
    }  
    public Connection getConnection(){  
        return connection;  
    }  
}
```

Declare a private static instance of the class

Make the constructor private

Add a static method to get an instance of the singleton class



The application

```
public class CustomerDAO {  
    Connection conn;  
  
    public CustomerDAO() {  
        conn = ConnectionPool.getPool().getConnection();  
    }  
    public void save(){  
        conn.open();  
    }  
}
```

```
public class Connection {  
    public void open(){  
        System.out.println("open connection to DB");  
    }  
}
```

```
public class ProductDAO {  
    Connection conn;  
  
    public ProductDAO() {  
        conn = ConnectionPool.getPool().getConnection();  
    }  
    public void save(){  
        conn.open();  
    }  
}
```

```
public class Application {  
  
    public static void main(String[] args) {  
        CustomerDAO cdao = new CustomerDAO();  
        cdao.save();  
        ProductDAO pdao = new ProductDAO();  
        pdao.save();  
    }  
}
```

Eager and lazy instantiation

```
public class ConnectionPool {  
    private static ConnectionPool pool = new ConnectionPool();  
    //this is a pool with only 1 connection  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {}  
  
    public static ConnectionPool getPool() {  
        return pool;  
    }  
    public Connection getConnection(){  
        return connection;  
    }  
}
```

Eager instantiation

```
public class ConnectionPool {  
    private static ConnectionPool pool;  
    // this is a pool with only 1 connection  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {}  
  
    public static ConnectionPool getPool() {  
        if (pool == null) {  
            pool = new ConnectionPool();  
        }  
        return pool;  
    }  
  
    public Connection getConnection() {  
        return connection;  
    }  
}
```

Lazy instantiation

Issues with singleton

- With reflection you can still create more instances of the singleton
 - Make the singleton reflection safe
- If 2 threads create a singleton at almost the same time, you might end up with 2 instances
 - Make the singleton thread safe
- With serialization and deserialization we might end up with 2 instances
 - Make the singleton serialization safe

Reflection

```
public class ReflectionSingletonTest {  
  
    public static void main(String[] args) {  
        ConnectionPool instanceOne = ConnectionPool.getPool();  
        ConnectionPool instanceTwo = null;  
        try {  
            Constructor[] constructors = ConnectionPool.class.getDeclaredConstructors();  
            for (Constructor constructor : constructors) {  
                //Below code will break the singleton pattern  
                constructor.setAccessible(true);  
                instanceTwo = (ConnectionPool) constructor.newInstance();  
                break;  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        System.out.println(instanceOne.getClass().getName() + " with hashCode: " + instanceOne.hashCode());  
        System.out.println(instanceTwo.getClass().getName() + " with hashCode: " + instanceTwo.hashCode());  
    }  
}
```

Two instances of
the ConnectionPool

reflection.ConnectionPool with hashCode: 366712642
reflection.ConnectionPool with hashCode: 1829164700

Make the singleton reflection safe

```
public class ConnectionPool {  
    private static ConnectionPool pool;  
    // this is a pool with only 1 connection  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {  
        // Prevent form the reflection api.  
        if (pool != null) {  
            throw new RuntimeException("Use getInstance() method to get the single instance of this  
                class.");  
        }  
    }  
  
    public static synchronized ConnectionPool getPool() {  
        if (pool == null) {  
            pool = new ConnectionPool();  
        }  
        return pool;  
    }  
  
    public Connection getConnection() {  
        return connection;  
    }  
}
```

Throw exception if
constructor is
called twice

```
java.lang.reflect.InvocationTargetException  
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)  
... 5 more  
Exception in thread "main" java.lang.NullPointerException  
reflection.safe.ConnectionPool with hascode: 2018699554 at  
reflection.safe.ReflectionSingletonTest.main(ReflectionSingletonTest.java:22)
```

Thread safety

```
public class SingletonTest {  
    public static void main(String[] args) {  
        //Thread 1  
        Thread t1 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                ConnectionPool instance1 = ConnectionPool.getPool();  
                System.out.println("Instance 1 hash:" + instance1.hashCode());  
            }  
        });  
  
        //Thread 2  
        Thread t2 = new Thread(new Runnable() {  
            @Override  
            public void run() {  
                ConnectionPool instance2 = ConnectionPool.getPool();  
                System.out.println("Instance 2 hash:" + instance2.hashCode());  
            }  
        });  
  
        //start both the threads  
        t1.start();  
        t2.start();  
    }  
}  
  
Instance 1 hash:1870487130  
Instance 2 hash:354710606
```

Two instances of
the ConnectionPool

Thread safety solution 1

```
public class ConnectionPool {  
    private static ConnectionPool pool;  
    // this is a pool with only 1 connection  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {  
        // Prevent form the reflection api.  
        if (pool != null) {  
            throw new RuntimeException("Use getInstance() method to get the single instance  
                of this class.");  
        }  
    }  
  
    public static synchronized ConnectionPool getPool() {  
        if (pool == null) {  
            pool = new ConnectionPool();  
        }  
        return pool;  
    }  
  
    public Connection getConnection() {  
        return connection;  
    }  
}
```

synchronized

Performance
problem

Instance 2 hash:892687863
Instance 1 hash:892687863

Thread safety solution 2

```
public class ConnectionPool {  
    private static ConnectionPool pool;  
    // this is a pool with only 1 connection  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {  
        // Prevent form the reflection api.  
        if (pool != null) {  
            throw new RuntimeException("Use getInstance() method to get the single instance  
                of this class.");  
        }  
    }  
  
    public static ConnectionPool getPool() {  
        // Double check locking pattern  
        if (pool == null) { // Check for the first time  
            synchronized (ConnectionPool.class) { // Check for the second time.  
                if (pool == null) pool = new ConnectionPool();  
            }  
        }  
        return pool;  
    }  
  
    public Connection getConnection() {  
        return connection;  
    }  
}
```

Only use synchronized if
the pool is not created yet

Instance 2 hash:892687863
Instance 1 hash:892687863

Serialization

```
public class SingletonTest {  
    public static void main(String[] args) {  
        try {  
            ConnectionPool instance1 = ConnectionPool.getPool();  
            ObjectOutputStream out = new ObjectOutputStream(new FileOutputStream("filename.ser"));  
            out.writeObject(instance1);  
            out.close();  
  
            // deserialize from file to object  
            ObjectInput in = new ObjectInputStream(new FileInputStream("filename.ser"));  
            ConnectionPool instance2 = (ConnectionPool) in.readObject();  
            in.close();  
  
            System.out.println("instance1 hashCode=" + instance1.hashCode());  
            System.out.println("instance2 hashCode=" + instance2.hashCode());  
  
        } catch (IOException | ClassNotFoundException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

```
public class ConnectionPool implements Serializable{
```

```
public class Connection implements Serializable{
```

instance1 hashCode=865113938
instance2 hashCode=1831932724

Two instances of
the ConnectionPool

Serialization solution

```
public class ConnectionPool {  
    private static ConnectionPool pool;  
    private Connection connection = new Connection();  
  
    private ConnectionPool() {  
        // Prevent form the reflection api.  
        if (pool != null) {  
            throw new RuntimeException("Use getInstance() method to get the single instance  
                of this class.");  
        }  
    }  
  
    public static ConnectionPool getPool() {  
        // Double check locking pattern  
        if (pool == null) { // Check for the first time  
            synchronized (ConnectionPool.class) { // Check for the second time.  
                if (pool == null) pool = new ConnectionPool();  
            }  
        }  
        return pool;  
    }  
  
    public Connection getConnection() {  
        return connection;  
    }  
  
    // This method is called immediately after an object of this class is deserialized.  
    protected Object readResolve() {  
        return getPool();  
    }  
}
```

Implement the
readResolve() method

instance1 hashCode=865113938
instance2 hashCode=865113938

Main point

- A singleton is a class that can have only one instance.
- Pure consciousness is the unified basis of all of creation.

Connecting the parts of knowledge with the wholeness of knowledge

1. The decorator class decorates a target class.
2. The factory pattern, builder pattern and singleton pattern are patterns that help in constructing objects

-
3. **Transcendental consciousness** is the field of all knowledge.
 4. **Wholeness moving within itself:** In unity consciousness, one appreciates the inherent underlying unity that underlies all the diversity of creation.
- 

Lesson 10 Framework design

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

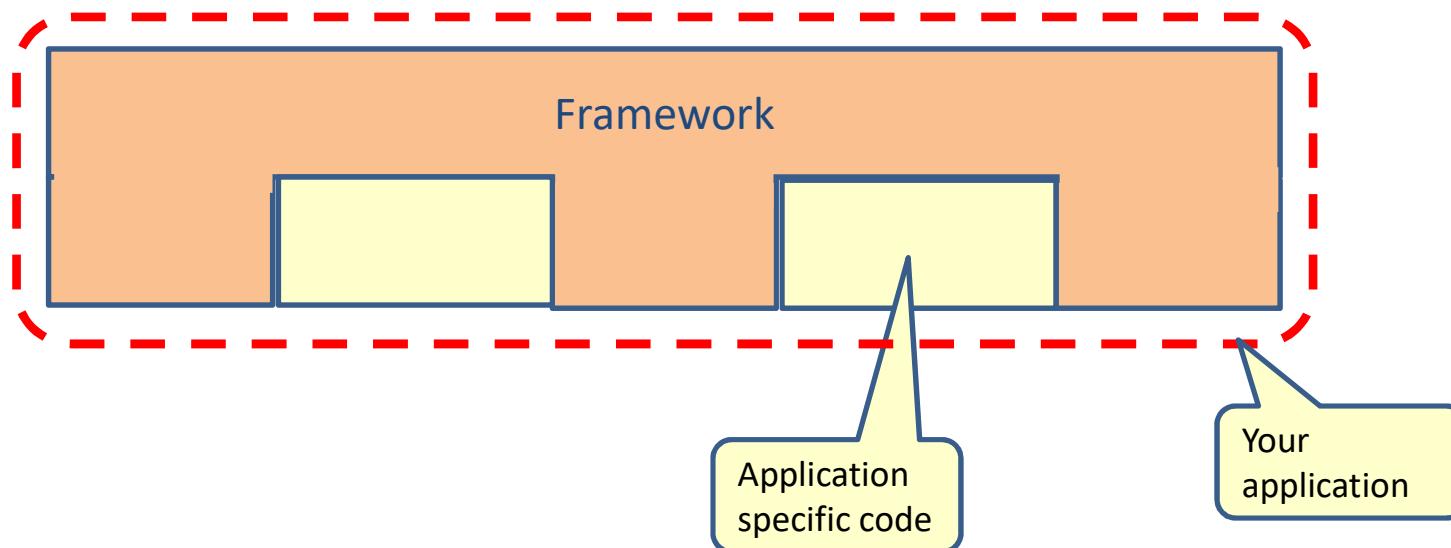
Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Framework

- A framework is a **reusable semi complete** application for a **specific domain**



Framework examples

Web frameworks

- SpringMVC
- Angular
- React
- Vue
- ...

ORM frameworks

- Hibernate
- Open JPA
- EclipseLink
- ...

Testing frameworks

- JUnit
- TestNG
- Mockito
- RestAssured
- Cucumber
- ...

Spring related frameworks

- Spring
- Spring boot
- Spring security
- ...

Logging frameworks

- Log4J 2
- LogBack
- SLF4J
- ...

Game engine/frameworks

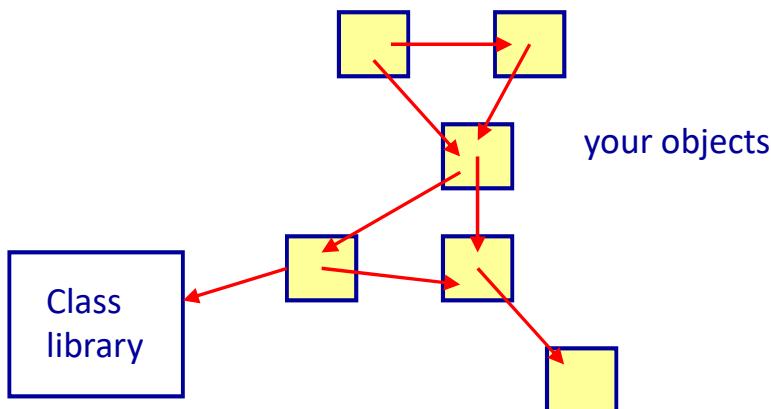
- Unreal Engine
- Unity
- Godot
- ...

Why frameworks

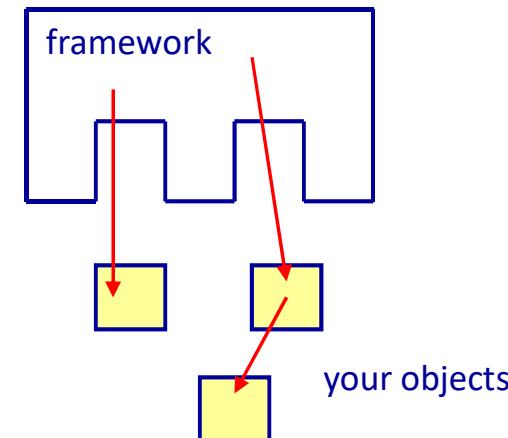
- They embody expertise
 - Developer can focus on the problem domain
- Reuse
- Reliability
 - Reusing a stable framework increases reliability.
- Standardization

Framework vs. Library

- Inversion of Control (IoC)
 - Hollywood principle: Don't call us, we'll call you
 - The framework has control over your code

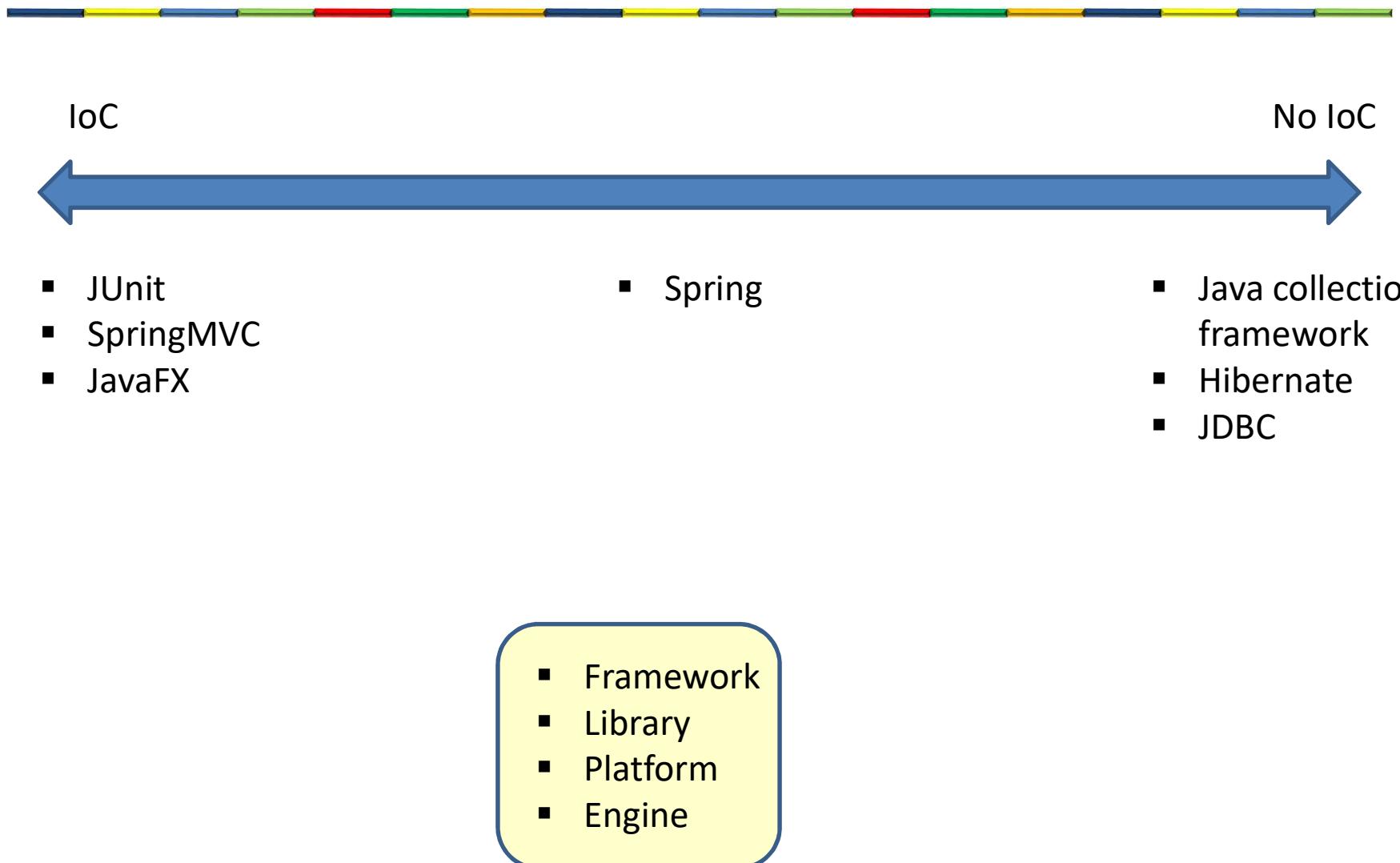


Your code calls the class library



IoC: The framework calls your code

Inversion of Control (IoC)



Example of unit testing

```
import static org.junit.Assert.*;
import org.junit.*

public class CounterTest {
    private Counter counter;

    @Before
    public void setUp() throws Exception {
        counter = new Counter();
    }

    @Test
    public void testIncrement(){
        assertEquals("Counter.increment does not work correctly",1,counter.increment());
        assertEquals("Counter.increment does not work correctly",2,counter.increment());
    }

    @Test
    public void testDecrement(){
        assertEquals("Counter.decrement does not work correctly",-1,counter.decrement());
        assertEquals("Counter.decrement does not work correctly",-2,counter.decrement());
    }
}
```

Initialization

Test method

Test method

```
public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }

    public int decrement(){
        return --counterValue;
    }

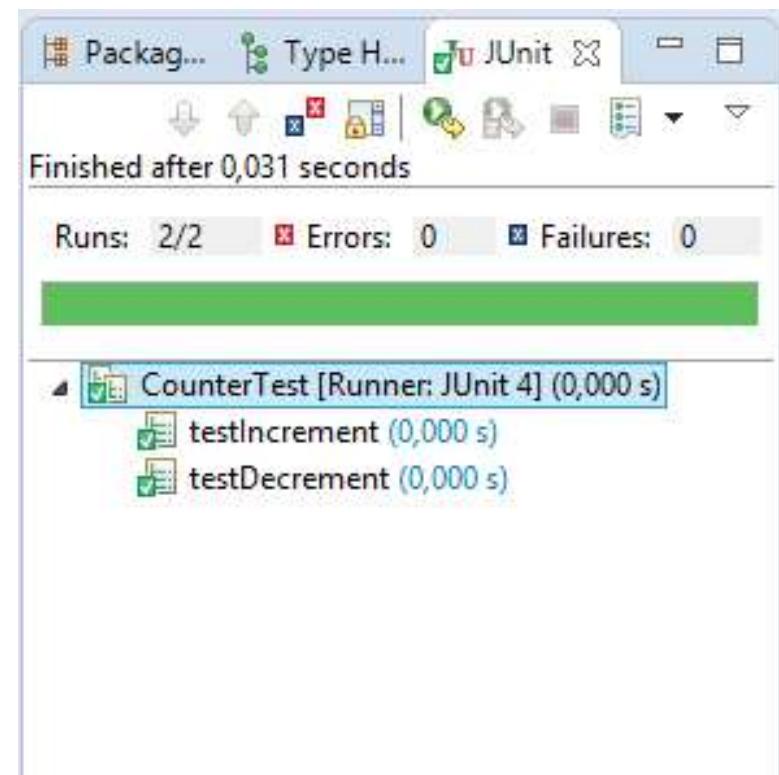
    public int getCounterValue() {
        return counterValue;
    }
}
```

Running the test

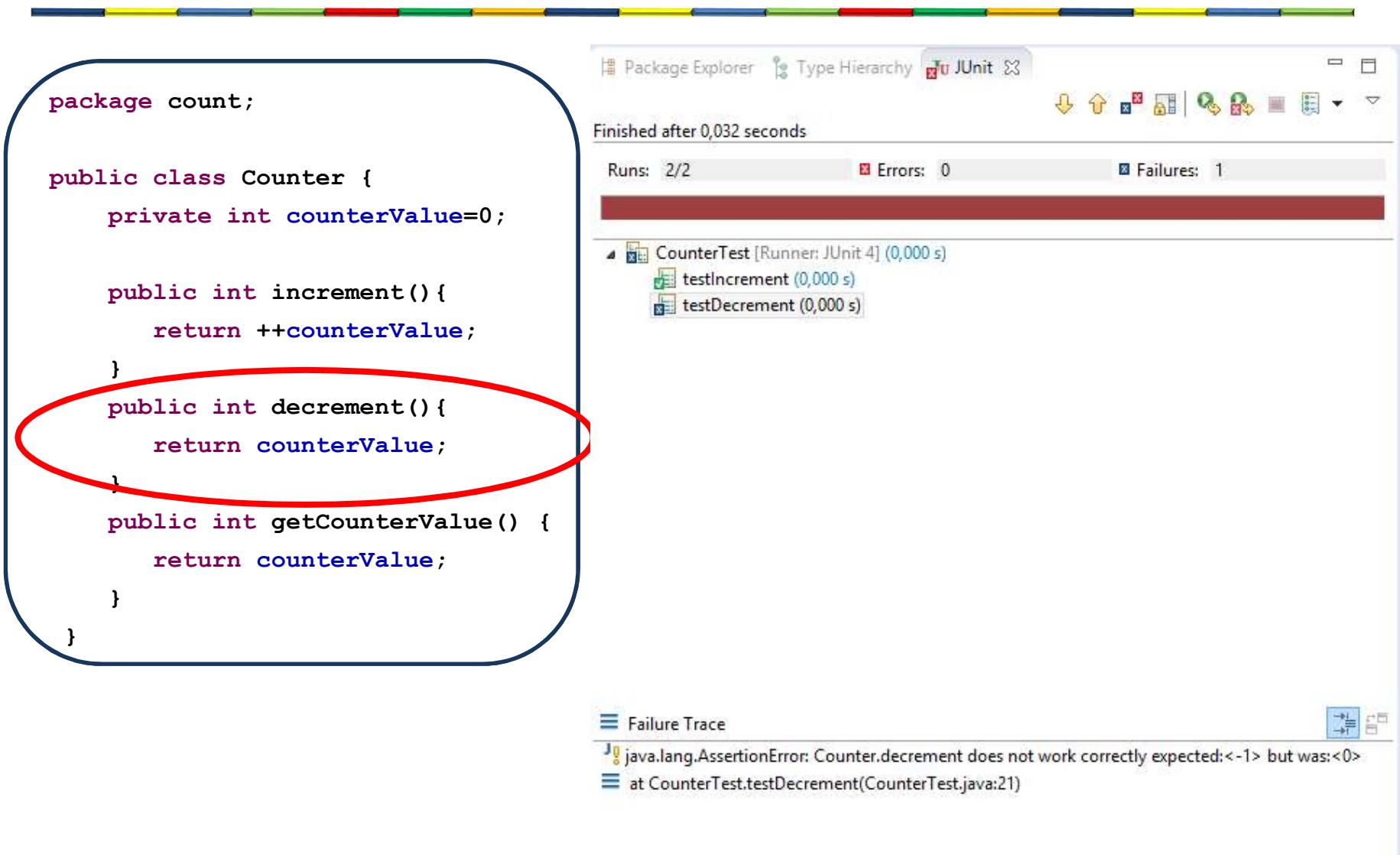
```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }
    public int decrement(){
        return --counterValue;
    }
    public int getCounterValue() {
        return counterValue;
    }
}
```



Running the test



The image shows a screenshot of the Eclipse IDE interface. On the left, a code editor displays the `Counter` class with a red oval highlighting the `decrement` method. The code is as follows:

```
package count;

public class Counter {
    private int counterValue=0;

    public int increment(){
        return ++counterValue;
    }

    public int decrement(){
        return counterValue;
    }

    public int getCounterValue() {
        return counterValue;
    }
}
```

On the right, the JUnit view shows the test results:

- Finished after 0,032 seconds
- Runs: 2/2
- Errors: 0
- Failures: 1

The failure is listed under `CounterTest`:

- `testIncrement (0,000 s)`
- `testDecrement (0,000 s)`

The failure trace is shown at the bottom:

```
java.lang.AssertionError: Counter.decrement does not work correctly expected:<-1> but was:<0>
at CounterTest.testDecrement(CounterTest.java:21)
```

Framework classification

- Technical frameworks (horizontal frameworks)



Testing



GUI



Logging

- Business domain frameworks (vertical frameworks)



Shopping



Finance

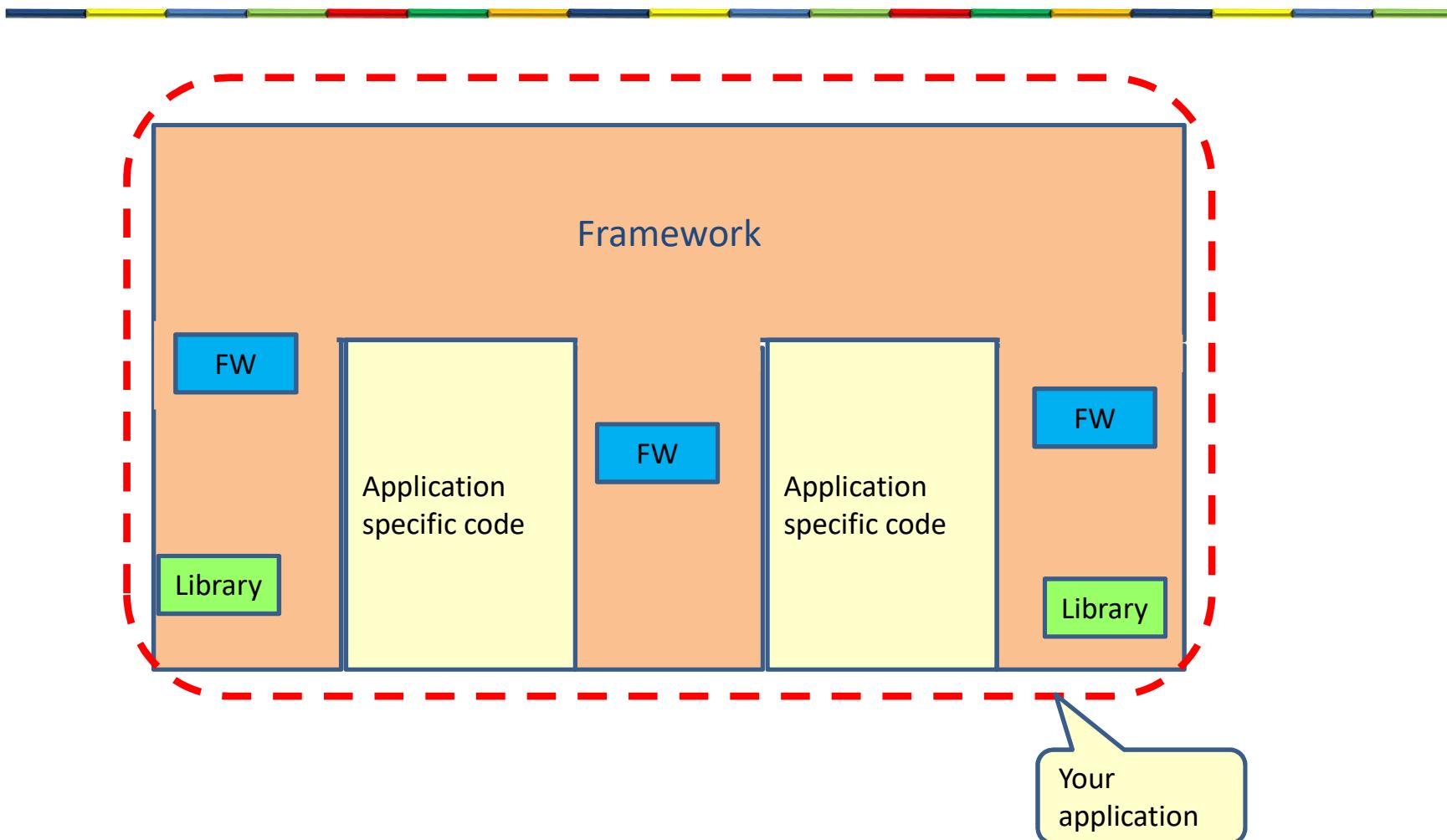


Avionics

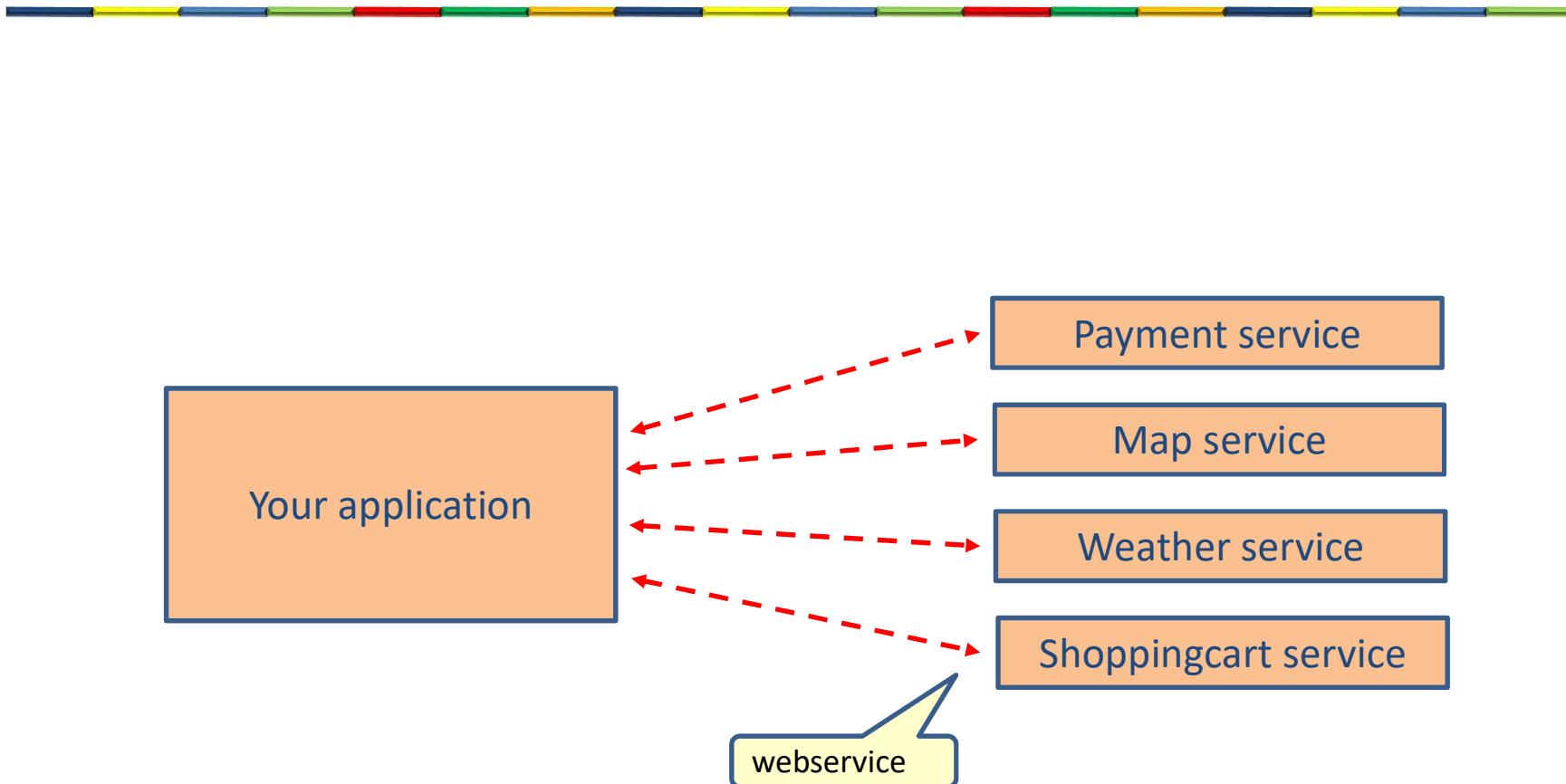
Characteristics of reuse

- To make something generic is 3 to 10 times more expensive than to make something specific
- High risk
- Is everyone aware that this framework exists?
- A framework is a product
 - That need documentation and tests
 - That need maintenance (project, budget)

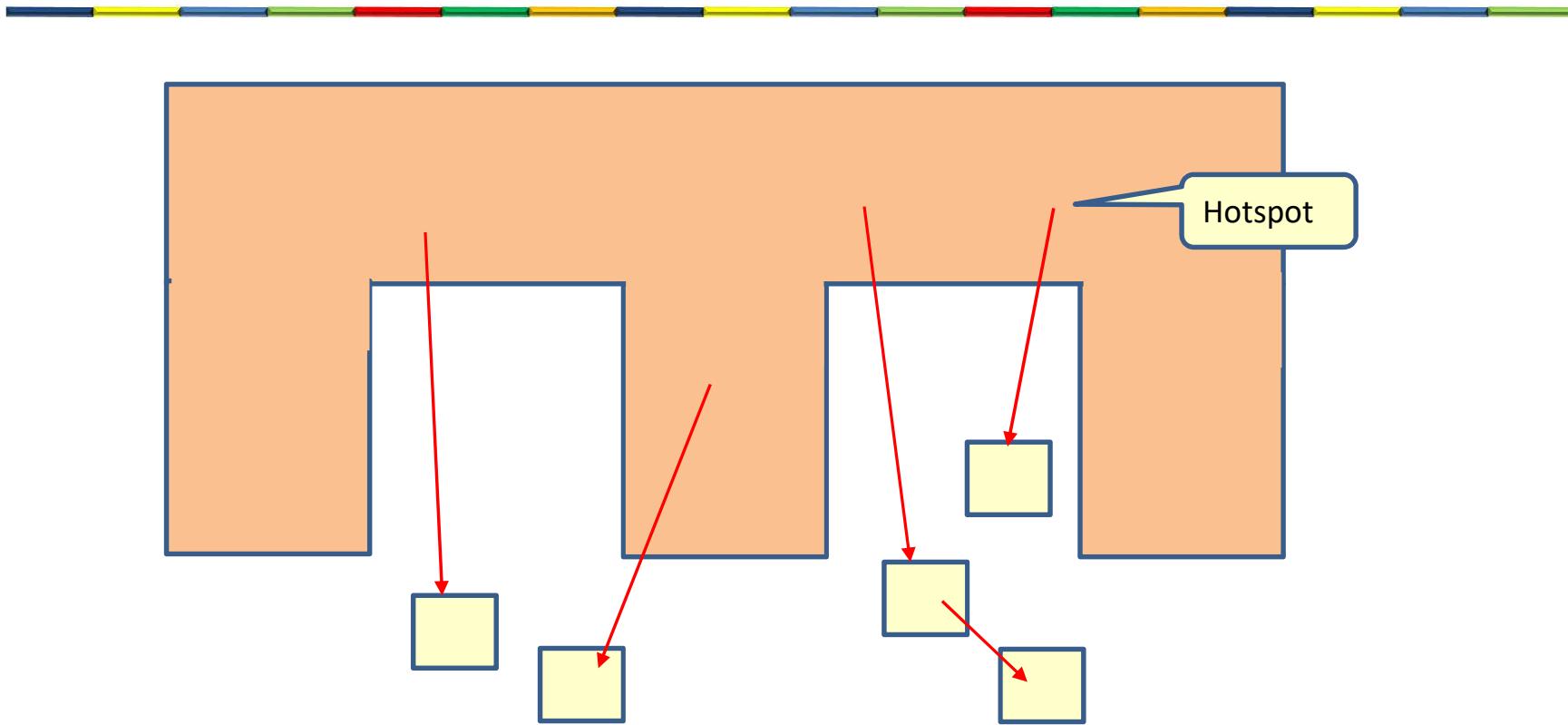
Frameworks + libraries



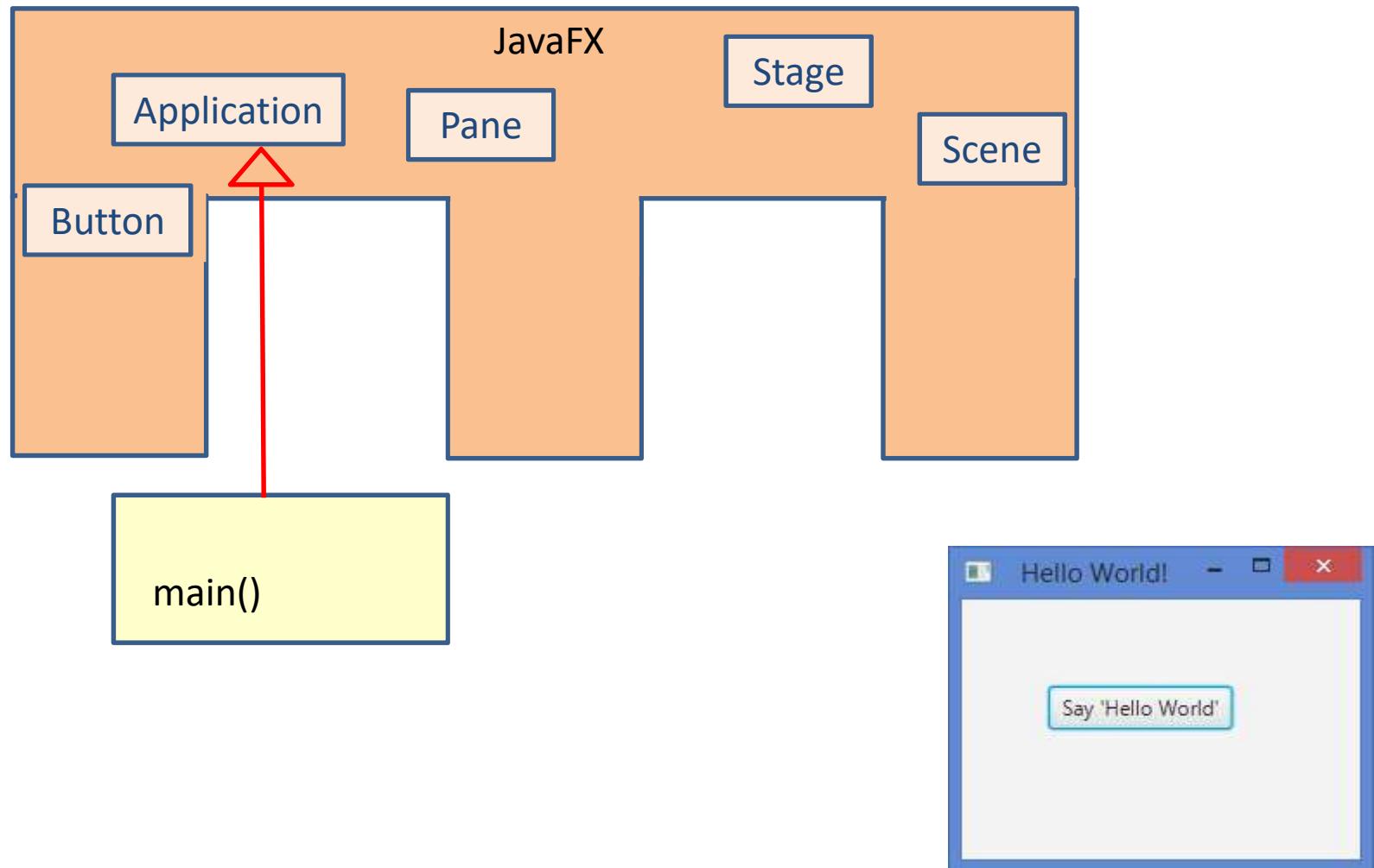
Supporting services



Hotspot (plugin point)



Using JavaFX framework



Using JavaFX framework

```
public class HelloWorld extends Application {  
    @Override  
    public void start(Stage primaryStage) {  
        Button button = new Button();  
        button.setText("Say 'Hello World'");  
        button.relocate(50, 50);  
        button.setOnAction(new EventHandler<ActionEvent>() {  
            @Override  
            public void handle(ActionEvent event) {  
                System.out.println("Hello World!");  
            }  
        });  
        Pane root = new Pane();  
        root.getChildren().add(button);  
    }  
    Scene scene = new Scene(root, 230, 150);  
    primaryStage.setTitle("Hello World!");  
    primaryStage.setScene(scene);  
    primaryStage.show();  
}  
public static void main(String[] args) {  
    Launch(args);  
}
```

Stage

Extend Application

Pane

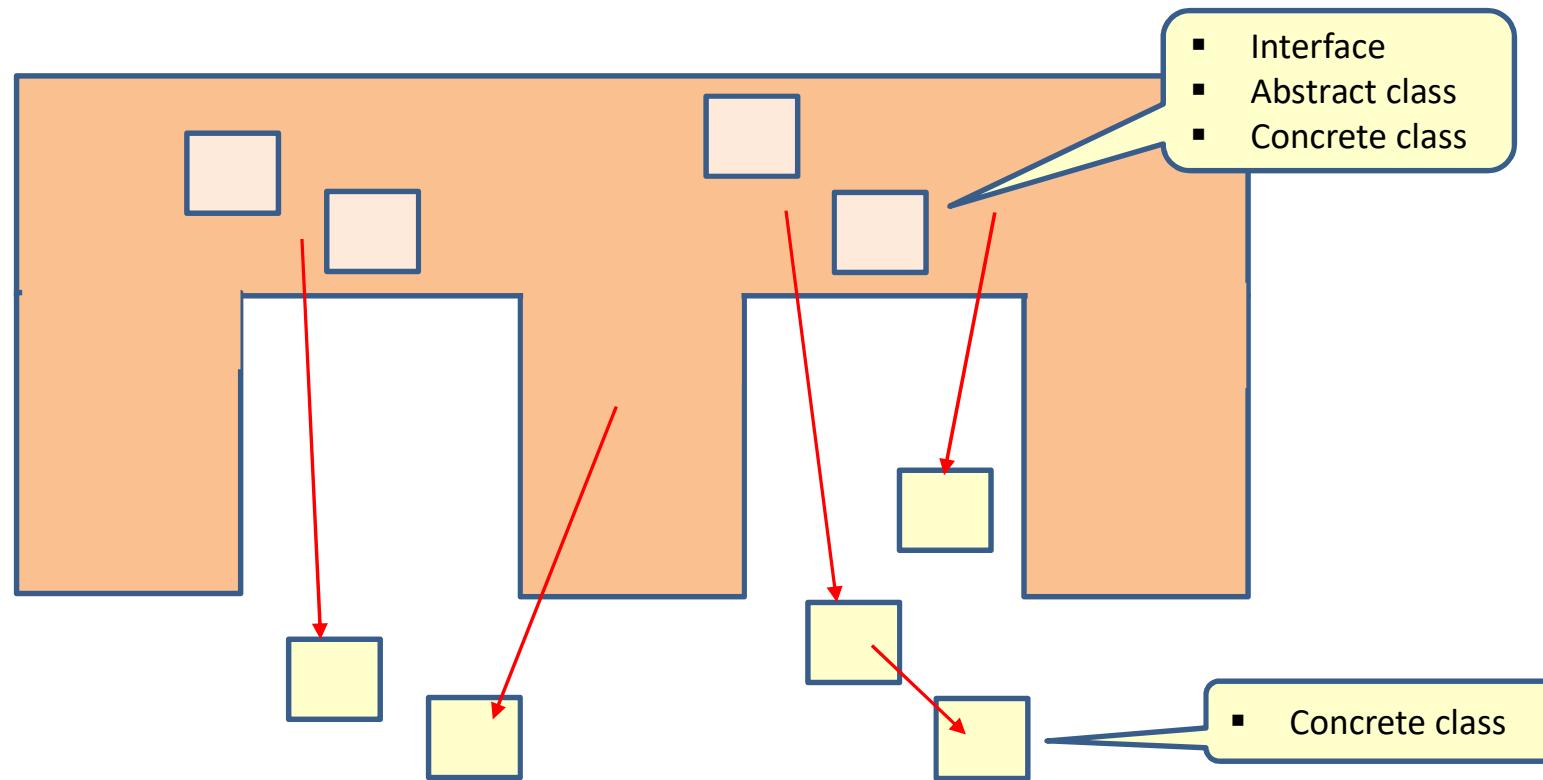
Scene

Launch the application



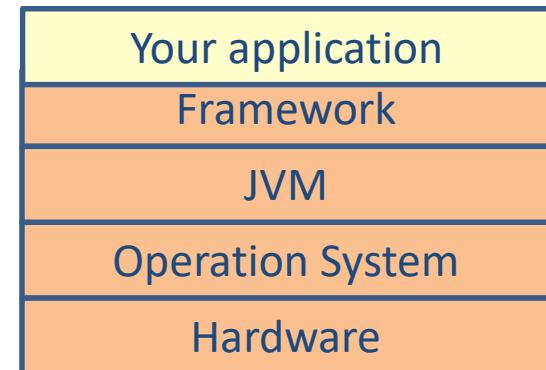
Hello World!

Framework implementation



Disadvantage of frameworks

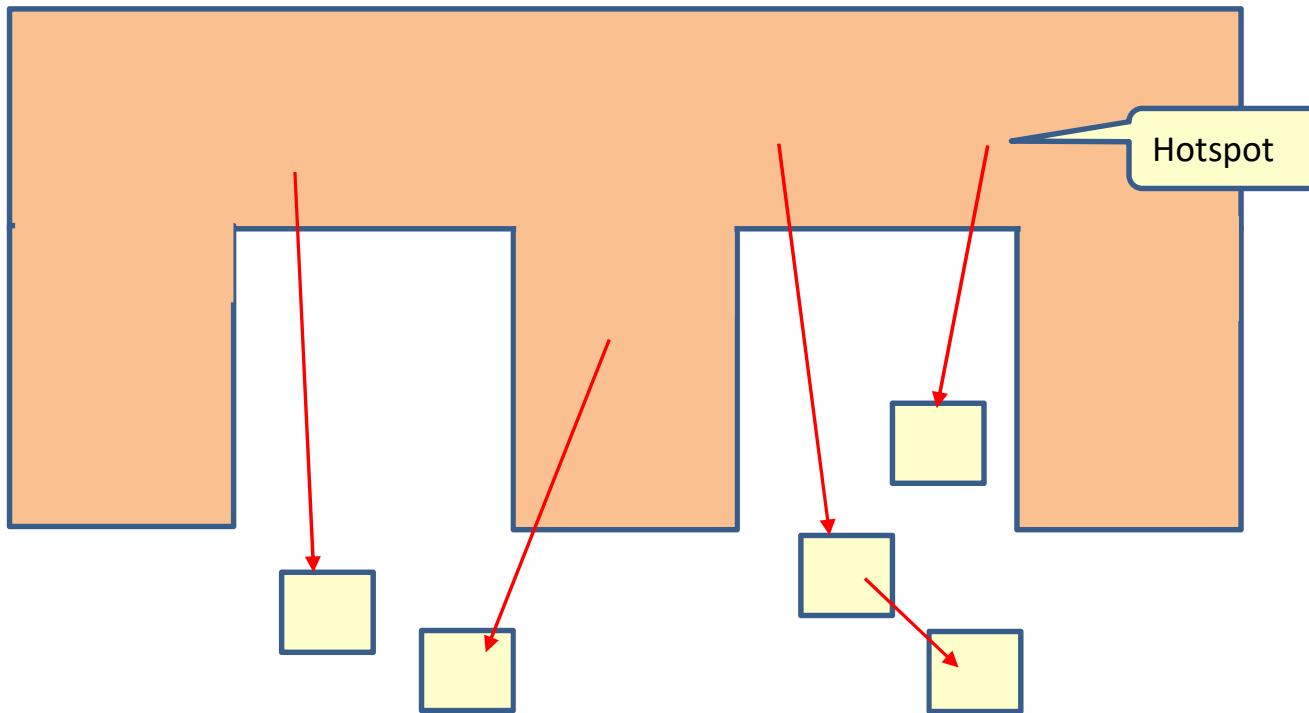
- Another layer of abstraction
 - You don't know the internal details of the FW
 - The framework can contain errors
- Steep learning curve



Main point

- Application development is much easier and faster when you reuse a framework rather than writing the application from scratch.
- Life is much easier, simpler and enjoyable if you make use of the framework of Nature, the Unified Field of all the laws of nature. Established in being (coherence), perform action.

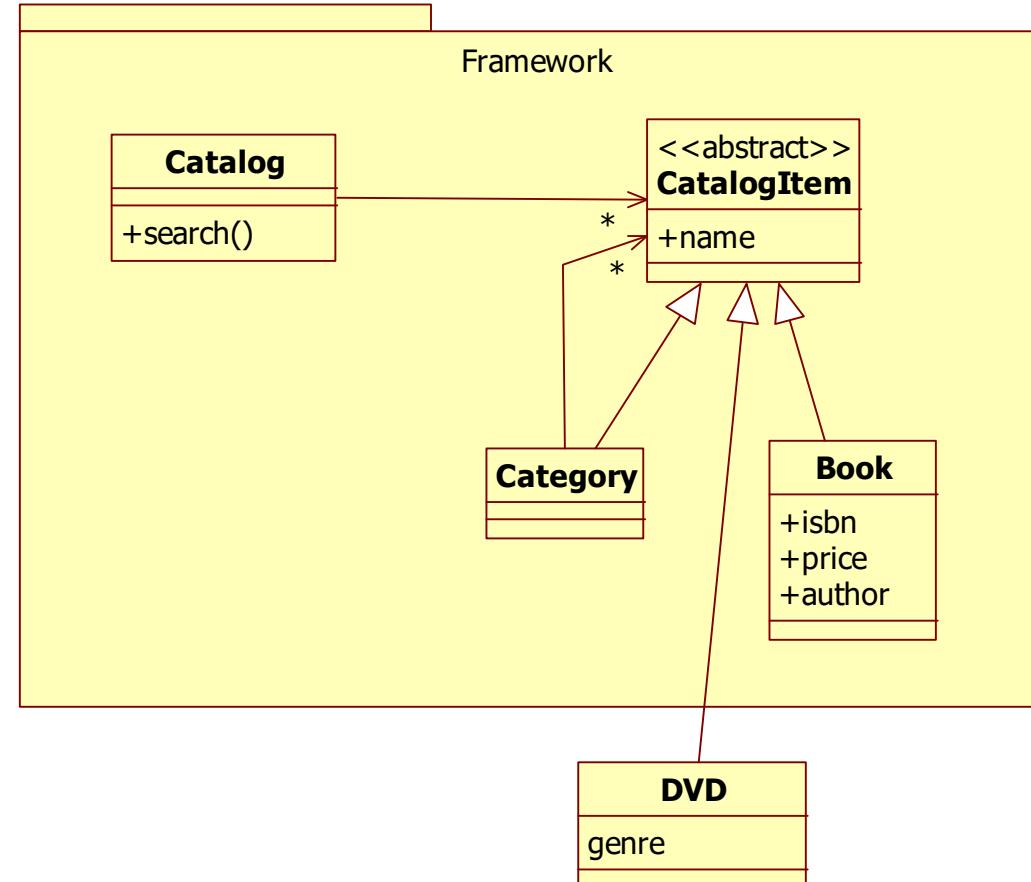
Hotspot (plugin point)



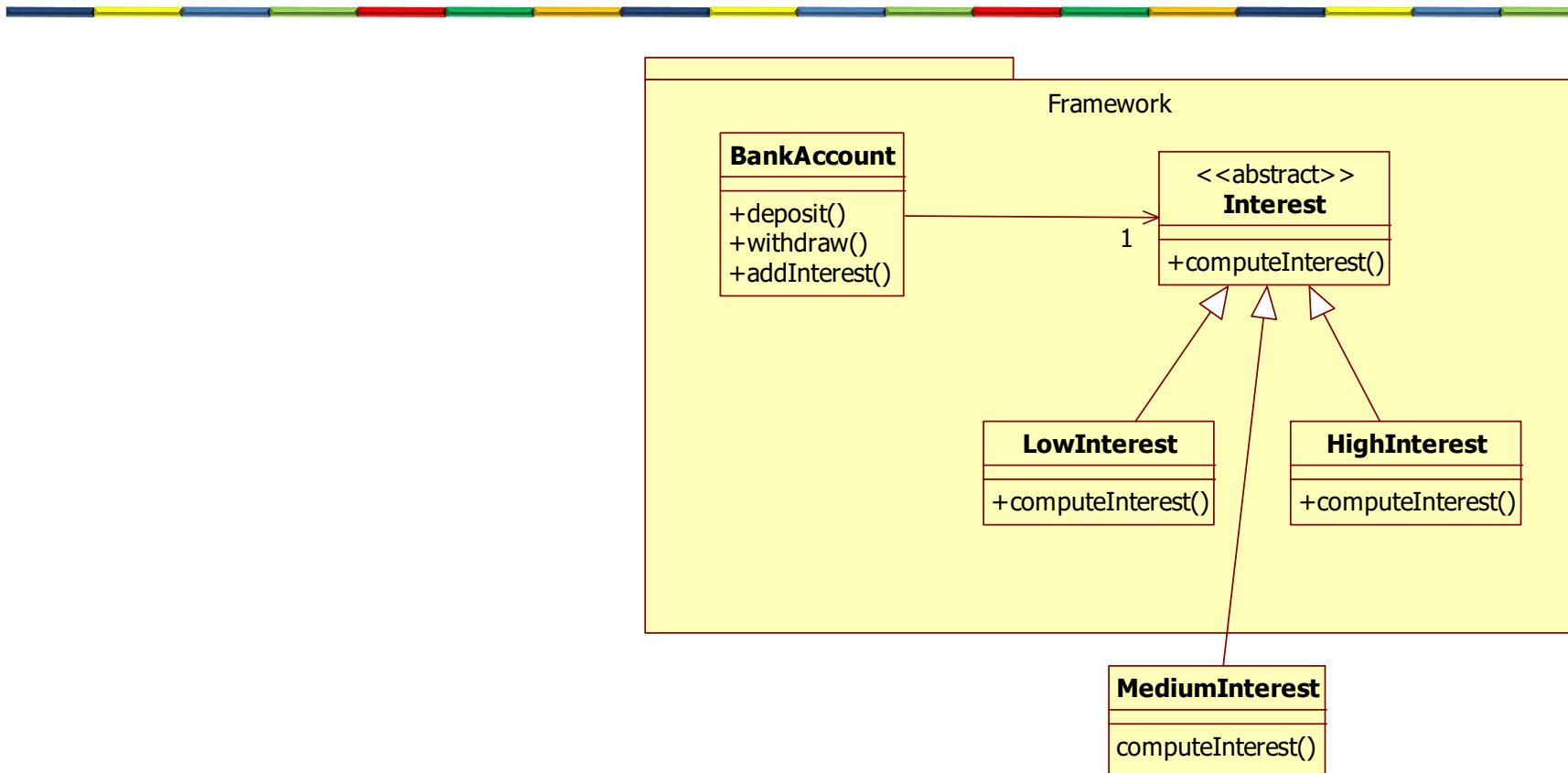
How to make hotspots?

- Plugin new algorithms
 - Strategy pattern, Chain of responsibility pattern
- Plugin new state behavior
 - State pattern
- Plugin new listeners
 - Observer pattern
- Translate between your code and FW code
 - Adapter pattern
- Plugin new actions
 - Command pattern
- Plugin new traversal algorithm
 - Iterator pattern
- Create new objects
 - Factory
- Add classes to a tree structure
 - Composite pattern

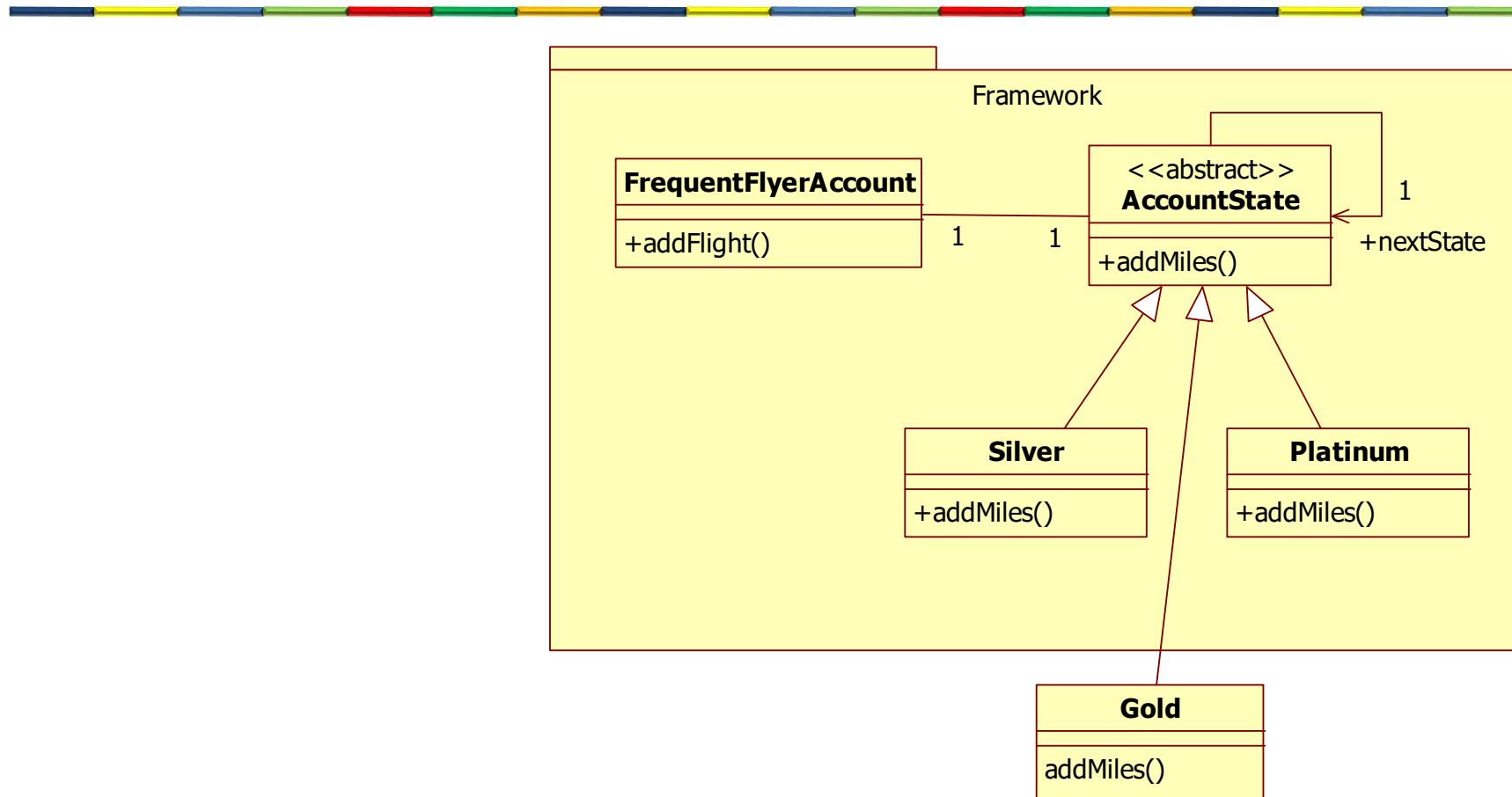
Plugin points: Composite pattern



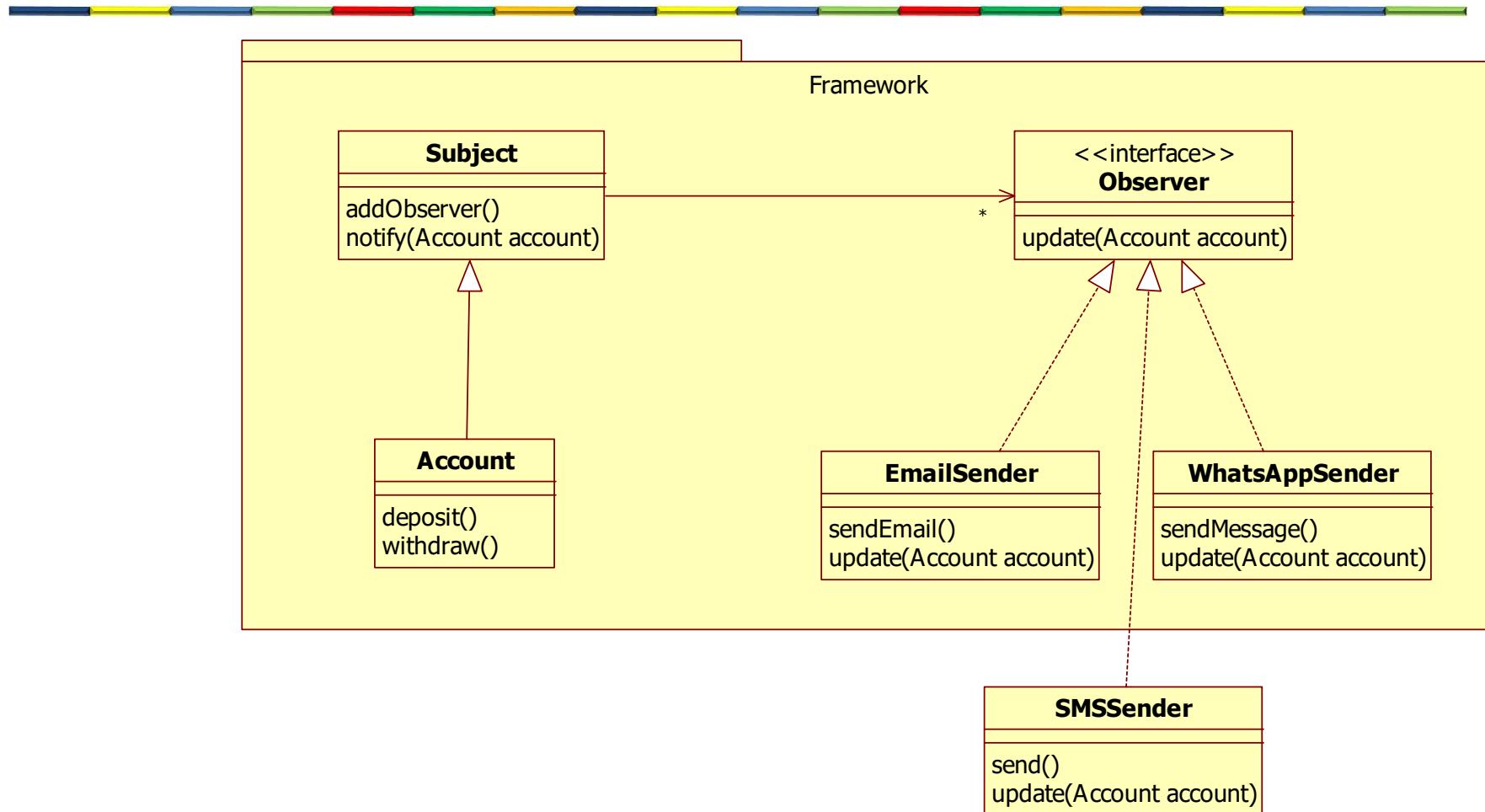
Plugin points: Strategy pattern



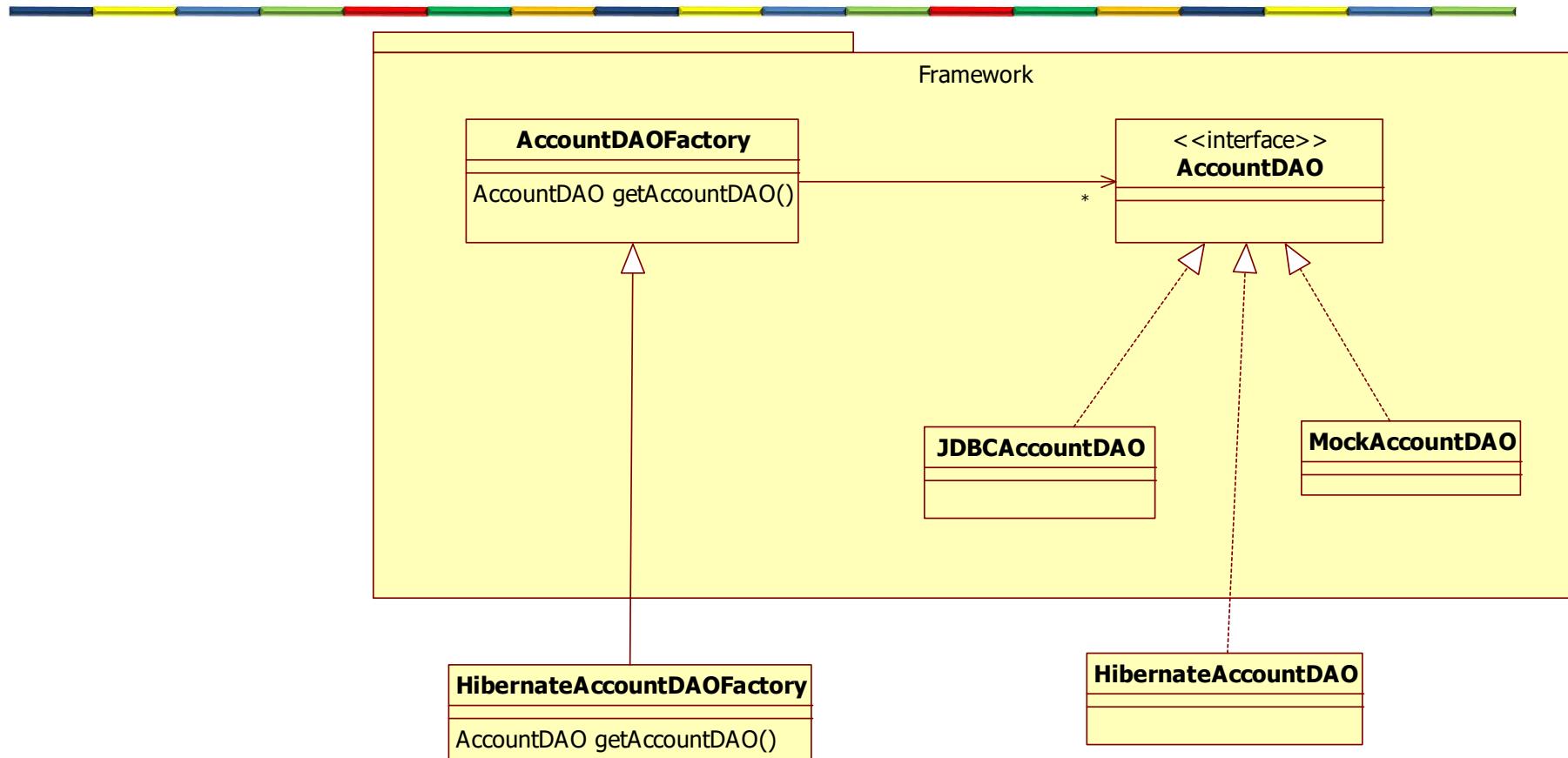
Plugin points: State pattern



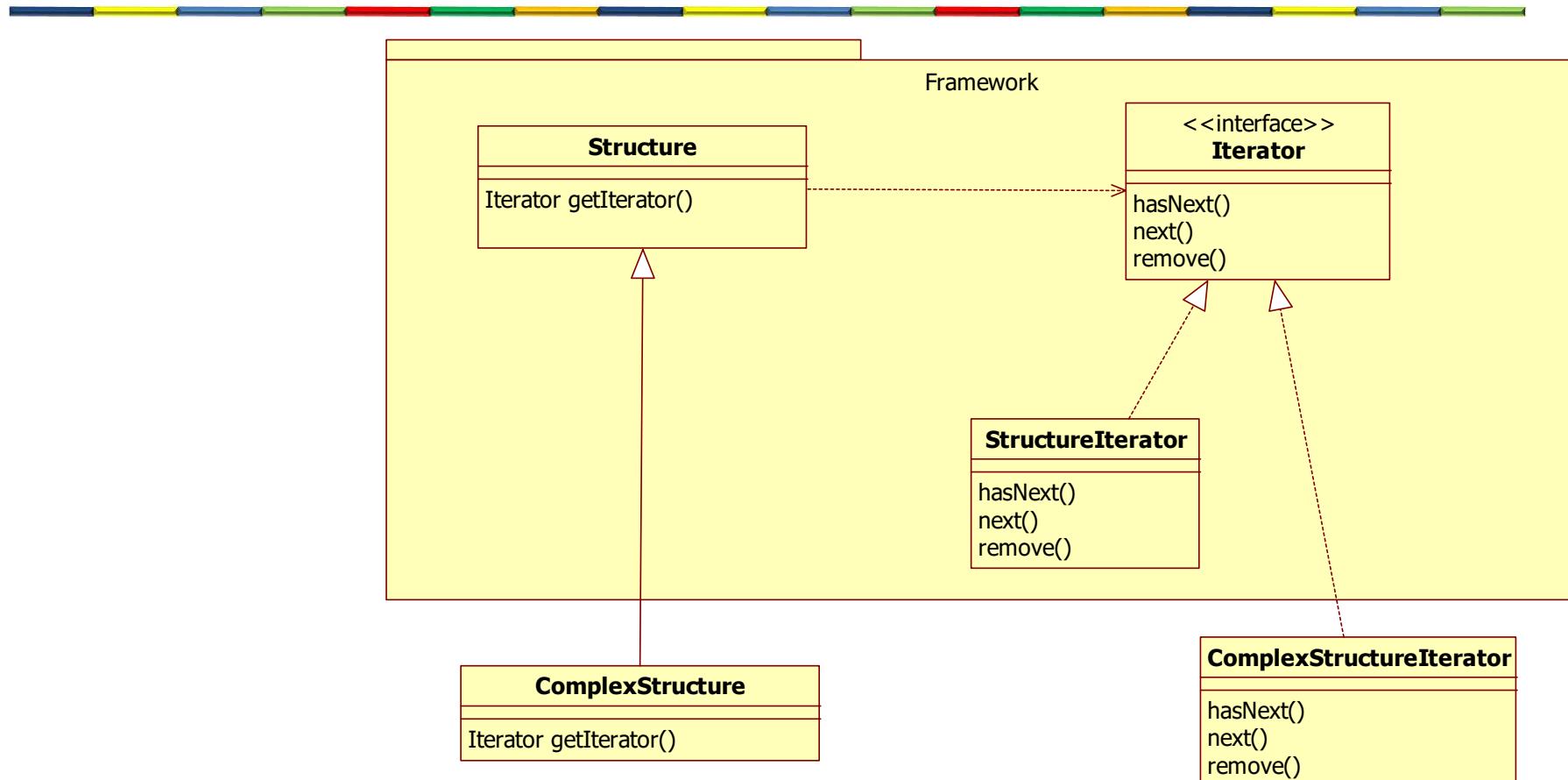
Plugin points: Observer pattern



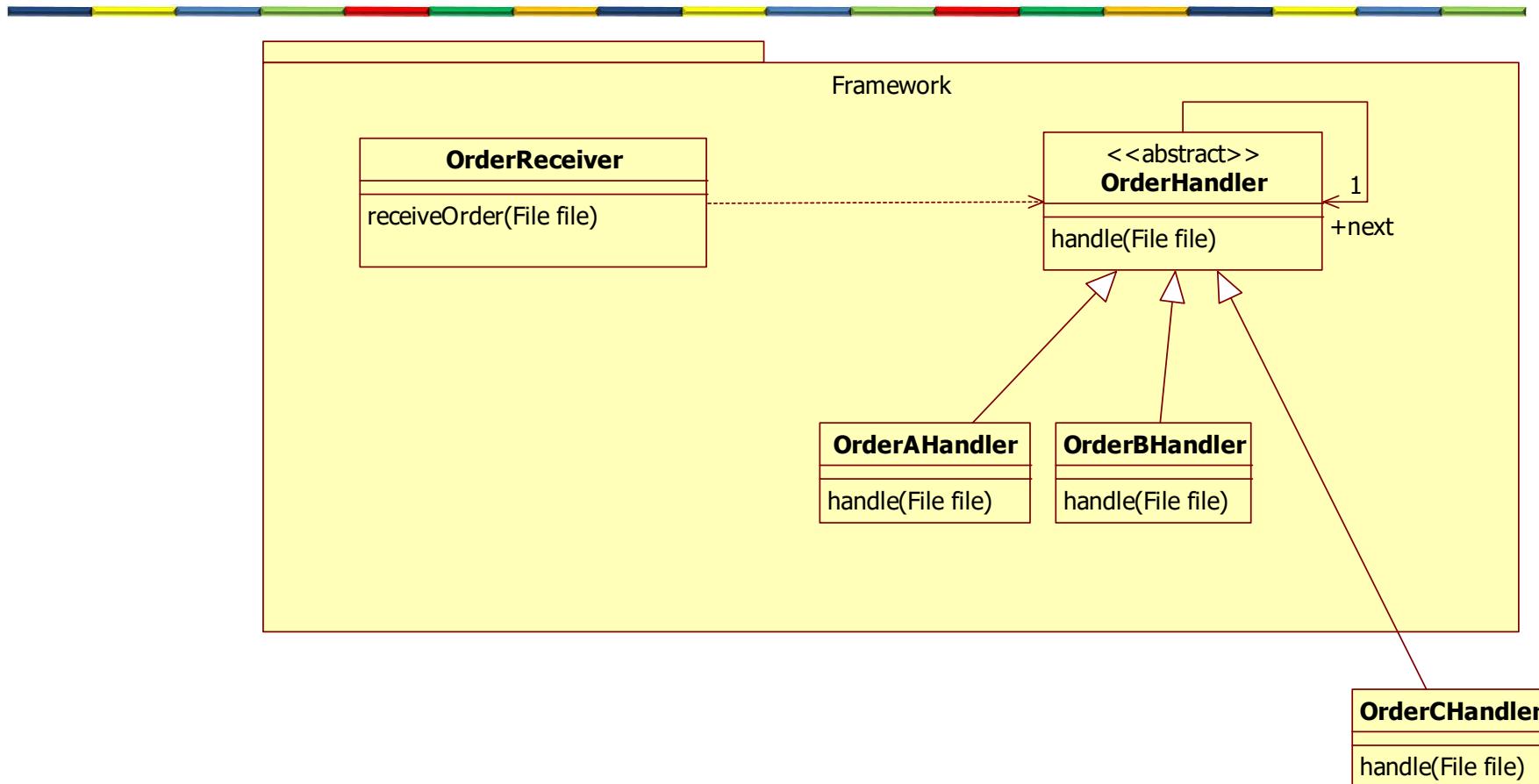
Plugin points: Factory pattern



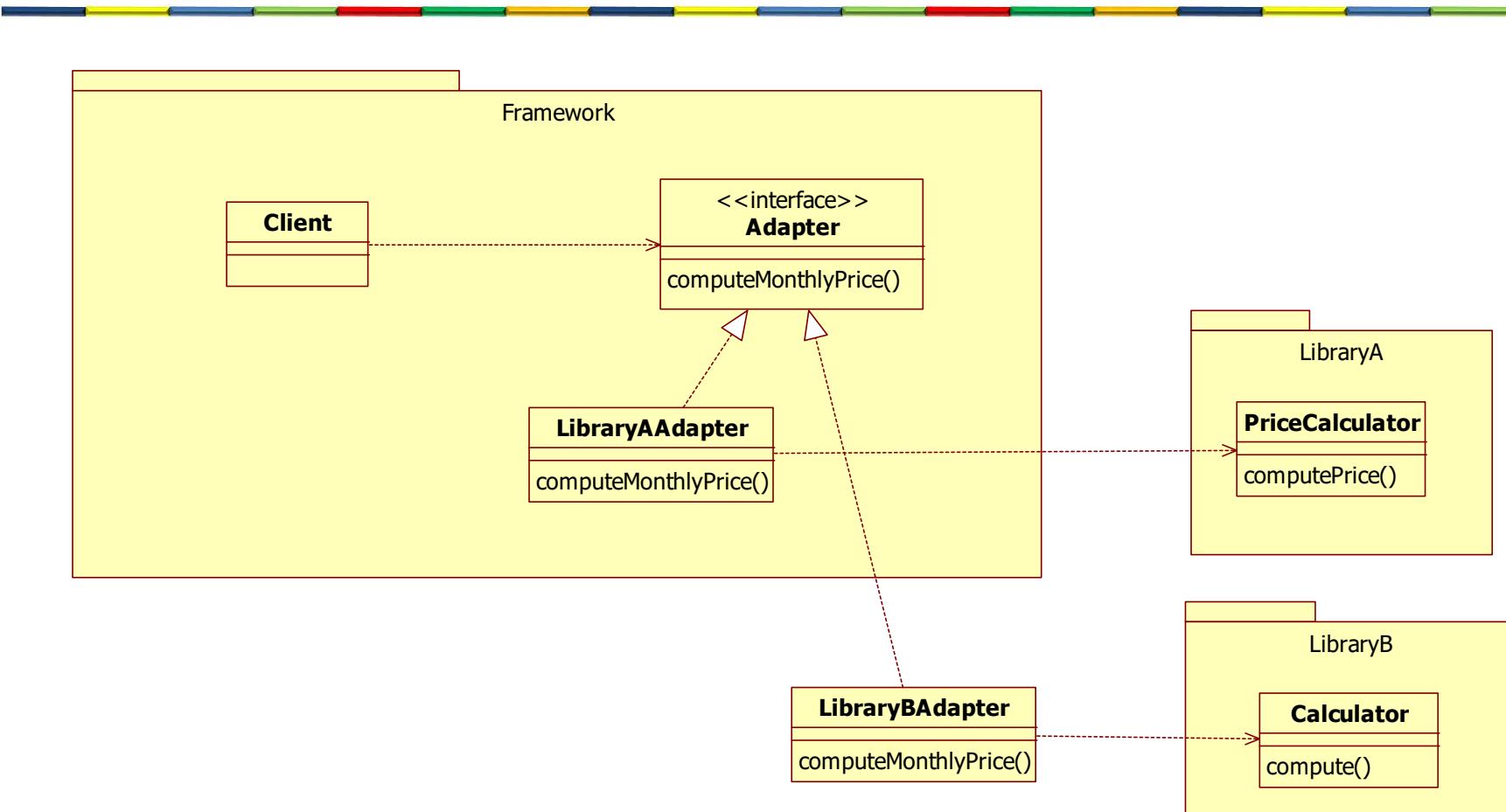
Plugin points: Iterator pattern



Plugin points: COR pattern

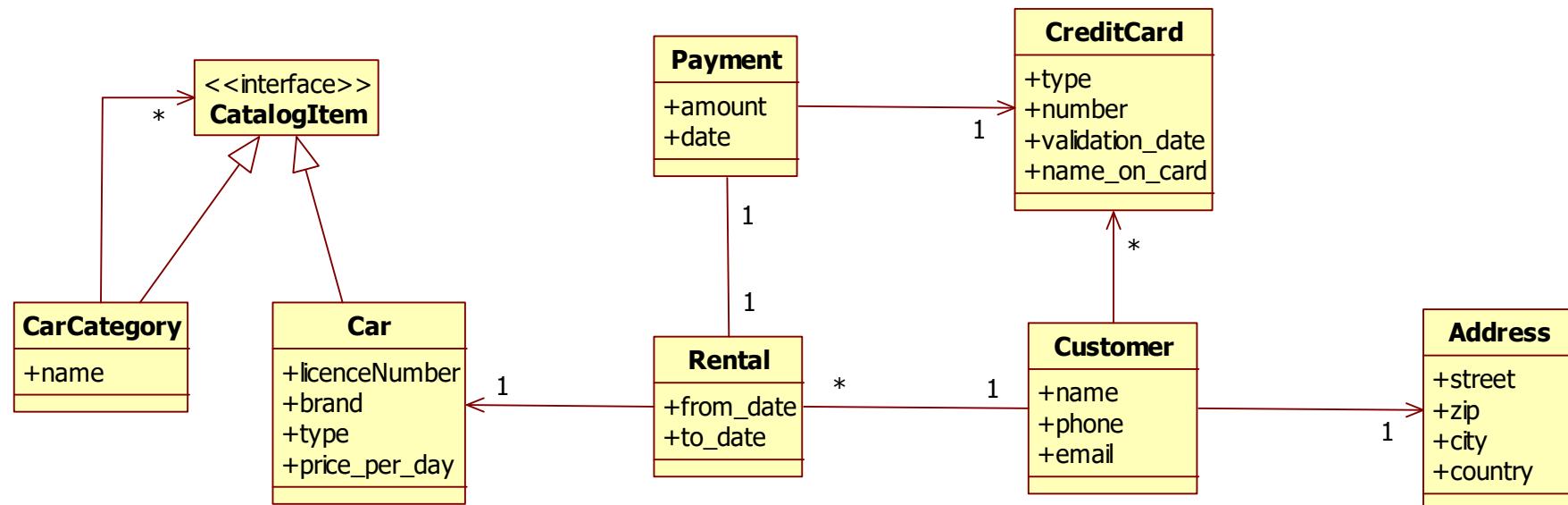


Plugin points: Adapter pattern

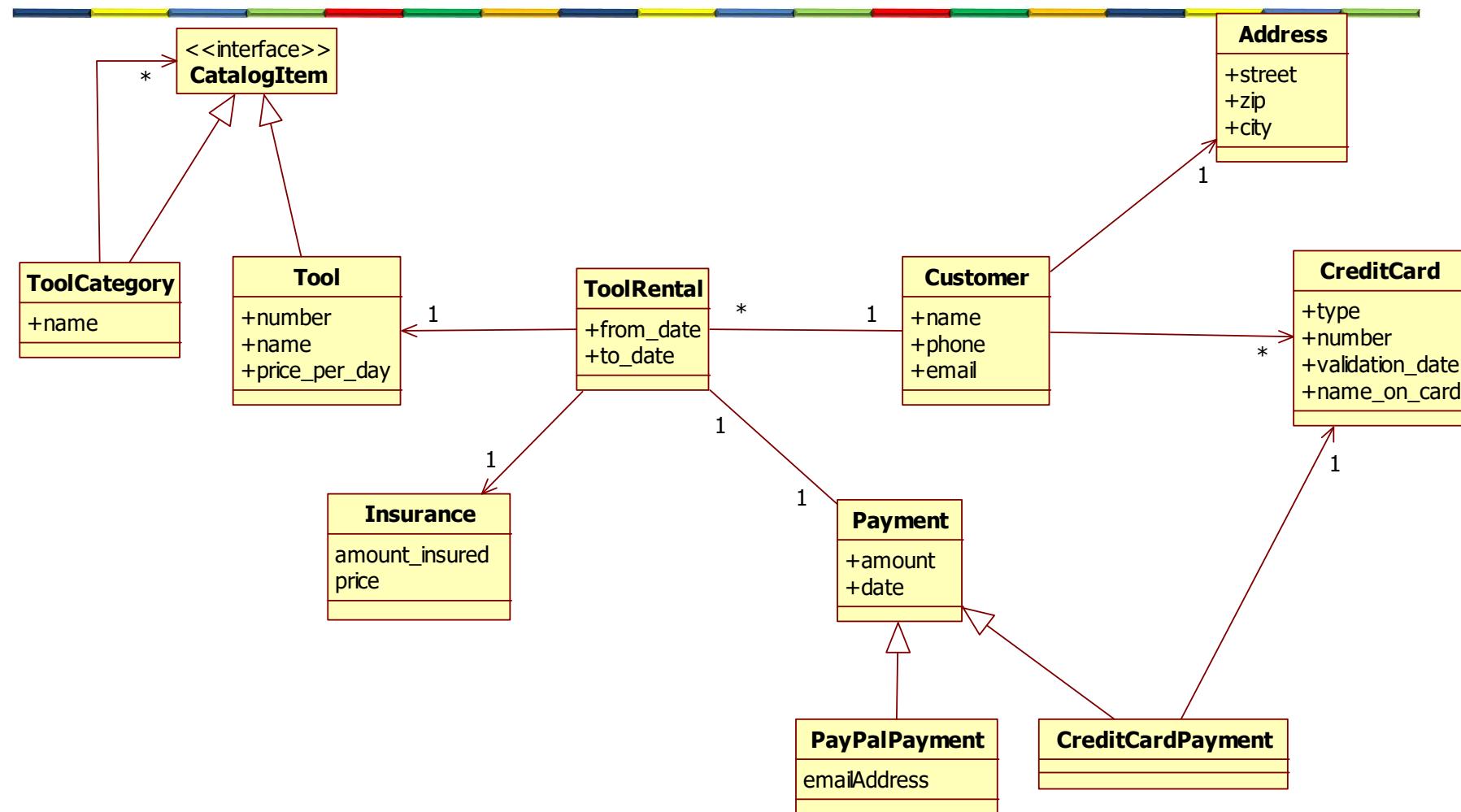


RENTAL FRAMEWORK

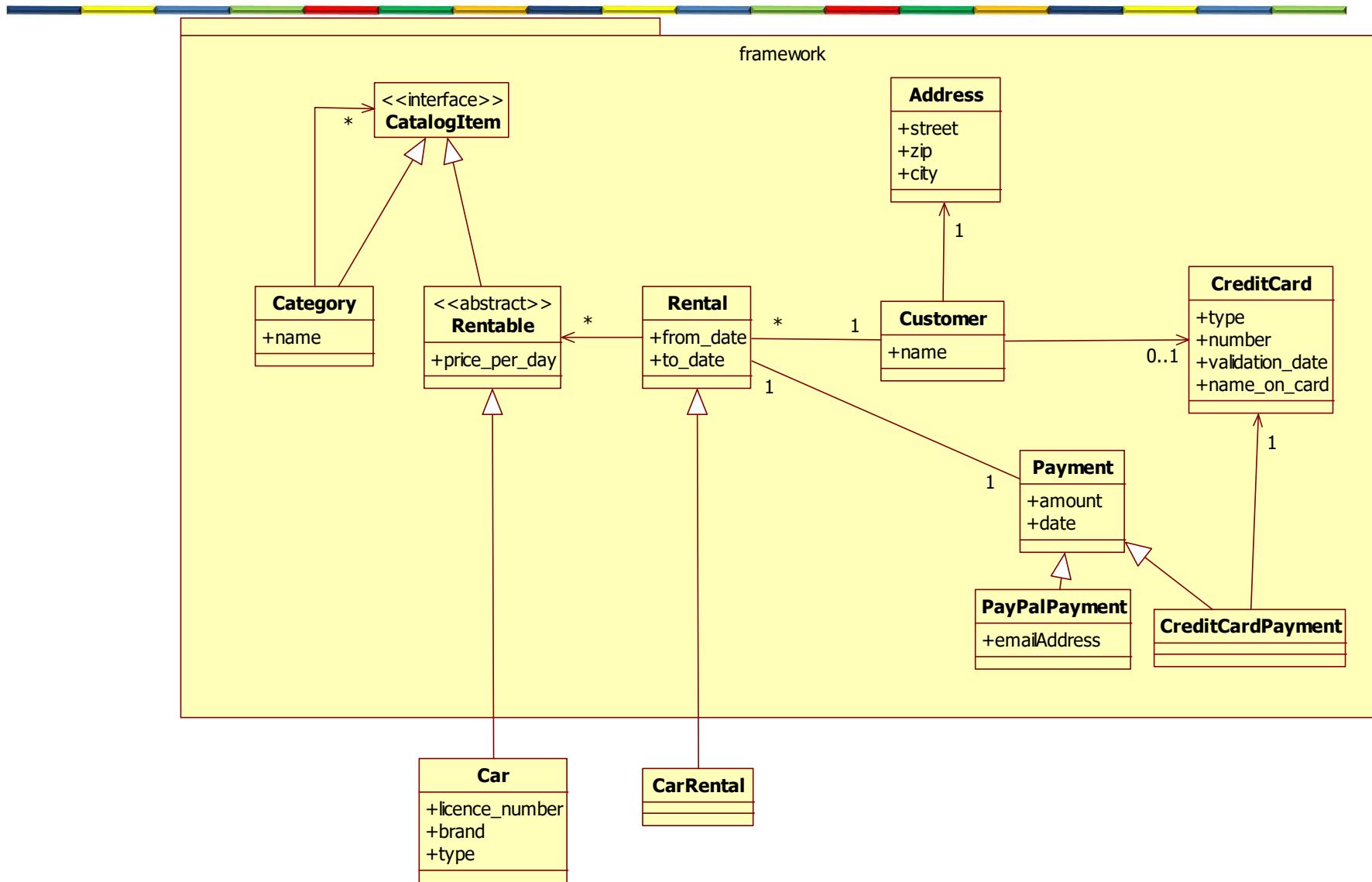
Car rental application



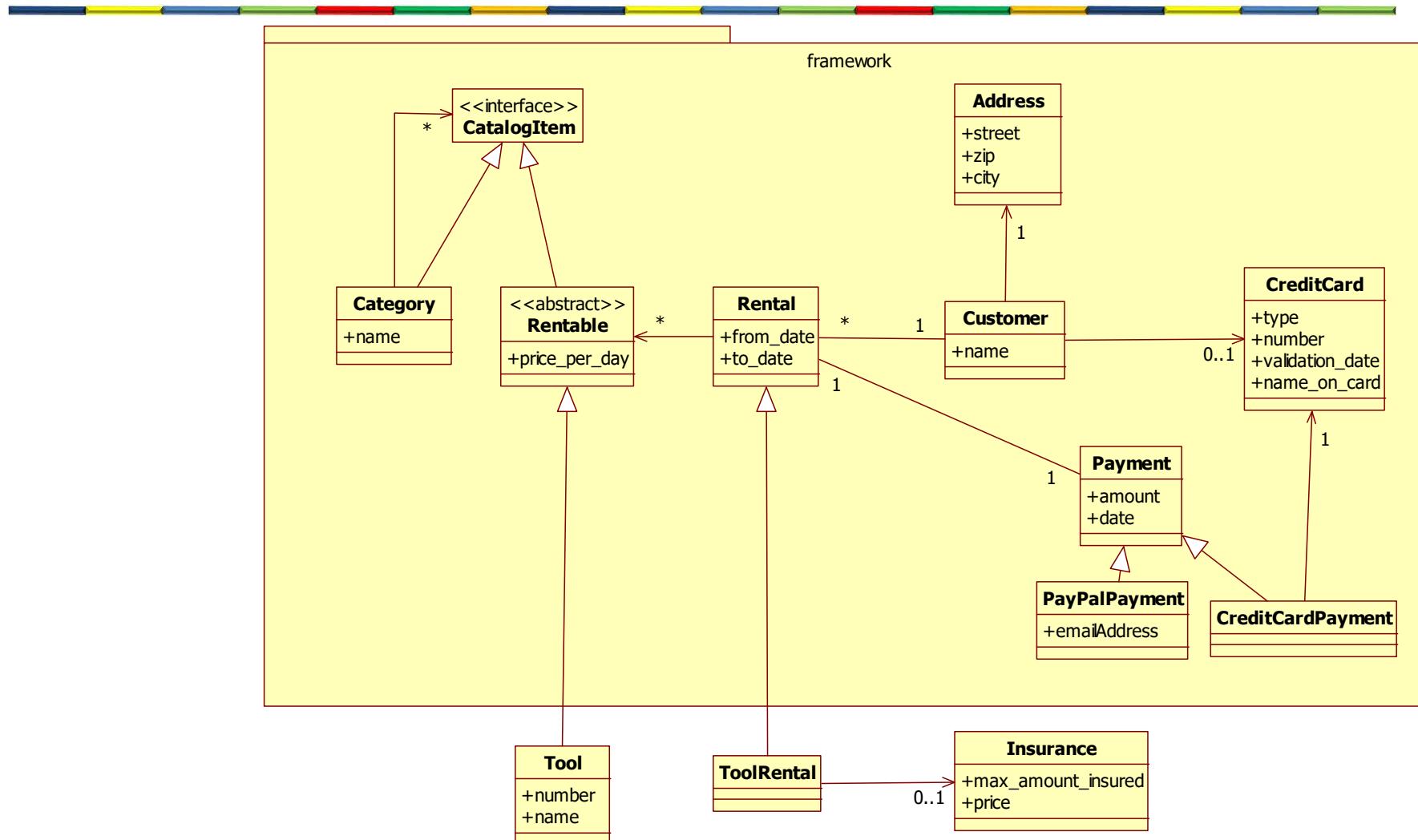
Tool rental application



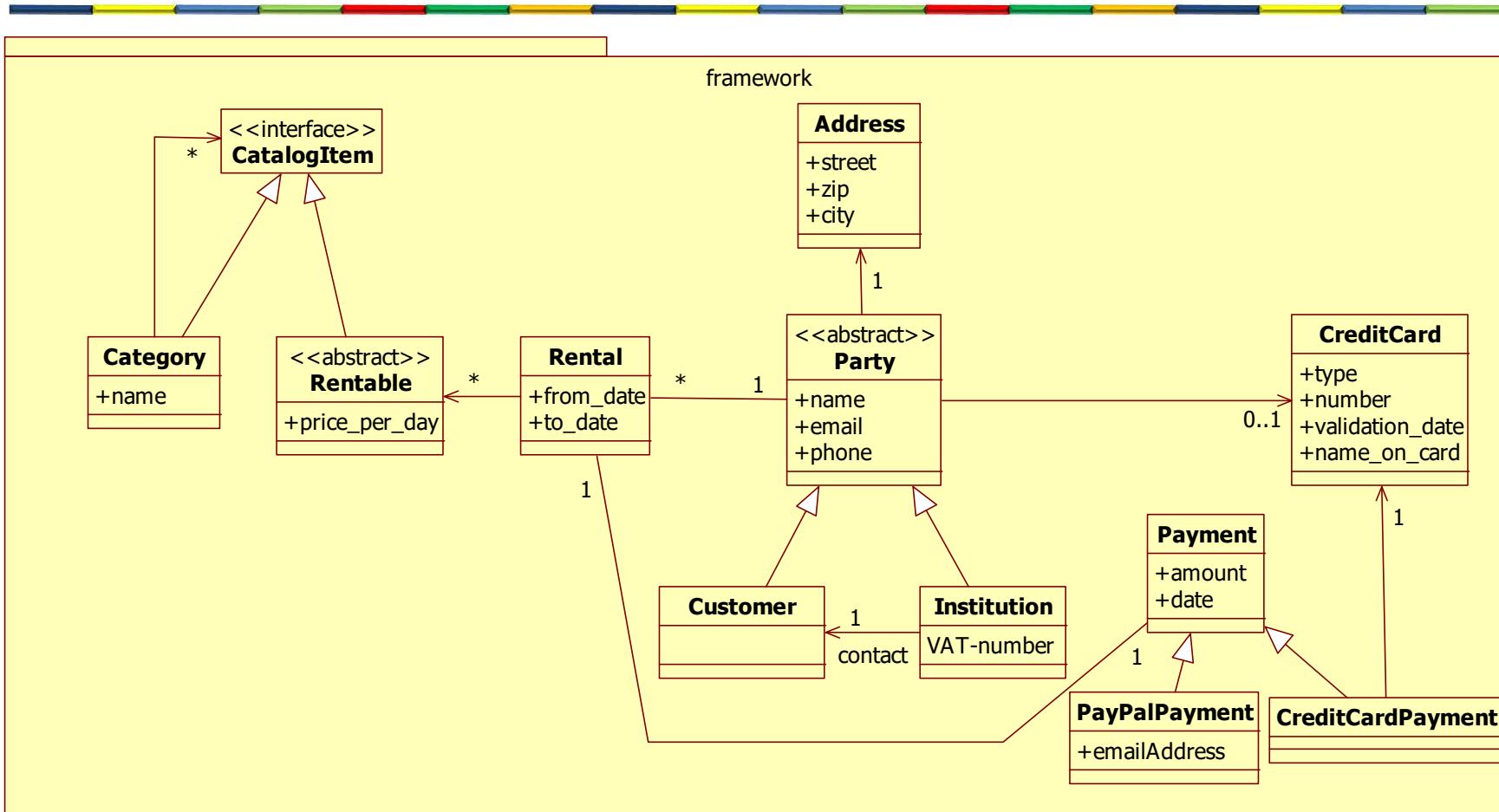
Framework + Car rental application



Framework + Tool rental application

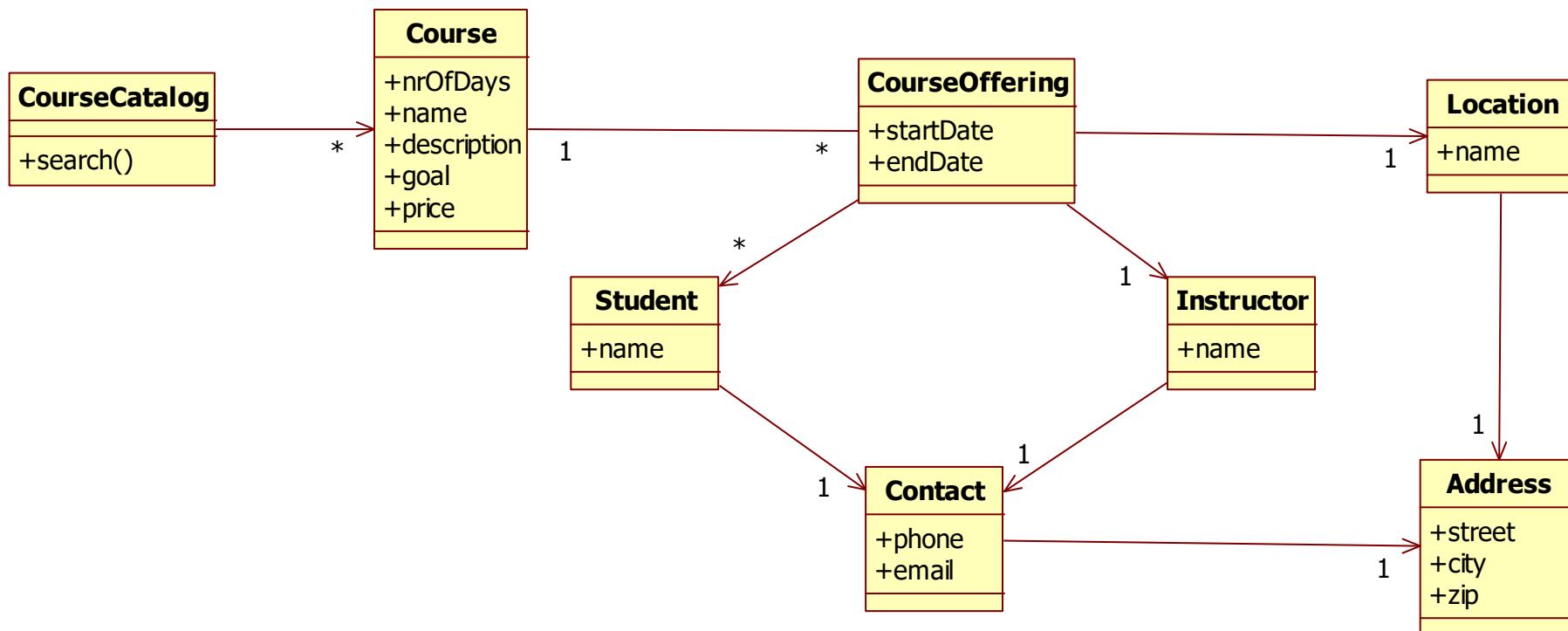


Party pattern

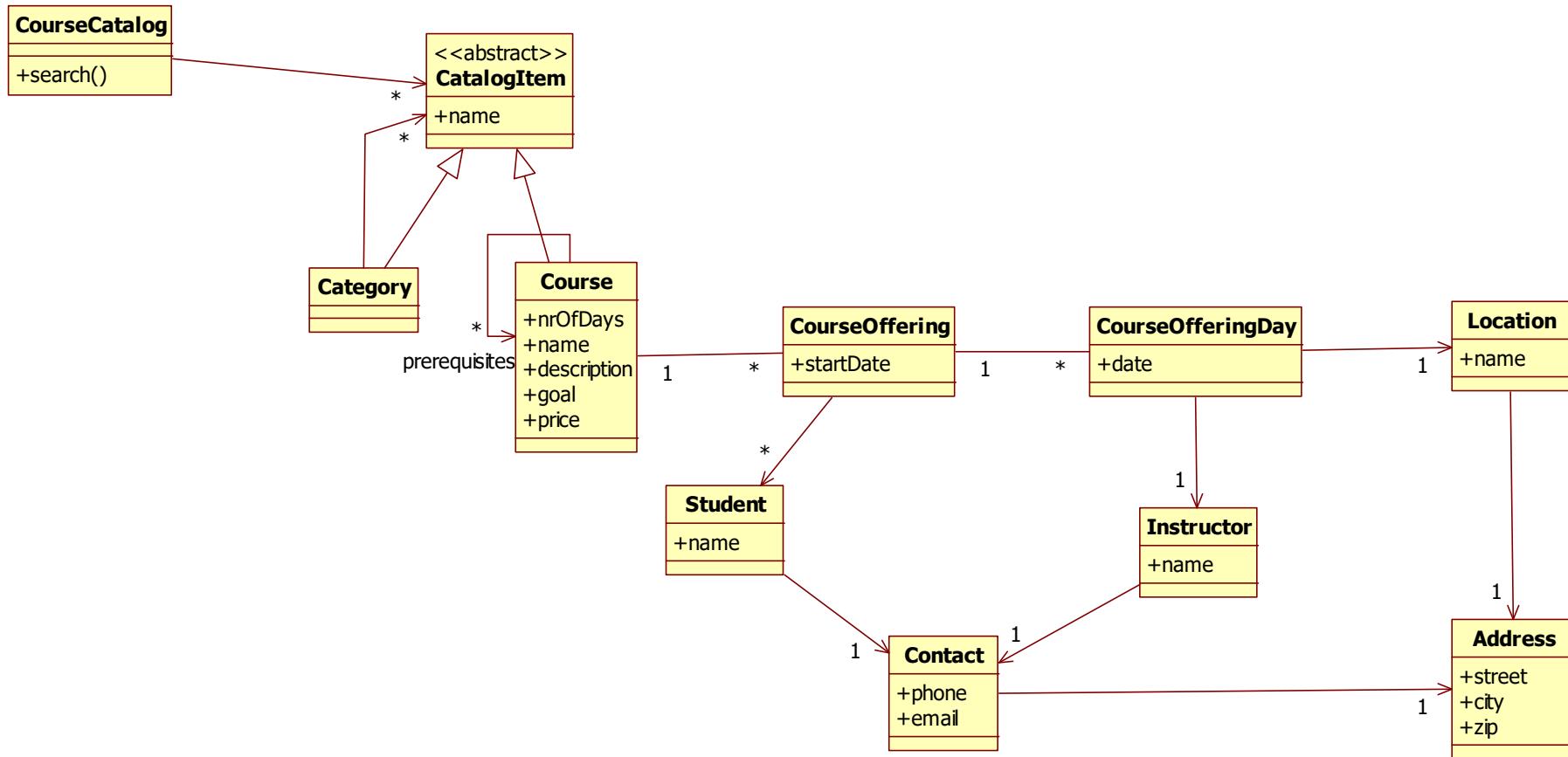


COURSE REGISTRATION FRAMEWORK

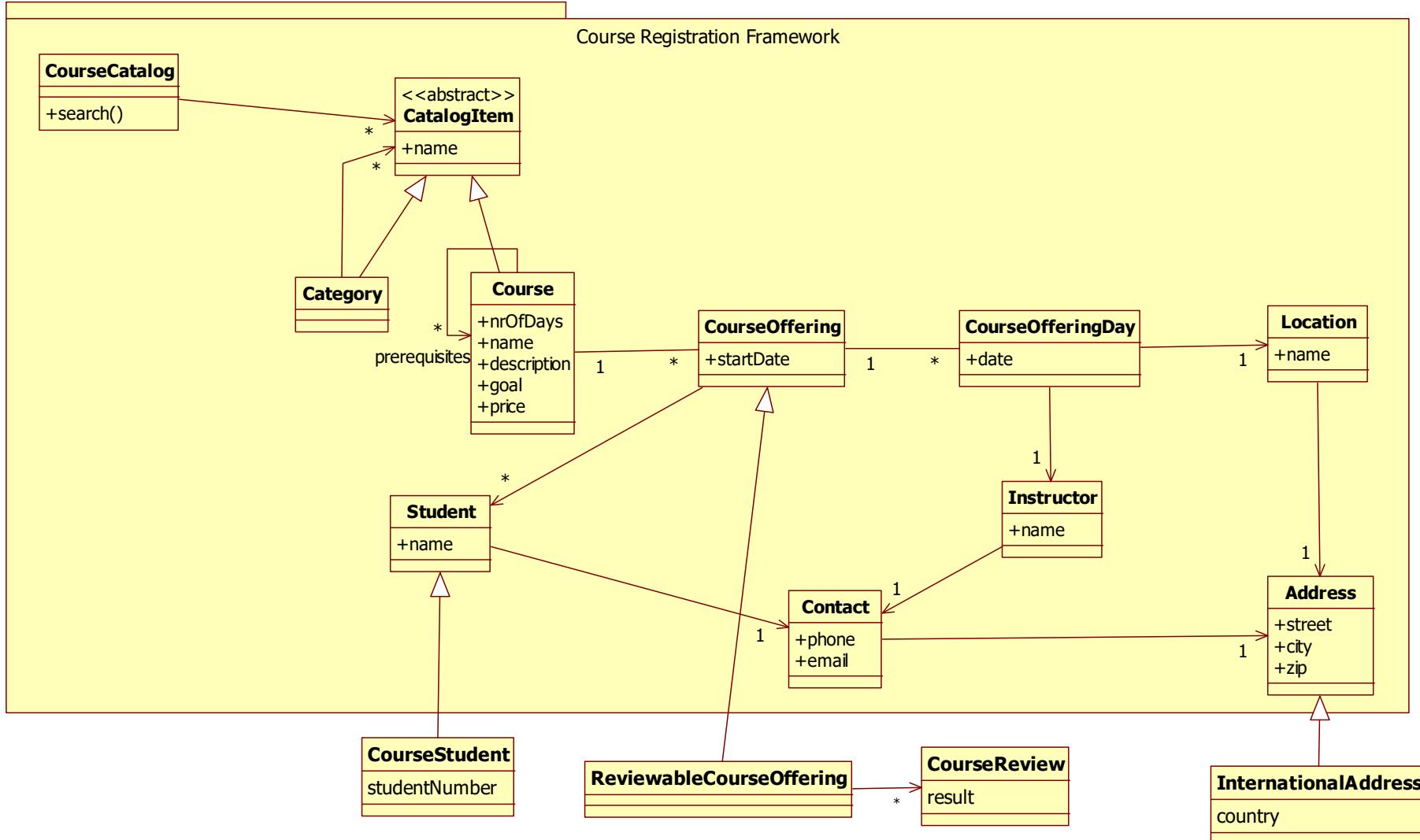
Simple course registration system



Advanced course registration system

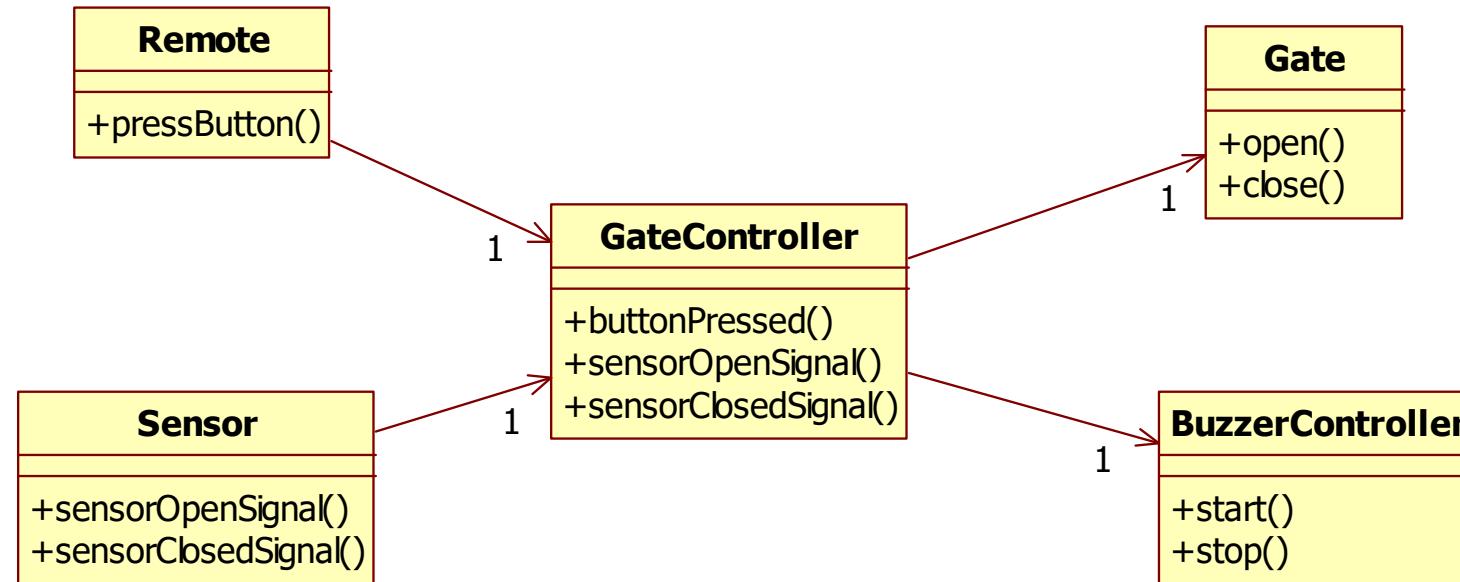


Course registration framework

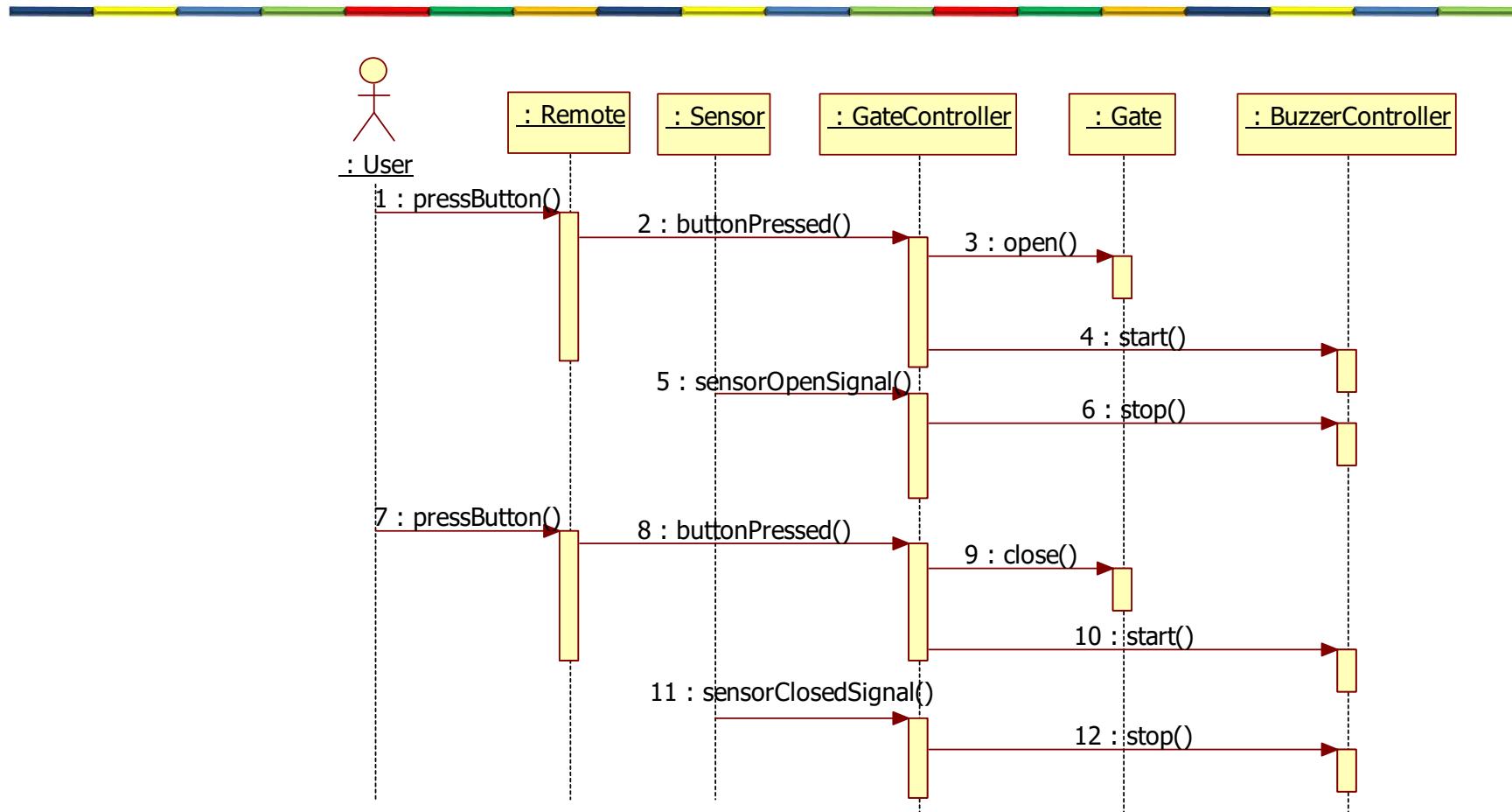


GATE CONTROLLER FRAMEWORK

Gate controller application



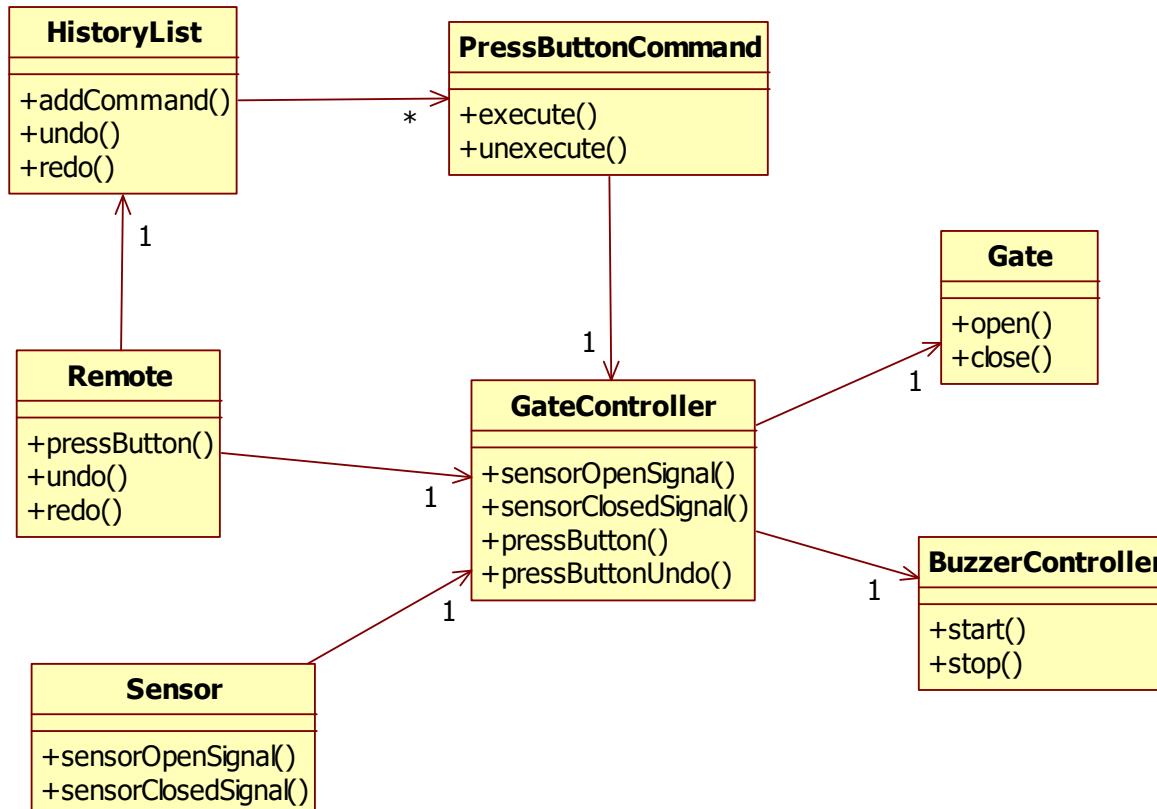
Gate controller application



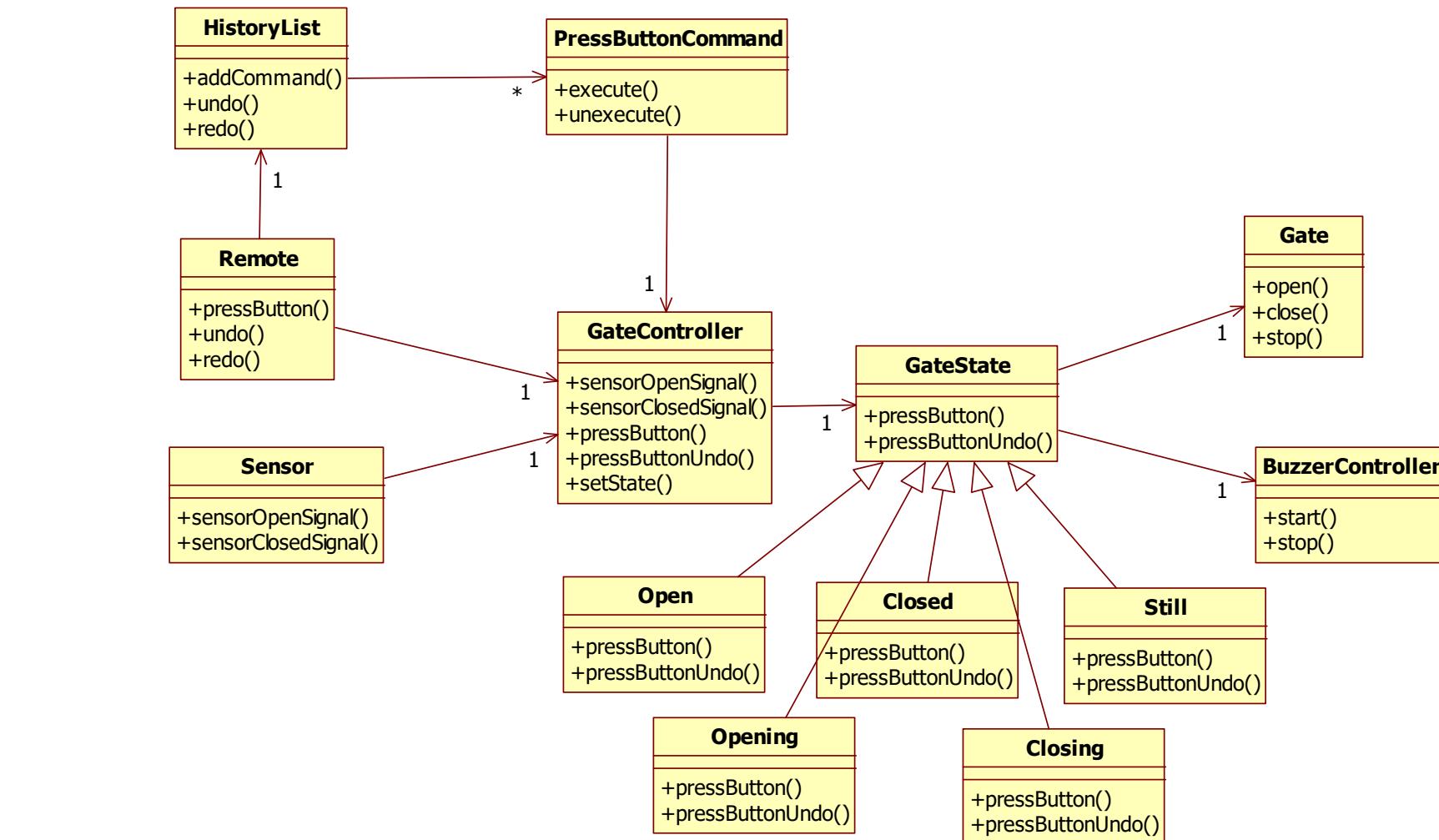
GateController framework

- Add undo/redo button
- Support different gate states (still, 75% open)
- Support multiple signaling devices (buzzers, lights, etc.)
- Support different gates

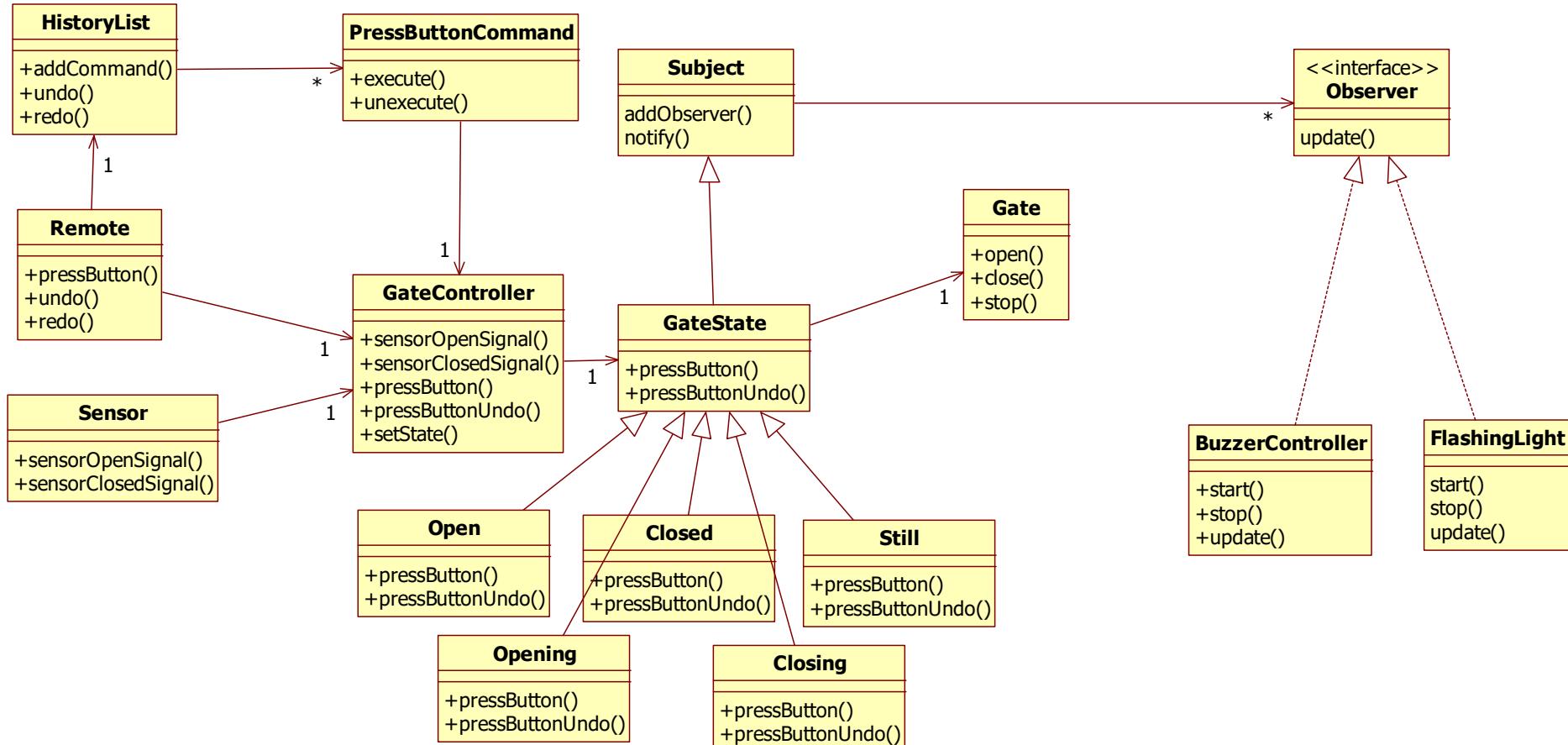
Support undo/redo button



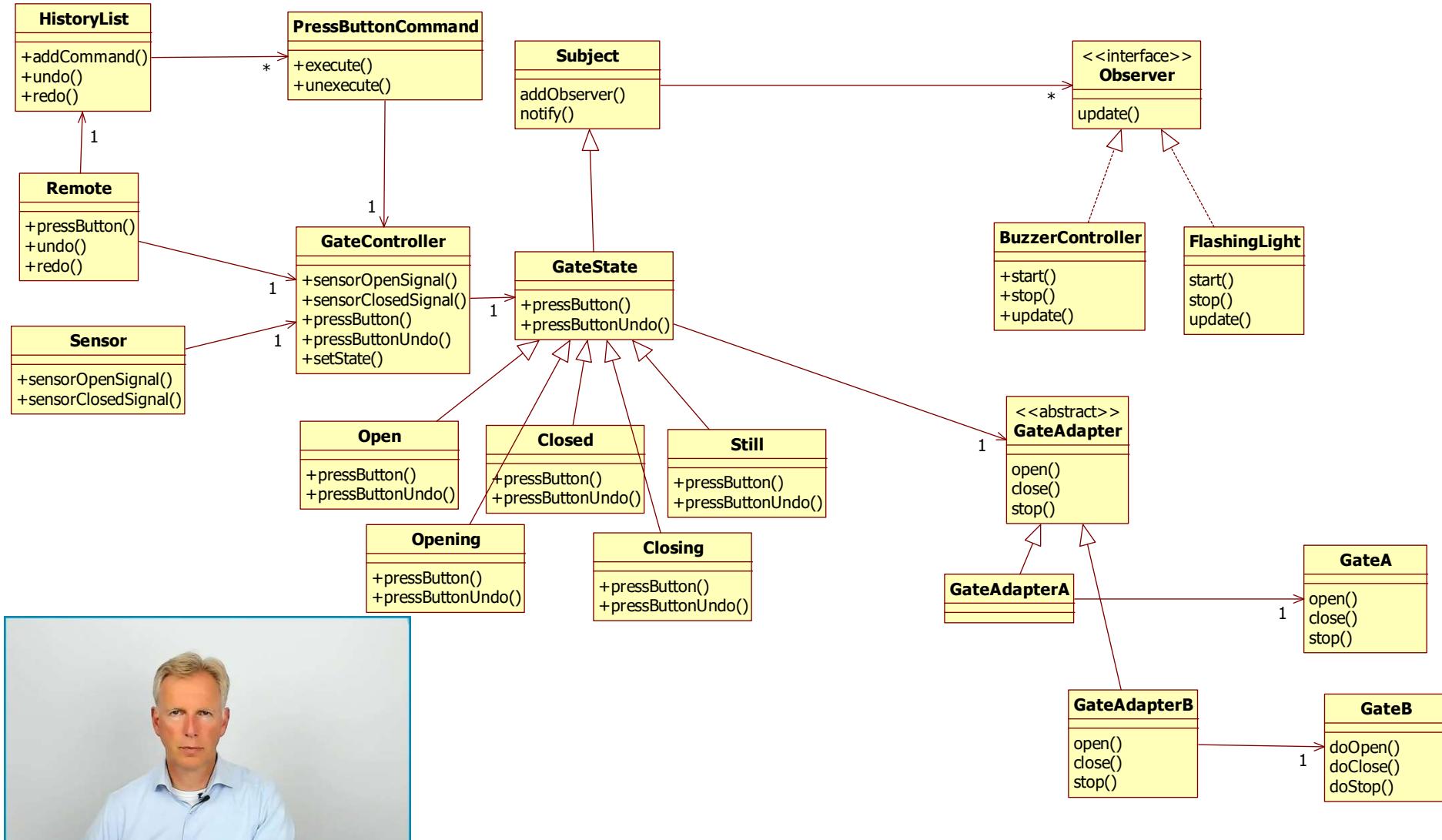
Support different gate states (still, 75% open, half open)



Support multiple signaling devices (buzzers, lights, etc.)



Support different gates



Main point

- A Framework captures domain specific expertise in abstract and concrete classes.
- The Unified Field which is the home of all the laws of nature, captures the intelligence of the whole universe.

Connecting the parts of knowledge with the wholeness of knowledge

1. Frameworks embody expertise: this frees developers who are not necessarily experts in a certain area from the complexity of the underlying details.
2. Frameworks are based on patterns. These patterns create the plugin points for the framework.



3. **Transcendental consciousness** is the home of all the Laws of Nature which govern the entire universe.
4. **Wholeness moving within itself:** In unity consciousness one spontaneously perceives the eternally silent, fully awake field of Pure Consciousness in the midst of all diversity.

Lesson 11 Framework implementation



L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

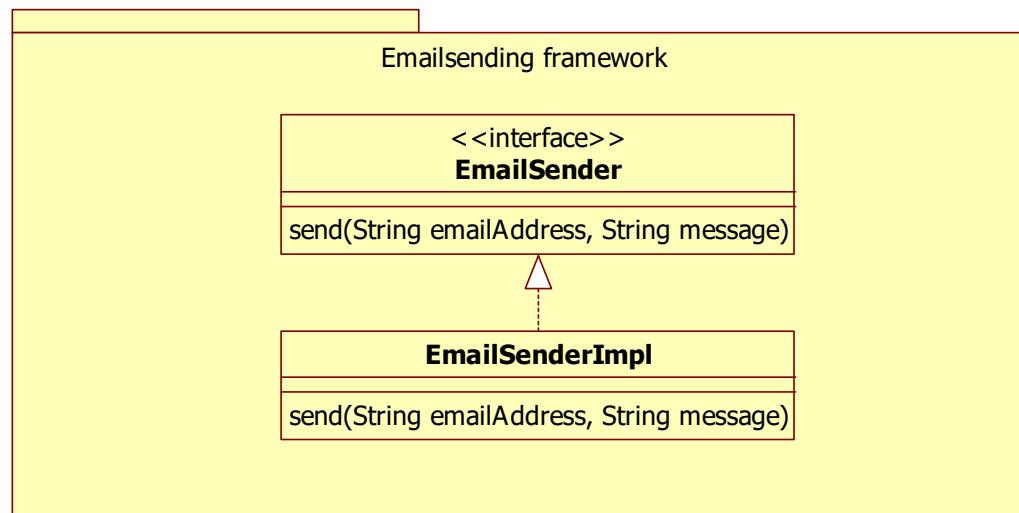
Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

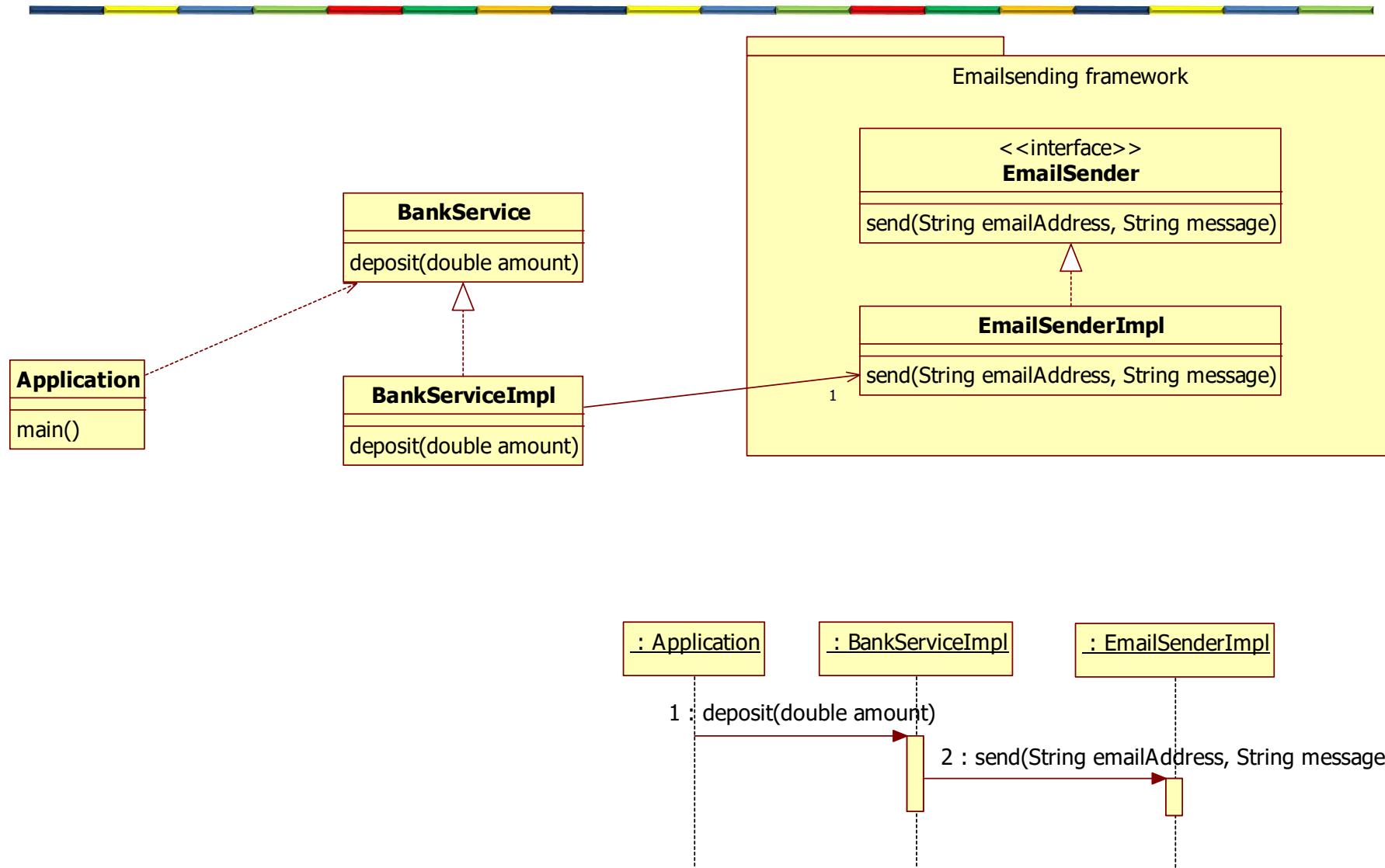
Final

Simple framework

```
public interface EmailSender {  
    void send(String emailAddress, String message);  
}  
  
public class EmailSenderImpl implements EmailSender {  
  
    public void send(String emailAddress, String message) {  
        System.out.println("sending email to "+emailAddress+ " , message="+message);  
    }  
}
```



Using the framework



Using the framework

```
public class Application {

    public static void main(String[] args) {
        BankService bankService = new BankServiceImpl();
        EmailSender emailSender = new EmailSenderImpl();
        bankService.setEmailSender(emailSender);

        bankService.deposit(100.0);
    }
}
```

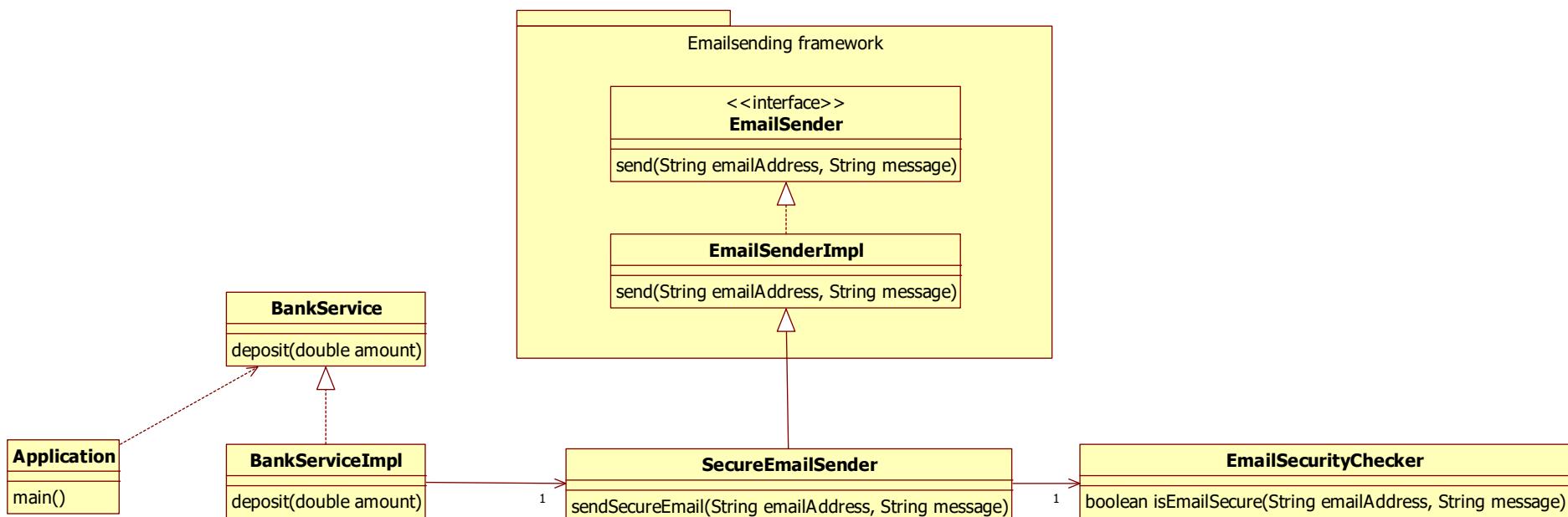
```
public interface BankService {
    void deposit(double amount);
    void setEmailSender(EmailSender emailService);
}
```

```
public class BankServiceImpl implements BankService {
    private EmailSender emailSender;

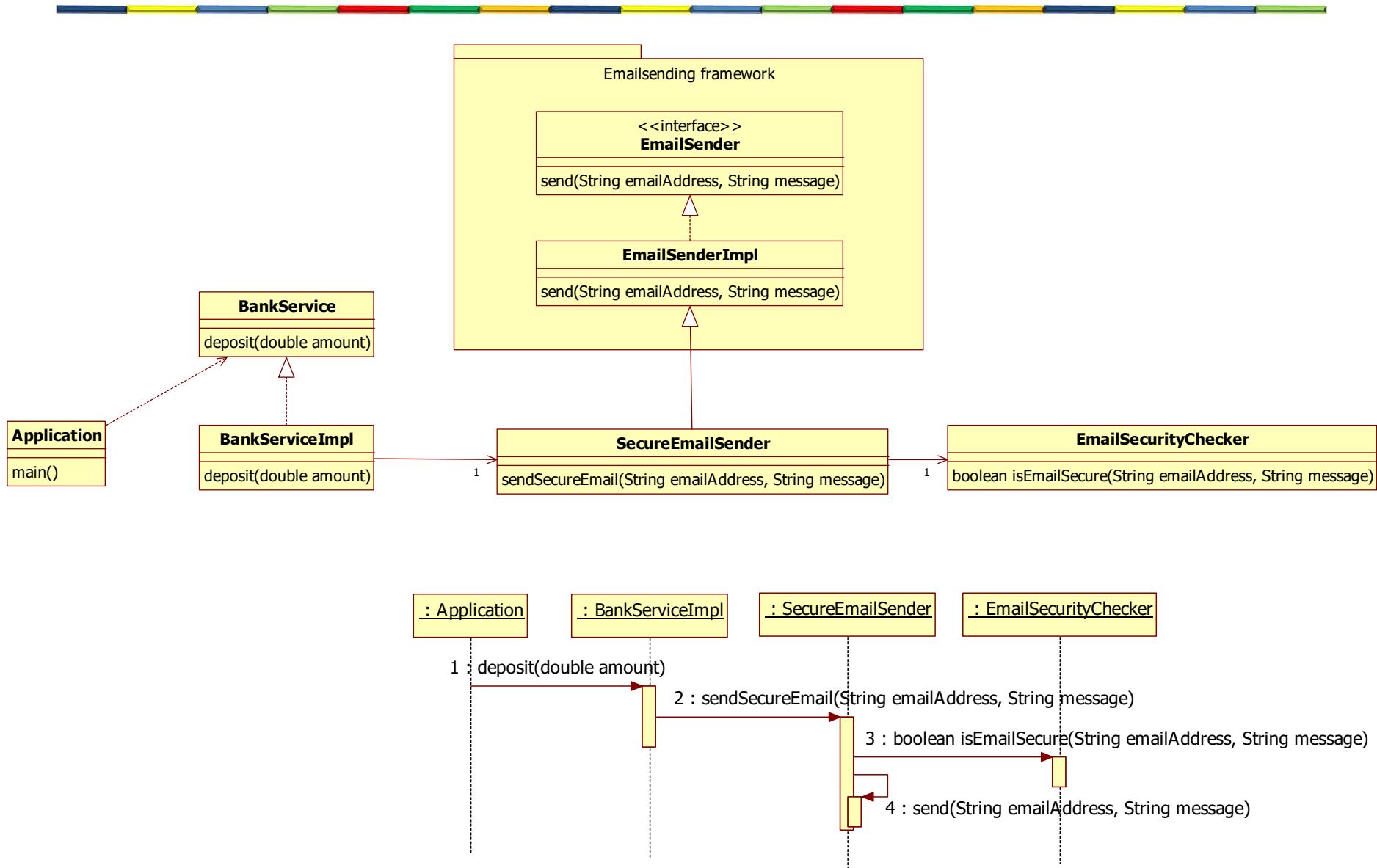
    public void setEmailSender(EmailSender emailSender) {
        this.emailSender = emailSender;
    }

    public void deposit(double amount) {
        emailSender.send("customer@gmail.com", "deposit of $" + amount);
    }
}
```

Using the framework



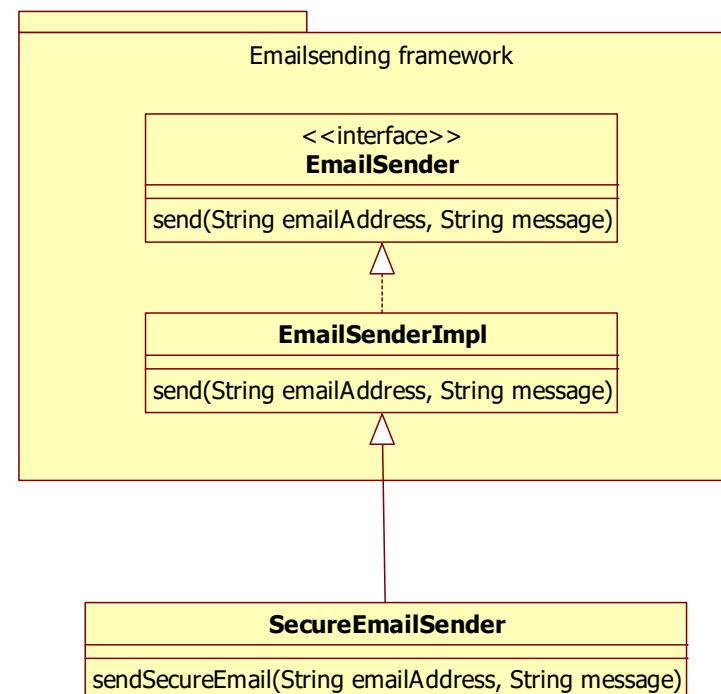
Inheritance



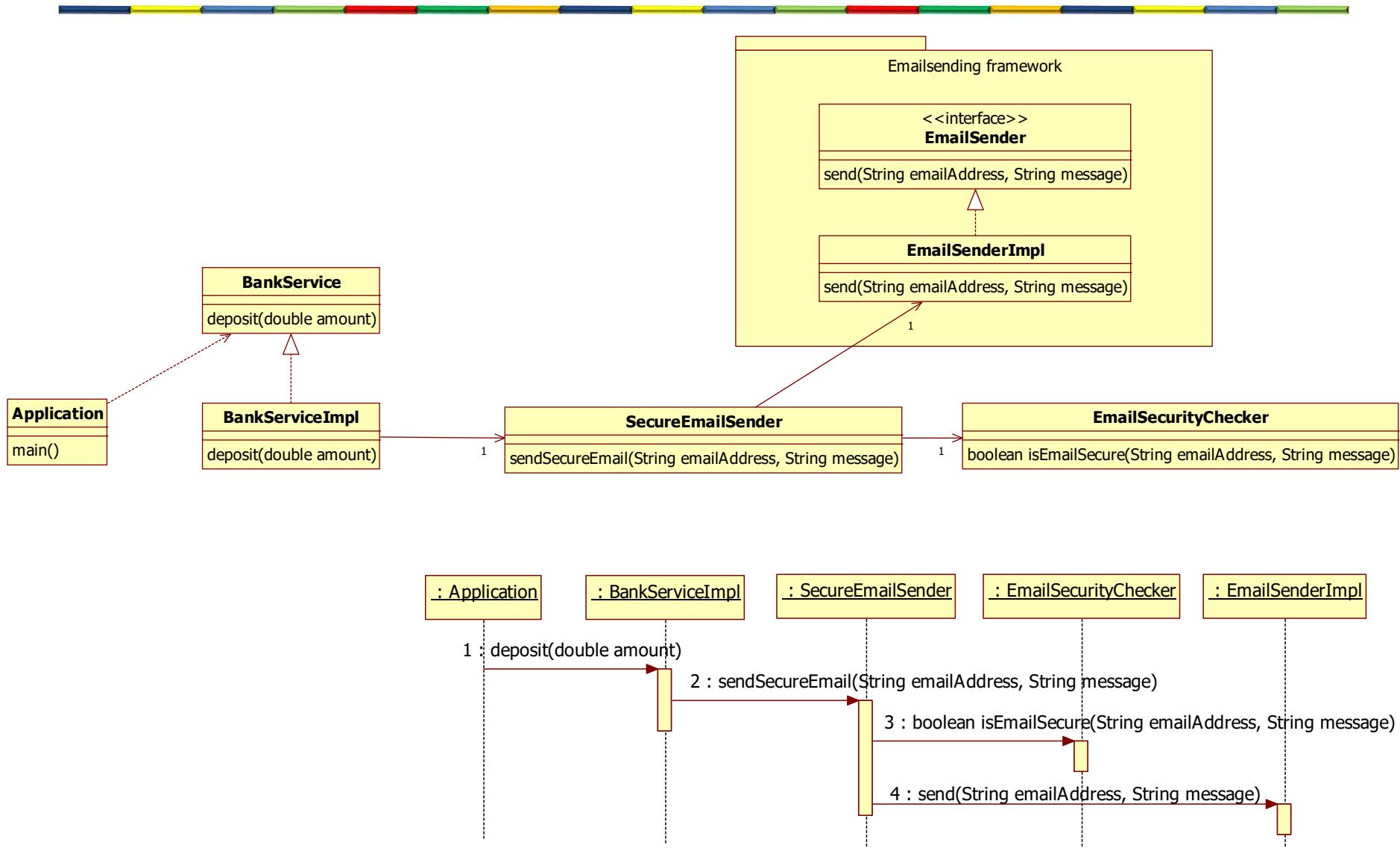
Inheritance

```
public class SecureEmailSender extends EmailSenderImpl{
    EmailSecurityChecker emailSecurityChecker = new EmailSecurityChecker();

    public void sendSecureEmail(String emailAddress, String message) {
        if (emailSecurityChecker.isEmailSecure(emailAddress, message)) {
            send(emailAddress, message);
        }
    }
}
```

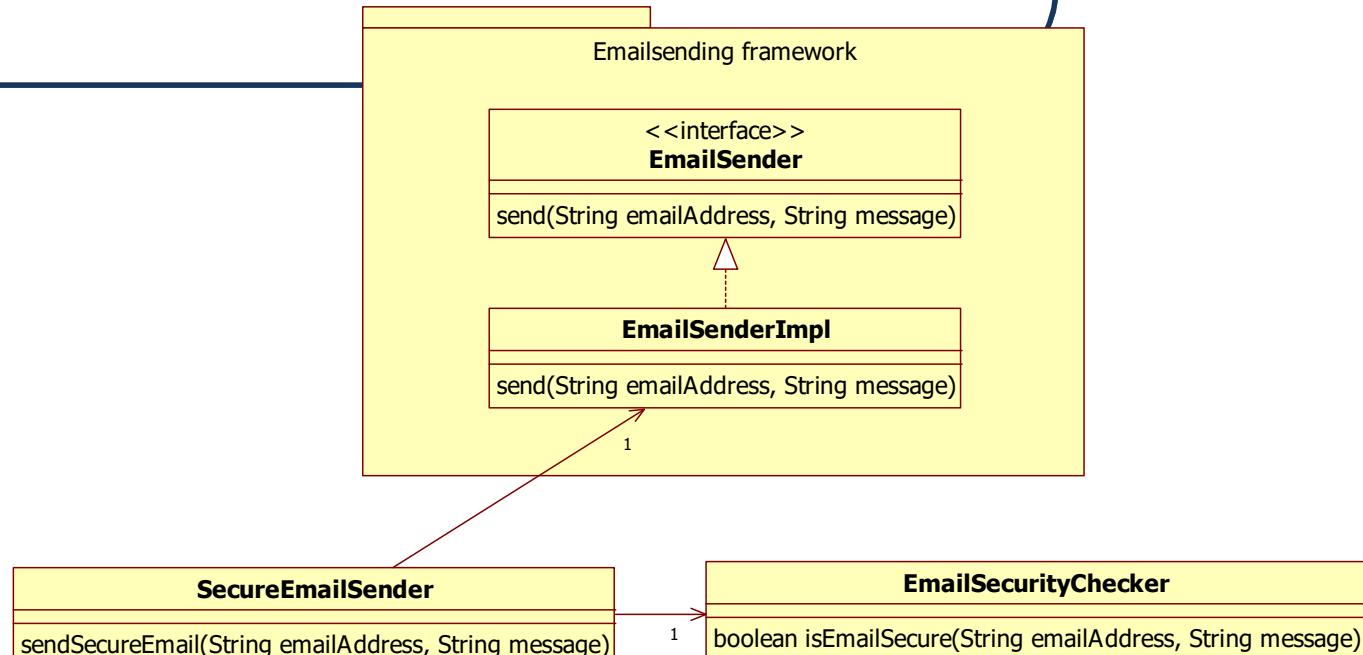


Composition



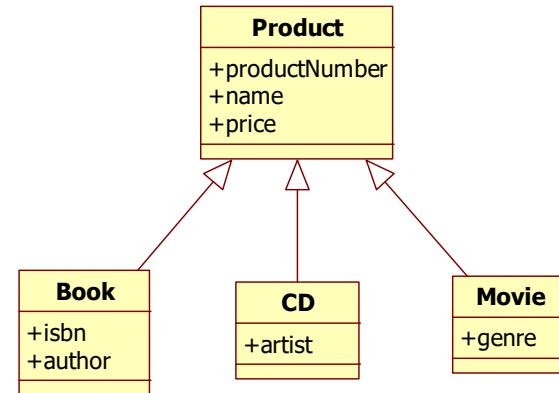
Composition

```
public class SecureEmailSender {  
    EmailSecurityChecker emailSecurityChecker = new EmailSecurityChecker();  
    EmailSender emailSender;  
  
    public void setEmailSender(EmailSender emailSender) {  
        this.emailSender = emailSender;  
    }  
  
    public void sendSecureEmail(String emailAddress, String message) {  
        if (emailSecurityChecker.isEmailSecure(emailAddress, message)) {  
            emailSender.send(emailAddress, message);  
        }  
    }  
}
```

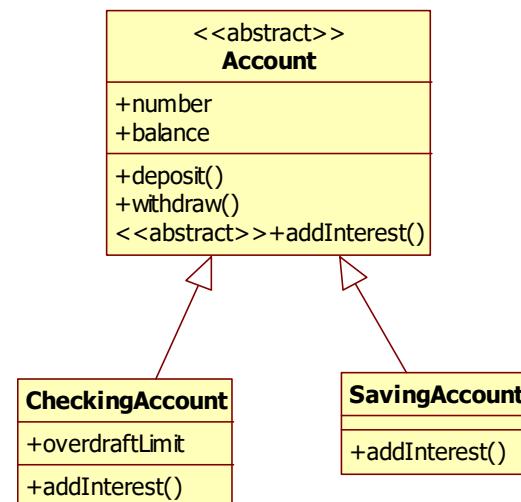


Advantages of inheritance

- Code reusability
 - Minimize the amount of duplicate code



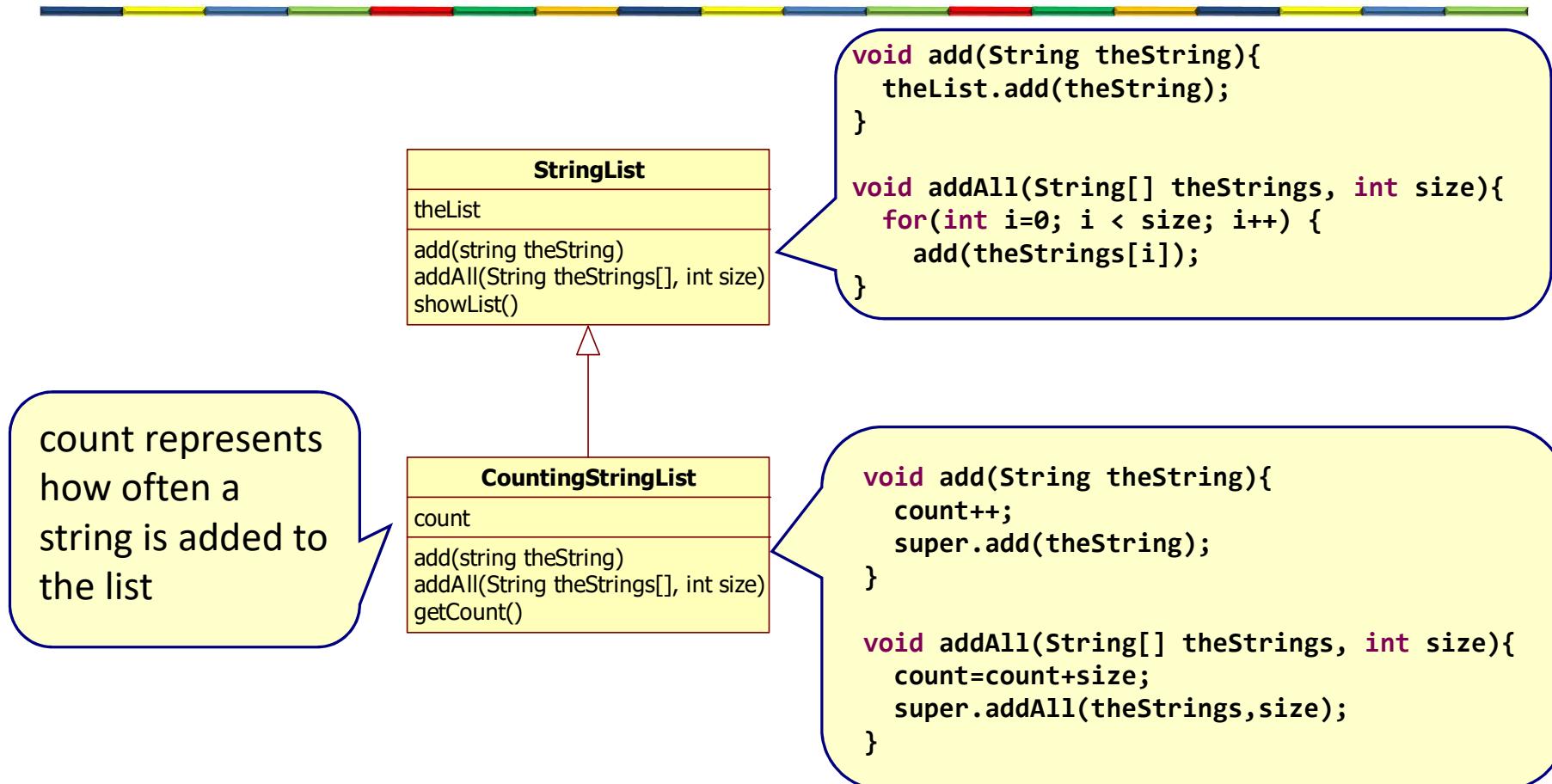
- Code flexibility
 - Classes that inherit from a common superclass can be used interchangeably (polymorphism)



Disadvantages of inheritance

- The base class and sub class are tightly coupled
 - Every change in the base class ripples down to the subclasses
 - Subclasses are entirely dependent on their superclass
 - If you write the subclass you need to understand the base class
 - Breaks encapsulation
- Multiple inheritance problem

Inheritance problem 1

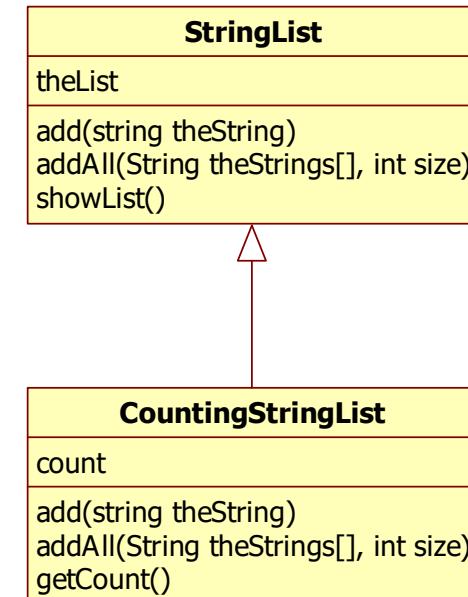


Inheritance problem 1

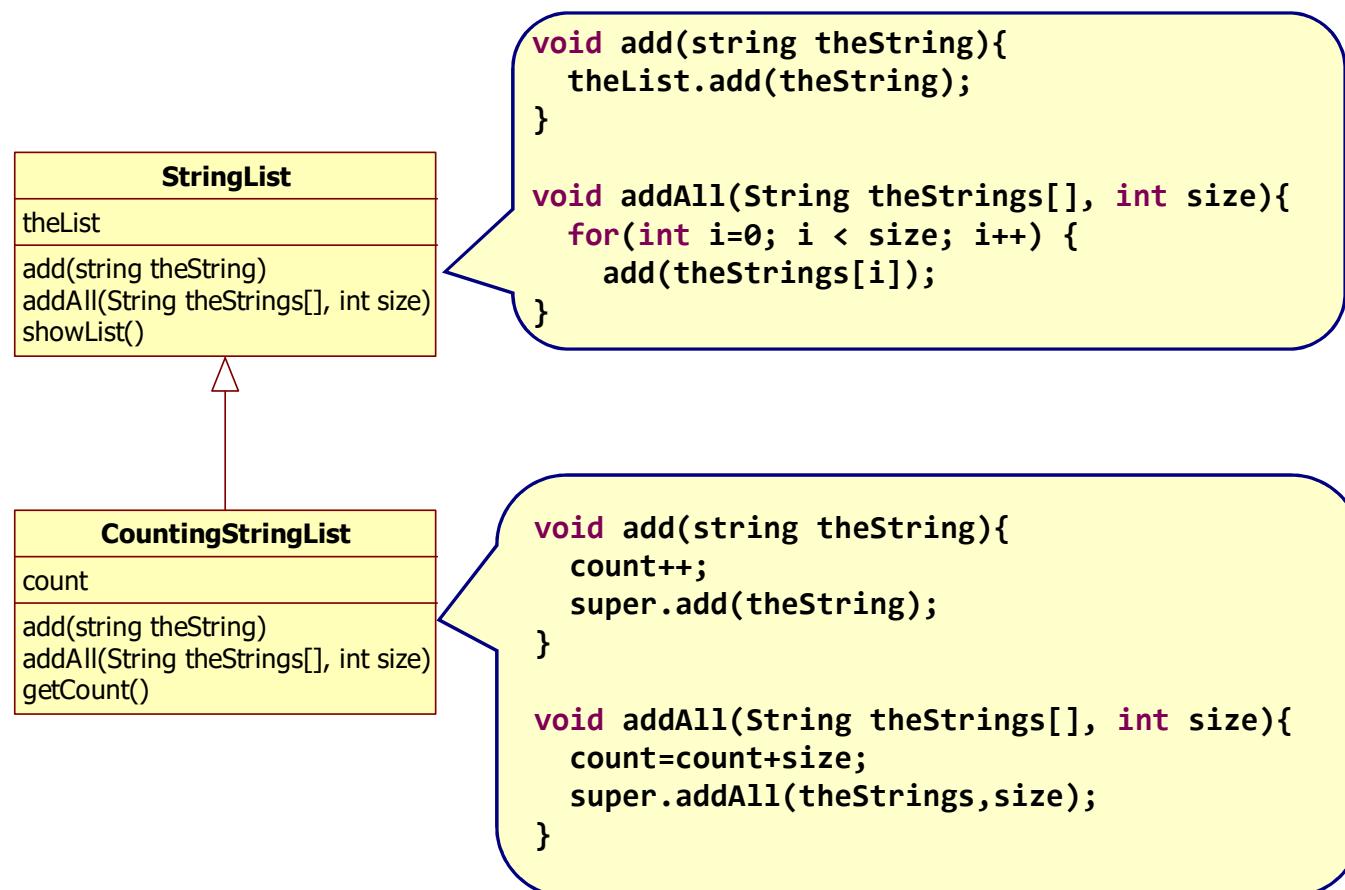
```
int main(...) {
    CountingStringList list = new CountingStringList();
    string s= "Mango";
    list.add(s);
    string s2= "Apple";
    list.add(s2);
    String[] fruit = {"Banana", "Orange"};
    list.addAll(fruit,2);
    list.showList();
    System.out.println("count="+list.getCount());
}
```

Mango
Apple
Banana
Orange
count=6

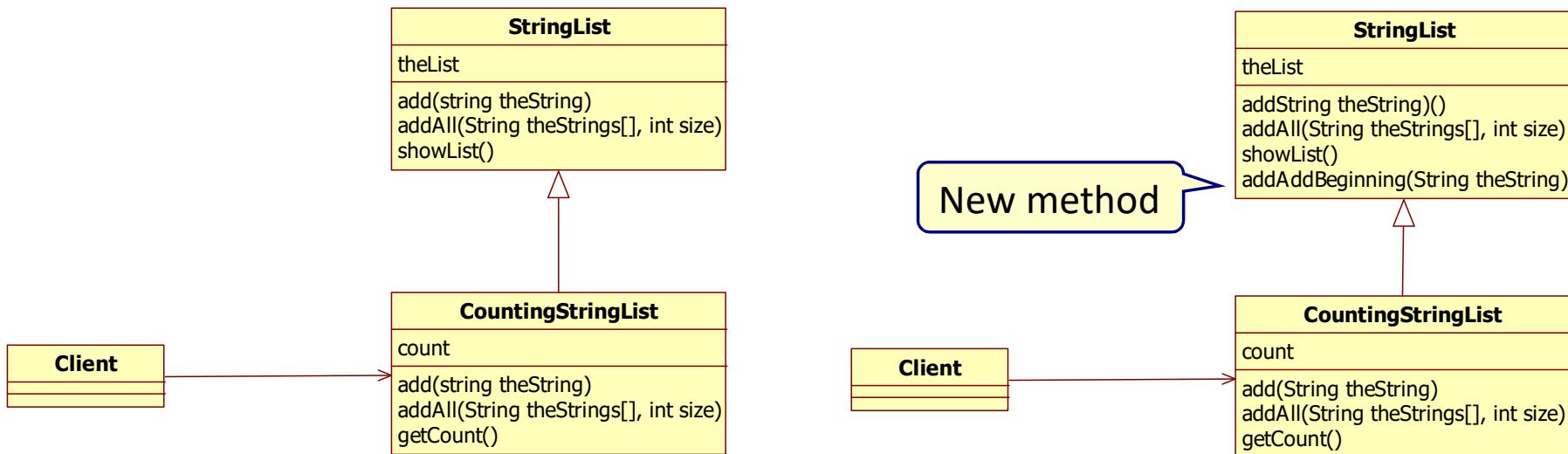
count=6, I expected 4



What is the problem?

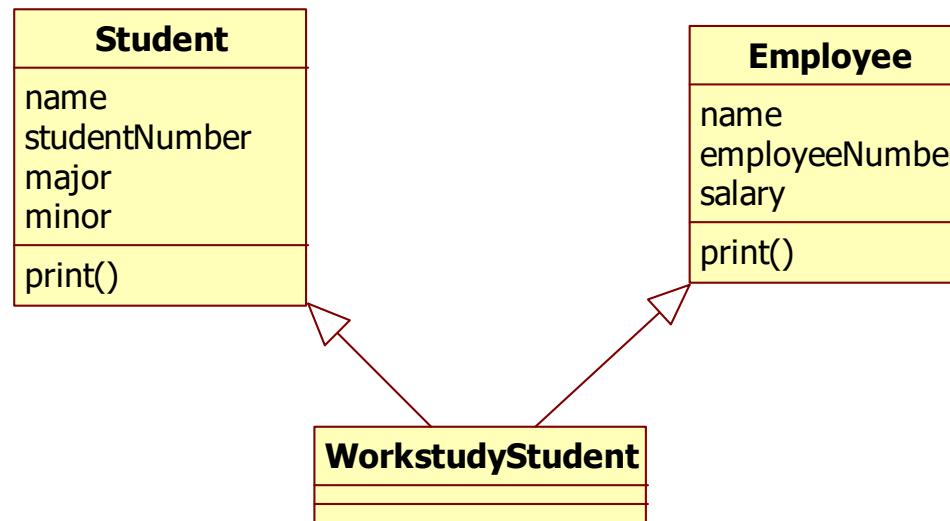


Inheritance problem 2



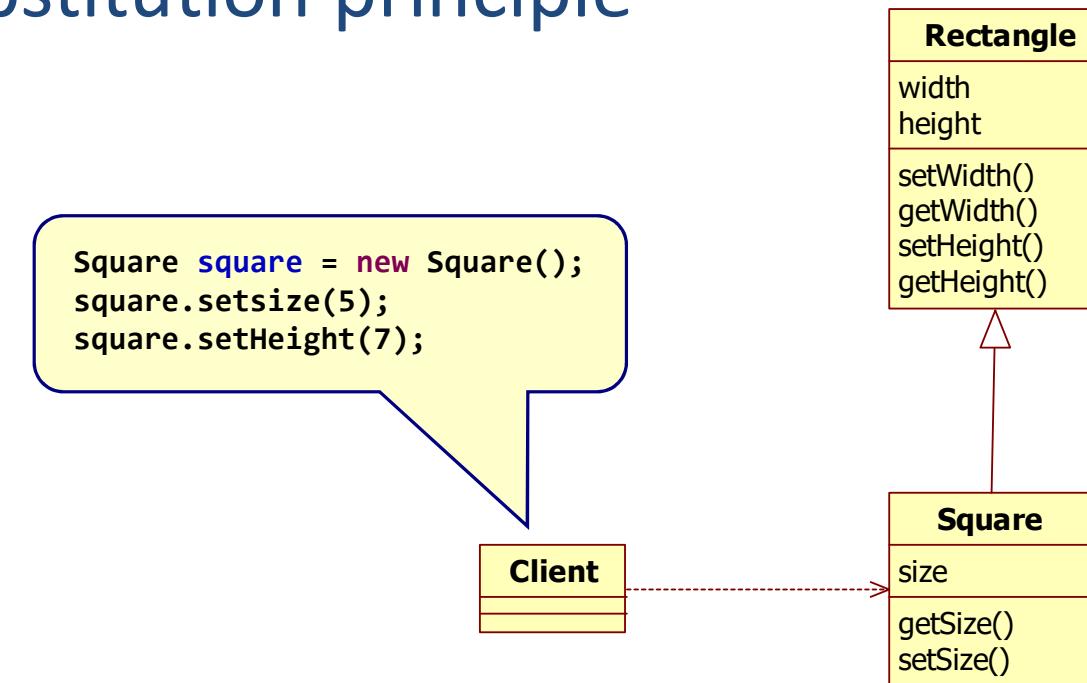
Inheritance problem 3

- Multiple inheritance is not allowed in Java



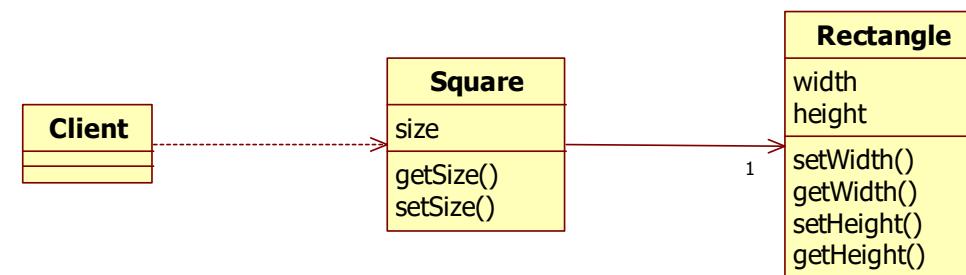
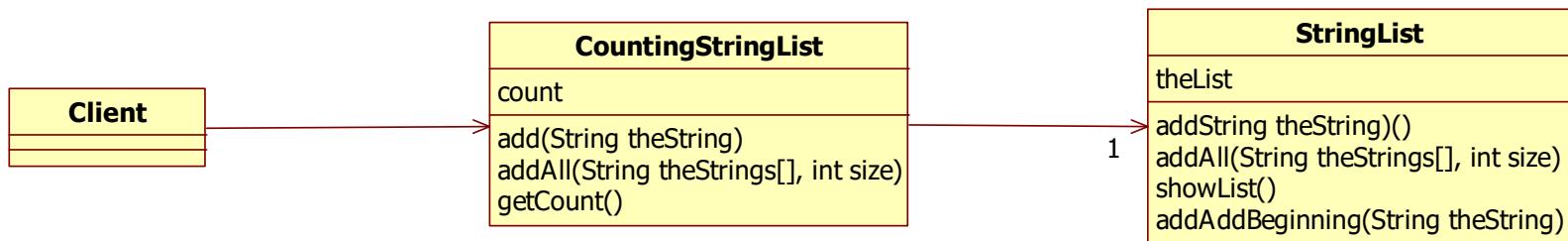
Inheritance problem 4

- Liskov substitution principle



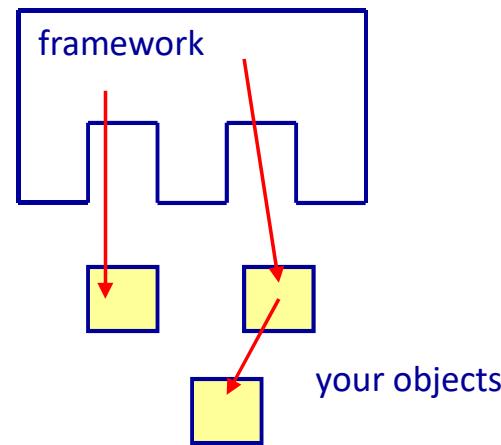
Solution to all inheritance problems

- Favor composition over inheritance



INVERSION OF CONTROL

Inversion of Control (IoC)

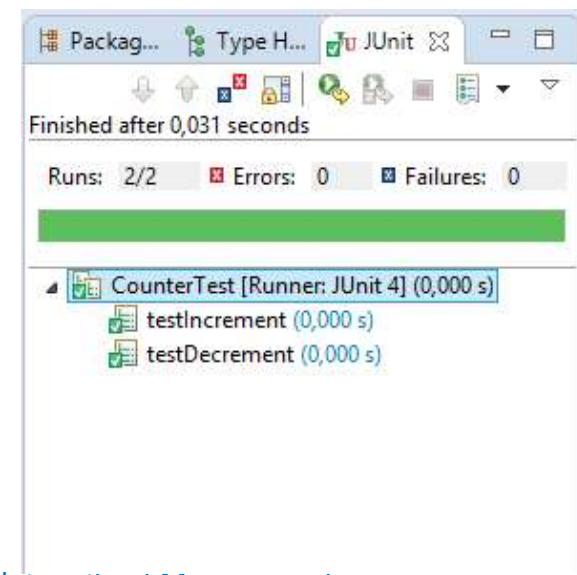


IoC: The framework calls your code

Example of JUnit framework

```
public class CounterTest {  
    @Test  
    public void testIncrement(){  
        Counter counter = new Counter();  
        assertEquals(1,counter.increment());  
        assertEquals(2,counter.increment());  
    }  
  
    @Test  
    public void testDecrement(){  
        Counter counter = new Counter();  
        assertEquals(-1,counter.decrement());  
        assertEquals(-2,counter.decrement());  
    }  
}
```

```
public class Counter {  
    private int counterValue=0;  
  
    public int increment(){  
        return ++counterValue;  
    }  
    public int decrement(){  
        return --counterValue;  
    }  
}
```

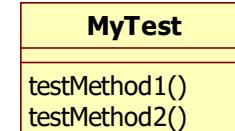
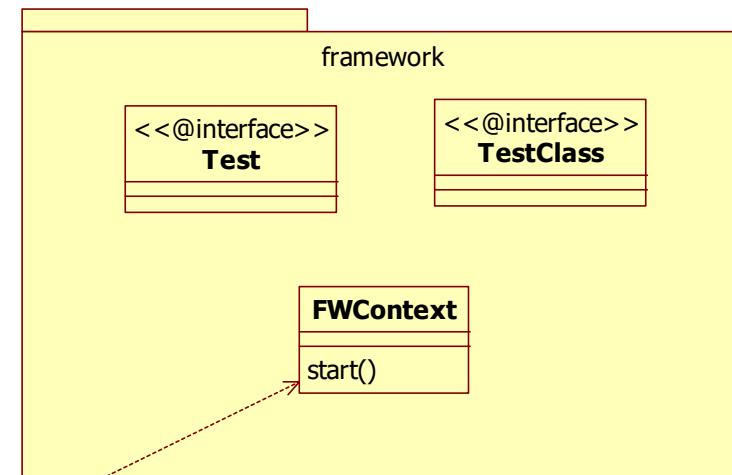


Inversion of Control framework

```
@TestClass
public class MyTest {

    @Test
    public void testMethod1() {
        System.out.println("perform test method 1");
    }

    @Test
    public void testMethod2() {
        System.out.println("perform test method 2");
    }
}
```

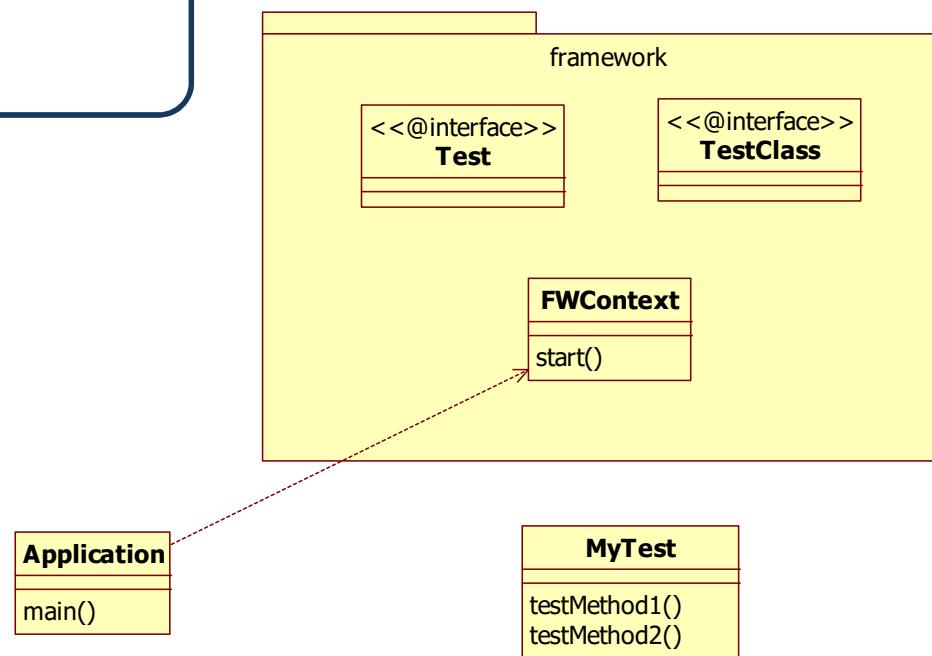


Inversion of Control framework

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test { }
```

Create your own annotations in Java

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface TestClass { }
```



Define your own annotations

Retention: defines the visibility of the annotation

- SOURCE—Annotation is visible only at the source level and will be ignored by the compiler.
- CLASS—Annotation is visible by the compiler at compile time, but will be ignored by the VM.
- RUNTIME—Annotation is visible by the VM so they can be read only at run-time.

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Test {
}
```

Target: where can I apply this annotation?

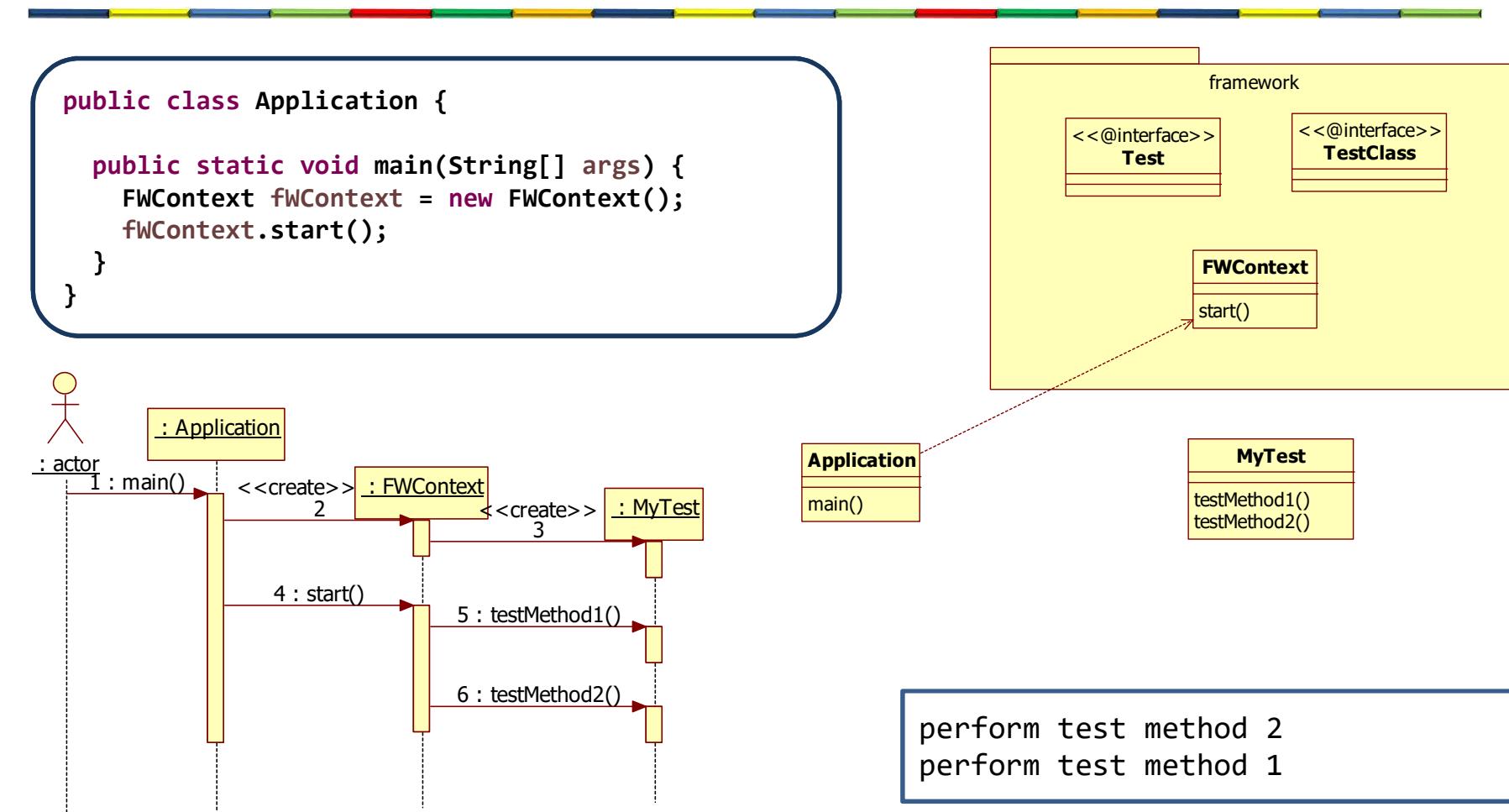
```
@Target(value={TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE})
```

Classpath scanning

```
public class FWContext {  
    private static List<Object> objectMap = new ArrayList<>();  
  
    public FWContext() {  
        try {  
            // find and instantiate all classes annotated with the @TestClass annotation  
            Reflections reflections = new Reflections("");  
            Set<Class<?>> types = reflections.getTypesAnnotatedWith(TestClass.class);  
            for (Class<?> implementationClass : types) {  
                objectMap.add((Object) implementationClass.newInstance());  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    public void start() {  
        try {  
            for (Object theTestClass : objectMap) {  
                // find all methods annotated with the @Test annotation  
                for (Method method : theTestClass.getClass().getDeclaredMethods()) {  
                    if (method.isAnnotationPresent(Test.class)) {  
                        method.invoke(theTestClass);  
                    }  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Classpath scanning

Inversion of Control framework



DEPENDENCY INJECTION

Instantiate an object directly

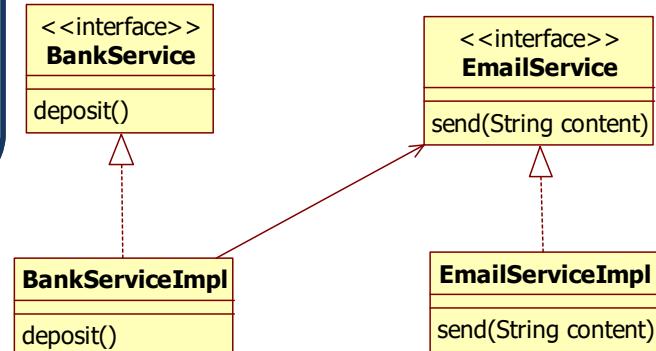
```
public interface BankService {  
    public void deposit();  
}
```

```
public class BankServiceImpl implements BankService {  
    private EmailService emailService = new EmailServiceImpl();  
  
    public void deposit() {  
        emailService.send("deposit");  
    }  
}
```

```
public interface EmailService {  
    void send(String content);  
}
```

```
public class EmailServiceImpl implements EmailService {  
  
    public void send(String content) {  
        System.out.println("sending email: "+content);  
    }  
}
```

Instantiate the EmailService

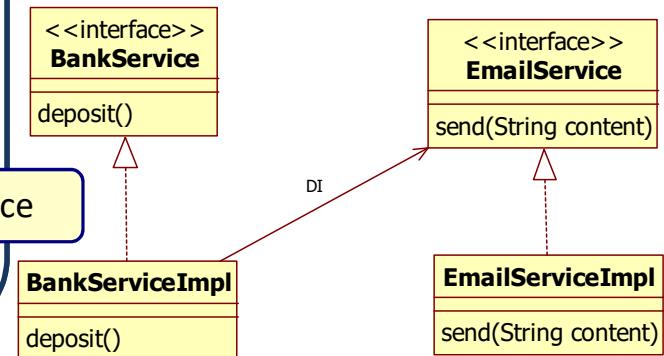


Dependency Injection

```
public interface BankService {  
    public void deposit();  
}
```

```
public class BankServiceImpl implements BankService {  
    private EmailService emailService;  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void deposit() {  
        emailService.send("deposit");  
    }  
}
```

We can inject any EmailService

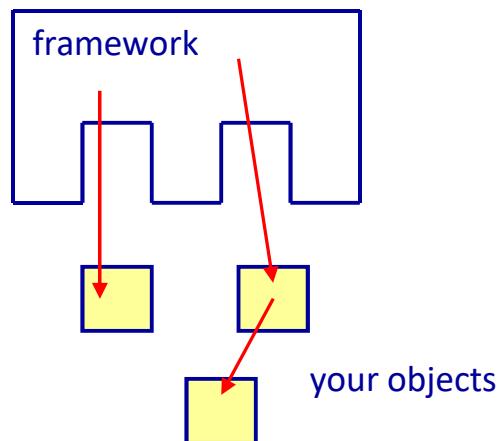


```
public interface EmailService {  
    void send(String content);  
}
```

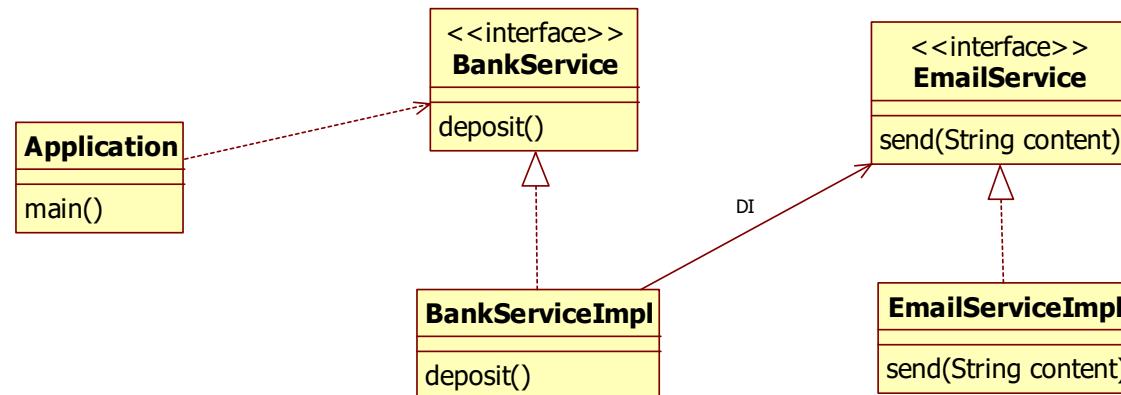
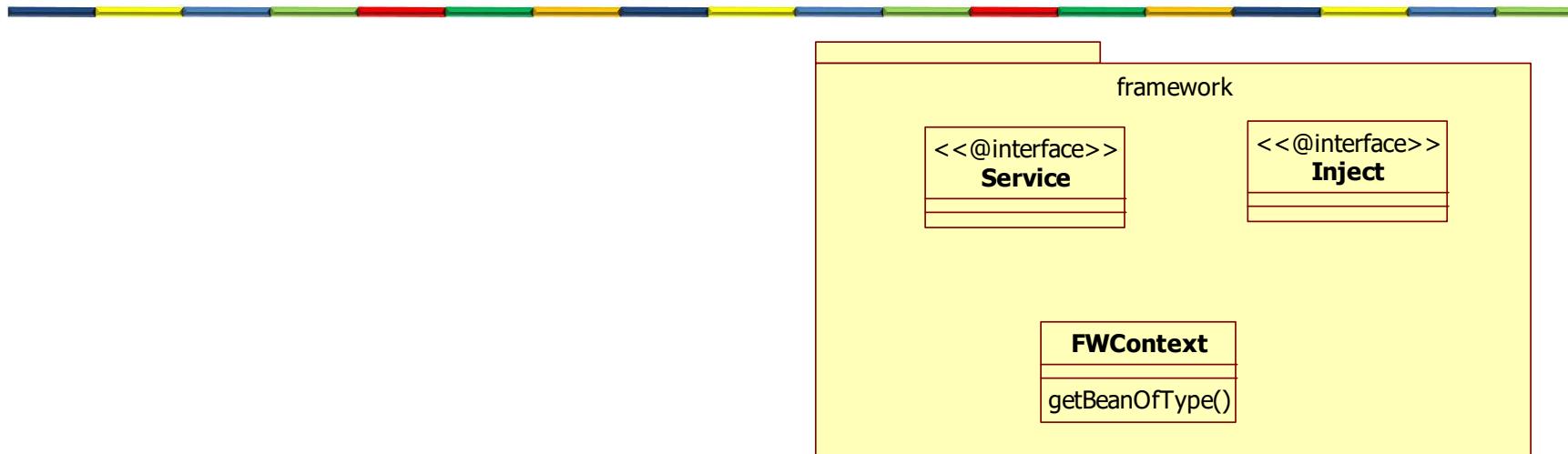
```
public class EmailServiceImpl implements EmailService {  
  
    public void send(String content) {  
        System.out.println("sending email: "+content);  
    }  
}
```

Framework implementation

- IoC: The framework instantiates our application classes
- Dependency injection: The framework wires our objects together



Dependency Injection framework



Dependency Injection framework

```
public interface BankService {  
    public void deposit();  
}
```

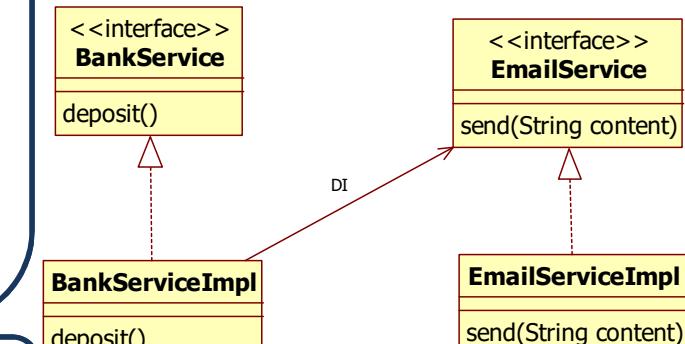
```
@Service  
public class BankServiceImpl implements BankService {  
    @Inject  
    private EmailService emailService;  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void deposit() {  
        emailService.send("deposit");  
    }  
}
```

```
public interface EmailService {  
    void send(String content);  
}
```

```
@Service  
public class EmailServiceImpl implements EmailService {  
  
    public void send(String content) {  
        System.out.println("sending email: "+content);  
    }  
}
```

The framework will instantiate this class

The framework will inject the EmailService

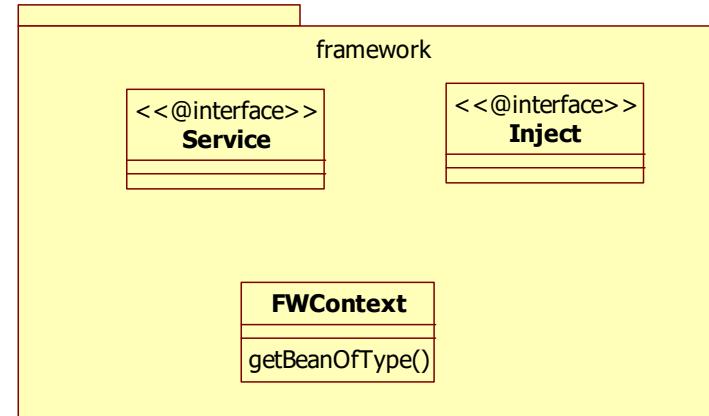


The framework will instantiate this class

Dependency Injection framework

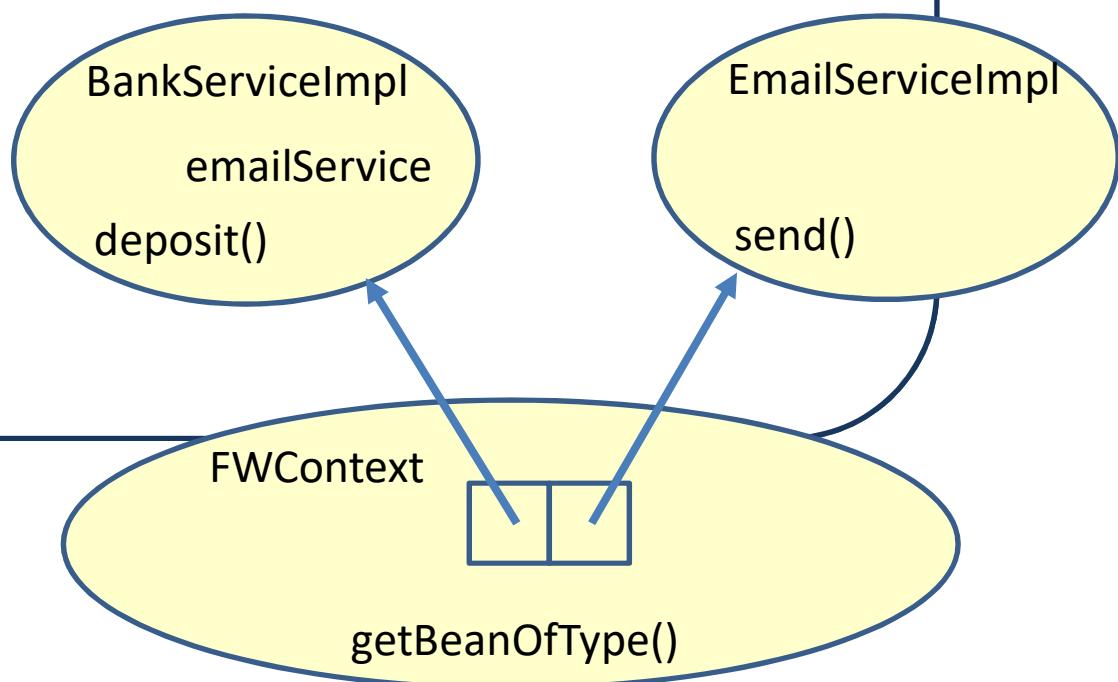
```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Service { }
```

```
@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject { }
```



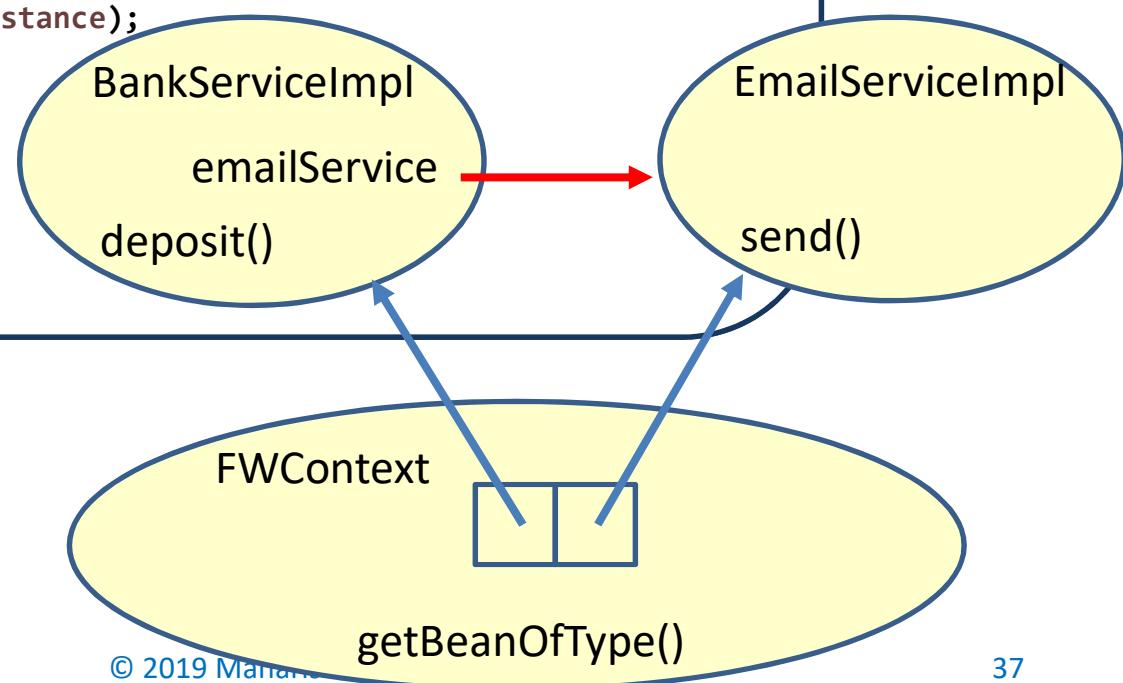
Context class

```
public class FWContext {  
  
    private static List<Object> objectMap = new ArrayList<>();  
  
    public FWContext() {  
        try {  
            // find and instantiate all classes annotated with the @Service annotation  
            Reflections reflections = new Reflections("");  
            Set<Class<?>> types = reflections.getTypesAnnotatedWith(Service.class);  
            for (Class<?> implementationClass : types) {  
                objectMap.add((Object) implementationClass.newInstance());  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        performDI();  
    }  
}
```



Context class

```
private void performDI() {  
    try {  
        for (Object theTestClass : objectMap) {  
            // find annotated fields  
            for (Field field : theTestClass.getClass().getDeclaredFields()) {  
                if (field.isAnnotationPresent(Inject.class)) {  
                    // get the type of the field  
                    Class<?> theFieldType = field.getType();  
                    //get the object instance of this type  
                    Object instance = getBeanOfType(theFieldType);  
                    //do the injection  
                    field.setAccessible(true);  
                    field.set(theTestClass, instance);  
                }  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

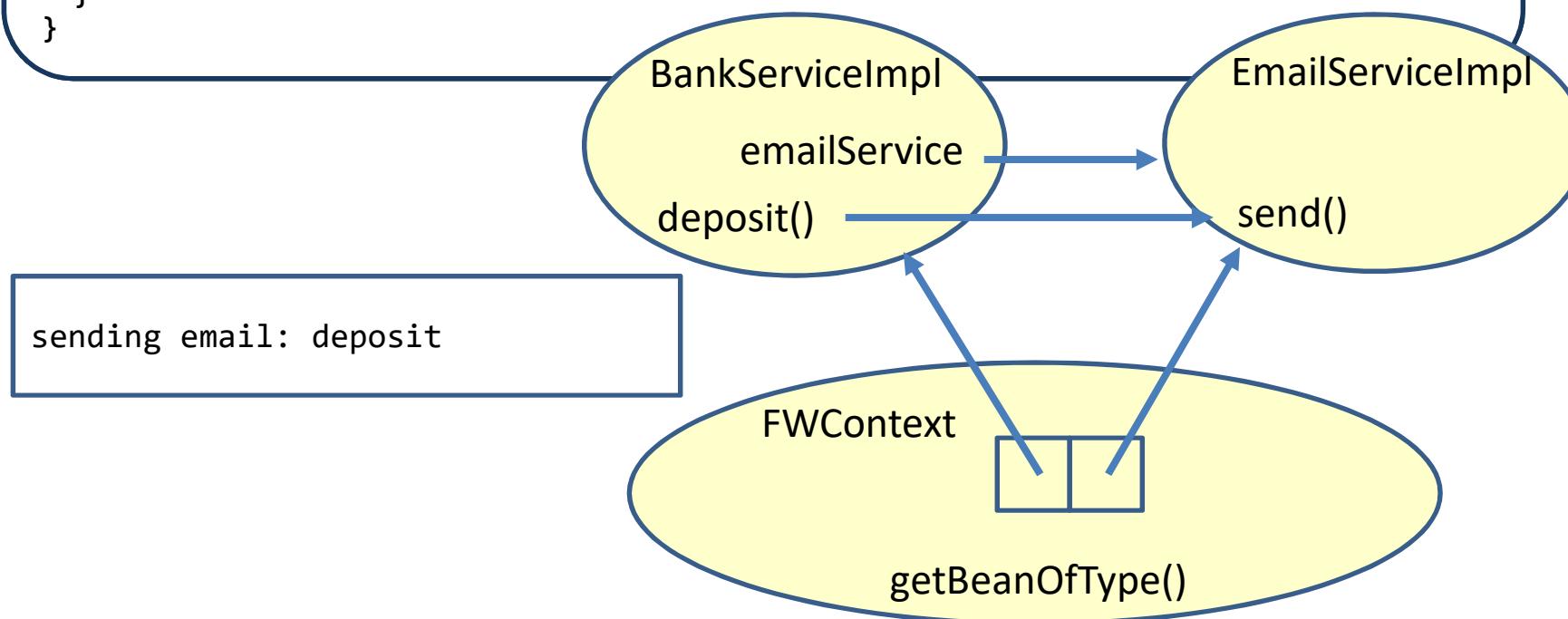


Context class

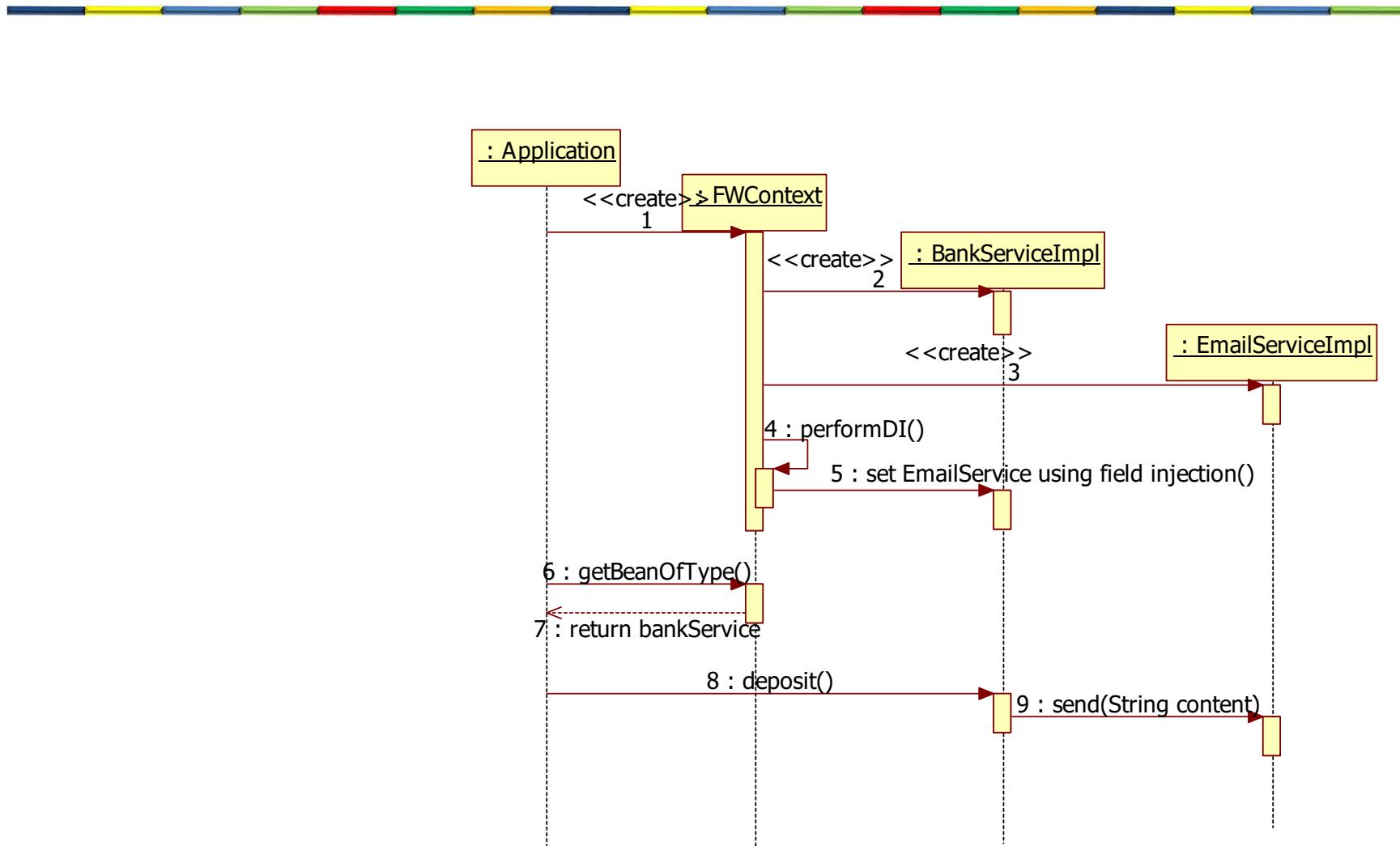
```
public Object getBeanOfType(Class interfaceClass) {  
    Object service = null;  
    try {  
        for (Object theTestClass : objectMap) {  
            Class<?>[] interfaces = theTestClass.getClass().getInterfaces();  
  
            for (Class<?> theInterface : interfaces) {  
                if (theInterface.getName().contentEquals(interfaceClass.getName()))  
                    service = theTestClass;  
            }  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
    return service;  
}
```

Dependency Injection framework

```
public class Application {  
  
    public static void main(String[] args) {  
        FWContext fwContext = new FWContext();  
  
        BankService bankService = (BankService) fwContext.getBeanOfType(BankService.class);  
        if (bankService != null)  
            bankService.deposit();  
    }  
}
```



Dependency Injection framework



Main point

- Dependency injection gives us flexibility in wiring objects together.
- Daily contact with pure consciousness results in more and more happiness by spontaneous right action.

HIDE THE CONTEXT

Make it simpler

```
public class Application {  
    public static void main(String[] args) {  
        FWContext fwContext = new FWContext();  
  
        BankService bankService = (BankService) fwContext.getBeanOfType(BankService.class);  
        if (bankService != null)  
            bankService.deposit();  
    }  
}
```

Let's hide the context
from the application code

Make it simpler

```
public class Application {  
  
    public static void main(String[] args) {  
        FWContext fwContext = new FWContext();  
  
        BankService bankService = (BankService) fwContext.getBeanOfType(BankService.class);  
        if (bankService != null)  
            bankService.deposit();  
    }  
}
```



```
public class Application implements Runnable{  
    @Inject  
    BankService bankService;  
  
    public static void main(String[] args) {  
        FWApplication.run(Application.class);  
    }  
  
    @Override  
    public void run() {  
        bankService.deposit();  
    }  
}
```

Inject the application class(es)

Create the context and perform DI

Start the application

FWApplication

```
public class FWApplication {

    public static void run(Class applicationClass) {
        // create the context
        FWContext fwContext = new FWContext();
        try {
            // create instance of the application class
            Object applicationObject = (Object) applicationClass.newInstance();
            // find annotated fields
            for (Field field : applicationObject.getClass().getDeclaredFields()) {
                if (field.isAnnotationPresent(Inject.class)) {
                    // get the type of the field
                    Class<?> theFieldType = field.getType();
                    // get the object instance of this type
                    Object instance = fwContext.getBeanOfType(theFieldType);
                    // do the injection
                    field.setAccessible(true);
                    field.set(applicationObject, instance);
                }
            }
            //call the run() method
            if (applicationObject instanceof Runnable)
                ((Runnable)applicationObject).run();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

INJECTION OF PRIMITIVE TYPES

EmailServiceImpl

```
@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject {
    String value() default "";
}
```

Add an attribute with name 'value' to the annotation

```
@Service
public class EmailServiceImpl implements EmailService{
    @Inject(value="message")
    String theMessage;

    public void send(String content) {
        System.out.println("sending email: "+content+" , message="+theMessage);
    }
}
```

Inject the message specified in the config.properties file

config.properties

```
message=Hello
bankname=First National Bank
```

BankServiceImpl

```
@Service
public class BankServiceImpl implements BankService{
    @Inject
    private EmailService emailService;

    @Inject(value="bankname")
    String bankName;

    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void deposit() {
        emailService.send("deposit to "+bankName);
    }
}
```

Inject the bankName specified
in the config.properties file

```
@Retention(RUNTIME)
@Target(FIELD)
public @interface Inject {
    String value() default "";
}
```

config.properties

```
message=Hello
bankname=First National Bank
```

ConfigFileReader

```
public class ConfigFileReader {  
  
    static Properties getConfigProperties() {  
        Properties prop = null;  
  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            prop = new Properties();  
            prop.load(new FileInputStream(rootPath + "/config.properties"));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return prop;  
    }  
}
```

config.properties

```
message=Hello  
bankname=First National Bank
```

FWContext

```
public class FWContext {  
  
    private static List<Object> objectMap = new ArrayList<>();  
  
    public FWContext() {  
        try {  
            // find and instantiate all classes annotated with the @Service annotation  
            Reflections reflections = new Reflections("");  
            Set<Class<?>> types = reflections.getTypesAnnotatedWith(Service.class);  
            for (Class<?> implementationClass : types) {  
                objectMap.add((Object) implementationClass.newInstance());  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        performReferenceDI();  
        performStringDI();  
    }  
    ...  
}
```

Inject all object references

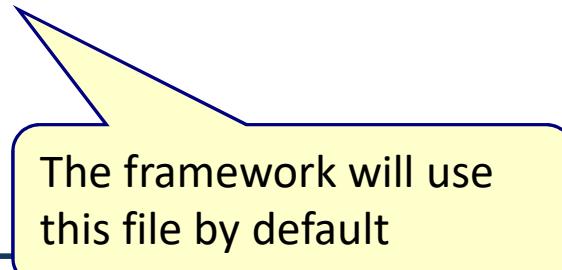
Inject all String attributes

FWContext

```
private void performStringDI() {
    Properties properties = ConfigFileReader.getConfigProperties();
    try {
        for (Object theTestClass : objectMap) {
            // find annotated fields
            for (Field field : theTestClass.getClass().getDeclaredFields()) {
                if (field.isAnnotationPresent(Inject.class)) {
                    // get the type of the field
                    Class<?> theFieldType = field.getType();
                    if (field.getType().getName().contentEquals("java.lang.String")) {
                        // get attribute value
                        String attrValue = field.getAnnotation(Inject.class).value();
                        // get the property value
                        String toBeInjectedString = properties.getProperty(attrValue);
                        // do the injection
                        field.setAccessible(true);
                        field.set(theTestClass, toBeInjectedString);
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Convention over configuration

```
public class ConfigFileReader {  
  
    static Properties getConfigProperties() {  
        Properties prop = null;  
  
        String rootPath = Thread.currentThread().getContextClassLoader().getResource("").getPath();  
        try {  
            prop = new Properties();  
            prop.load(new FileInputStream(rootPath + "/config.properties"));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
        return prop;  
    }  
}
```



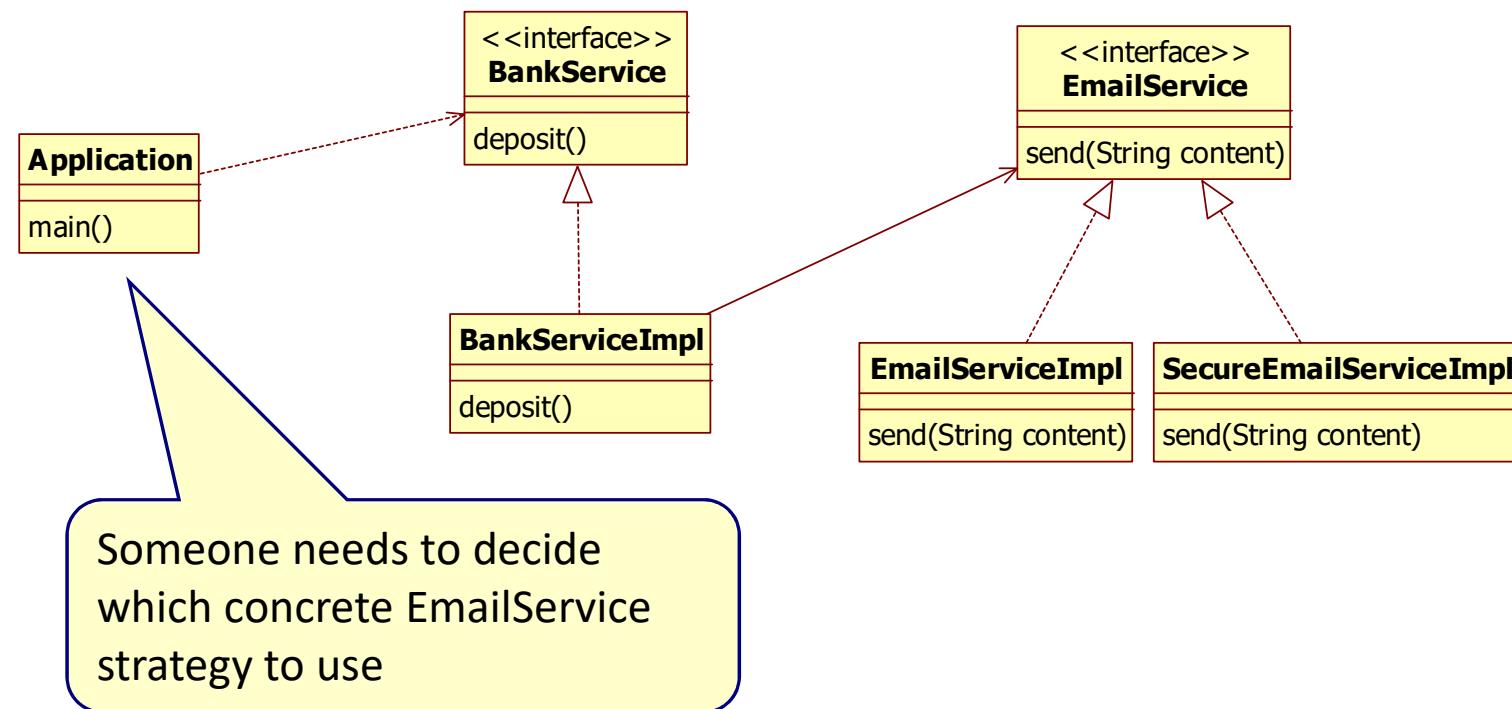
The framework will use this file by default

config.properties

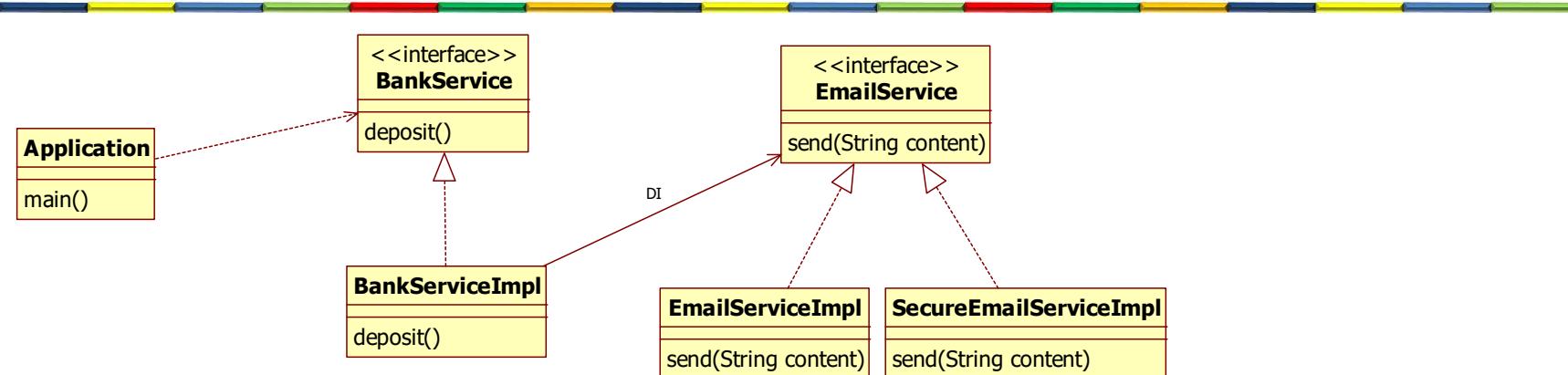
```
message=Hello  
bankname=First National Bank
```

PROFILES

Strategy pattern



Two classes that implement the same interface



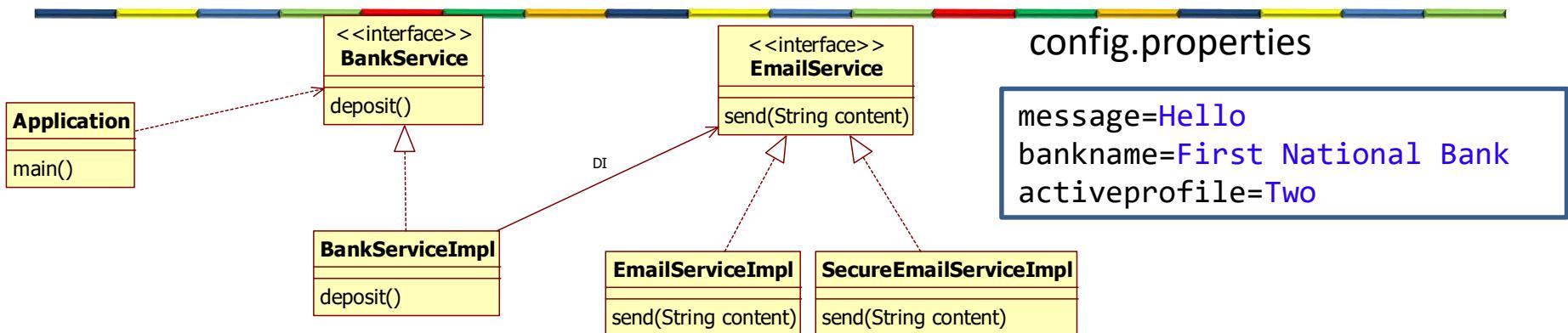
```
@Service
public class EmailServiceImpl implements EmailService{
    @Inject(value="message")
    String theMessage;

    public void send(String content) {
        System.out.println("sending email: "+content+" , message="+theMessage);
    }
}
```

```
@Service
public class SecureEmailServiceImpl implements EmailService{
    @Inject(value="message")
    String theMessage;

    public void send(String content) {
        System.out.println("sending secure email: "+content+" , message="+theMessage);
    }
}
```

Profiles



```
@Service
@Profile(value="One")
public class EmailServiceImpl implements EmailService{
    @Inject(value="message")
    String theMessage;

    public void send(String content) {
        System.out.println("sending email: "+content+ " , message=" +theMessage);
    }
}

@Service
@Profile(value="Two")
public class SecureEmailServiceImpl implements EmailService{
    @Inject(value="message")
    String theMessage;

    public void send(String content) {
        System.out.println("sending secure email: "+content+ " , message=" +theMessage);
    }
}
```

Working with profiles

```
public class FWContext {  
  
    private static List<Object> objectMap = new ArrayList<>();  
    Properties properties;  
    String activeProfile;  
  
    public FWContext() {  
        try {  
            properties = ConfigFileReader.getConfigProperties();  
            activeProfile= properties.getProperty("activeprofile");  
            ...  
        }  
  
        public Object getBeanOfType(Class interfaceClass) {  
            ...  
        }  
    }  
}
```

Get the activeprofile from the configuration files

Working with profiles

```
public Object getBeanOfType(Class interfaceClass) {
    List<Object> objectList = new ArrayList<Object>();
    try {
        for (Object theTestClass : objectMap) {
            Class<?>[] interfaces = theTestClass.getClass().getInterfaces();

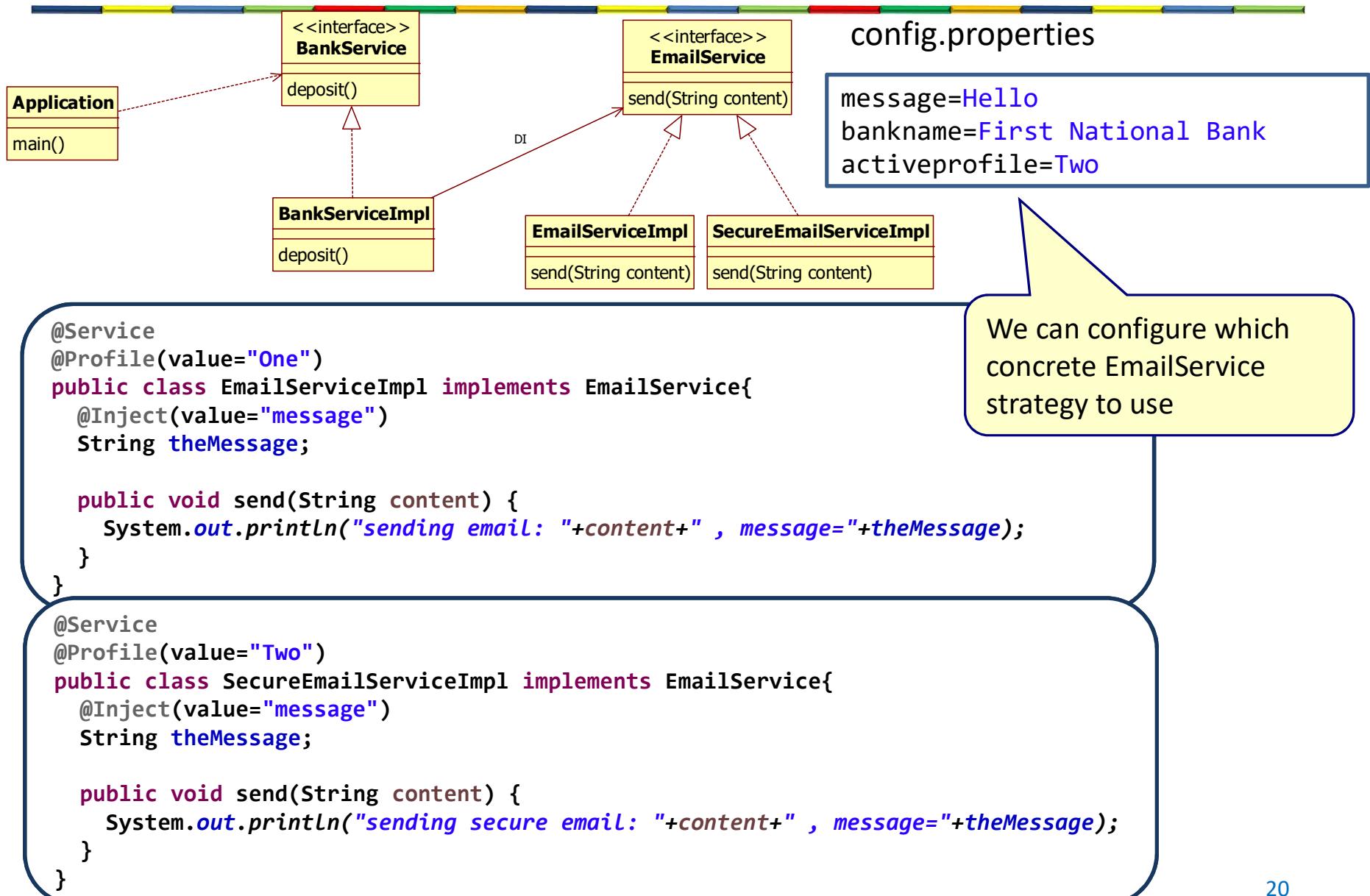
            for (Class<?> theInterface : interfaces) {
                if (theInterface.getName().contentEquals(interfaceClass.getName()))
                    objectList.add(theTestClass);
            }
        }
    } catch (Exception e) {
        e.printStackTrace();
    }

    if (objectList.size() < 1) return null;
    if (objectList.size() == 1) return objectList.get(0);
    if (objectList.size() > 1) {
        for (Object theObject : objectList) {
            String profilevalue = theObject.getClass().getAnnotation(Profile.class).value();
            if (profilevalue.contentEquals(activeProfile)) {
                return theObject;
            }
        }
    }
    return null;
}
```

Get all classes that implement the provided interface

If multiple classes implement the provided interface, find the class with the activeprofile

Framework with profiles



Main point

- Frameworks make heavily use of:
 - Inversion of Control
 - Classpath scanning
 - Dependency injection
 - Convention over configuration
- The Unified Field contains all the laws of nature.

Connecting the parts of knowledge with the wholeness of knowledge

1. Frameworks often use dependency injection to wire objects together
2. Dependency injection together with profiles gives us the open-closed principle

-
- **Transcendental consciousness** is the never changing field at the basis of all evolution.
 - **Wholeness moving within itself:** In unity consciousness one realizes that the perfect underlying structure of the entire creation is just the same structure of one's own pure consciousness.



Lesson 12 Spring framework

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Aim of the Spring framework

- Make enterprise Java application development as **easy** as possible following good programming practices
- Has support for
 - Data access
 - Remoting
 - Scheduling
 - ...

A basic Spring application

```
package basic;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new
            ClassPathXmlApplicationContext("springconfig.xml");
        CustomerService customerService = context.getBean("customerService", CustomerService.class);
        customerService.sayHello();
    }
}
```

```
package basic;

public class CustomerService {
    public void sayHello(){
        System.out.println("Hello from CustomerService");
    }
}
```

Create an
ApplicationContext
based on
springconfig.xml

Get the bean with
id="customerService"
from the
ApplicationContext

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="customerService" class="basic.CustomerService" />
</beans>
```

springconfig.xml

Bean declaration

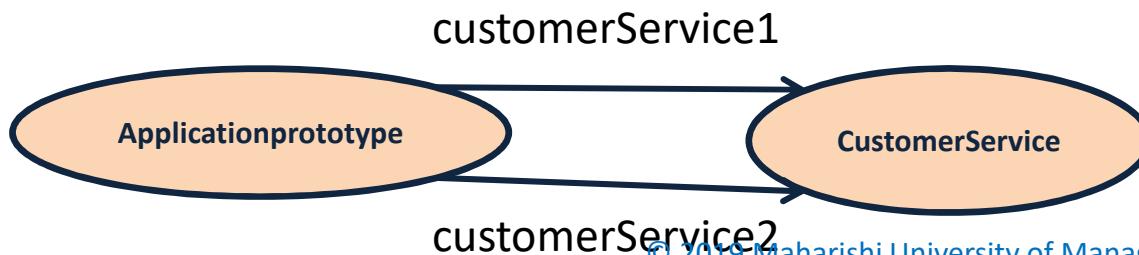
Spring beans are default singletons

```
public class Application{  
    public static void main(String[] args) {  
        ApplicationContext context = new  
            ClassPathXmlApplicationContext("module2/singleton/springconfig.xml");  
        CustomerService customerService1 = context.getBean("customerService", CustomerService.class);  
        CustomerService customerService2 = context.getBean("customerService", CustomerService.class);  
        System.out.println("customerService1 ="+ customerService1);  
        System.out.println("customerService2 ="+ customerService2);  
    }  
}
```

```
public class CustomerService {  
    public CustomerService() {  
    }  
}
```

```
<bean id="customerService" class="module2.singleton.CustomerService" />
```

```
customerService1 =module2.singleton.CustomerService@29e357  
customerService2 =module2.singleton.CustomerService@29e357
```



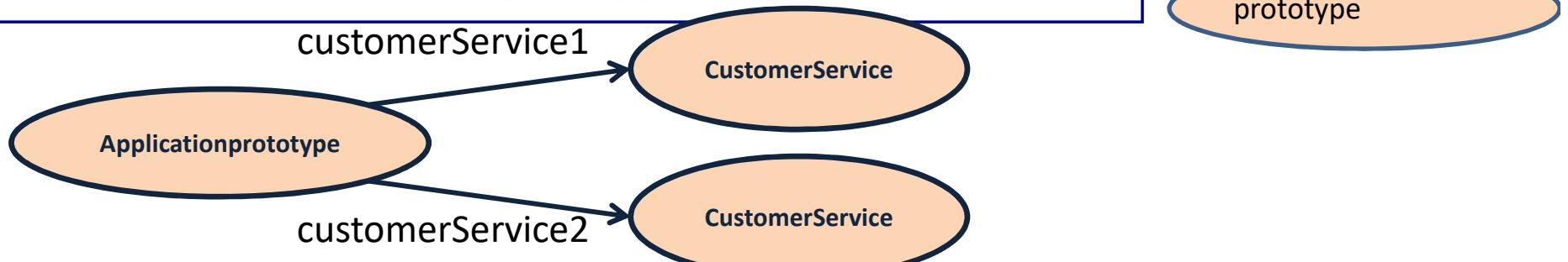
Prototype beans

```
public class Application{
    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("module2/prototype/springconfig.xml");
        CustomerService customerService1 = context.getBean("customerService", CustomerService.class);
        CustomerService customerService2 = context.getBean("customerService", CustomerService.class);
        System.out.println("customerService1 =" + customerService1);
        System.out.println("customerService2 =" + customerService2);
    }
}
```

```
public class CustomerService {
    public CustomerService() {
    }
}
```

```
<bean id="customerService" class="module2.prototype.CustomerService" scope="prototype" />
```

```
customerService1 =module2.prototype.CustomerService@1632847
customerService2 =module2.prototype.CustomerService@e95a56
```



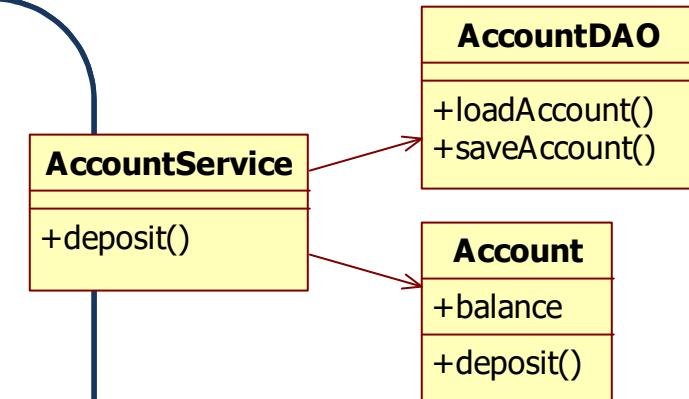
DEPENDENCY INJECTION

Different way's to “wire” 2 object together

1. Instantiate an object directly
2. Use an interface
3. Use a factory class
4. Use Spring Dependency Injection

1. Instantiate an object directly

```
public class AccountService {  
    private AccountDAO accountDAO;  
  
    public AccountService() {  
        accountDAO = new AccountDAO();  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account=accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

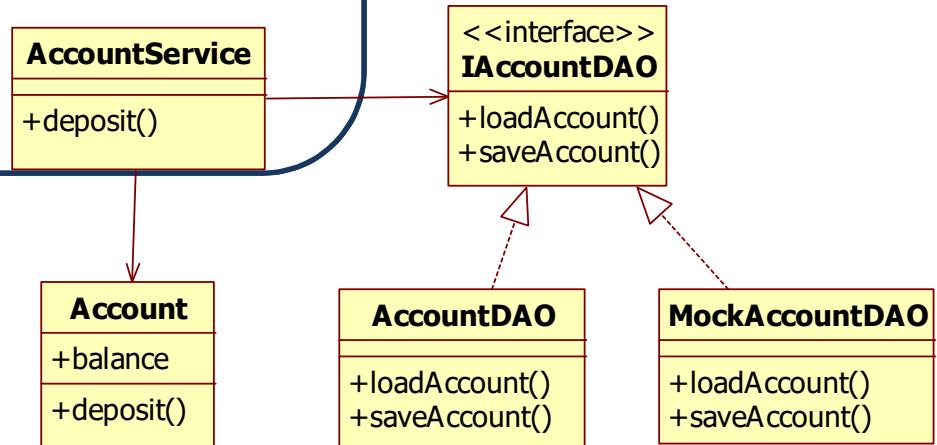


- The relation between `AccountService` and `AccountDAO` is hard coded
 - If you want to change the `AccountDAO` implementation, you have to change the code

2. Use an Interface

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService() {  
        accountDAO = new AccountDAO();  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account=accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

accountDAO is of type
IAccountDAO

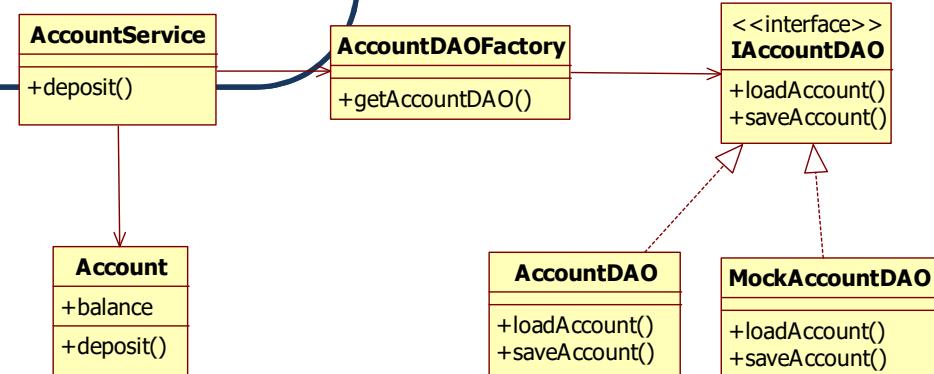


- The relation between **AccountService** and **AccountDAO** is still hard-coded
 - We have more flexibility, but if you want to change the **AccountDAO** implementation to the **MockAccountDAO**, you have to change the code

3. Use a factory class

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService() {  
        accountDAO = AccountDAOFactory.getAccountDAO();  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account=accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

The factory creates
The accountDAO object



- The relation between AccountService and AccountDAO is still hard coded
 - We have more flexibility, but if you want to change the AccountDAO implementation to the MockAccountDAO, you have to change code in the factory

4. Use Spring Dependency Injection

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public void setAccountDAO(IAccountDAO accountDAO) {  
        this.accountDAO = accountDAO;  
    }  
  
    public void deposit(long accountNumber, double amount){  
        Account account=accountDAO.loadAccount(accountNumber);  
        account.deposit(amount);  
        accountDAO.saveAccount(account);  
    }  
}
```

accountDAO is injected by the Spring framework

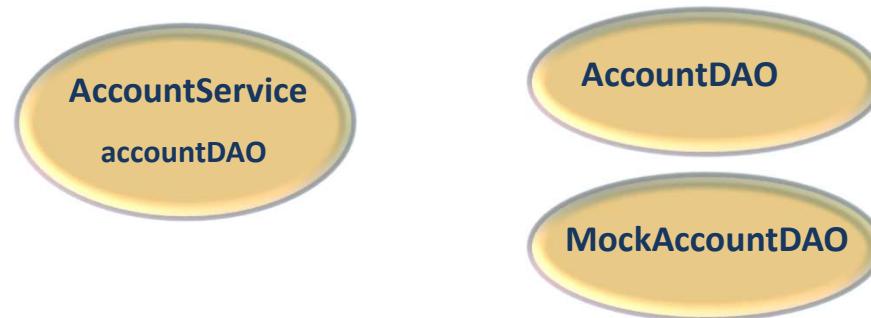
```
<bean id="accountService" class="AccountService">  
    <property name="accountDAO" ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />  
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

- The attribute accountDAO is configured in XML and the Spring framework takes care that accountDAO references the AccountDAO object.

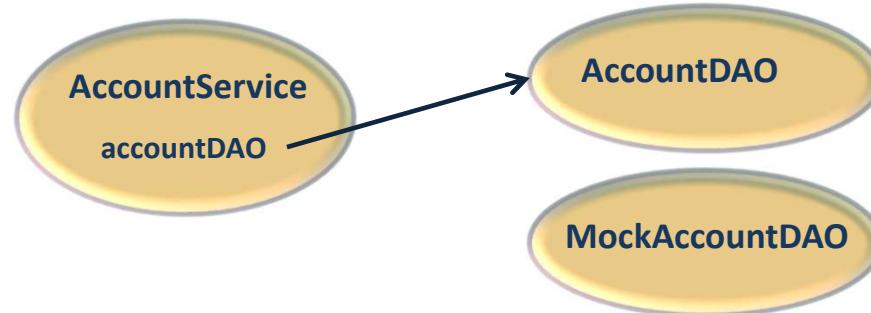
How does DI work?

```
<bean id="accountService" class="AccountService">
    <property name="accountDAO" ref="accountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

1. Spring instantiates all beans in the XML configuration file



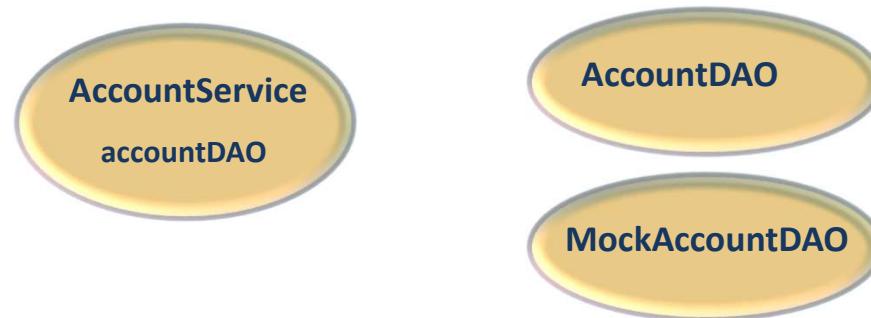
2. Spring then connects the accountDAO attribute to the AccountDAO instance



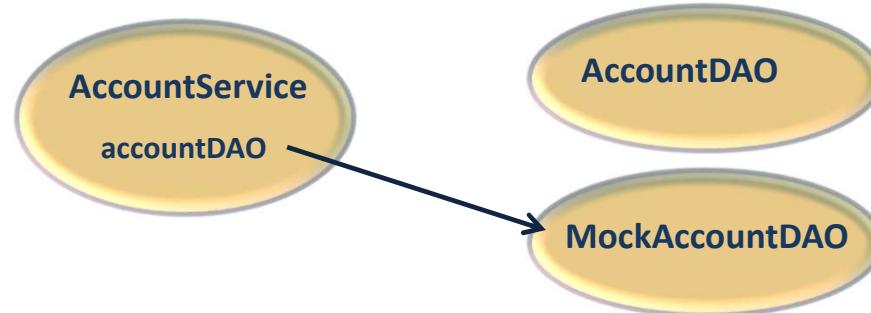
Change the wiring

```
<bean id="accountService" class="AccountService">
  <property name="accountDAO" ref="mockAccountDAO" />
</bean>
<bean id="accountDAO" class="AccountDAO" />
<bean id="mockAccountDAO" class="MockAccountDAO" />
```

1. Spring instantiates all beans in the XML configuration file



2. Spring then connects the accountDAO attribute to the MockAccountDAO instance



Advantages of Dependency Injection

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public void setAccountDAO(IAccountDAO accountDAO) {  
        this.accountDAO = accountDAO;  
    }  
}
```

```
<bean id="accountService" class="AccountService">  
    <property name="accountDAO" ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />
```

- Flexibility: it is easy to change the wiring between objects without changing code
- Unit testing becomes easier
- Code is clean

DIFFERENT TYPES OF DI

Types of DI

- Setter injection
- Constructor injection
- Autowiring

Setter Injection

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public void setAccountDAO(IAccountDAO accountDAO) {  
        this.accountDAO = accountDAO;  
    }  
}
```

A setter method
is needed

```
<bean id="accountService" class="AccountService">  
    <property name="accountDAO" ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />
```

Use <property .../>

Constructor Injection

```
public class AccountService {  
    private IAccountDAO accountDAO;  
  
    public AccountService(IAccountDAO accountDAO) {  
        this.accountDAO = accountDAO;  
    }  
}
```

A constructor
is needed to set
accountDAO

```
<bean id="accountService" class="AccountService">  
    <constructor-arg ref="accountDAO" />  
</bean>  
<bean id="accountDAO" class="AccountDAO" />
```

Use <constructor-arg .../>

Autowiring

- Spring figures out how to wire beans together
- 3 types of autowiring
 - By Name
 - By Type
 - Constructor

Autowiring by name

```
public class CustomerService {  
    private EmailService emailService;  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

```
public class EmailService {  
  
    public void sendEmail(){  
        System.out.println("sendEmail");  
    }  
}
```

Autowire by name uses setter injection, so we need a setter method

Spring will inject the bean with id="emailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byName" />  
<bean id="emailService" class="mypackage.EmailService"/>
```

Autowiring by type

```
public class CustomerService {  
    private EmailService emailService;  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
}
```

```
public class EmailService {  
  
    public void sendEmail(){  
        System.out.println("sendEmail");  
    }  
}
```

Autowire by type uses setter injection, so we need a setter method

Spring will inject the bean with type EmailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="byType" />  
<bean id="eService" class="mypackage.EmailService"/>
```

Constructor autowiring

```
public class CustomerService {  
    private EmailService emailService;  
  
    public CustomerService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

The constructor has 1 attribute of type EmailService

```
public class EmailService {  
  
    public void sendEmail(){  
        System.out.println("sendEmail");  
    }  
}
```

Spring will inject the bean with type EmailService" into the attribute 'emailService'

```
<bean id="customerService" class="mypackage.CustomerService" autowire="constructor"/>  
<bean id="eService" class="mypackage.EmailService"/>
```

Annotation based Autowiring by constructor

```
public class CustomerService {  
    private EmailService emailService;  
  
    @Autowired  
    public CustomerService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

@Autowire indicates to Spring that the emailService attribute should be injected by type via the constructor

```
public class EmailService {  
  
    public void sendEmail(){  
        System.out.println("sendEmail");  
    }  
}
```

This tag tells Spring to look for configuration annotations in the declared beans

```
<context:annotation-config>  
<bean id="customerService" class="mypackage.CustomerService"/>  
<bean id="eService" class="mypackage.EmailService"/>
```

Annotation based Autowiring by type

```
public class CustomerService {  
    private EmailService emailService;  
  
    @Autowired  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

@Autowire indicates to Spring that the emailService attribute should be injected by type via the setter method

```
public class EmailService {  
  
    public void sendEmail(){  
        System.out.println("sendEmail");  
    }  
}
```

This tag tells Spring to look for configuration annotations in the declared beans

```
<context:annotation-config>  
<bean id="customerService" class="mypackage.CustomerService"/>  
<bean id="eService" class="mypackage.EmailService"/>
```

Field injection

```
public class CustomerService {  
    @Autowired  
    @Qualifier("myEmailService")  
    private EmailService emailService;  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

autowire by name

```
public class EmailService {  
  
    public void sendEmail(){  
        System.out.println("sendEmail");  
    }  
}
```

```
<context:annotation-config>  
<bean id="customerService" class="mypackage.CustomerService"/>  
<bean id="myEmailService" class="mypackage.EmailService"/>
```

Field injection

```
public class CustomerService {  
    @Autowired  
    private EmailService emailService;  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

autowire by type

```
public class CustomerService {  
    @Inject  
    private EmailService emailService;  
  
    public void addCustomer(){  
        emailService.sendEmail();  
    }  
}
```

Injection of primitive values

```
public class CustomerServiceImpl implements CustomerService {  
    private String defaultCountry;  
    private long numberOfCustomers;  
  
    public void setDefaultCountry(String defaultCountry) {  
        this.defaultCountry = defaultCountry;  
    }  
    public String getDefaultCountry() {  
        return defaultCountry;  
    }  
    public long getNumberOfCustomers() {  
        return numberOfCustomers;  
    }  
    public void setNumberOfCustomers(long numberOfCustomers) {  
        this.numberOfCustomers = numberOfCustomers;  
    }  
}
```

```
<bean id="customerService" class="mypackage.CustomerServiceImpl">  
    <property name="defaultCountry" value="USA"/>  
    <property name="numberOfCustomers" value="56982"/>  
</bean>
```

Automatic conversion from
String to long

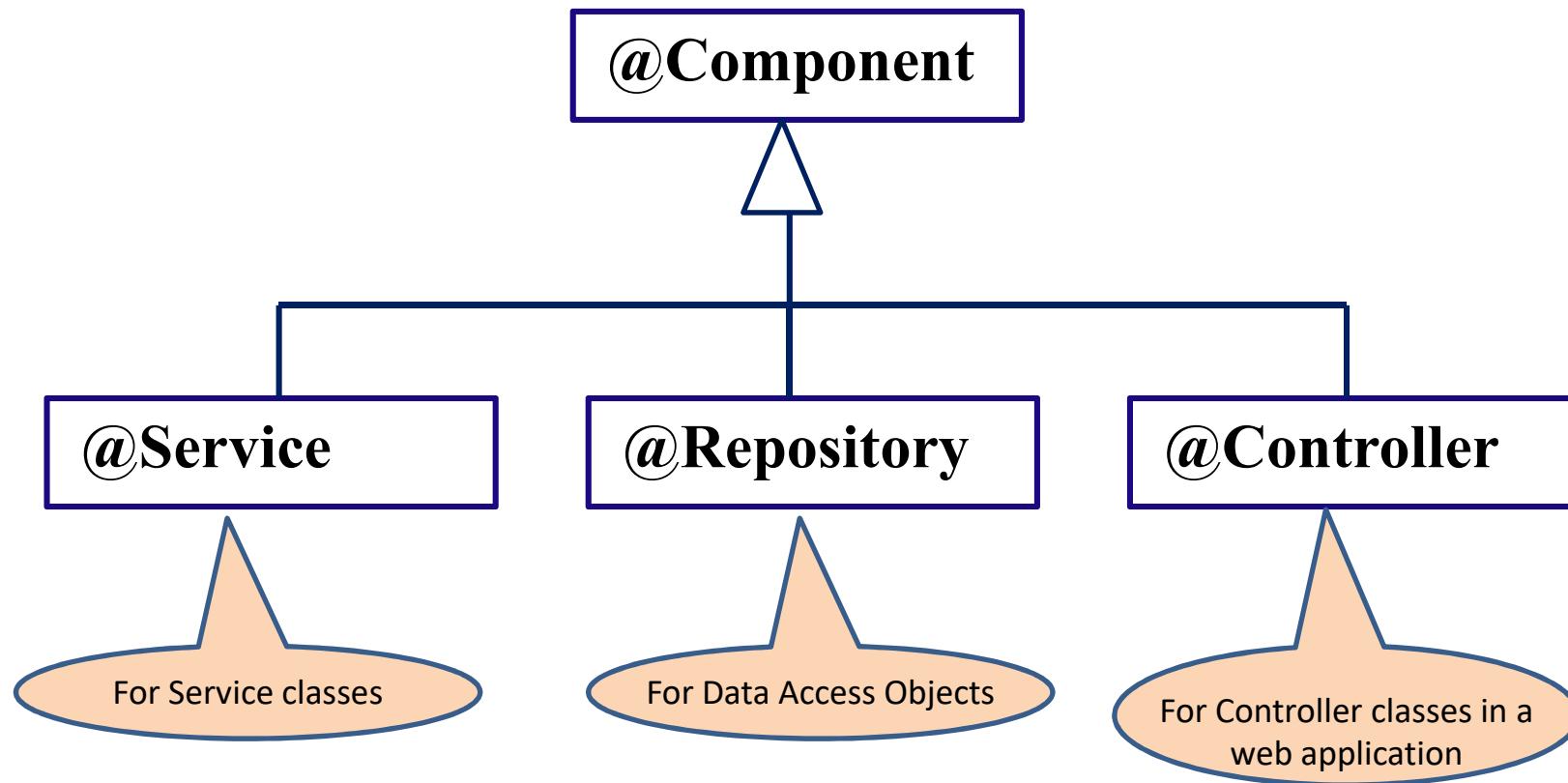
DEPENDENCY INJECTION WITH CLASSPATH SCANNING

Classpath scanning

- Define beans with annotations instead of defining them with XML
- All classes with the annotations
 - `@Component`
 - `@Service`
 - `@Repository`
 - `@Controller`

become spring beans

Classpath scanning annotations



Classpath scanning example (1/2)

```
@Service ("customerService")
public class CustomerServiceImpl implements CustomerService{
    private EmailService emailService;

    @Autowired
    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void addCustomer() {
        emailService.sendEmail();
    }
}
```

@Service annotation

The EmailService is injected

```
@Service ("emailService")
public class EmailService implements IEmailService {

    public void sendEmail() {
        System.out.println("sendEmail");
    }
}
```

@Service annotation

Classpath scanning example (2/2)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <context:component-scan base-package="module3.classpathscanning.basic"/>
    <context:annotation-config />
</beans>
```

No beans declared

```
public class Application {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("springconfig.xml");
        CustomerService customerService = context.getBean("customerService", CustomerService.class);
        customerService.addCustomer();
    }
}
```

@Value

```
@Service ("emailService")
public class EmailServiceImpl implements EmailService{
    @Value("smtp.mailserver.com")
    private String emailServer;

    public void sendEmail() {
        System.out.println("send email to server: "+ emailServer);
    }
}
```

Set the Value of an attribute

DEPENDENCY INJECTION WITH JAVA CONFIGURATION

Java Configuration

- Spring beans can also be configured in Java (and annotations) instead of XML

```
@Configuration
public class AppConfig {
    @Bean
    public CustomerService customerService() {
        CustomerService customerService = new CustomerServiceImpl();
        customerService.setEmailService(emailService());
        return customerService;
    }
    @Bean
    public EmailService emailService() {
        return new EmailServiceImpl();
    }
}
```

Similar configuration

```
<bean id="customerService" class="module3.di.CustomerServiceImpl">
    <property name="emailService" ref="emailService"/>
</bean>
<bean id="emailService" class="module3.di.EmailServiceImpl" />
```

Java configuration example (1/2)

```
public class CustomerServiceImpl implements CustomerService{
    private EmailService emailService;

    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }

    public void addCustomer() {
        emailService.sendEmail();
    }
}
```

```
public class EmailService implements IEmailService {

    public void sendEmail() {
        System.out.println("sendEmail");
    }
}
```

Java configuration example (2/2)

```
@Configuration  
public class AppConfig {  
    @Bean  
    public CustomerService customerService(){  
        CustomerService customerService = new CustomerServiceImpl();  
        customerService.setEmailService(emailService());  
        return customerService;  
    }  
    @Bean  
    public EmailService emailService(){  
        return new EmailServiceImpl();  
    }  
}
```

Create a bean with the name
"customerService"

Set the property emailService

AnnotationConfigApplicationContext

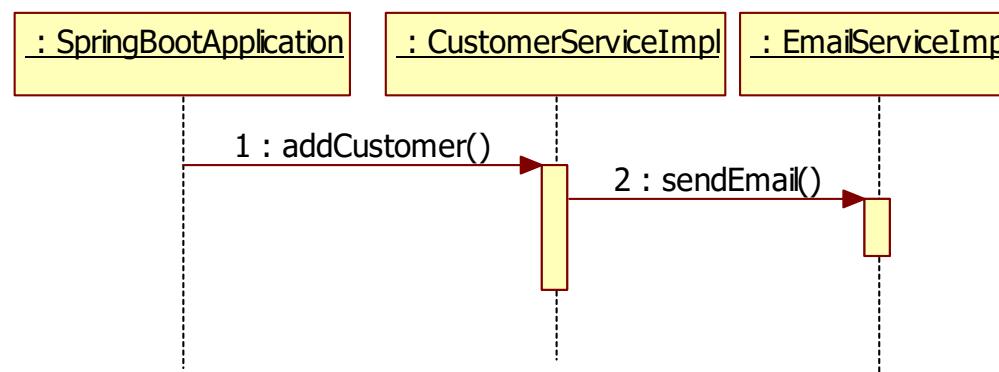
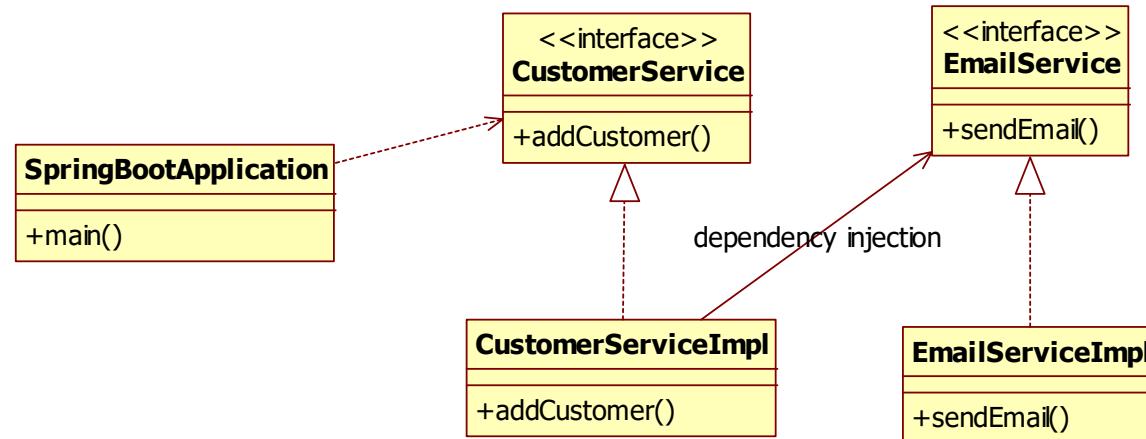
```
public class Application {  
    public static void main(String[] args) {  
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);  
        CustomerService customerService =  
            context.getBean("customerService", CustomerService.class);  
  
        customerService.addCustomer();  
    }  
}
```

3 WAYS TO CONFIGURE SPRING APPLICATIONS

3 ways of Spring configuration

- XML configuration
- Classpath scanning and Autowiring
- Java configuration

Example application



The implementation

```
public interface EmailService {  
    void sendEmail();  
}
```

```
public class EmailServiceImpl implements EmailService{  
    public void sendEmail() {  
        System.out.println("Sending email");  
    }  
}
```

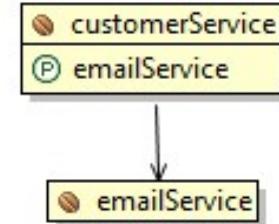
```
public interface CustomerService {  
    void addCustomer();  
}
```

```
public class CustomerServiceImpl implements CustomerService {  
  
    private EmailService emailService;  
  
    public void setEmailService(EmailService emailService) {  
        this.emailService = emailService;  
    }  
    public void addCustomer() {  
        emailService.sendEmail();  
    }  
}
```

Option 1: XML configuration

```
<bean id="customerService" class="xml.CustomerServiceImpl">
    <property name="emailService" ref="emailService" />
</bean>
<bean id="emailService" class="xml.EmailServiceImpl" />
```

Spring Beans



```
public class CustomerServiceImpl implements CustomerService {

    private EmailService emailService;

    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }
    public void addCustomer() {
        emailService.sendEmail();
    }
}
```

XML configuration

- Advantages
 - Configuration separate from Java code
 - All configuration in one place
 - Tools can use the XML for graphical views
 - Easy to change the configuration
- Disadvantages
 - Large verbose XML file(s)
 - XML and Java has to fit together
 - No compile time type safety
 - Less refactor-friendly

Option 2: Classpath scanning and Autowiring

The diagram illustrates the configuration and implementation of Spring beans. At the top, a horizontal bar is divided into colored segments: dark blue, yellow, green, red, green, yellow, blue, red, green, yellow, blue, green. Below this is a 'Spring Beans' section containing two beans: 'emailServiceimpl' and 'customerServiceimpl'. A line connects the 'emailServiceimpl' bean to a code snippet in a box. Another line connects the 'customerServiceimpl' bean to a code snippet in a rounded rectangle.

```
<context:component-scan base-package="scanning"/>
<context:annotation-config />
```

```
@Service
public class CustomerServiceImpl implements CustomerService {
    @Autowired
    private EmailService emailService;

    public void addCustomer() {
        emailService.sendEmail();
    }
}
```

Classpath scanning and Autowiring

- Advantages
 - All information (configuration and logic) in one place: the Java code
 - Simpler as XML
 - More type safe
- Disadvantage
 - Configuration in the Java code
 - Configuration is harder to change
 - Not a clear overview
 - You have to recompile

Option 3: Java configuration

```
@Configuration
public class AppConfig {
    @Bean
    public CustomerService customerService(){
        CustomerService customerService = new CustomerServiceImpl();
        customerService.setEmailService(emailService());
        return customerService;
    }
    @Bean
    public EmailService emailService(){
        return new EmailServiceImpl();
    }
}
```

```
public class CustomerServiceImpl implements CustomerService {

    private EmailService emailService;

    public void setEmailService(EmailService emailService) {
        this.emailService = emailService;
    }
    public void addCustomer() {
        emailService.sendEmail();
    }
}
```

Java configuration

- Advantages

- Configuration separate from Java code
- Type safe

- Disadvantage

- Configuration class can contain lot of java configuration code
- Configuration is harder to change
 - Not a clear overview
 - You have to recompile

Simpler configuration

- Java config + autowiring

```
@Configuration
public class AppConfig {
    @Bean
    public CustomerService customerService(){
        return new CustomerServiceImpl();
    }
    @Bean
    public EmailService emailService(){
        return new EmailServiceImpl();
    }
}
```

```
public class CustomerServiceImpl implements CustomerService {
    @Autowired
    private EmailService emailService;

    public void addCustomer() {
        emailService.sendEmail();
    }
}
```

Most simple configuration!

- Java config + classpath scanning + autowiring

```
@Configuration  
@ComponentScan  
public class AppConfig {  
}
```

```
@Service  
public class CustomerServiceImpl implements CustomerService {  
    @Autowired  
    private EmailService emailService;  
  
    public void addCustomer() {  
        emailService.sendEmail();  
    }  
}
```

```
@Service  
public class EmailServiceImpl implements EmailService{  
    public void sendEmail() {  
        System.out.println("Sending email");  
    }  
}
```

Main point

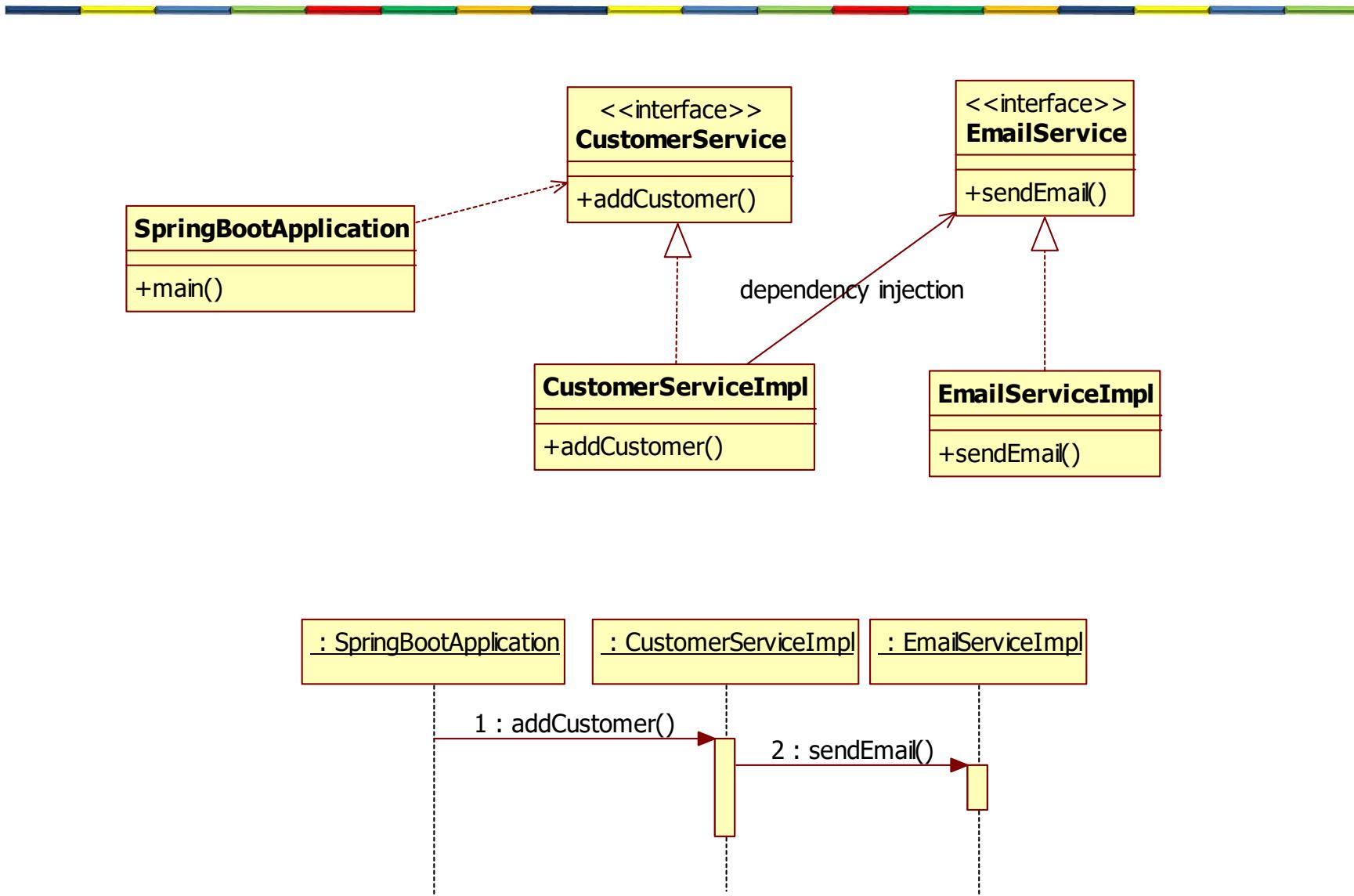
- Spring can be configured in different ways but the most simple configuration is done with classpath scanning, autowiring and Java configuration
- Our actions yields maximum results with minimum effort if we operate at the level of pure consciousness.

SPRING BOOT

Spring boot

- Framework that makes it easy to configure and run spring applications

Example application



Using annotations

```
public interface EmailService {  
    void sendEmail();  
}
```

```
@Service  
public class EmailServiceImpl implements EmailService{  
    public void sendEmail() {  
        System.out.println("Sending email");  
    }  
}
```

```
public interface CustomerService {  
    void addCustomer();  
}
```

```
@Service  
public class CustomerServiceImpl implements CustomerService{  
    @Autowired  
    private EmailService emailService;  
  
    public void addCustomer() {  
        emailService.sendEmail();  
    }  
}
```

Spring Boot option 1

```
@SpringBootApplication
public class SpringBootApplication {

    public static void main(String[] args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(SpringBootApplication.class);
        CustomerService customerService =
            context.getBean("customerServiceImpl",CustomerService.class);
        customerService.addCustomer();
    }
}
```

Same as
@Configuration,
@ComponentScan
@EnableAutoConfiguration

Create an ApplicationContext
based on the current
configuration class

Spring Boot option 2

```
@SpringBootApplication
public class SpringBootApplication {

    public static void main(String[] args) {
        ApplicationContext context = new
            AnnotationConfigApplicationContext(AppConfig.class);
        CustomerService customerService =
            context.getBean("customerServiceImpl",CustomerService.class);
        customerService.addCustomer();
    }
}
```

Create an ApplicationContext
based on an external
configuration class

```
@Configuration
@ComponentScan("customers")
public class AppConfig {
}
```

Spring Boot option 3

```
@SpringBootApplication
public class SpringBootProjectApplication implements CommandLineRunner {
    @Autowired
    private CustomerService customerService;

    public static void main(String[] args) {
        SpringApplication.run(SpringBootProjectApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        customerService.addCustomer();
    }
}
```

Implement the CommandLineRunner

Implement the run() method

Spring Boot configuration

- Spring Boot uses **application.properties** as the default configuration file

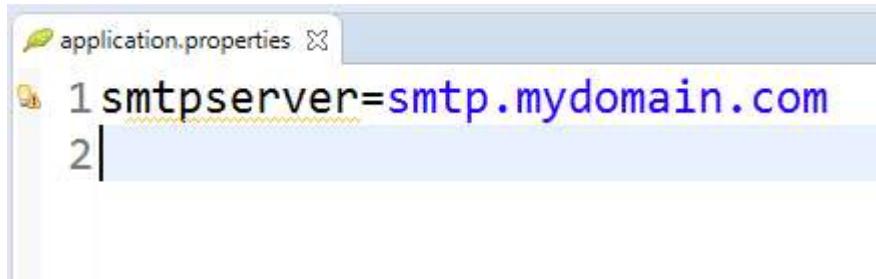


application.properties

```
public interface EmailService {  
    void sendEmail();  
}
```

```
@Service  
public class EmailServiceImpl implements EmailService{  
    @Value("${smtpserver}")  
    String smtpServer;  
  
    public void sendEmail() {  
        System.out.println("Sending email using smtp server "+smtpServer);  
    }  
}
```

Inject the value from the properties file



STRATEGY WITH SPRING

DI example

```
@Service
public class GreetingService {
    @Autowired
    private Greeting greeting;

    public String getTheGreeting() {
        return greeting.getGreeting();
    }
}
```

Spring does not know
which class to inject

```
@Component
public class GreetingOne implements Greeting{
    public String getGreeting() {
        return "Hello World";
    }
}
```

```
public interface Greeting {
    String getGreeting();
}
```

```
@Component
public class GreetingTwo implements Greeting{
    public String getGreeting() {
        return "Hi World";
    }
}
```

DI example

```
@SpringBootApplication
public class DemoProjectApplication implements CommandLineRunner {

    @Autowired
    private GreetingService greetingService;

    public static void main(String[] args) {
        SpringApplication.run(DemoProjectApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println(greetingService.getTheGreeting());
    }
}
```

```
*****
APPLICATION FAILED TO START
*****
```

Description:

Field greeting in demo.GreetingService required a single bean, but 2 were found:
- greetingOne: defined in file [C:\springtraining\workspace\DemoProject\target\classes\demo\GreetingOne.class]
- greetingTwo: defined in file [C:\springtraining\workspace\DemoProject\target\classes\demo\GreetingTwo.class]

Solution 1: use qualifier

```
@Service
public class GreetingService {
    @Autowired
    @Qualifier(value="greetingOne")
    private Greeting greeting;

    public String getTheGreeting() {
        return greeting.getGreeting();
    }
}
```

Solution 2: use profiles

```
@Service
public class GreetingService {
    @Autowired
    private Greeting greeting;

    public String getTheGreeting() {
        return greeting.getGreeting();
    }
}
```

Set the active profile in application.properties

```
1 spring.profiles.active=One
2 |
```

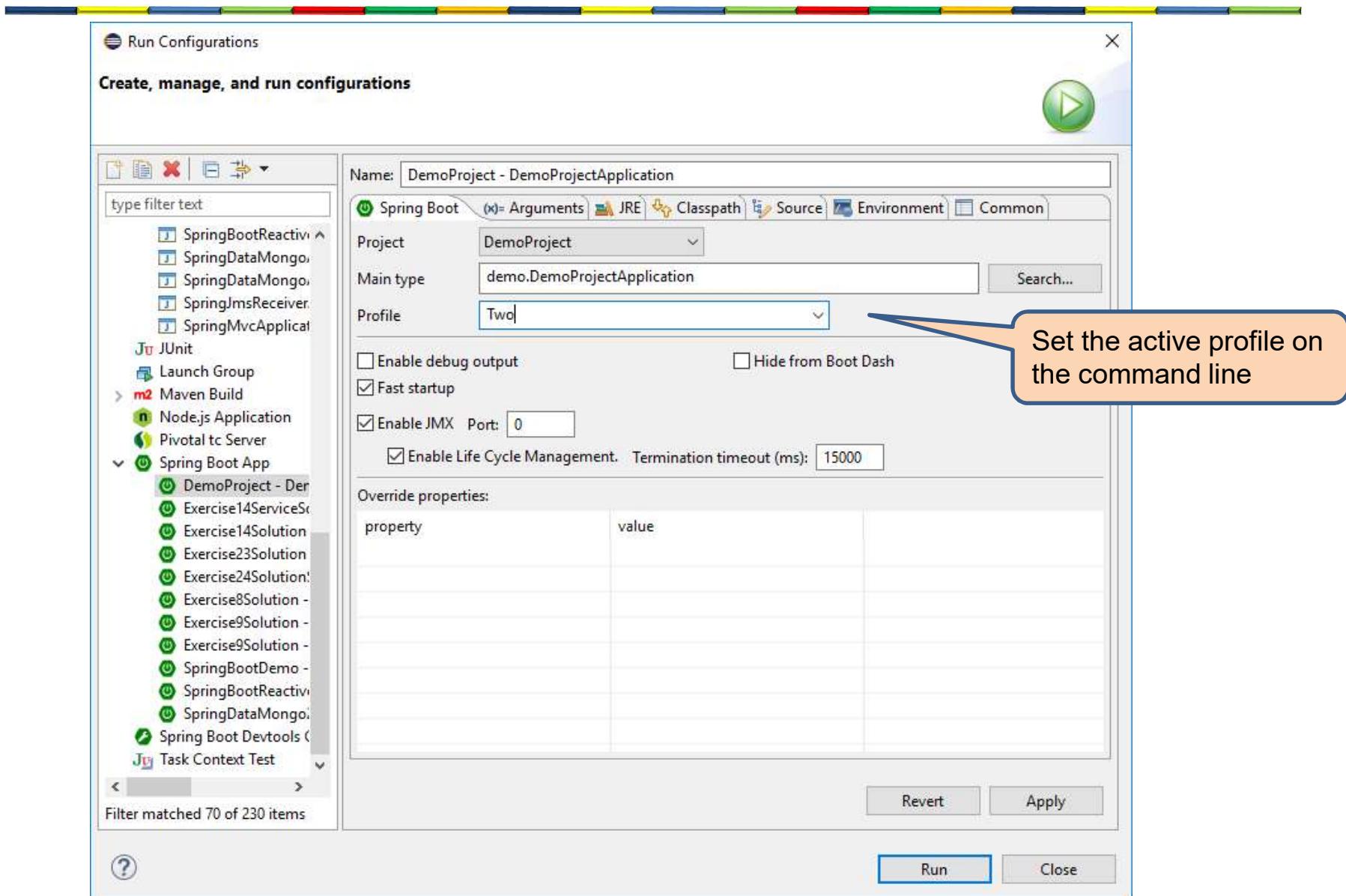
```
@Component
@Profile("One")
public class GreetingOne implements Greeting{
    public String getGreeting() {
        return "Hello World";
    }
}
```

Define a profile

```
@Component
@Profile("Two")
public class GreetingTwo implements Greeting{
    public String getGreeting() {
        return "Hi World";
    }
}
```

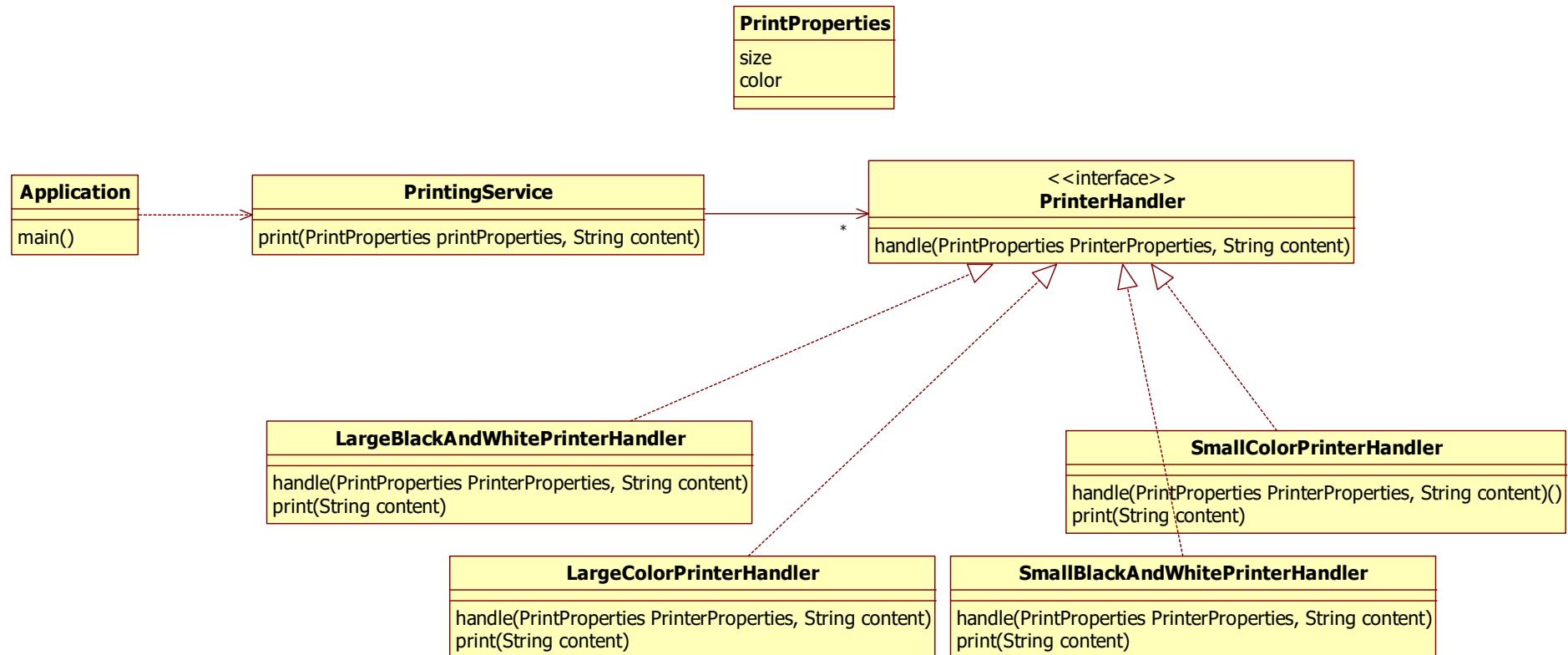
Define a profile

Activate a profile



CHAIN OF RESPONSIBILITY WITH SPRING

Chain of Responsibility with Spring



LargeBlackAndWhitePrinterHandler

```
@Component
public class LargeBlackAndWhitePrinterHandler implements PrinterHandler{

    public void print(String content){
        System.out.println("Large black and white printer prints: "+content);
    }

    public boolean handle(PrintProperties printerProperties, String content) {
        if (isBlackAndWhitePrinter(printerProperties) &&
            isLargePrinter(printerProperties)) {
            print(content);
            return true;
        }
        return false;
    }

    private boolean isLargePrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getSize()==Size.LARGE;
    }

    private boolean isBlackAndWhitePrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getColor()==Color.BLACKANDWHITE;
    }
}
```

LargeColorPrinterHandler

```
@Component
public class LargeColorPrinterHandler implements PrinterHandler{

    public void print(String content){
        System.out.println("Large color printer prints: "+content);
    }

    public boolean handle(PrintProperties printerProperties, String content) {
        if (isColorPrinter(printerProperties) &&
            isLargePrinter(printerProperties)) {
            print(content);
            return true;
        }
        return false;
    }

    private boolean isLargePrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getSize()==Size.LARGE;
    }

    private boolean isColorPrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getColor()==Color.COLOR;
    }
}
```

SmallBlackAndWhitePrinterHandler

```
@Component
public class SmallBlackAndWhitePrinterHandler implements PrinterHandler{

    public void print(String content){
        System.out.println("Large black and white printer prints: "+content);
    }

    public boolean handle(PrintProperties printerProperties, String content) {
        if (isBlackAndWhitePrinter(printerProperties) &&
            isSmallPrinter(printerProperties)) {
            print(content);
            return true;
        }
        return false;
    }

    private boolean isSmallPrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getSize()==Size.SMALL;
    }

    private boolean isBlackAndWhitePrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getColor()==Color.BLACKANDWHITE;
    }
}
```

SmallColorPrinterHandler

```
@Component
public class SmallColorPrinterHandler implements PrinterHandler{

    public void print(String content){
        System.out.println("Small color printer prints: "+content);
    }

    public boolean handle(PrintProperties printerProperties, String content) {
        if (isColorPrinter(printerProperties) && isSmallPrinter(printerProperties)) {
            print(content);
            return true;
        }
        return false;
    }

    private boolean isSmallPrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getSize()==Size.SMALL;
    }

    private boolean isColorPrinter(PrintProperties PrinterProperties) {
        return PrinterProperties.getColor()==Color.COLOR;
    }
}
```

PrinterHandler and PrinterProperties

```
public interface PrinterHandler {  
    public void handle(PrintProperties PrinterProperties, String content);  
}
```

```
public class PrintProperties {  
    enum Size {  
        SMALL, LARGE  
    }  
  
    enum Color {  
        BLACKANDWHITE, COLOR  
    }  
  
    private Size size;  
    private Color color;  
  
    public PrintProperties(Size size, Color color) {  
        this.size = size;  
        this.color = color;  
    }  
  
    ...  
}
```

PrintingService

```
@Service
public class PrintingService {
    @Autowired
    List<PrinterHandler> printerHandlers;

    public void print(PrintProperties printProperties, String content) {
        for (PrinterHandler printerHandler: printerHandlers) {
            if (printerHandler.handle(printProperties, content))
                break;
        }
    }
}
```

Application

```
@SpringBootApplication
public class Application implements CommandLineRunner {

    @Autowired
    private PrintingService printingService;

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        printingService.print(new PrintProperties(Size.SMALL, Color.COLOR), "Hello world");
        printingService.print(new PrintProperties(Size.LARGE, Color.COLOR), "Hello world");
        printingService.print(new PrintProperties(Size.SMALL, Color.BLACKANDWHITE), "Hello world");
        printingService.print(new PrintProperties(Size.LARGE, Color.BLACKANDWHITE), "Hello world");
    }
}
```

Small color printer prints: Hello world
Large color printer prints: Hello world
Small black and white printer prints: Hello world
Large black and white printer prints: Hello world

Main point

- The Chain of responsibility can be implemented in Spring with list injection
- We enliven those aspects in creation where we put our attention on.

Connecting the parts of knowledge with the wholeness of knowledge

1. The Spring framework instantiates your objects and wires them together with dependency injection.
2. Profiles allow you to change the wiring of beans using an external properties file

-
3. **Transcendental consciousness** is the field of pure consciousness which is home to all the laws of nature.
 4. **Wholeness moving within itself:** In unity consciousness one enjoys a permanent state of fulfilment while spontaneously benefiting oneself and society
- 

Lesson 13 Spring framework

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Crosscutting concern

- Check security for **every** service level method

```
public class CustomerService {  
  
    public void getAllCustomers() {  
        checkSecurity();  
        ...  
    }  
  
    public void getCustomer(long customerNumber) {  
        checkSecurity();  
        ...  
    }  
  
    public void addCustomer(long customerNumber, String firstName) {  
        checkSecurity();  
        ...  
    }  
  
    public void removeCustomer(long customerNumber) {  
        checkSecurity();  
        ...  
    }  
}
```

We have to call
checkSecurity() for all methods
of all service classes

Crosscutting concern

- Log **every** call to the database

```
public class AccountDAO {  
  
    public void saveAccount(Account account) {  
        ...  
        Logger.log("...");  
    }  
  
    public void updateAccount(Account account) {  
        ...  
        Logger.log("...");  
    }  
  
    public void loadAccount(long accountNumber) {  
        ...  
        Logger.log("...");  
    }  
  
    public void removeAccount(long accountNumber) {  
        ...  
        Logger.log("...");  
    }  
}
```

We have to call
Logger.log() for all methods of
all DAO classes

Good programming practice principles

DRY: Don't Repeat Yourself

- Write functionality at one place, and only at one place
- Avoid code scattering

SoC: Separation of Concern

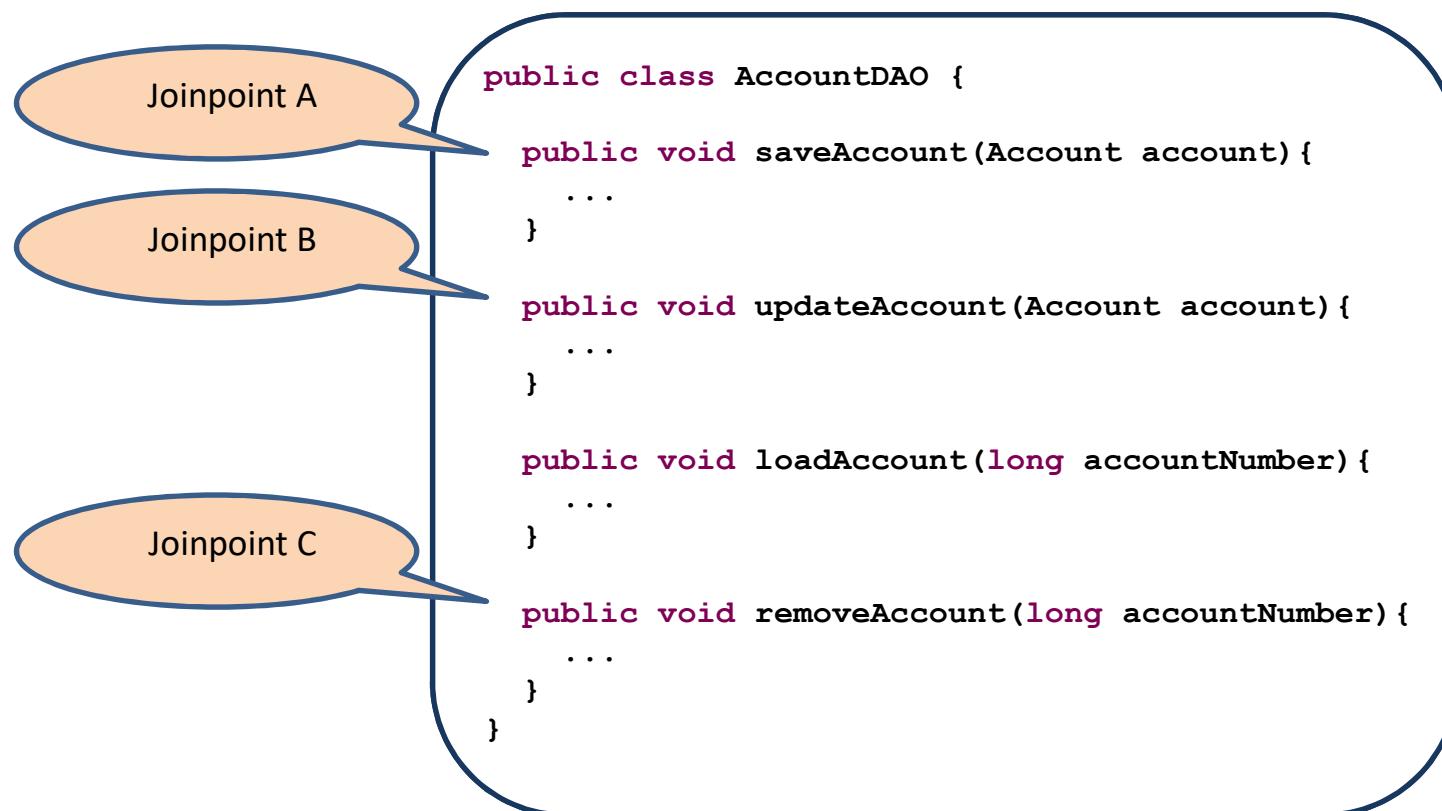
- Avoid code tangling

AOP concepts

- Joinpoint
- Pointcut
- Aspect
- Advice
- Weaving

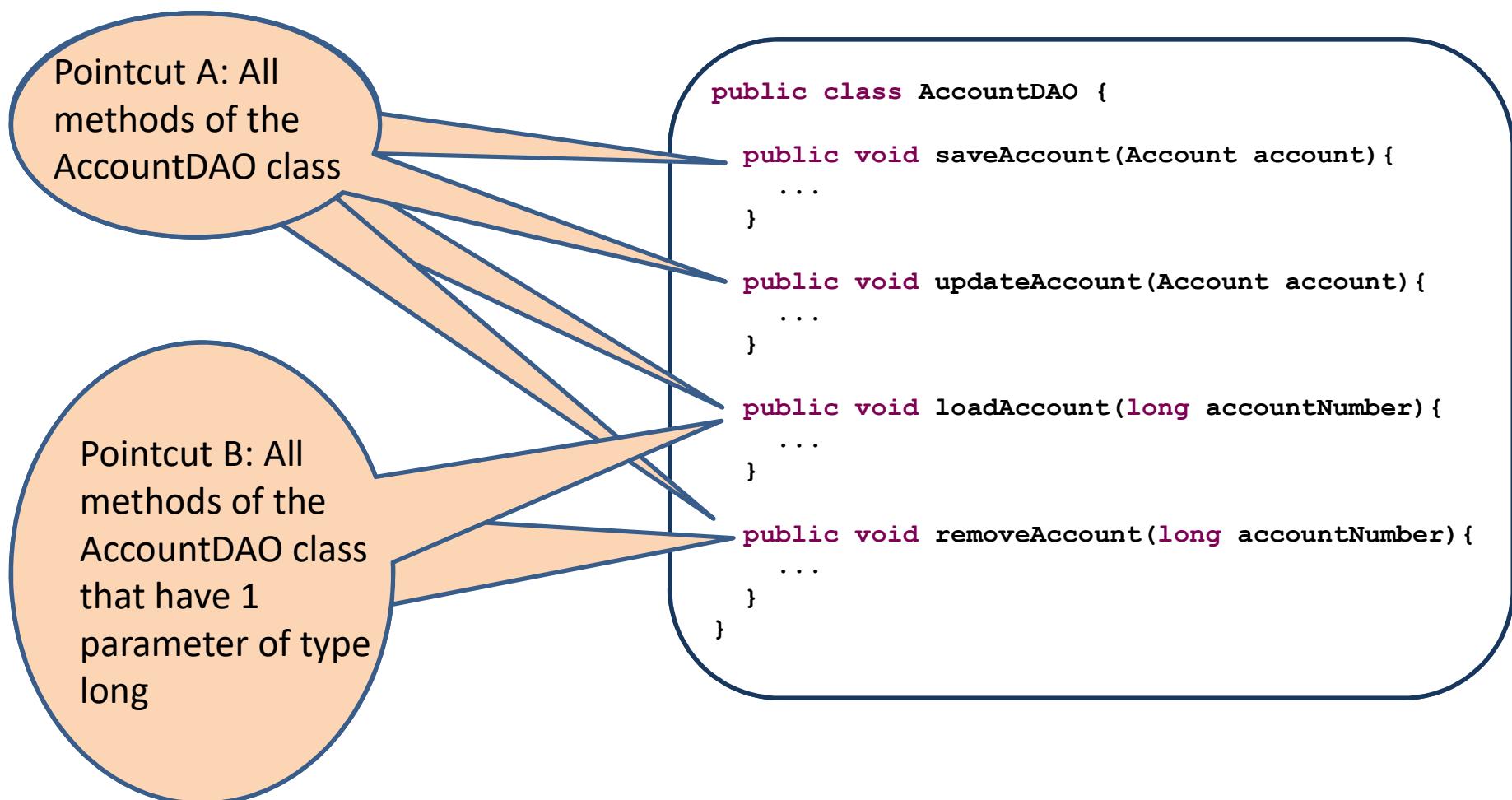
AOP concept: Joinpoint

- A specific method



AOP concept: Pointcut

- A collection of 1 or more joinpoints



AOP concept: Advice

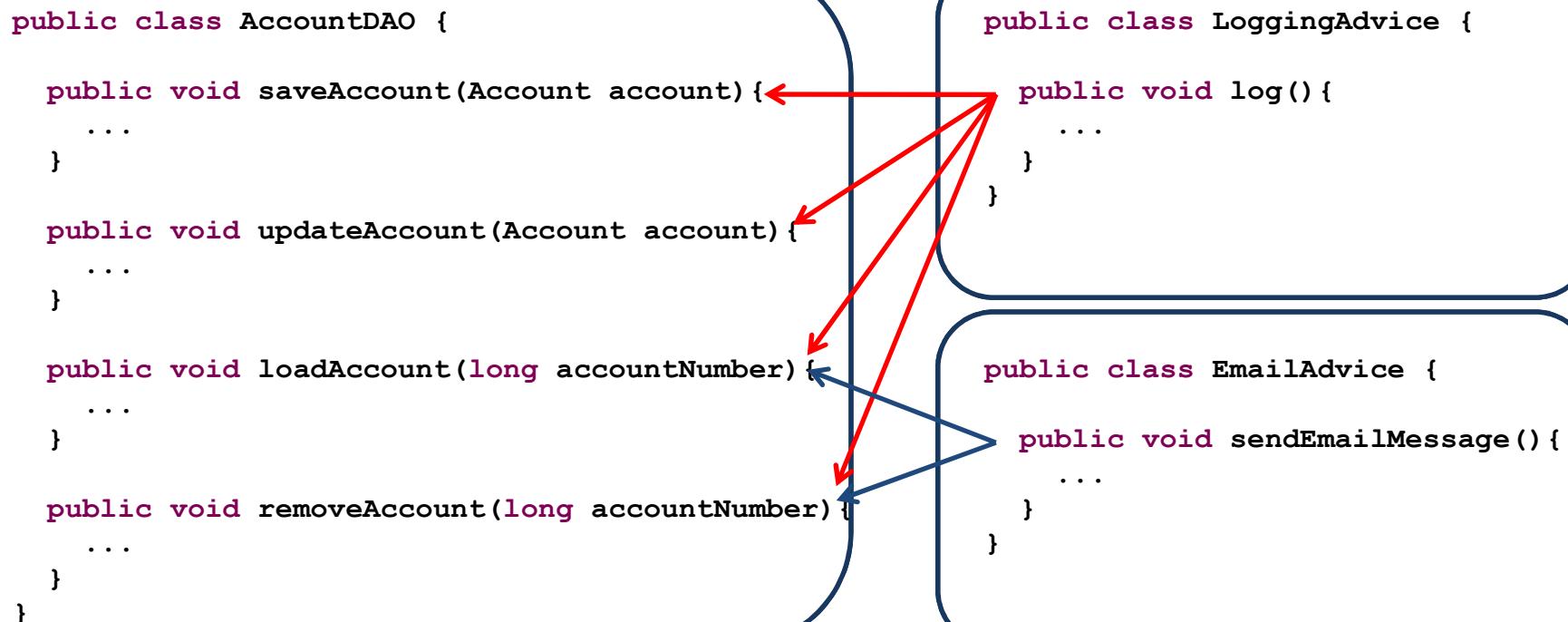
- The implementation of the crosscutting concern

```
public class LoggingAdvice {  
  
    public void log(){  
        ...  
    }  
}
```

```
public class EmailAdvice {  
  
    public void sendEmailMessage(){  
        ...  
    }  
}
```

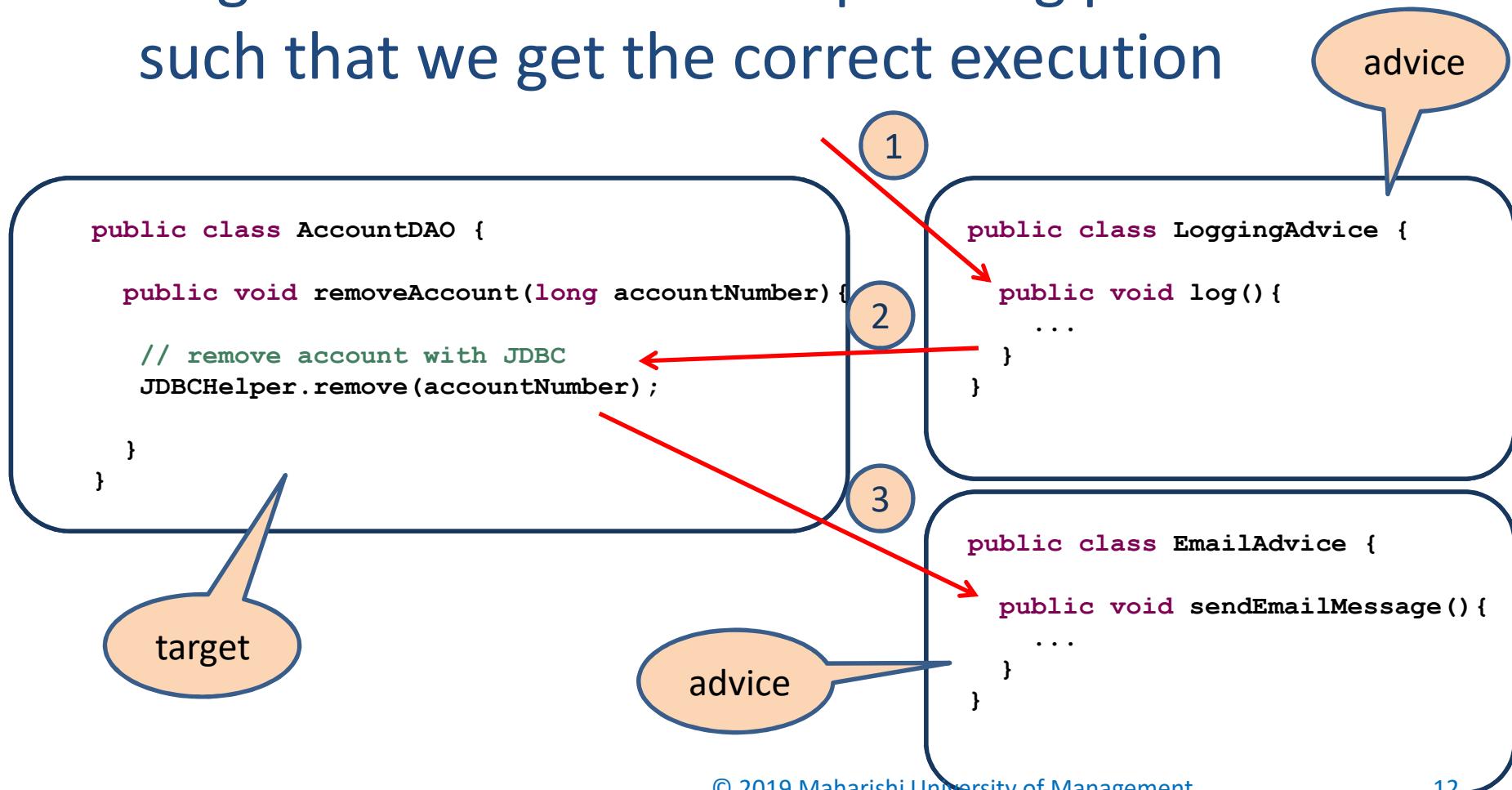
AOP concept: Aspect

- What crosscutting concern do I execute (=advice) at which locations in the code (=pointcut)
 - Aspect A: call the log() method of LoggingAdvice before every method call of AccountDAO
 - Aspect B: call the sendEmailMessage() method of EmailAdvice after every method call of AccountDAO that has one parameter of type long



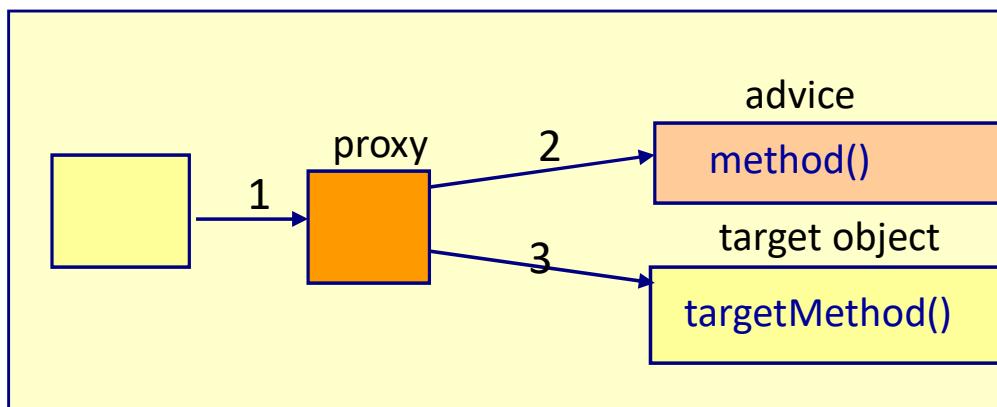
AOP concept: Weaving

- Weave the advice code together with the target code at the corresponding pointcuts such that we get the correct execution



Weaving

- Spring creates a dynamic proxy that weaves the advice and the target method together

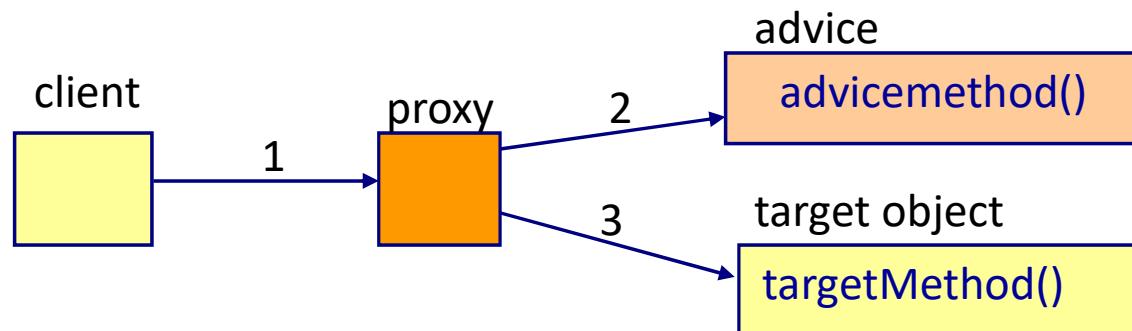


Advice types

- Before
- After returning
- After throwing
- After
- Around

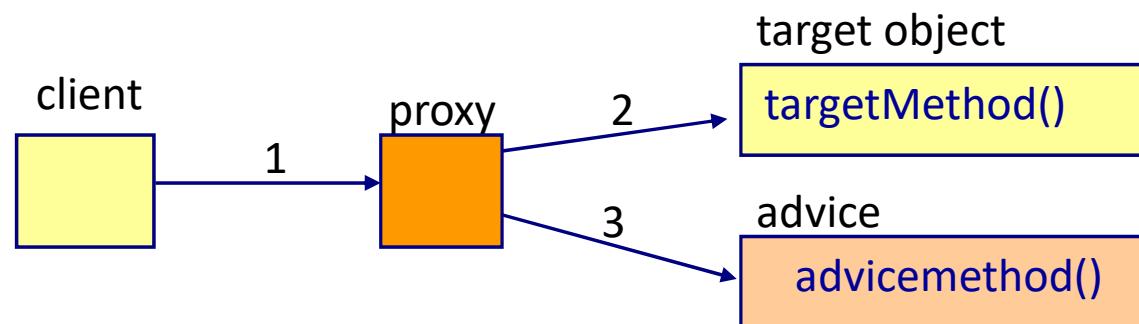
Before advice

- First call the advice method and then the business logic method



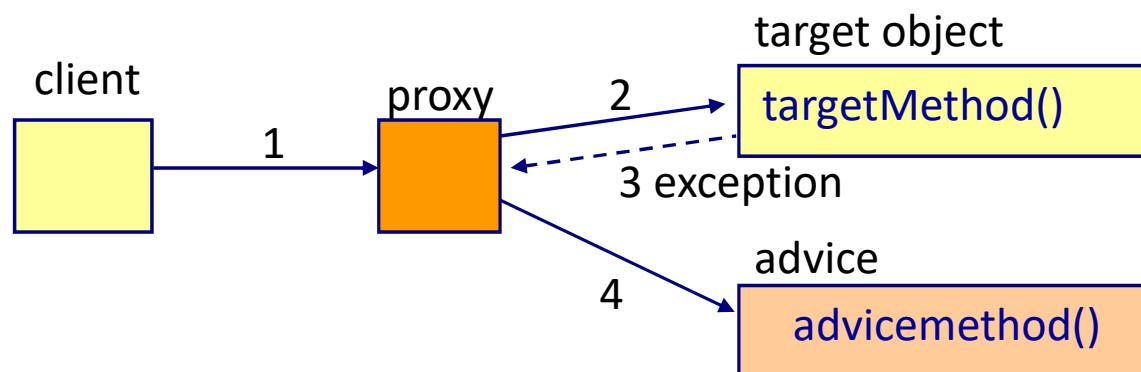
After returning advice

- First call the business logic method and when this business logic method returns normally without an exception, then call the advice method



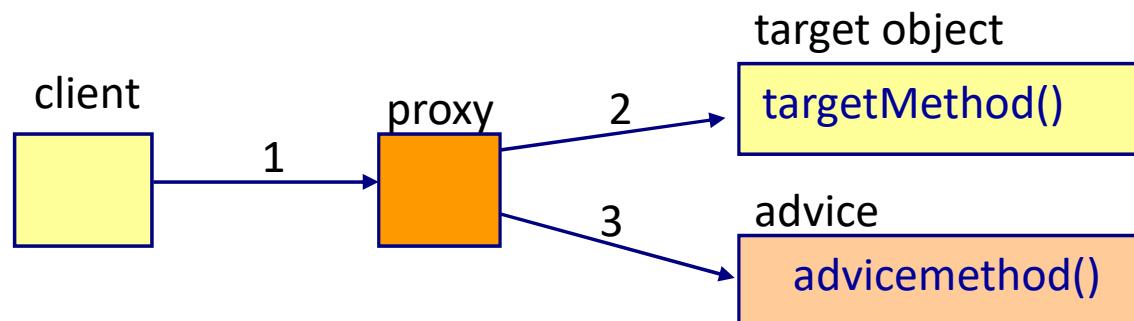
After throwing advice

- First call the business logic method and when this business logic method throws an exception, then call the advice method



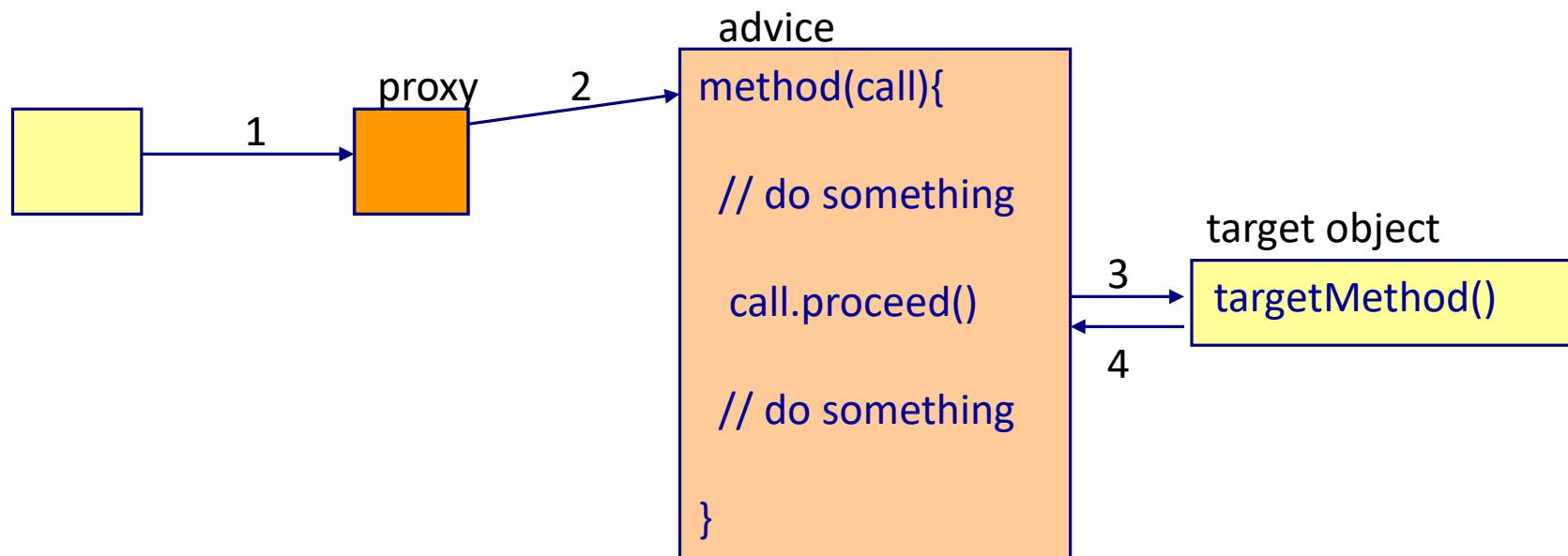
After advice

- First call the business logic method and then call the advice method (independent of how the business logic method returned: normally or with exception)



Around advice

- First call the advice method. The advice method calls the business logic method, and when the business logic method returns, we get back to the advice method



AOP with Spring Boot

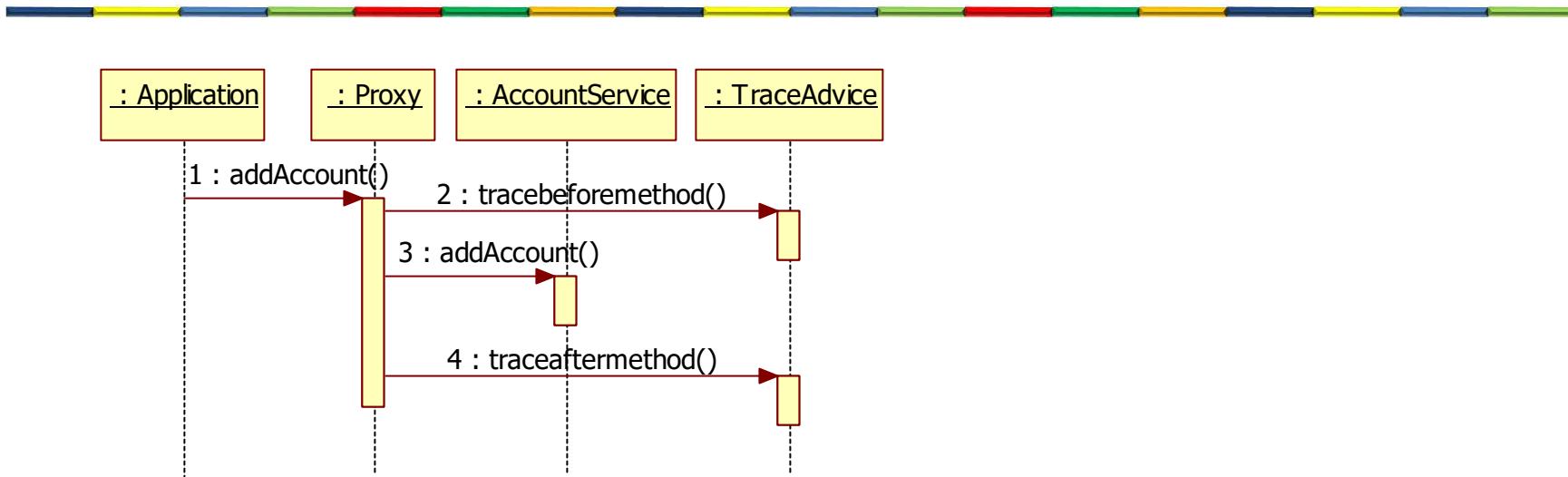
```
public class AccountService implements IAccountService{
    Collection<Account> accountList = new ArrayList();

    public void addAccount(String accountNumber, Customer customer) {
        Account account = new Account(accountNumber, customer);
        accountList.add(account);
        System.out.println("in execution of method addAccount");
    }
}
```

@Configuration

```
@Aspect
@Configuration
public class TraceAdvice {
    @Before("execution(* accountpackage.AccountService.*(..))")
    public void tracebeforemethod(JoinPoint joinpoint) {
        System.out.println("before execution of method "+joinpoint.getSignature().getName());
    }
    @After("execution(* accountpackage.AccountService.*(..))")
    public void traceaftermethod(JoinPoint joinpoint) {
        System.out.println("after execution of method "+joinpoint.getSignature().getName());
    }
}
```

AOP with Spring Boot



```
@Aspect
@Configuration
public class TraceAdvice {
    @Before("execution(* accountpackage.AccountService.*(..))")
    public void tracebeforemethod(JoinPoint joinpoint) {
        System.out.println("before execution of method "+joinpoint.getSignature().getName());
    }
    @After("execution(* accountpackage.AccountService.*(..))")
    public void traceaftermethod(JoinPoint joinpoint) {
        System.out.println("after execution of method "+joinpoint.getSignature().getName());
    }
}
```

Pointcut execution language

Pointcut execution language

```
@Aspect
public class TraceAdvice {
    @Before("execution(* accountpackage.AccountService.*(..))")
    public void tracebeforemethod(JoinPoint joinpoint) {
        System.out.println("before execution of method "+joinpoint.getSignature().getName());
    }
    @After("execution(* accountpackage.AccountService.*(..))")
    public void traceaftermethod(JoinPoint joinpoint) {
        System.out.println("after execution of method "+joinpoint.getSignature().getName());
    }
}
```

Pointcut execution language

- `@Before ("execution(public * *.*.*(..))")`

Visibility:

- Possibilities:
 - private
 - public
 - Protected
- Optional
- Cannot be *

Return type:

- The return type of the corresponding method(s)
- Not optional
- Can be *

package.class.method(args):

- Name of the package can also be *
- Name of the class can also be *
- Name of the method can also be *
- Arguments can be ..
- Not optional
- Can also be *.*(..)
- Can also be *(..)

Pointcut execution language examples

```
@After("execution(public * *(..))")
```

All public methods

```
@After("execution(public void *(..))")
```

All public methods that return void

```
@After("execution(* order.*.*(..))")
```

All methods from all classes in the order package

```
@After("execution(* *.*.create*(..))")
```

All methods that start with create

```
@After("execution(* *.*.Customer.*(..))")
```

All methods from the Customer class

Pointcut execution language examples

```
@After("execution(* order.Customer.*(..))")
```

All methods from the Customer class in the order package

```
@After("execution(* order.Customer.getPayment(..))")
```

The getPayment () method from the Customer class in the order package

```
@After("execution(* order.Customer.getPayment(int))")
```

The getPayment () method with a parameter of type int from the Customer class in the order package

```
@After("execution(* *.*.*(long, String))")
```

All methods from all classes that have 2 parameters, the first of type long, and the second of type String

Around example

```
@Around("execution(* *.*.*(..))")  
public Object profile (ProceedingJoinPoint call) throws Throwable{  
StopWatch clock = new StopWatch("");  
clock.start(call.toShortString());  
  
Object object= call.proceed();  
  
clock.stop();  
System.out.println(clock.prettyPrint());  
return object;  
}
```

Create and start a stopwatch

Call the business logic method

Stop the stopwatch and
print result

```
StopWatch ''': running time (millis) = 1  
-----  
ms      %      Task name  
-----  
00001  100%  execution(addAccount)
```

Getting the return value

- Works only for @AfterReturning

```
public class Customer {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

getName() returns a String

The pointcut expression

Add 'returning' parameter

```
@Aspect  
public class TraceAdvice {  
    @AfterReturning(pointcut="execution(* mypackage.Customer.getName(..))", returning="returnValue")  
    public void tracemethod(JoinPoint joinpoint, String retValue) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("return value =" + retValue);  
    }  
}
```

Add parameter to the advice method.
The name of the parameter must be the same as the name of the returning parameter of the @AfterReturning annotation

Getting the return value

```
public class Customer {  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
}
```

getAge() returns an integer

```
@Aspect  
public class TraceAdvice {  
    @AfterReturning(pointcut="execution(* mypackage.Customer.getAge(..))",returning="returnValue")  
    public void tracemethod(JoinPoint joinpoint, int returnValue) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("return value =" + returnValue);  
    }  
}
```

Add 'returning' parameter

returnValue is an int

Getting the exception

- Works only for @AfterThrowing

```
public class Customer {  
    public void myMethod() throws MyException{  
        throw new MyException("myexception");  
    }  
}
```

```
public class MyException extends Exception{  
    private String message;  
  
    public MyException(String message){  
        this.message=message;  
    }  
    public String getMessage(){  
        return message;  
    }  
}
```

```
@Aspect  
public class TraceAdvice {  
    @AfterThrowing(pointcut="execution(* mypackage.Customer.myMethod(..))", throwing="exception")  
    public void tracemethod(JoinPoint joinpoint, MyException exception) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("exception message =" + exception.getMessage());  
    }  
}
```

Add 'throwing' parameter

Add parameter to the advice method

Get parameters

```
public class Customer {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
@Aspect  
public class TraceAdvice {  
    @After("execution(* mypackage.Customer.setName(..)) & args(name)")  
    public void tracemethod(JoinPoint joinpoint, String name) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
    }  
}
```

Add 'args' parameter

Add parameter(s) to the advice method

Get parameters

```
public class Customer {  
    private String name;  
    private int age;  
  
    public void setNameAndAge(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
}
```

2 parameters

```
@Aspect  
public class TraceAdvice {  
    @Before("execution(* mypackage.Customer.setNameAndAge(..)) && args(name,age)")  
    public void tracemethod(JoinPoint joinpoint, String name, int age) {  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
        System.out.println("parameter age =" + age);  
    }  
}
```

Add name and age to the args parameter

Add 2 parameters to the advice method

Get parameters from the Joinpoint

```
public class Customer {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Get the arguments from the joinpoint

Take the first argument

```
@Aspect  
public class TraceAdvice {  
    @After("execution(* mypackage.Customer.setName(..))")  
    public void tracemethodA(JoinPoint joinpoint) {  
        Object[] args = joinpoint.getArgs();  
        String name = (String)args[0];  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("parameter name =" + name);  
    }  
}
```

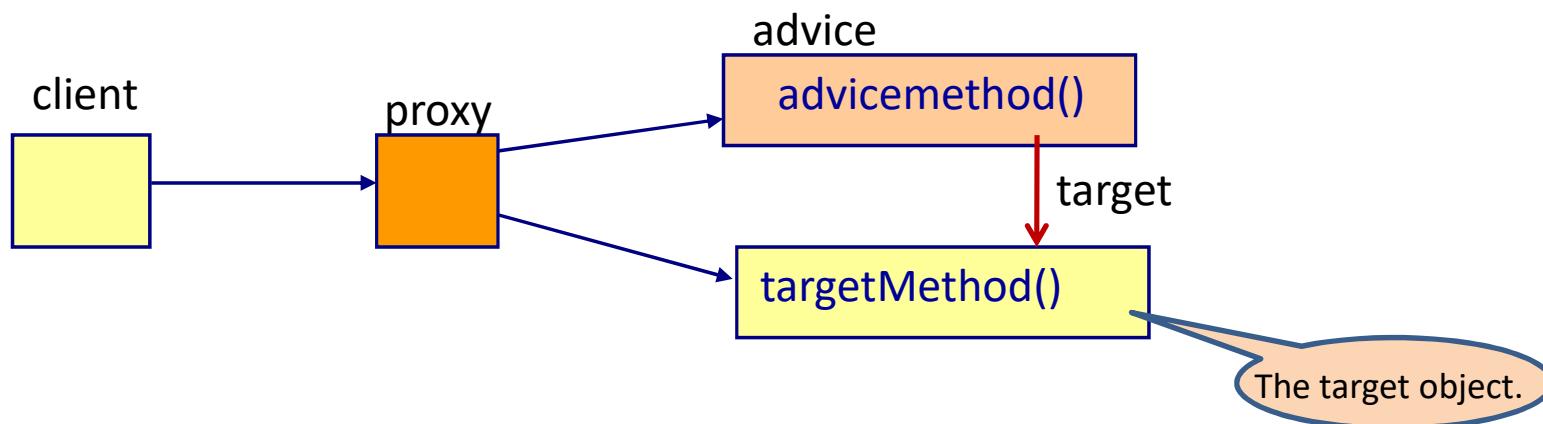
Get multiple parameters from the Joinpoint

```
public class Customer {  
    private String name;  
    private int age;  
  
    public void setNameAndAge(String name, int age){  
        this.name = name;  
        this.age = age;  
    }  
}
```

2 parameters

```
@Aspect  
public class TraceAdvice {  
    @Before("execution(* mypackage.Customer.setNameAndAge(..))")  
    public void tracemethod(JoinPoint joinpoint ) {  
        Object[] args = joinpoint.getArgs();  
        String name = (String)args[0];  
        int age = (Integer)args[1];  
        System.out.println("method =" +joinpoint.getSignature().getName());  
        System.out.println("parameter name =" +name);  
        System.out.println("parameter age =" +age);  
    }  
}
```

The target class



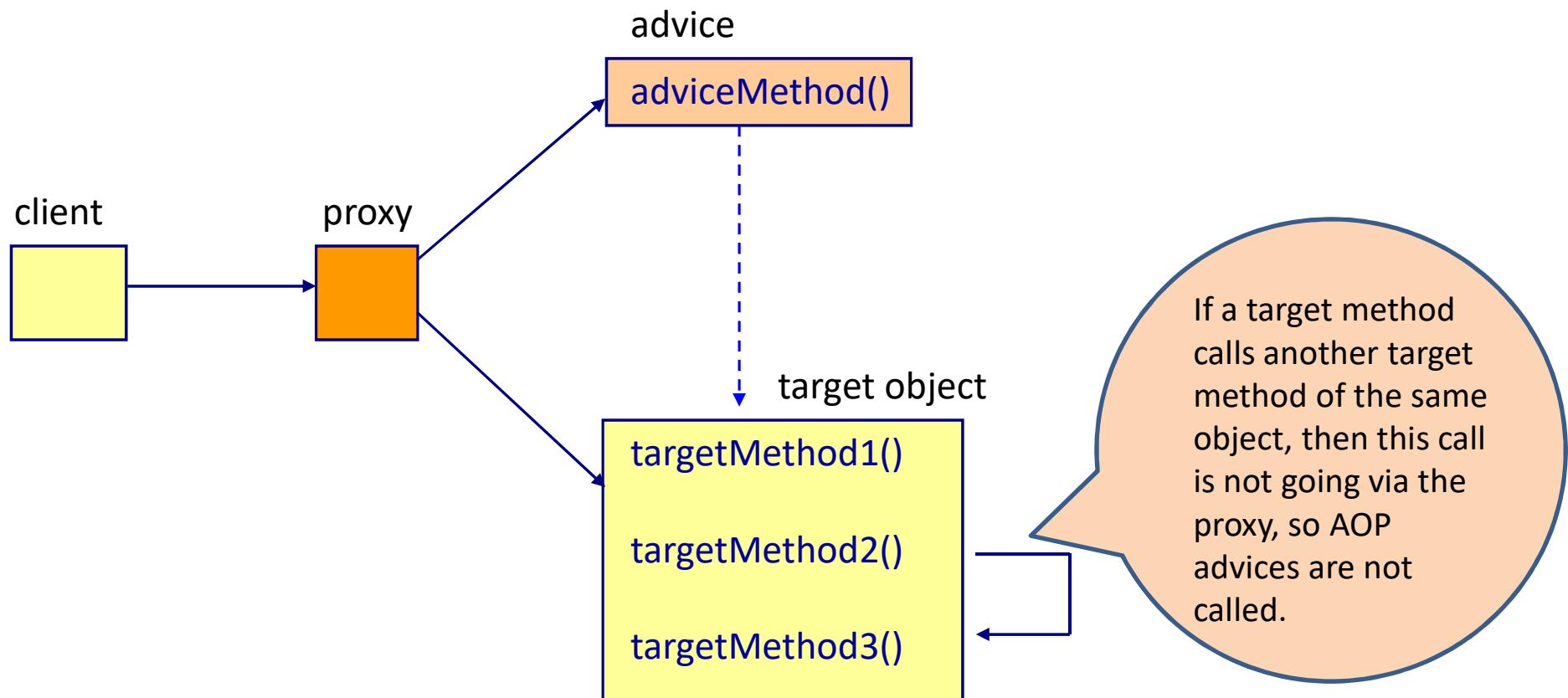
Get the target class

```
public class Customer {  
    private String name;  
    private int age;  
  
    public int getAge() {  
        return age;  
    }  
    public void setAge(int age) {  
        this.age = age;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

Get the target object from the joinpoint

```
@Aspect  
public class TraceAdvice {  
    @After("execution(* mypackage.Customer.setName(..))")  
    public void tracemethod(JoinPoint joinpoint) {  
        Customer customer = (Customer)joinpoint.getTarget();  
        System.out.println("method =" + joinpoint.getSignature().getName());  
        System.out.println("customer age =" + customer.getAge());  
    }  
}
```

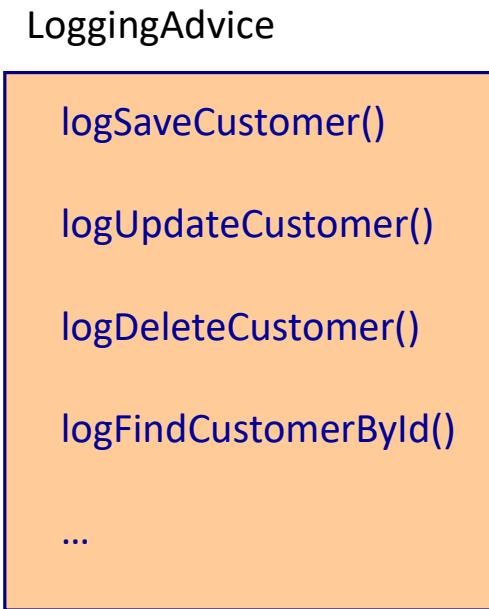
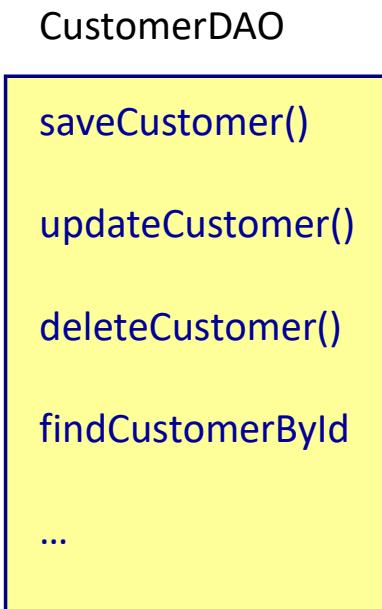
Disadvantage of a proxy



Advantages of AOP

- No code tangling
- No code scattering

Crosscutting concern needs to be generic



Disadvantages of AOP

- You don't have a clear overview of which code runs when
- AOP works only for generic logic that is always the same
- A pointcut expression is a string that is parsed at runtime
 - No compile time checking of the pointcut expression
- You make mistakes easily
- Problems with proxy-based AOP

Main point

- With Spring AOP we separate the logic at design time and we weave it together at runtime using a proxy.
- In the relative world everything seems to be separated while in reality everything is connected at its source, the unified field of pure consciousness.

Event publisher and listener

```
@Service
public class CustomerServiceImpl implements CustomerService {
    @Autowired
    private ApplicationEventPublisher publisher; Inject a publisher

    public void addCustomer() {
        publisher.publishEvent(new AddCustomerEvent("New customer is added"));
    }
}
```

```
@Service
public class Listener {
    @EventListener
    public void onEvent(AddCustomerEvent event) {
        System.out.println("received event :" + event.getMessage());
    }
}
```

Listen to AddCustomer events

Events

```
public class AddCustomerEvent {  
    private String message;  
  
    public AddCustomerEvent(String message) {  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
}
```

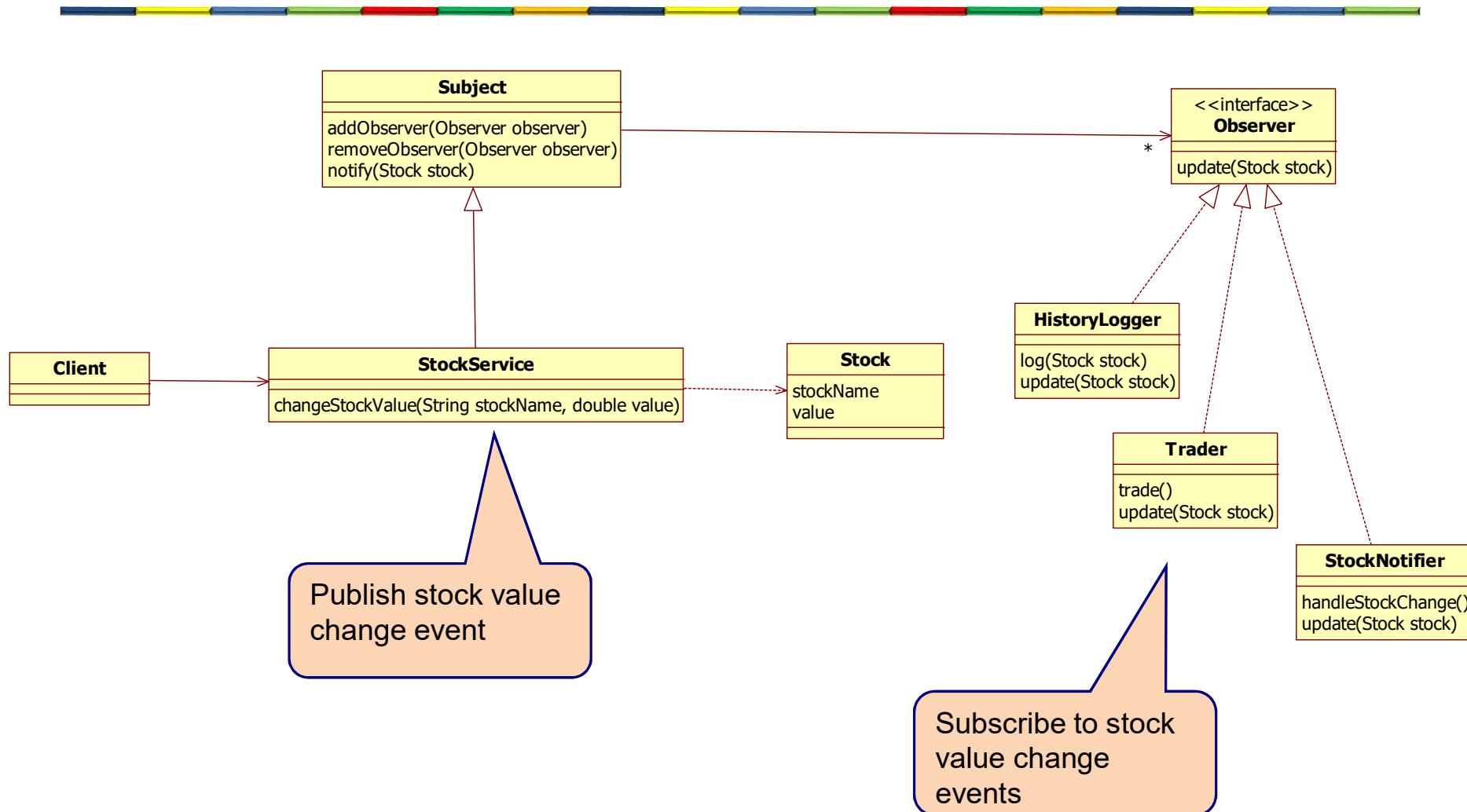
A simple event class

Asynchronous events

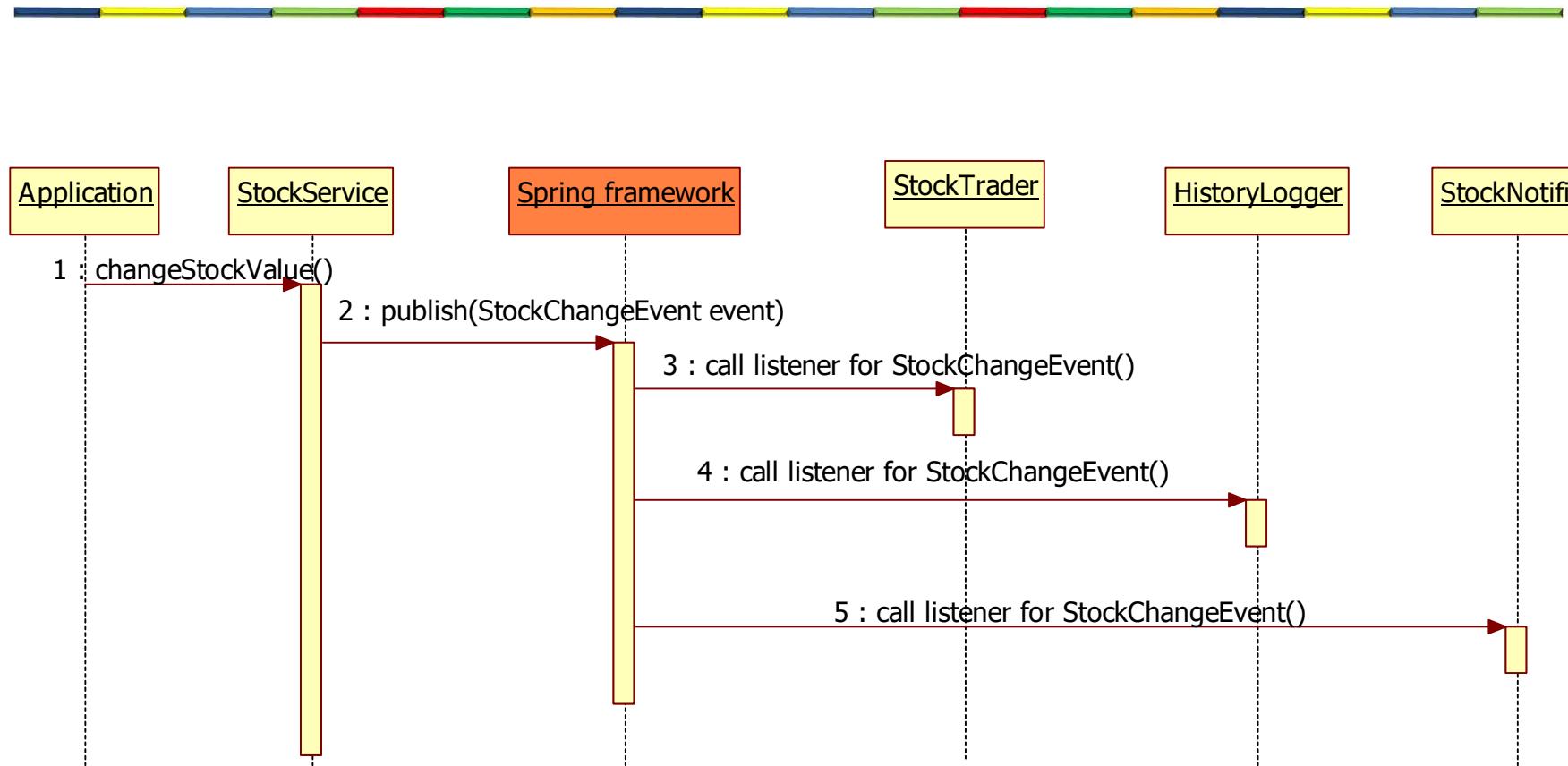
```
@Service
@EnableAsync
public class Listener {

    @Async
    @EventListener
    public void onEvent(AddCustomerEvent event) {
        System.out.println("received event :" + event.getMessage());
    }
}
```

Observer pattern



Spring publish-subscribe



Publisher



```
public class StockChangeEvent {  
    private String stockName;  
    private double newValue;  
  
    public StockChangeEvent(String stockName, double newValue) {  
        this.stockName = stockName;  
        this.newValue = newValue;  
    }  
  
    @Override  
    public String toString() {  
        return "StockChangeEvent [stockName=" + stockName + ", newValue=" + newValue + "]";  
    }  
}
```

```
@Service  
public class StockService {  
    @Autowired  
    private ApplicationEventPublisher publisher;  
  
    public void changeStockValue(String stockName, double value) {  
        publisher.publishEvent(new StockChangeEvent(stockName, value));  
    }  
}
```

Subscribers

```
@Component
public class HistoryLogger {
    @Async
    @EventListener
    public void log(StockChangeEvent stockChangeEvent) {
        System.out.println("HistoryLogger received event :" + stockChangeEvent);;
    }
}
```

```
@Component
public class StockTrader {
    @Async
    @EventListener
    public void trade(StockChangeEvent stockChangeEvent) {
        System.out.println("StockTrader received event :" + stockChangeEvent);;
    }
}
```

```
@Component
public class StockNotifier {
    @Async
    @EventListener
    public void handleChangeEvent(StockChangeEvent stockChangeEvent) {
        System.out.println("StockNotifier received event :" + stockChangeEvent);;
    }
}
```

The application

```
@SpringBootApplication
@EnableAsync
public class Application implements CommandLineRunner {

    @Autowired
    private StockService stockService;

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        stockService.changeStockValue("AMZN", 2310.80);
        stockService.changeStockValue("MSFT", 890.45);
    }
}
```

```
HistoryLogger received event :StockChangeEvent [stockName=AMZN, newValue=2310.8]
HistoryLogger received event :StockChangeEvent [stockName=MSFT, newValue=890.45]
StockTrader received event :StockChangeEvent [stockName=AMZN, newValue=2310.8]
StockTrader received event :StockChangeEvent [stockName=MSFT, newValue=890.45]
StockNotifier received event :StockChangeEvent [stockName=AMZN, newValue=2310.8]
StockNotifier received event :StockChangeEvent [stockName=MSFT, newValue=890.45]
```

Main point

- Publish-Subscribe is a very powerful technique that allows for loosely coupled publishers and subscribers.
- The nervous system of a human being is able to subscribe to the intelligence of pure consciousness.

Connecting the parts of knowledge with the wholeness of knowledge

1. With AOP you can avoid code tangling and code scattering
2. Spring events is a very easy way to implement the observer pattern.

-
3. **Transcendental consciousness** is the source of all activity
 4. **Wholeness moving within itself:** In Unity Consciousness, one experiences that one self (rishi), and all other objects (chhandas) and the operations between oneself and all other objects (devata) are expressions of one's own Self.

