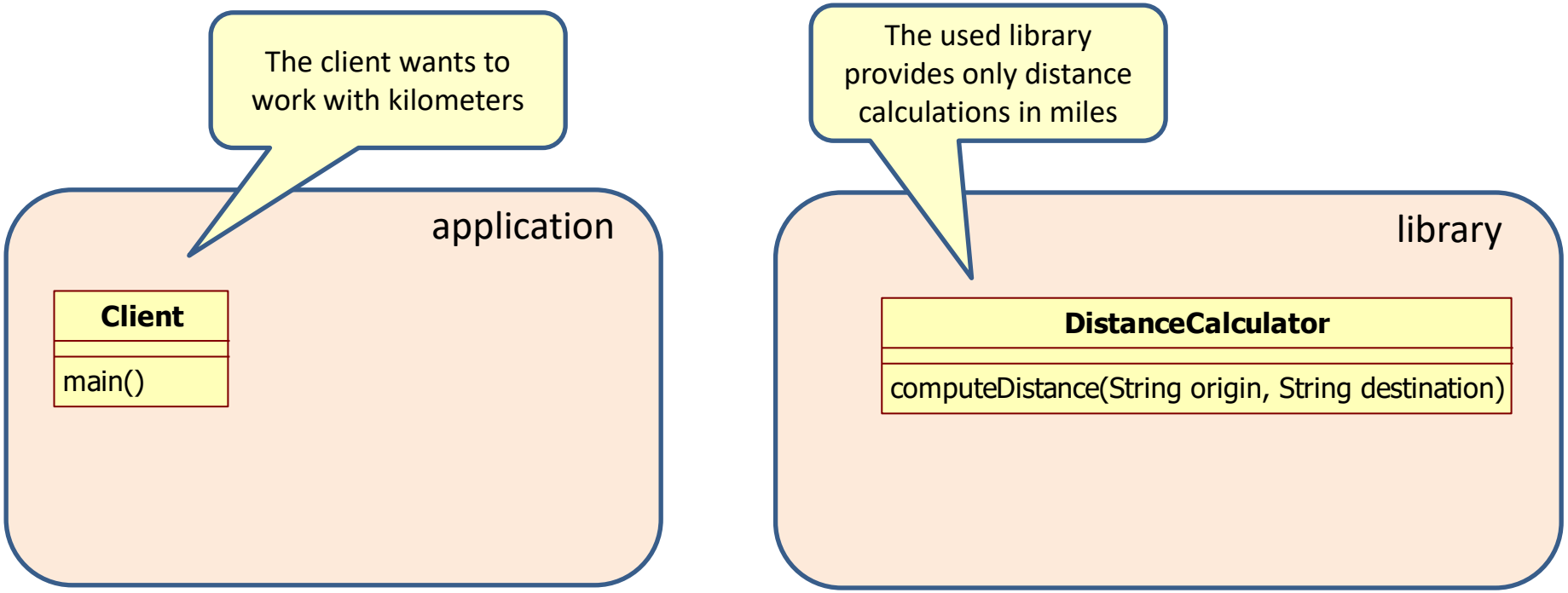
LESSON 8
ADAPTER, MEDIATOR, PROXY PATTERN

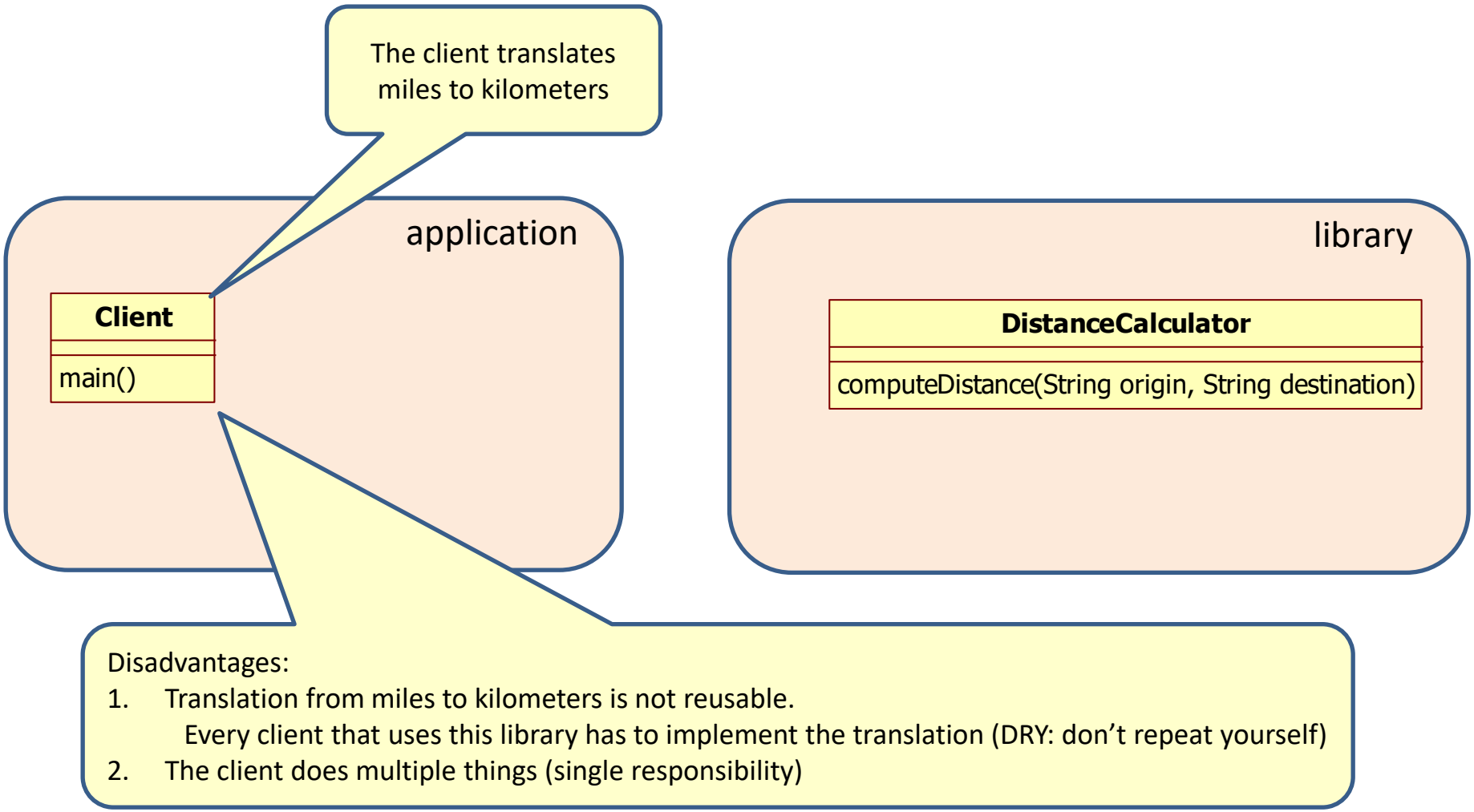
Adapter pattern

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

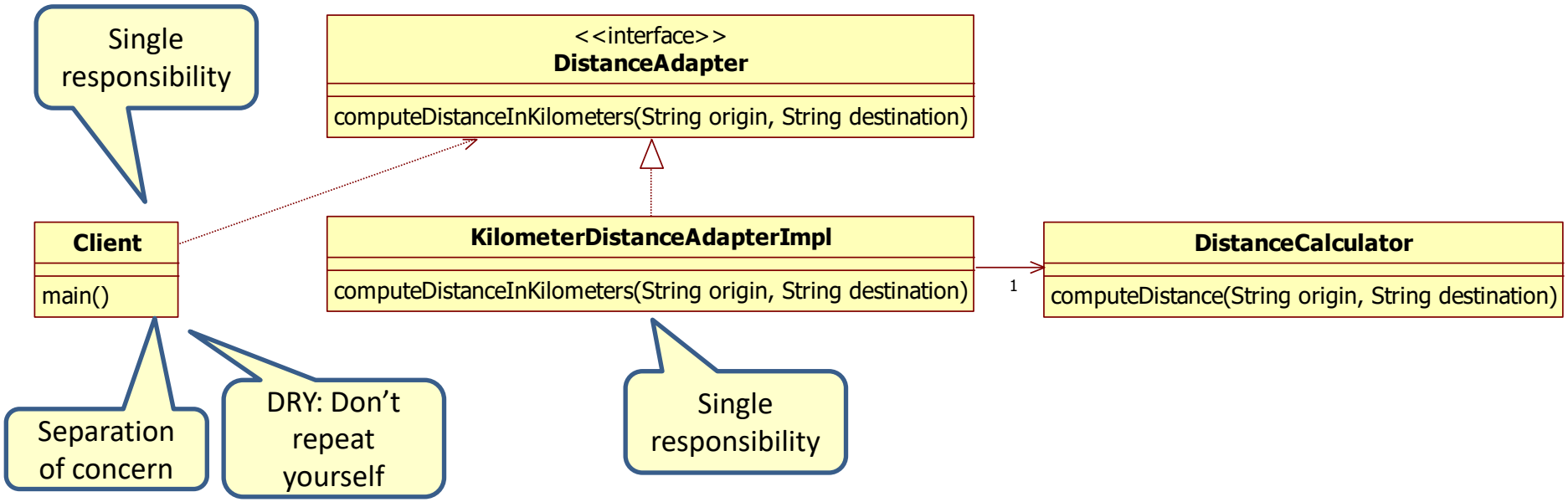
Design problem

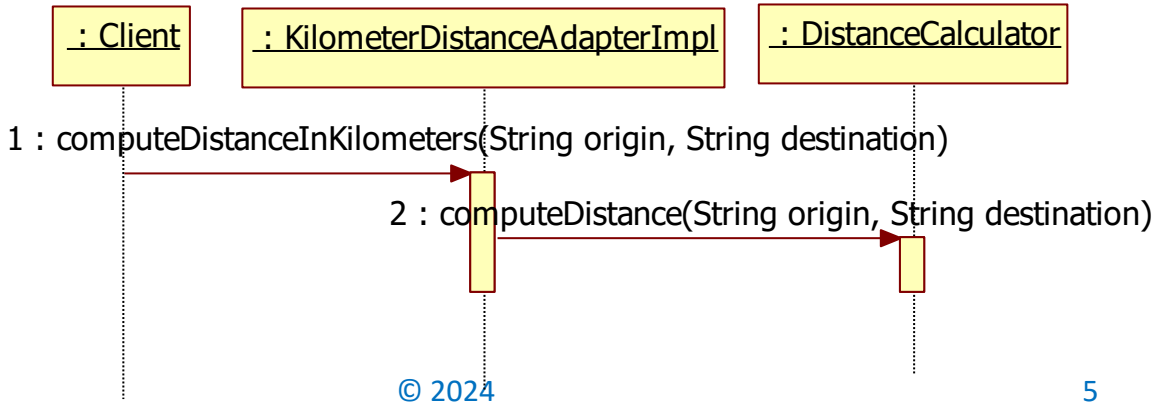


Solution


The client translates miles to kilometers
application
Client
main()
library
DistanceCalculator
computeDistance(String origin, String destination)
Disadvantages:
1. Translation from miles to kilometers is not reusable.
Every client that uses this library has to implement the translation (DRY: don't repeat yourself)
2. The client does multiple things (single responsibility)

Better solution: Adapter





Adapter + Adaptee

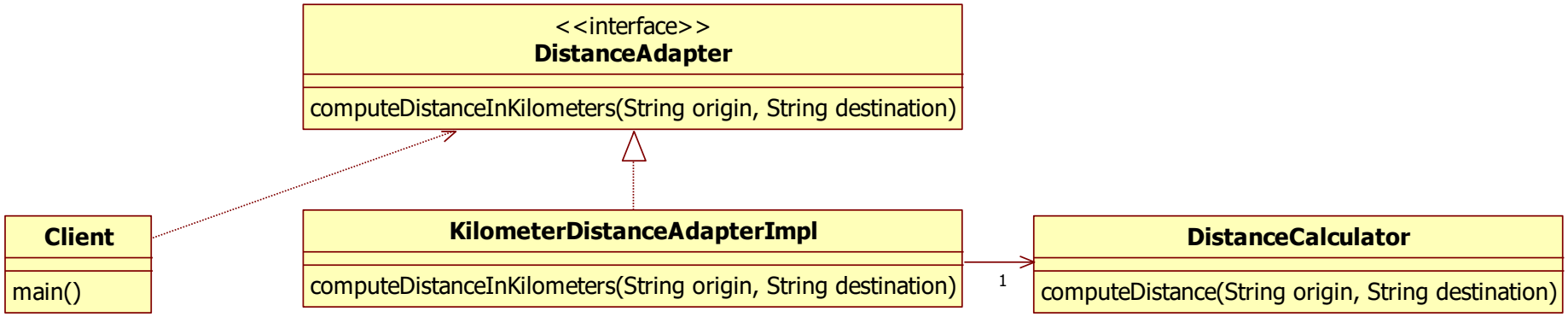
```
public class DistanceCalculator {
  public double computeDistance(String origin, String destination) {
    return (new Random()).nextInt(100);
  }
}
```

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

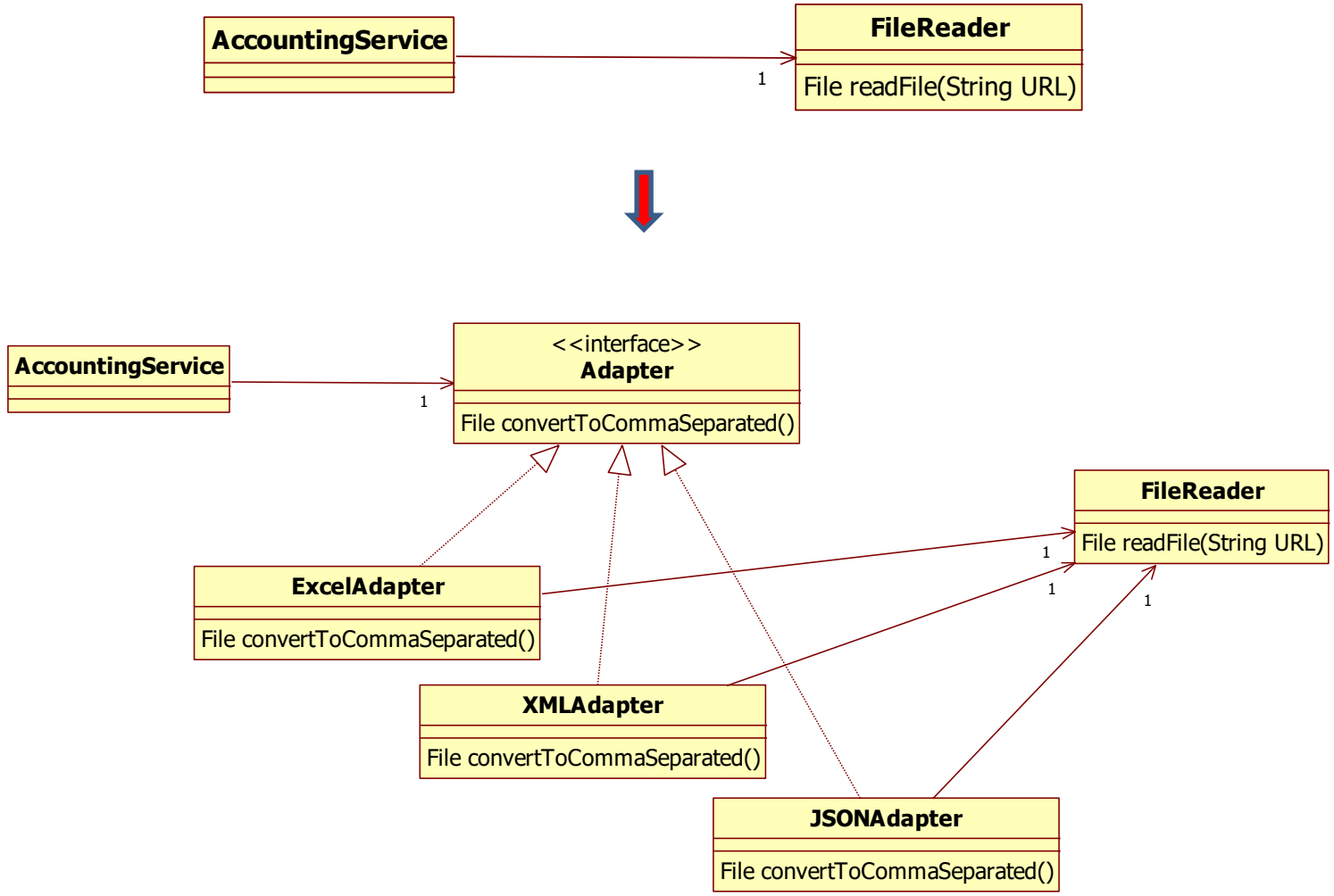
```
public class KilometerDistanceAdapterImpl implements DistanceAdapter {
  private DistanceCalculator distanceCalculator;

  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

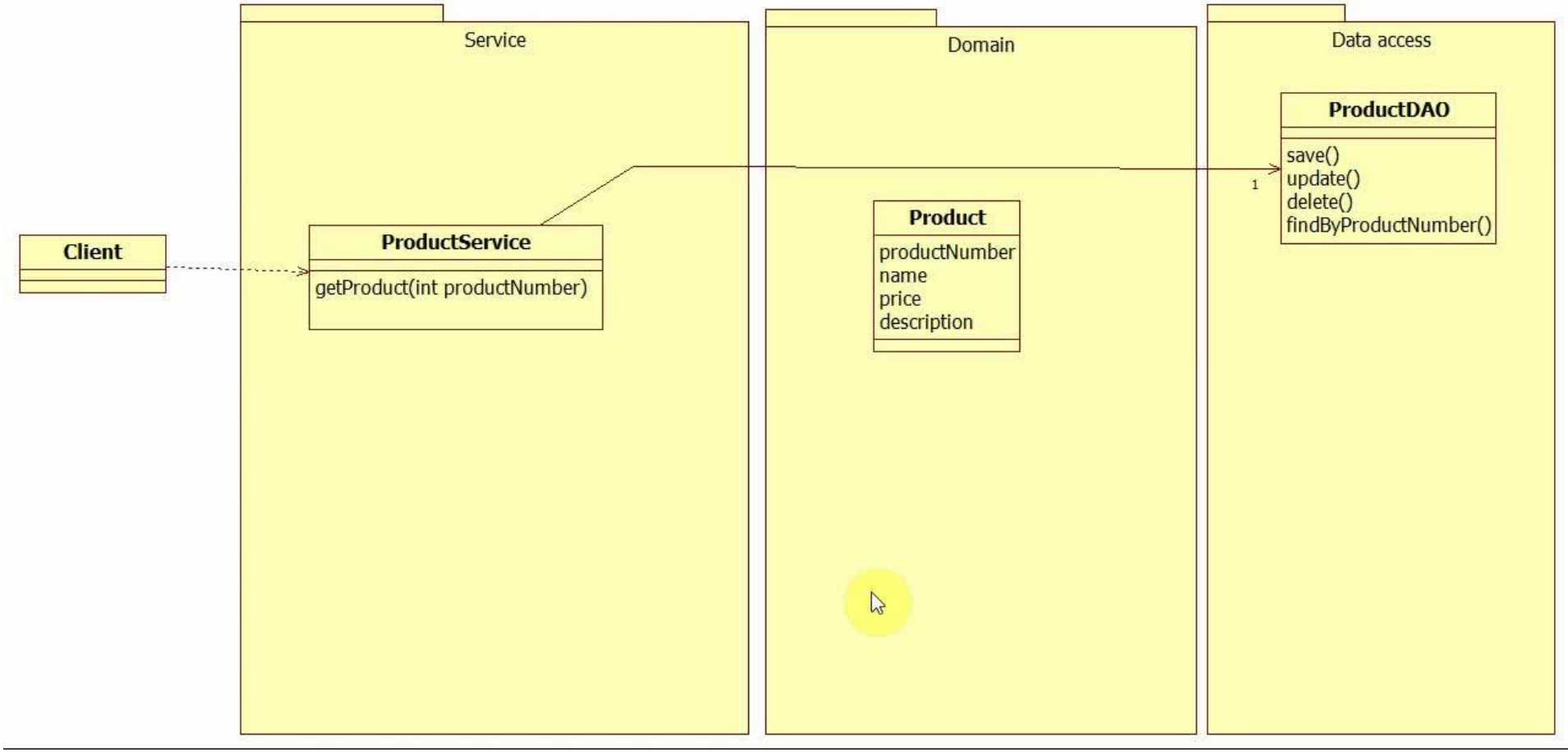
```
public class Client {

  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");
  }
}
```



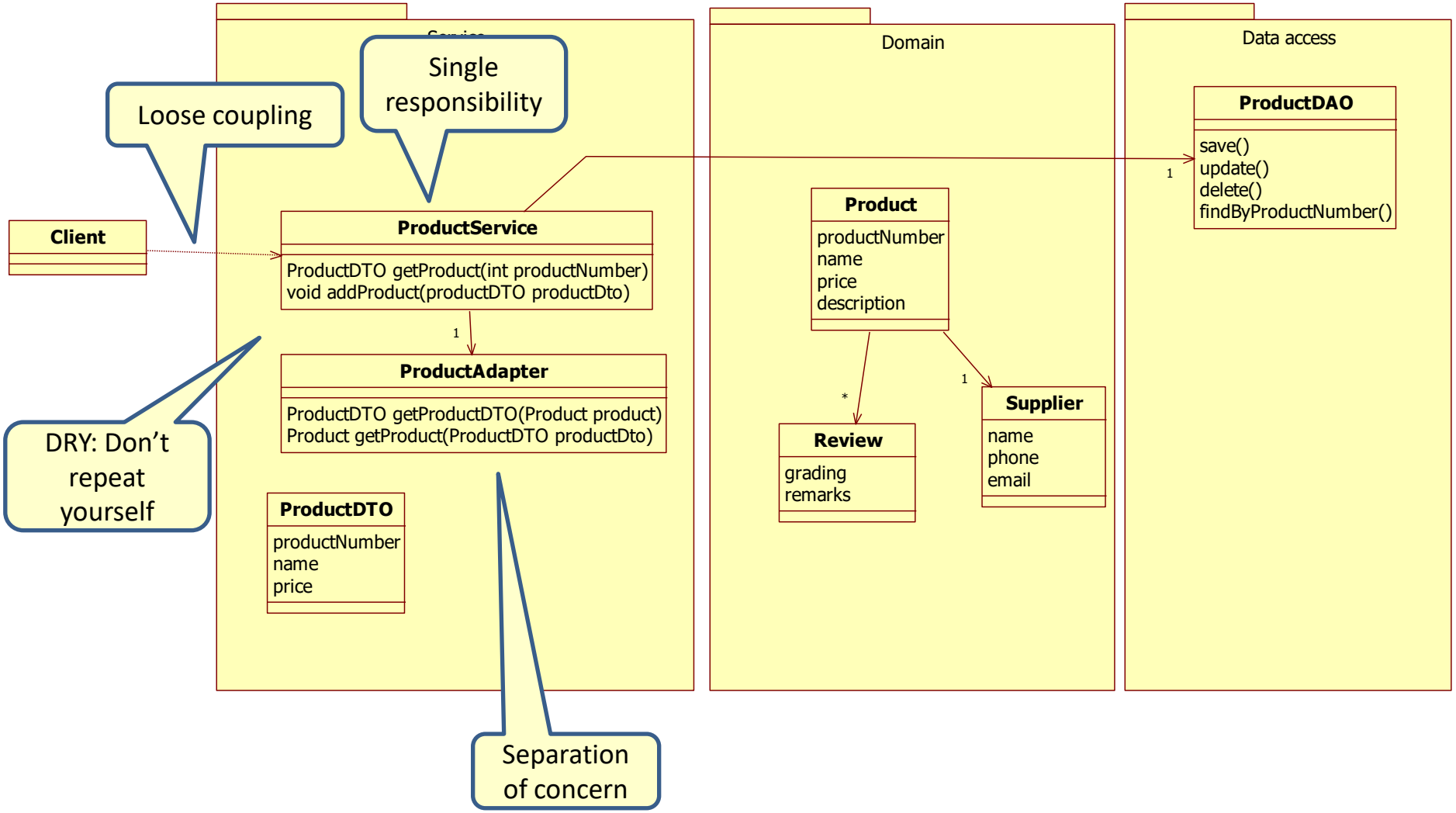
Where are adapters used



Where are adapters used



Where are adapters used



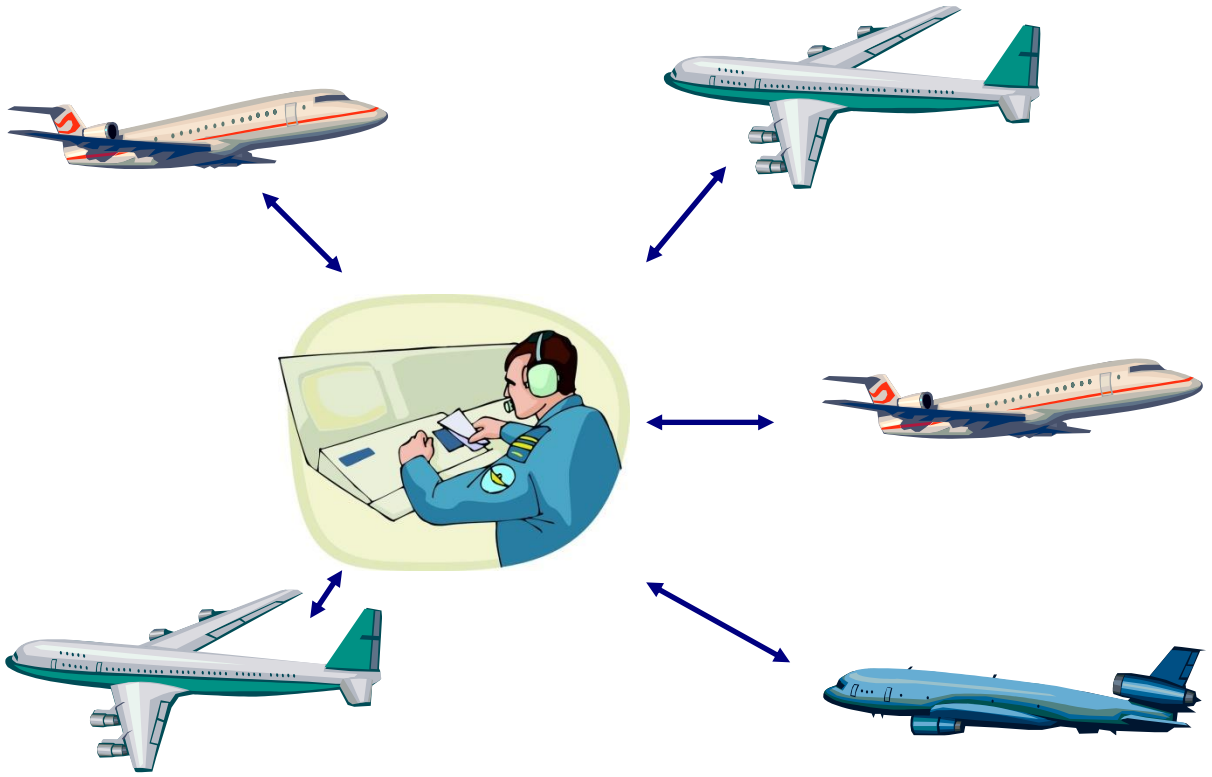
Main point

- 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.

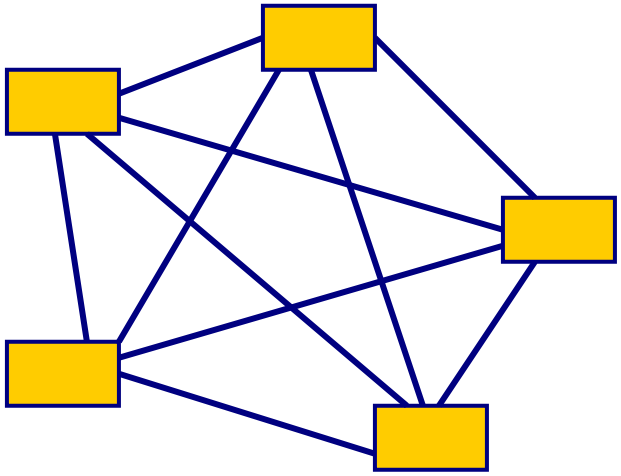
MEDIATOR

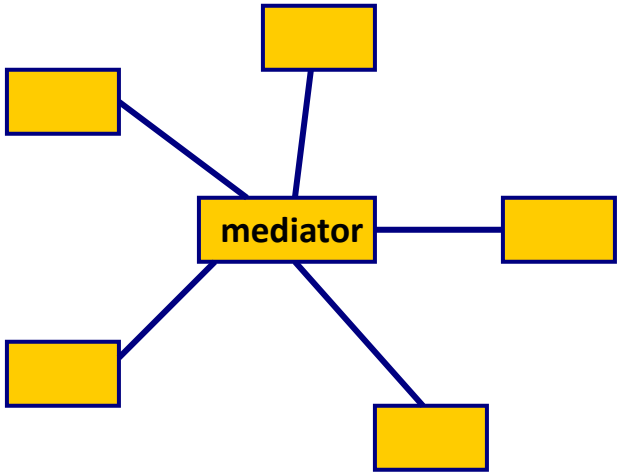
Mediator pattern

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

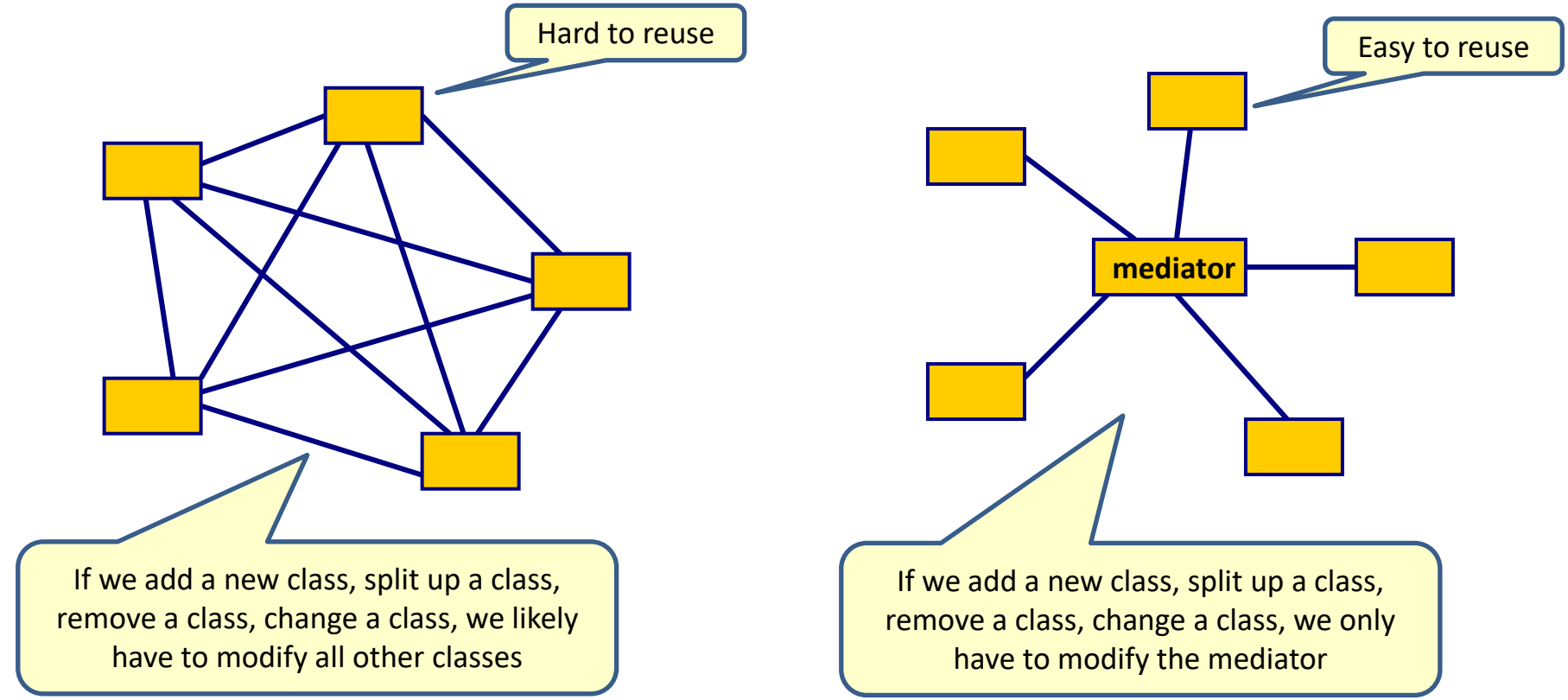


Mediator

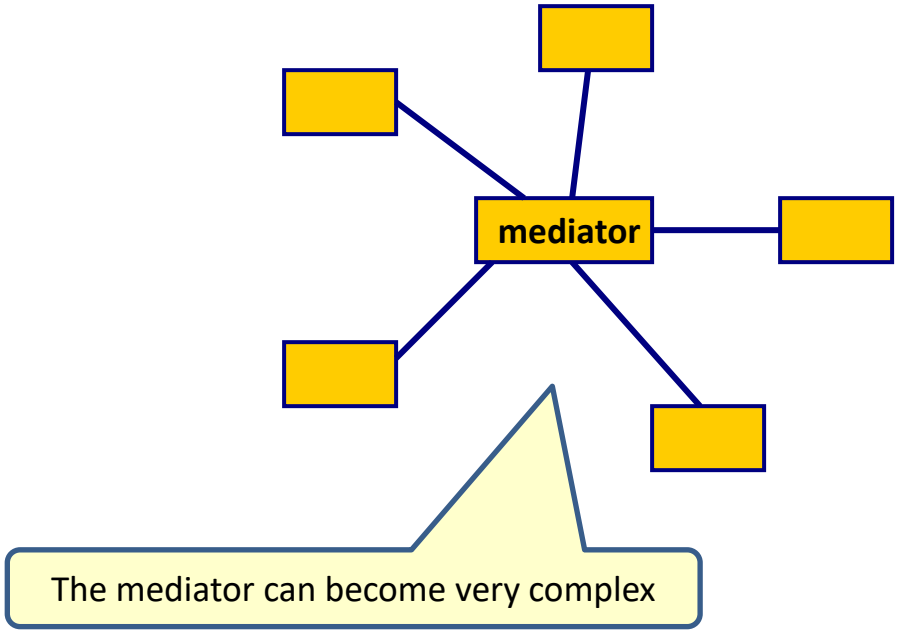




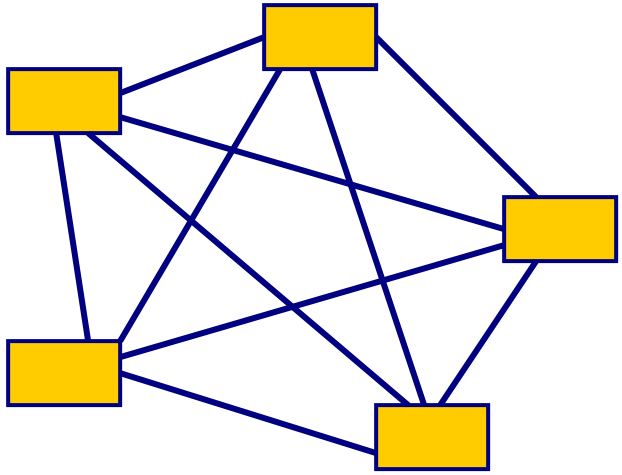
Advantage Mediator: loose coupling



Disadvantage Mediator

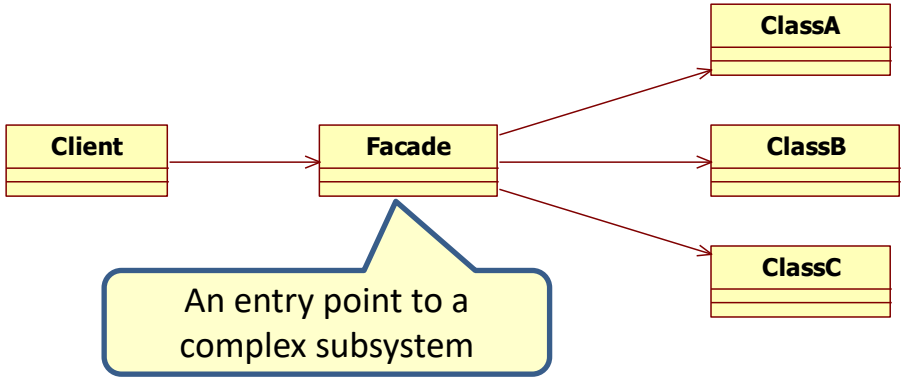


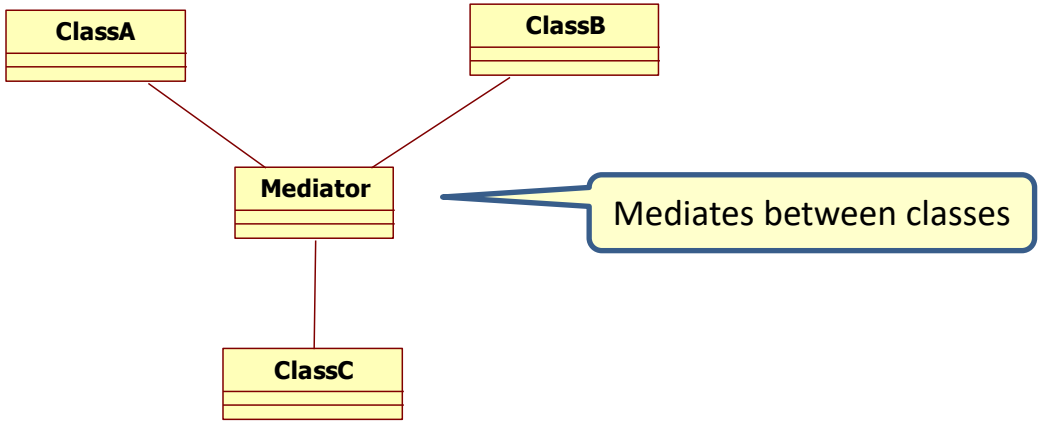
How did we get to this?



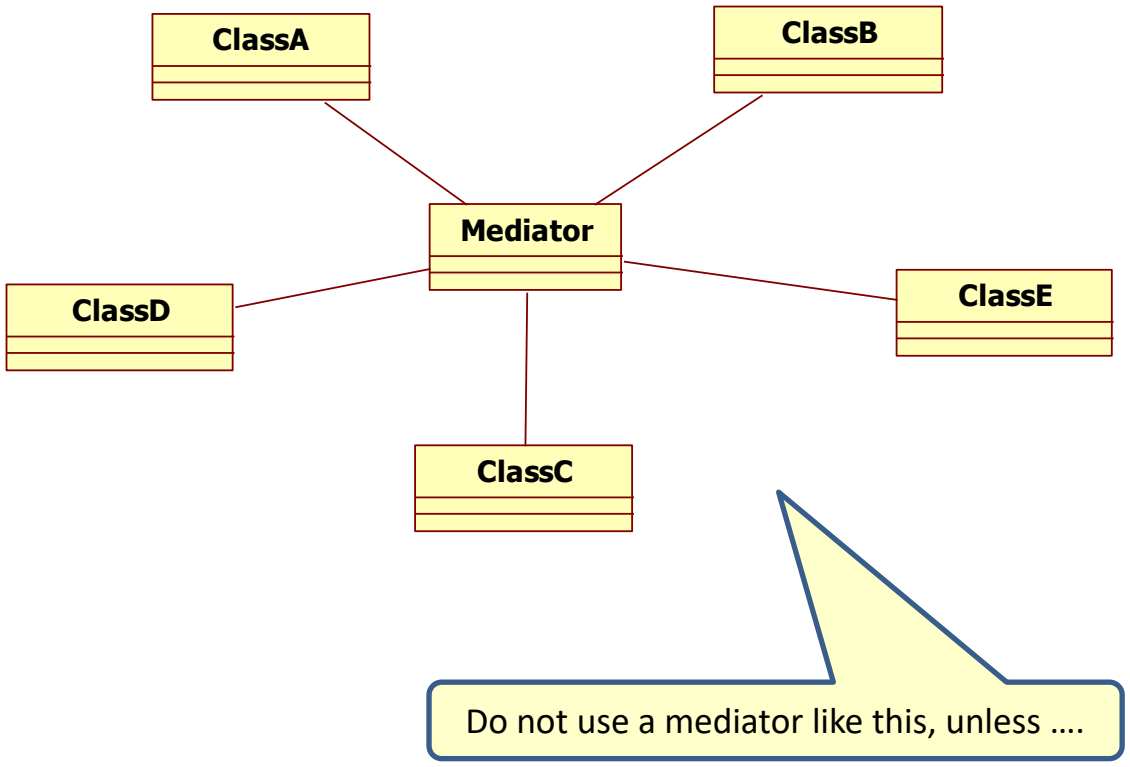
If you end up with a class diagram like this, something went wrong.
This is not good OO design!

Façade and Mediator



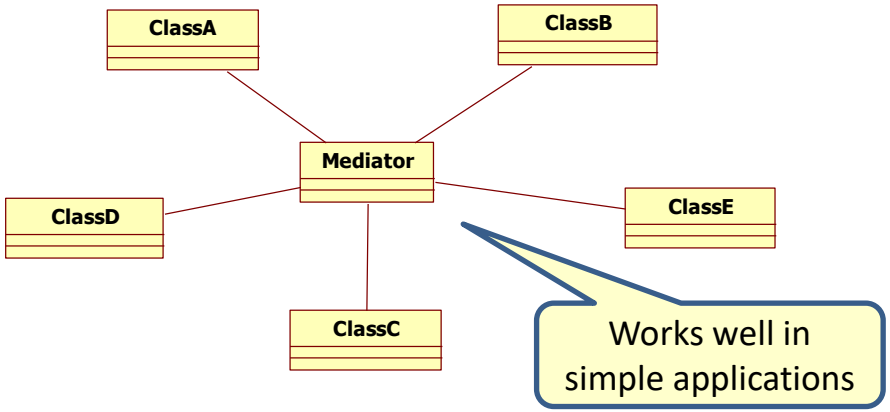


Examples of the mediator

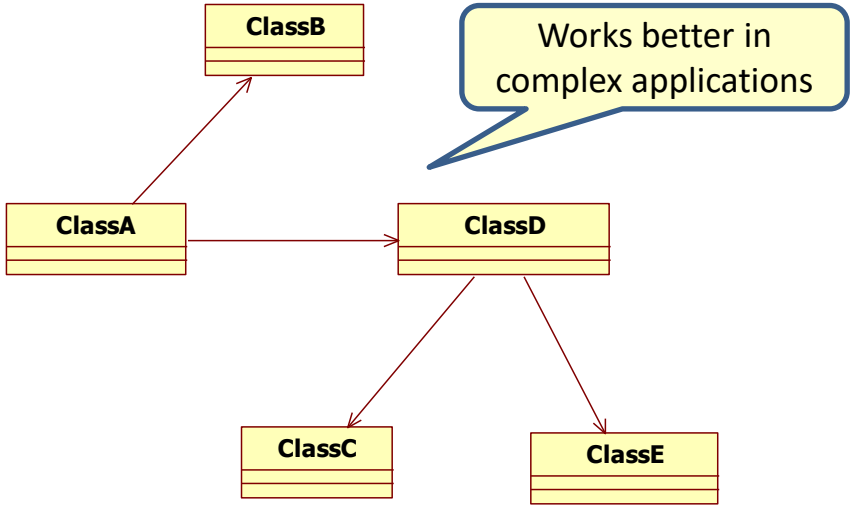


Orchestration vs. choreography

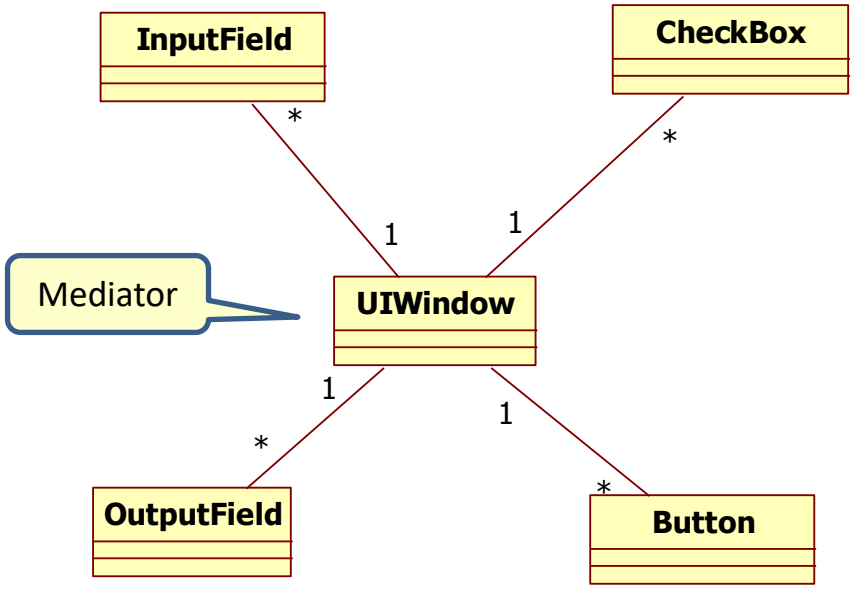
- Orchestration
  - One central brain

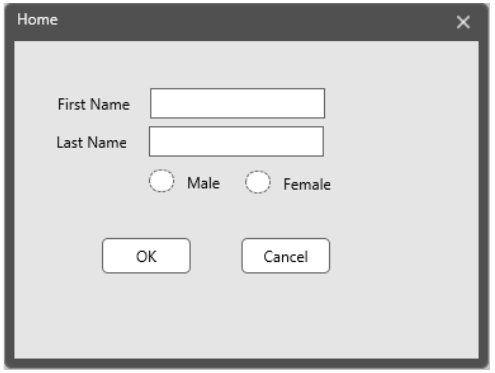


- Choreography
  - No central brain



Examples of the mediator





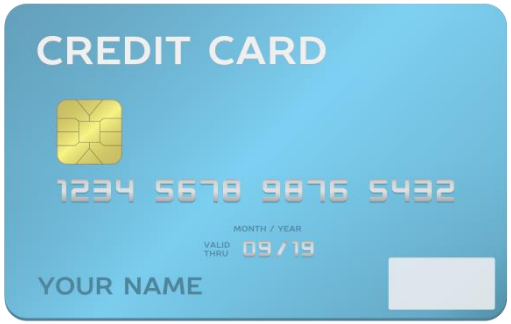
Main point

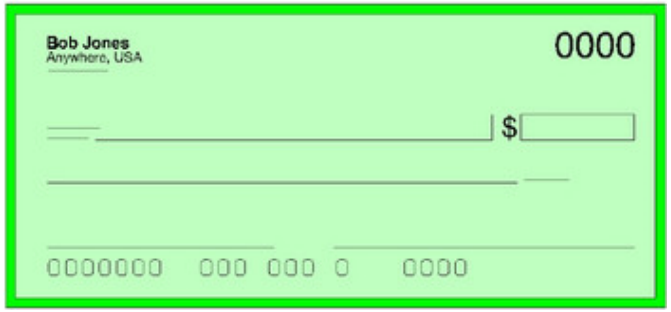
- The Mediator object is responsible for controlling and coordinating the interactions of a group of objects.
- The Unified Field is the source of creation that coordinates all interactions in the relative world.

PROXY

Proxy pattern

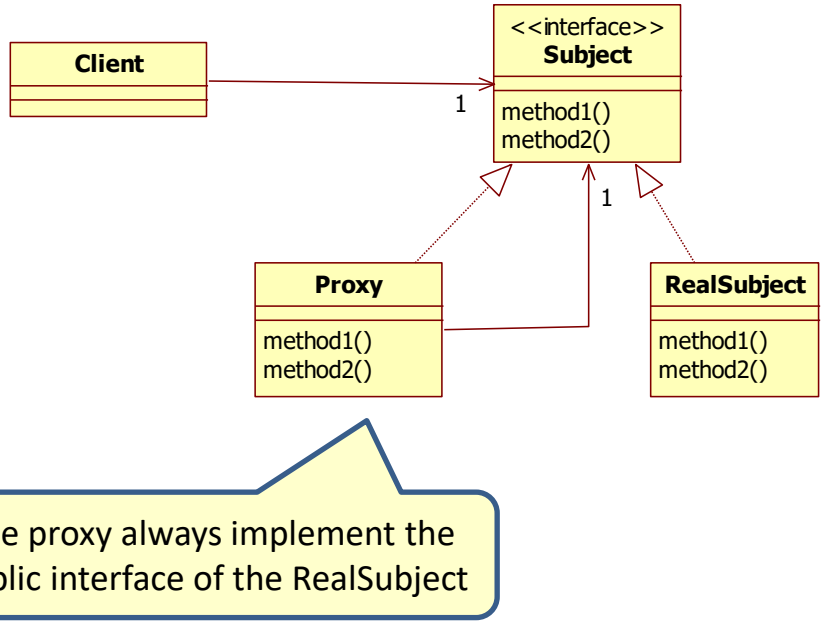
- Provides a surrogate or placeholder for another object.

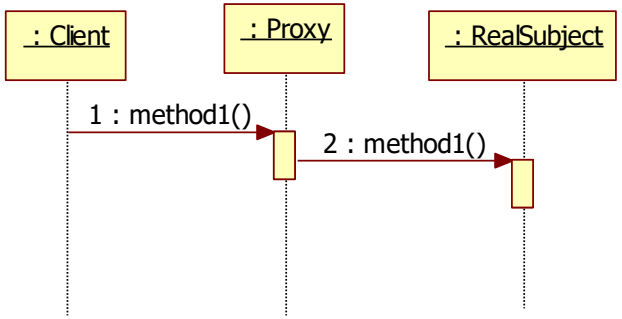




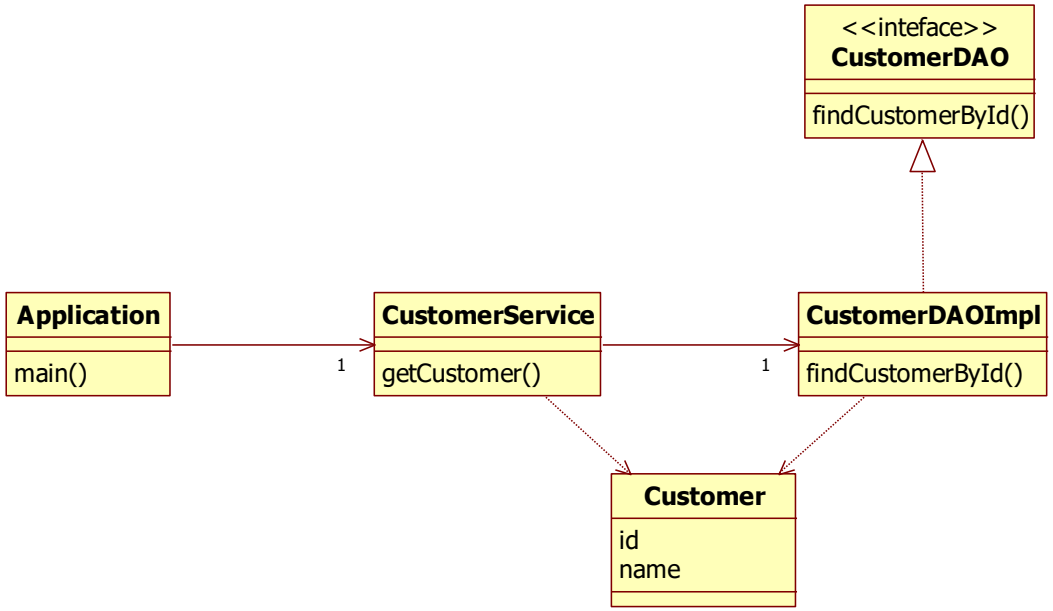
Proxy pattern

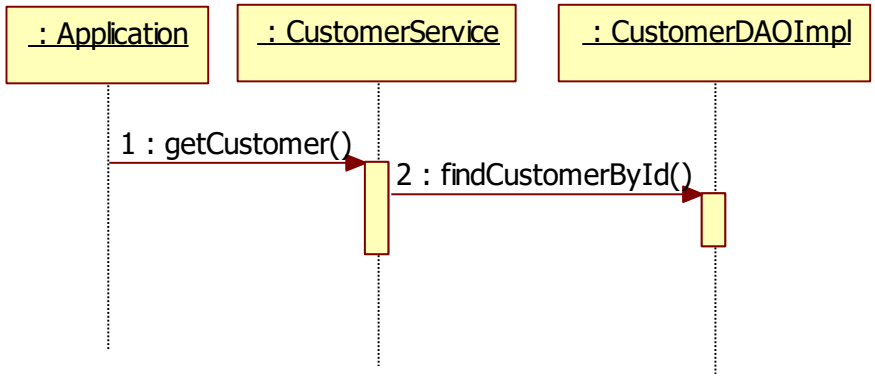
- Remote proxy
- Caching proxy
- Synchronization proxy
- Security proxy
- Transactional proxy
- Lazy load proxy
- Logging proxy
- …





Example without proxy

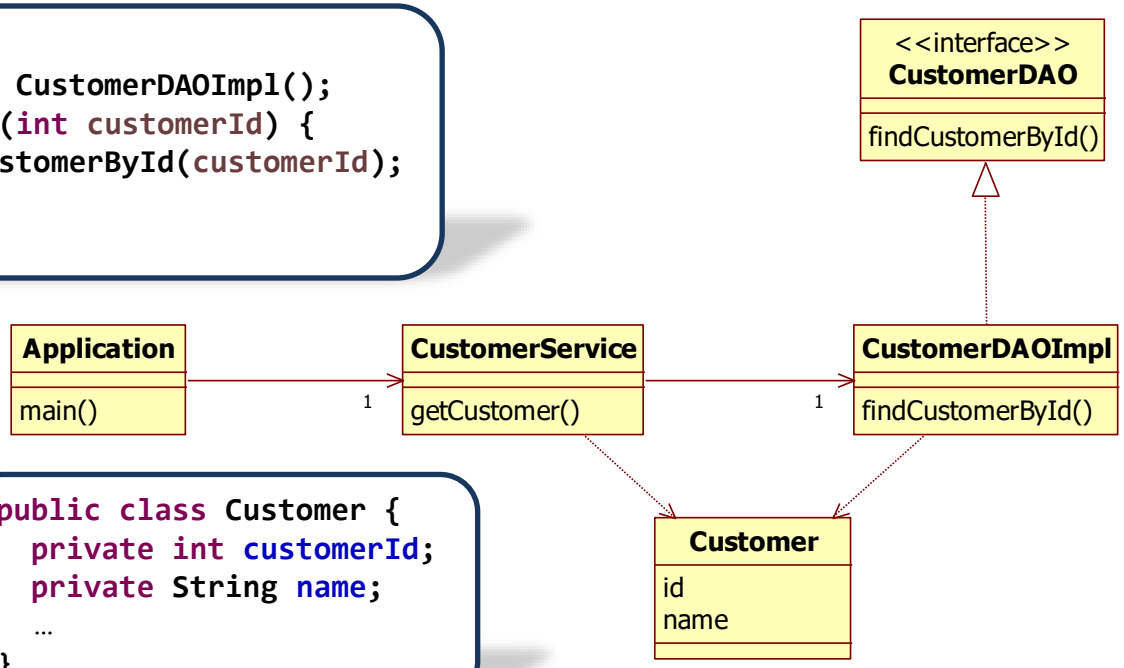




Example without proxy

```
public interface CustomerDAO {
  Customer findCustomerById(int customerId);
}
```

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

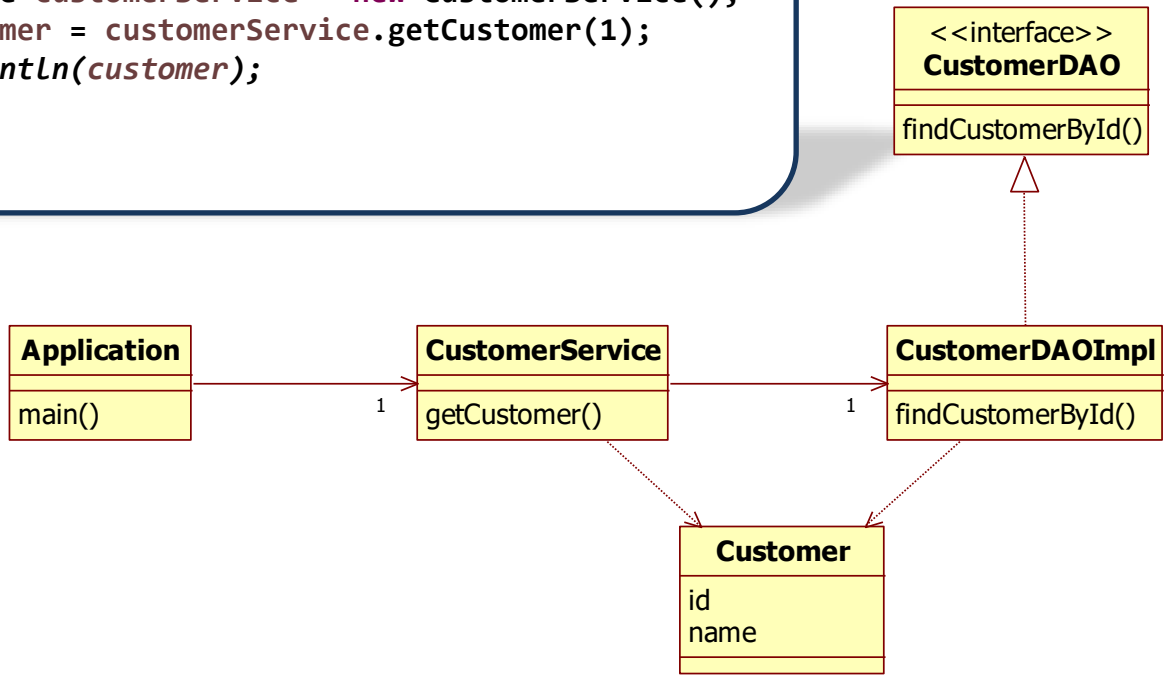



```
public class Customer {
  private int customerId;
  private String name;
  ...
}
```

```
public class CustomerDAOImpl implements CustomerDAO{
  public Customer findCustomerById(int customerId) {
    return new Customer(customerId, "Frank Brown");
  }
}
```

Example without proxy: application

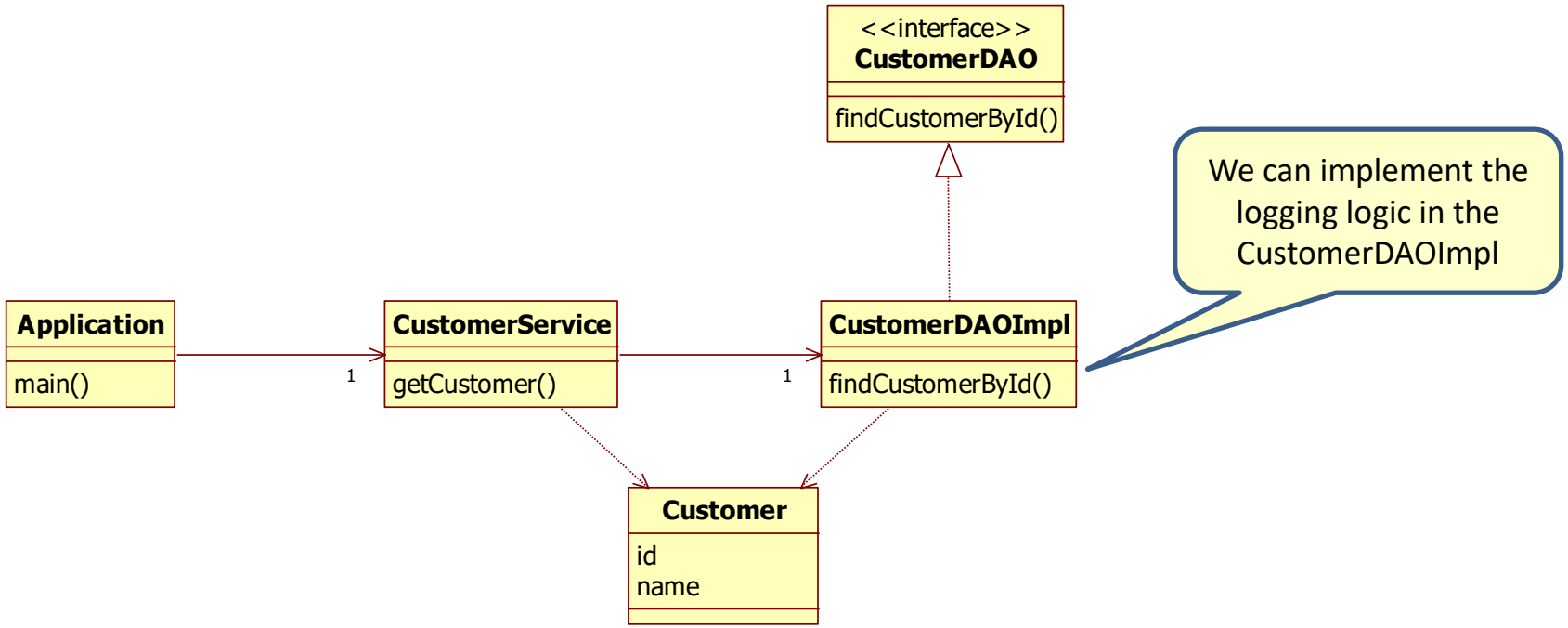
```
public class Application {
  public static void main(String[] args) {
    CustomerService customerService = new CustomerService();
    Customer customer = customerService.getCustomer(1);
    System.out.println(customer);
  }
}
```




```
Customer [customerId=1, name=Frank Brown]
```

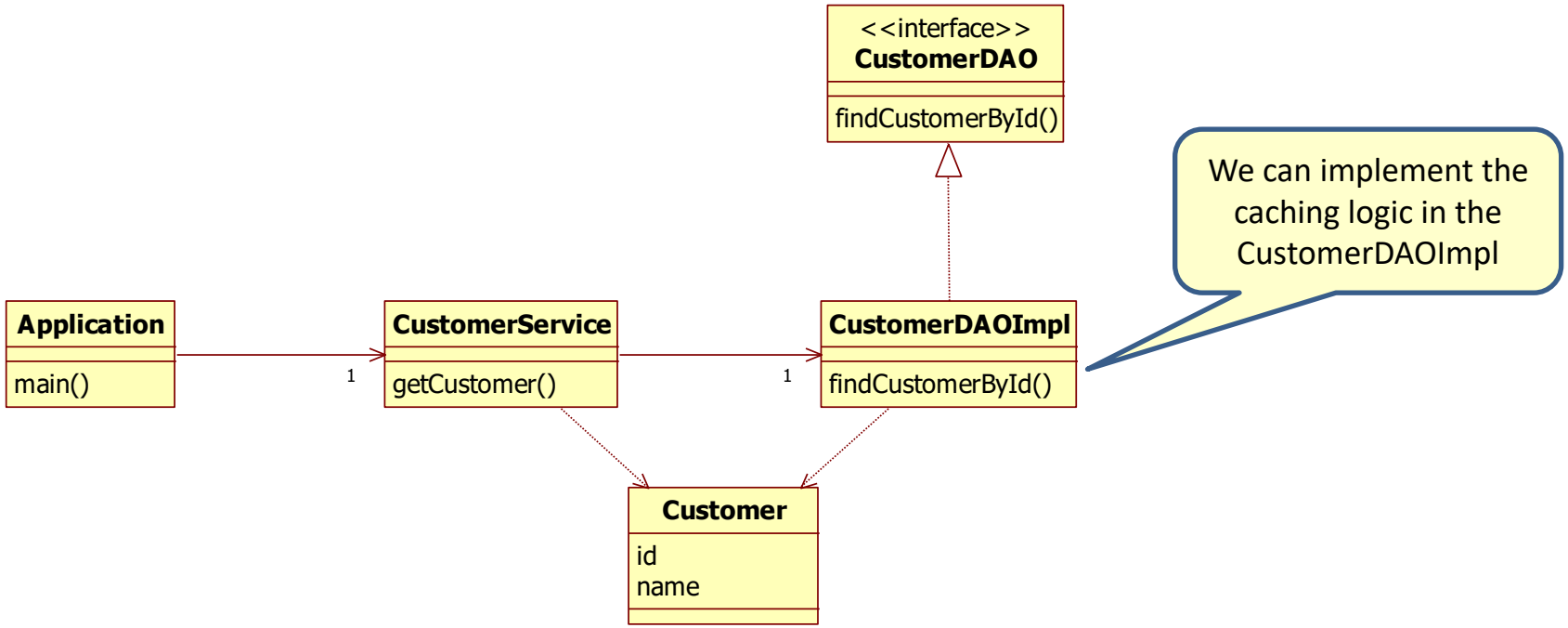
New requirement: logging

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



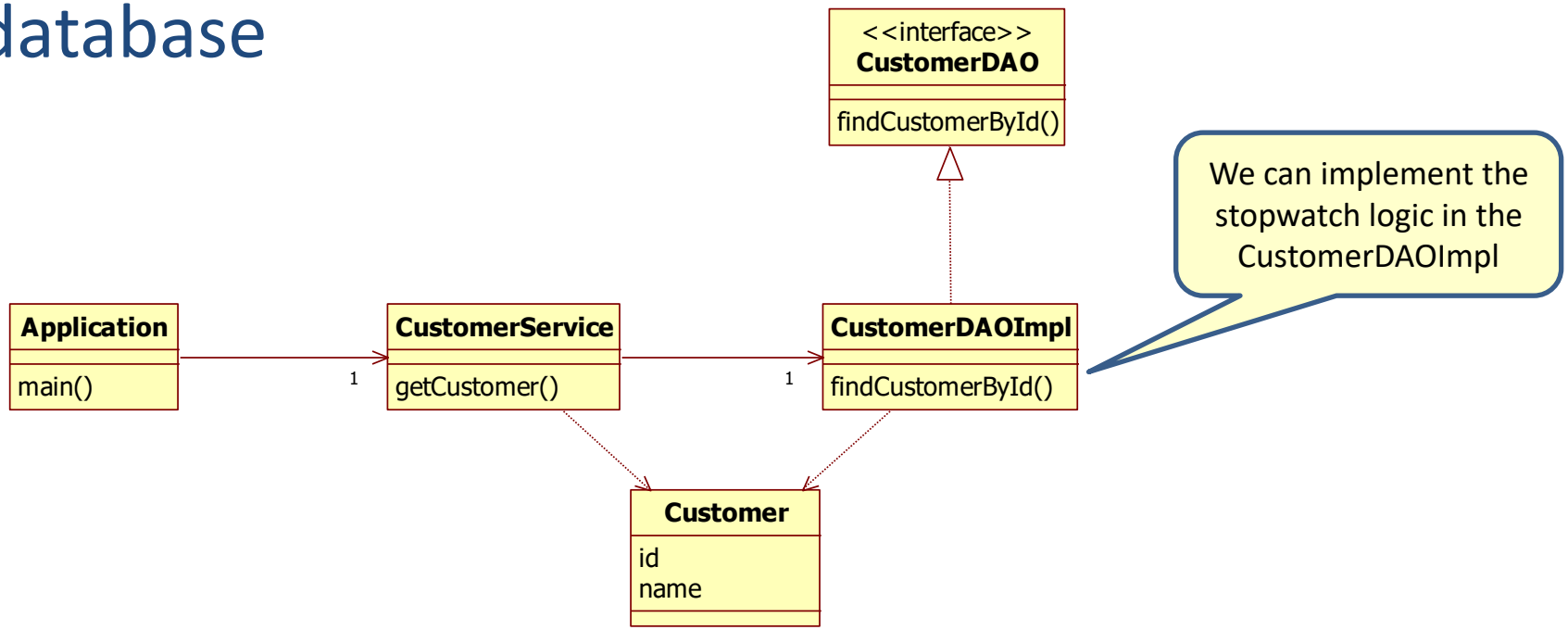
New requirement: caching

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



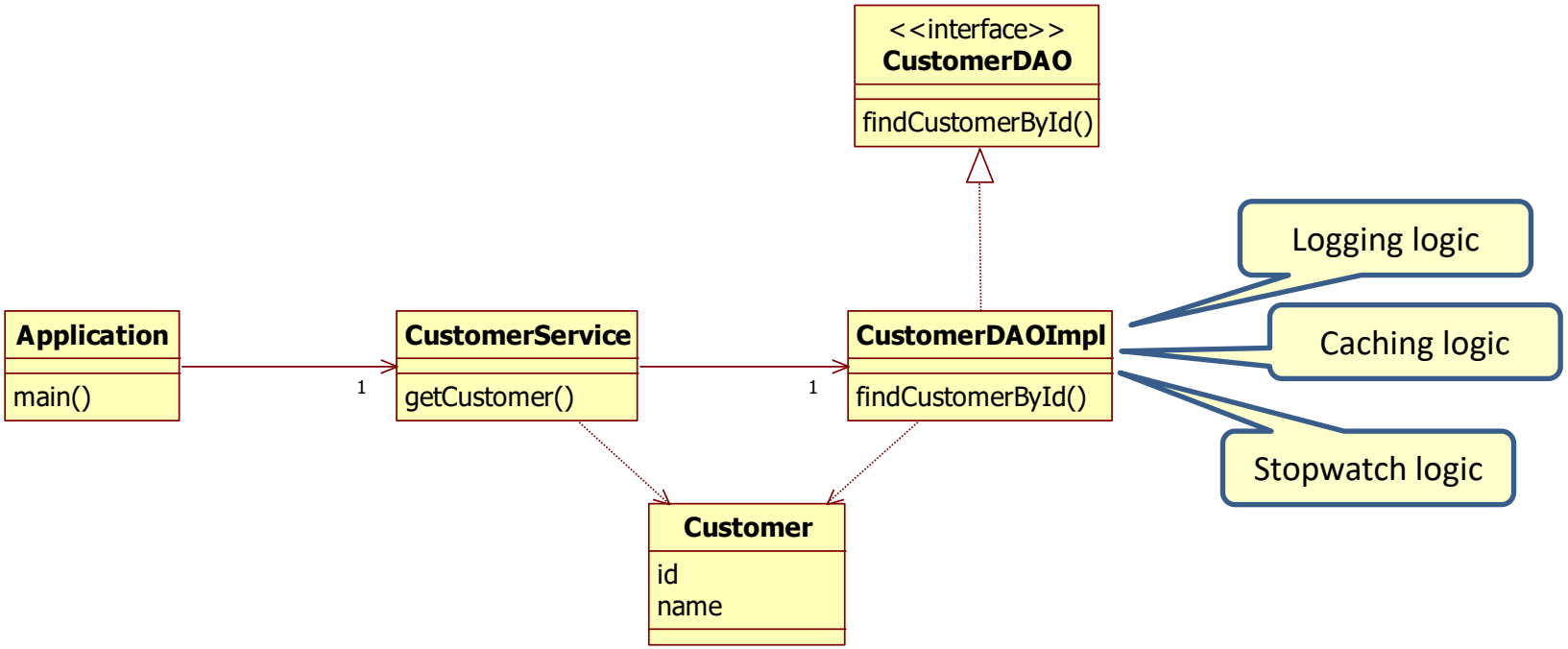
New requirement: time measurement

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



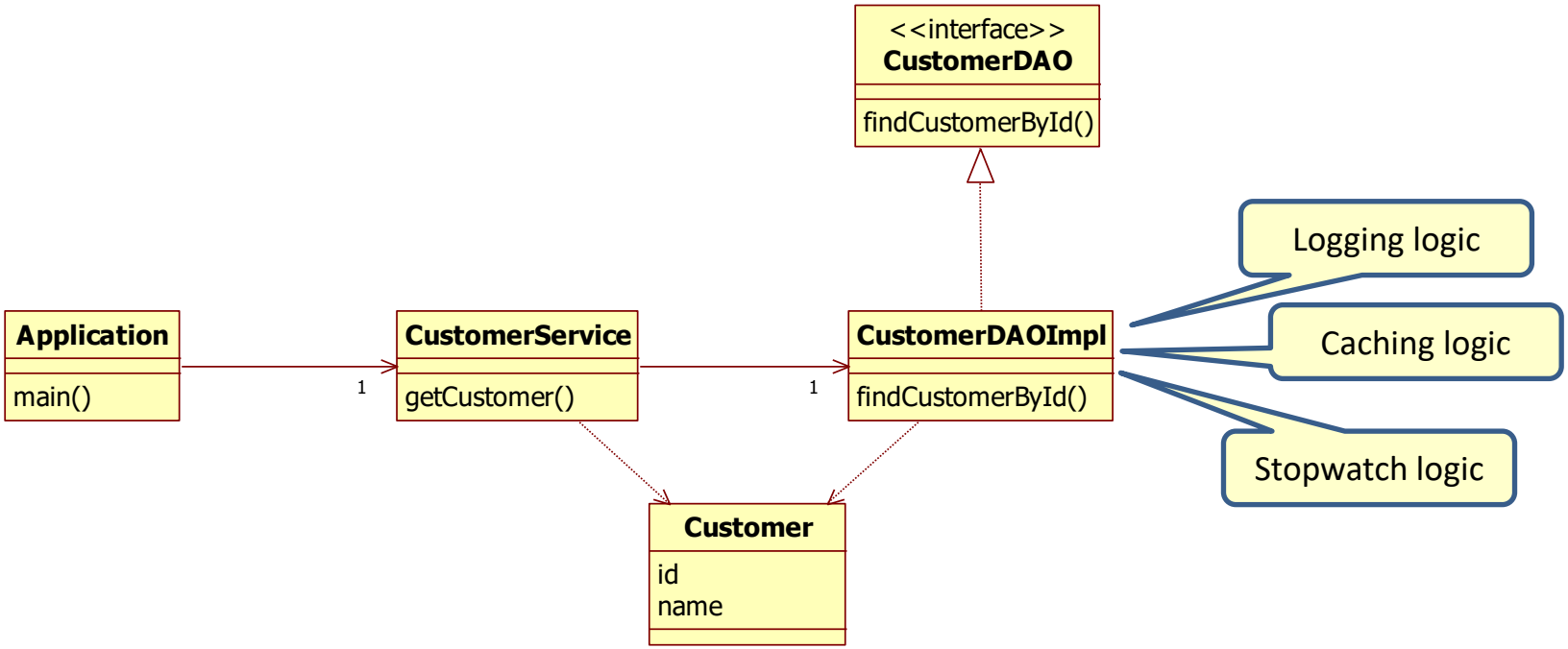
Single responsibility

- A class has one reason to change

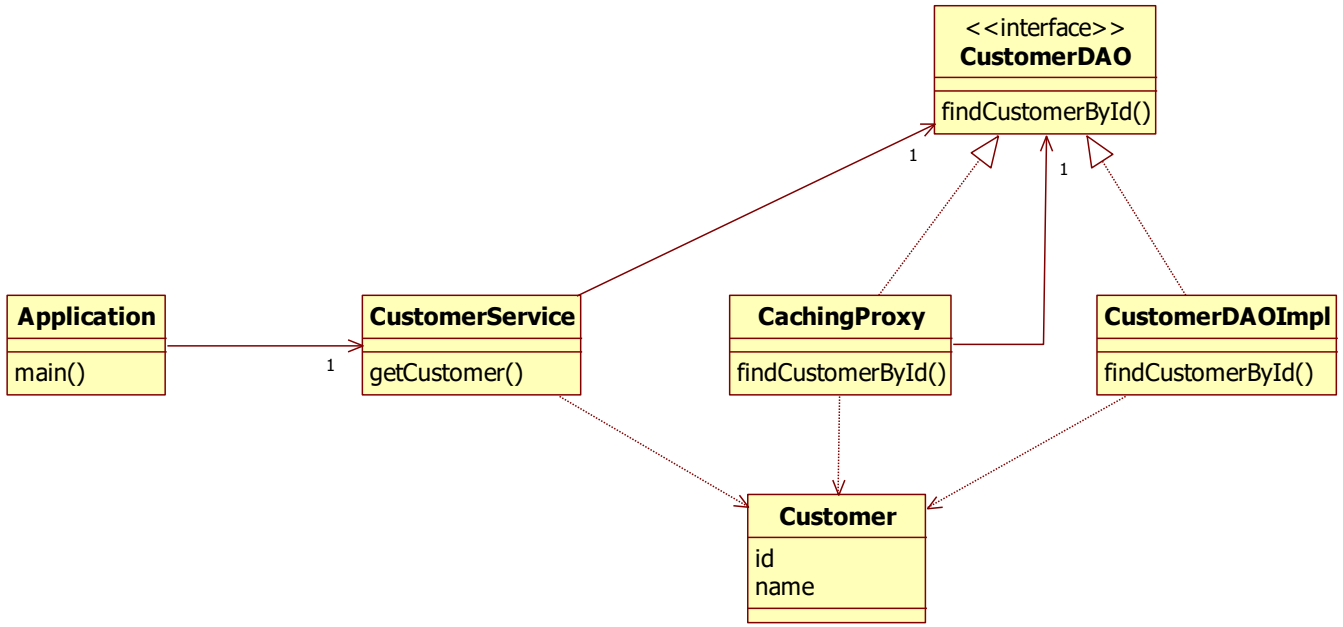


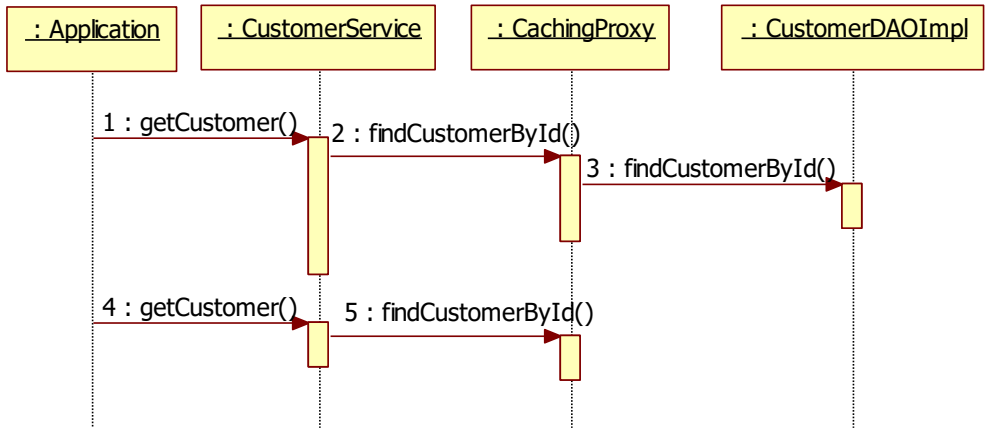
Open-closed principle

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



Caching proxy



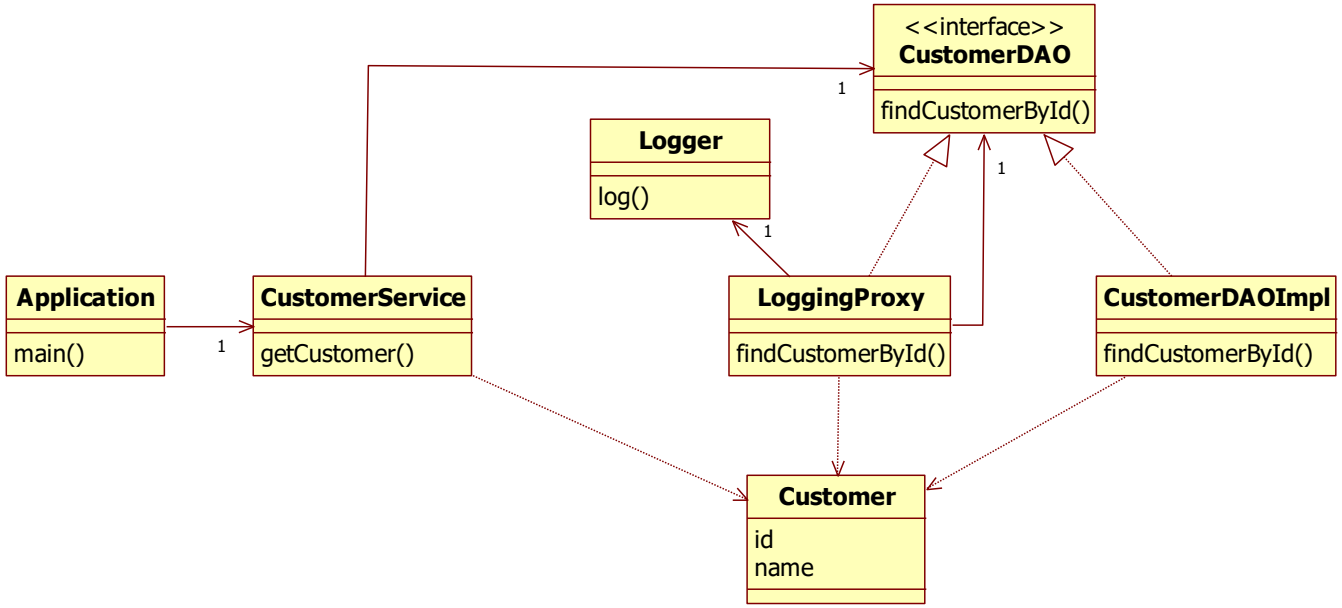


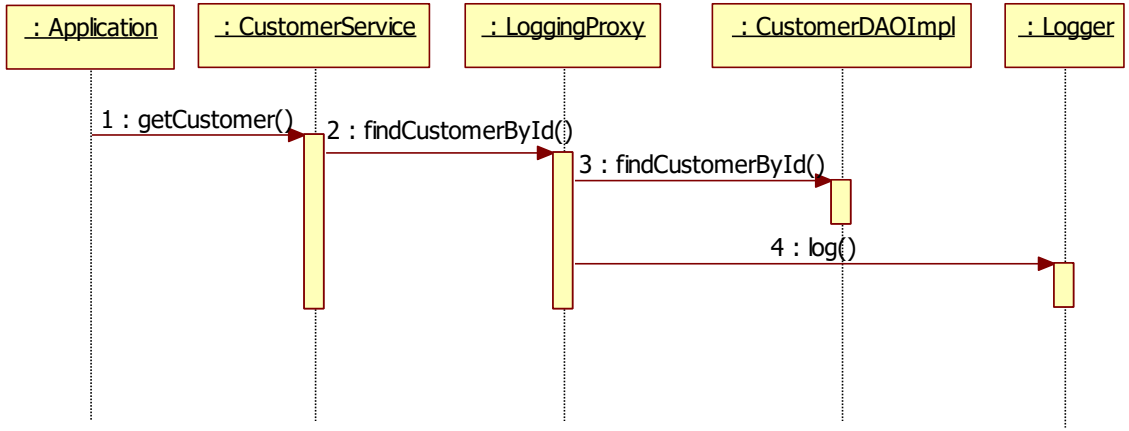
Caching proxy code

```
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





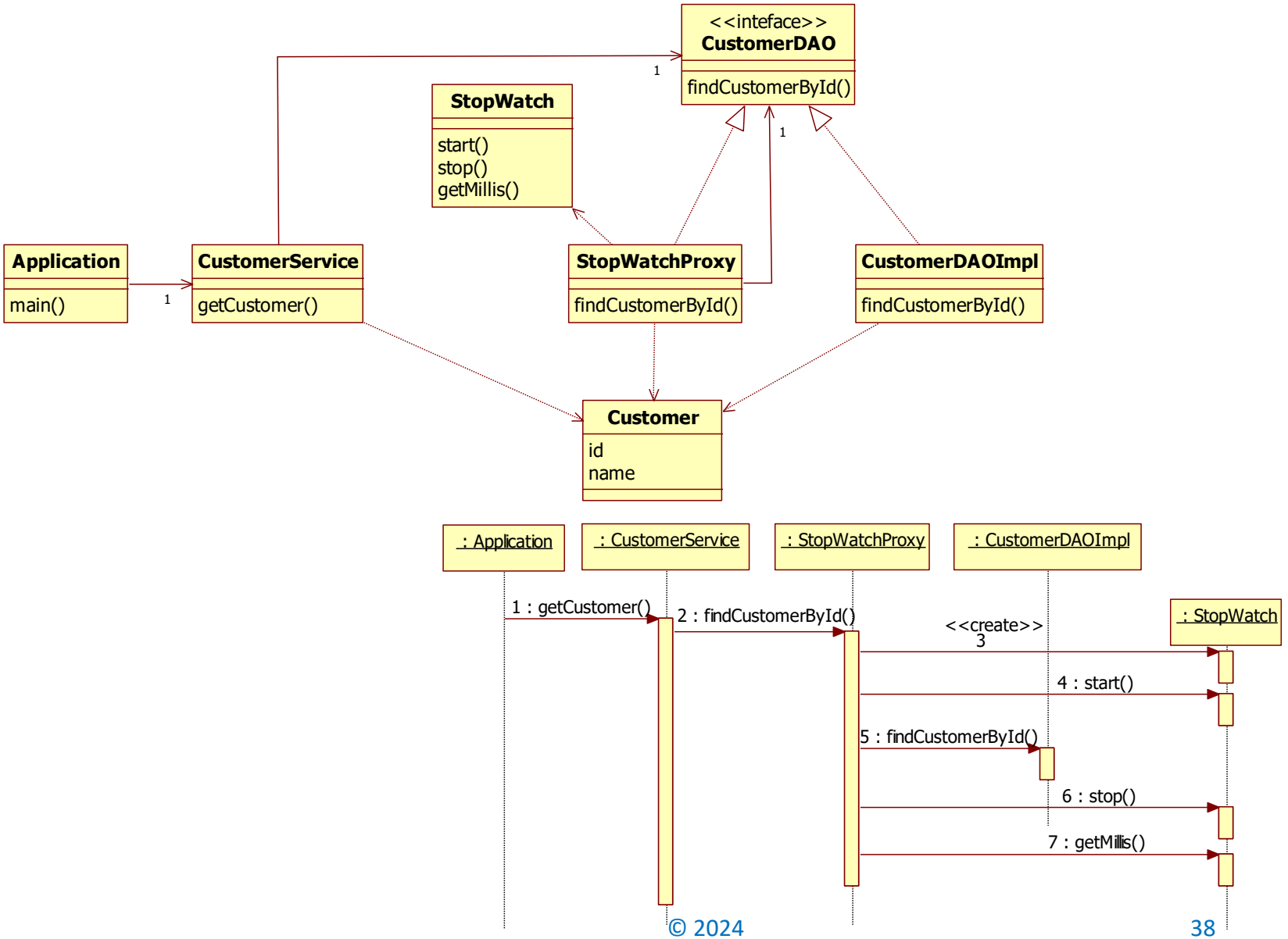
Logging proxy code

```
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

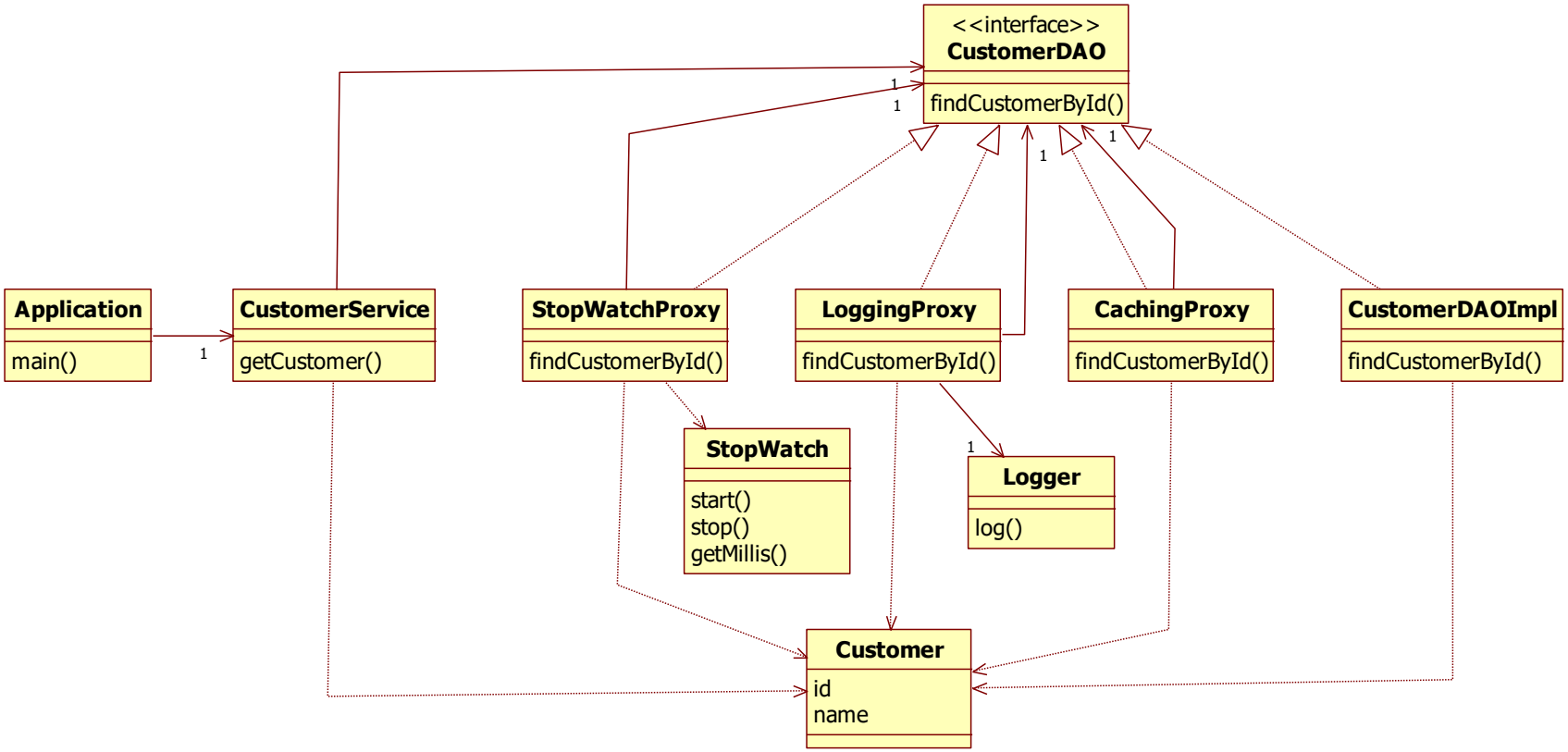
```
public class CustomerService {
  CustomerDAO customerDAO = new CustomerDAOImpl();
  CustomerDAO stopWatchProxy = new StopWatchProxy(customerDAO);

  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;
  }
}
```

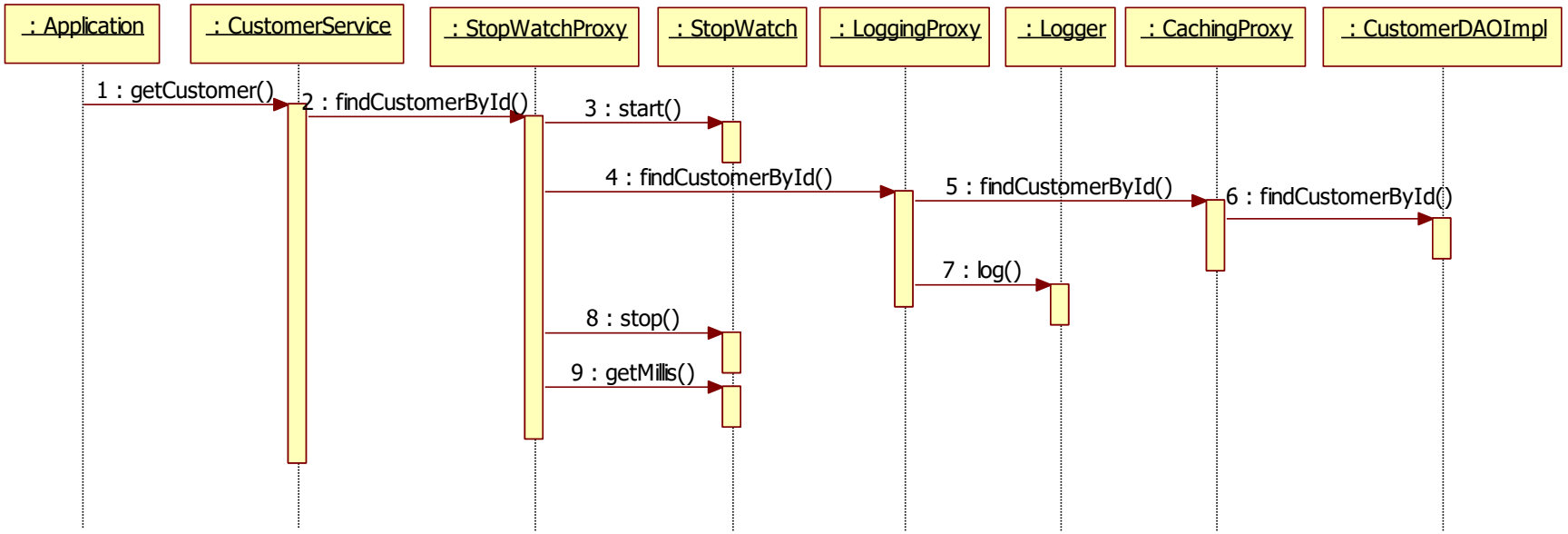
StopWatch code

```
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

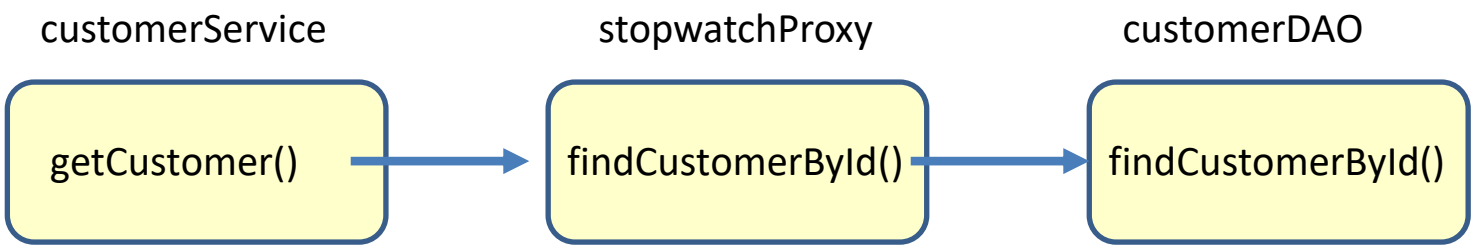
```
import java.lang.reflect.*;

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

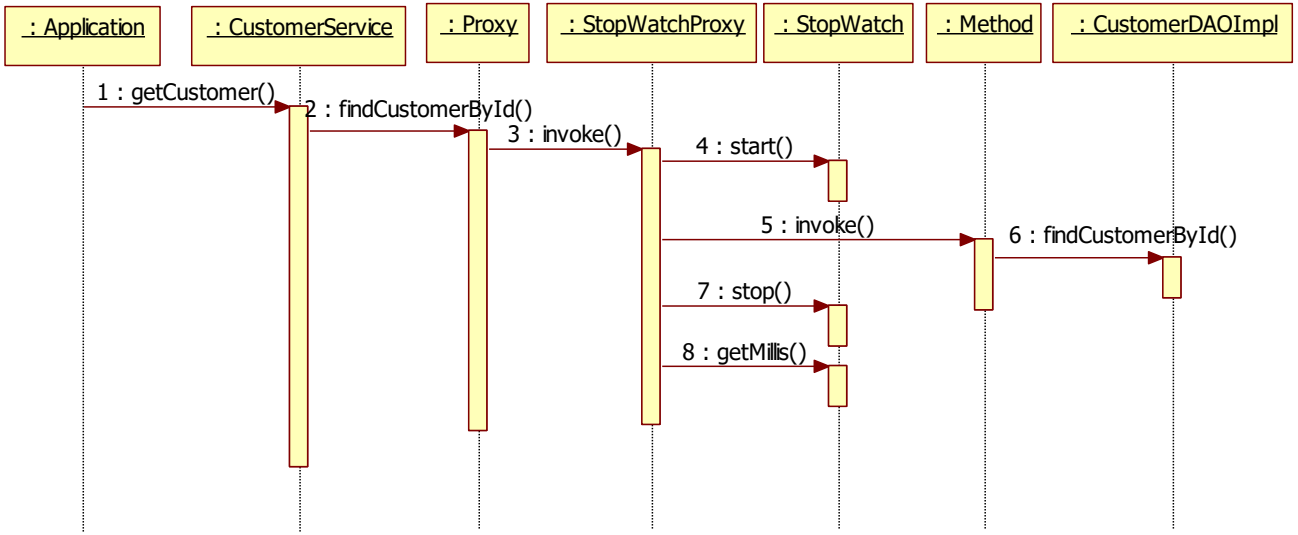
```
import java.lang.reflect.Proxy;

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.findCustomerById(customerId);
    }
}
```

Create a Proxy



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

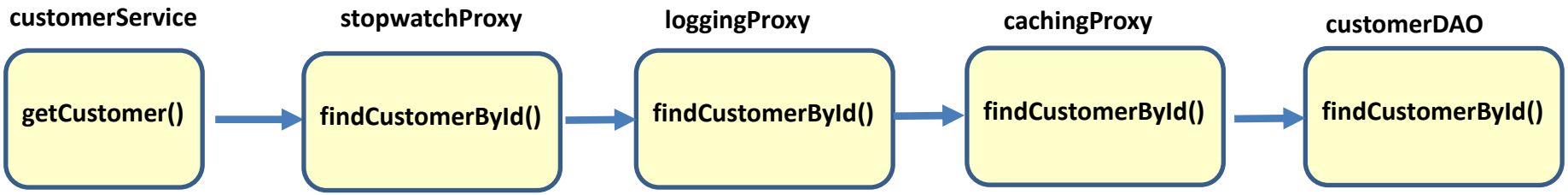
```
public class LoggingProxy implements InvocationHandler {
  private Object target;
  Logger logger = new Logger();

  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;
  }
}
```

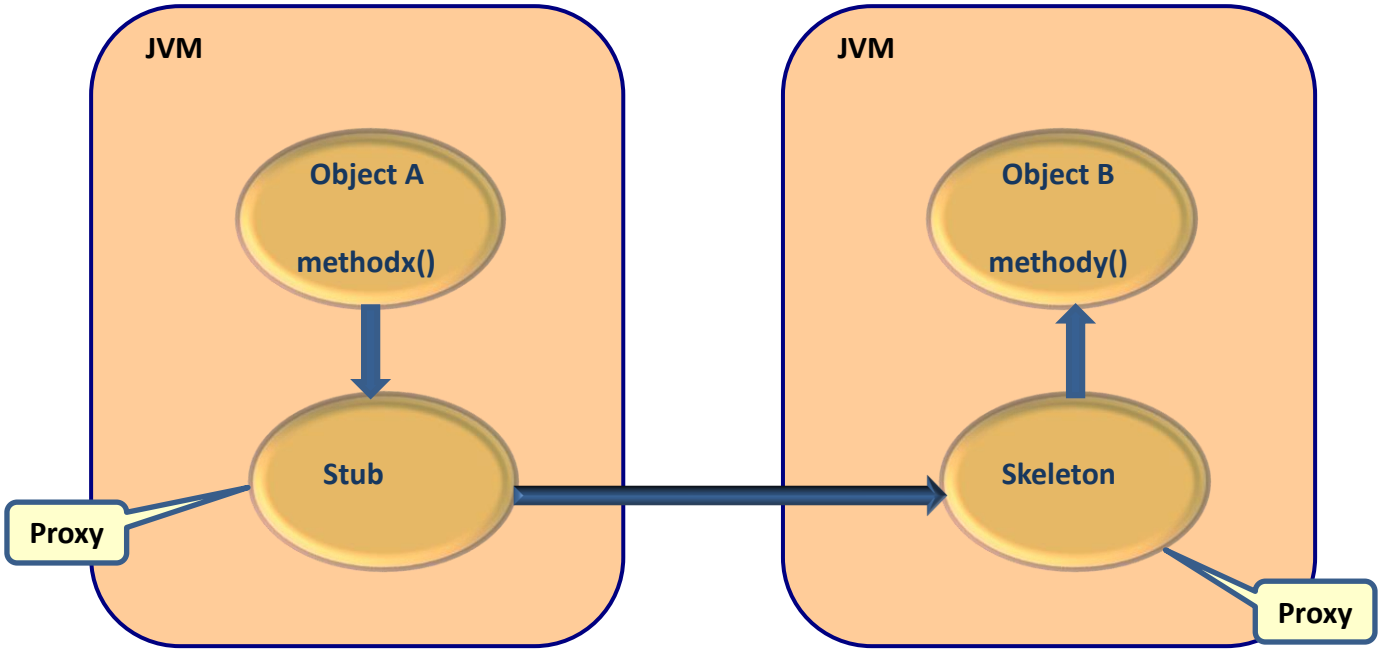
Nested dynamic proxies

```
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);
  }
}
```



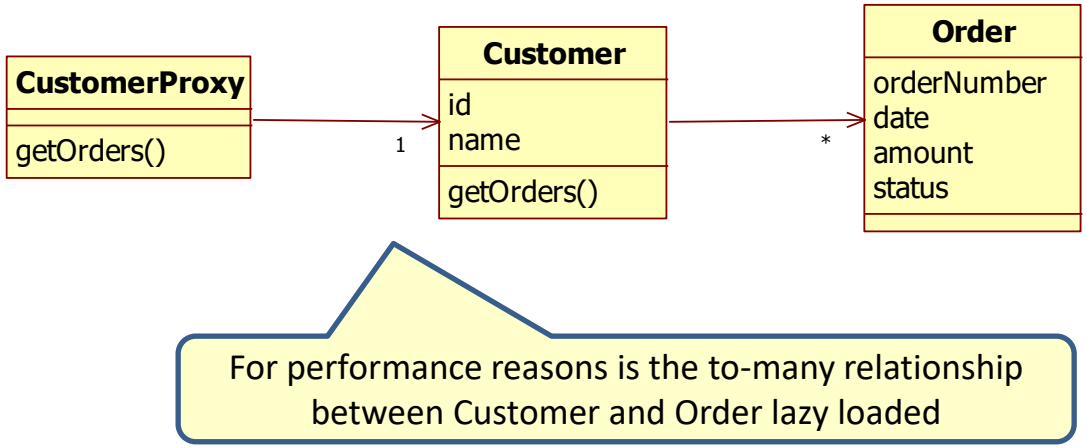
Where are proxies used: RPC

- Remote Procedure Calls
  - Remote Method Invocation (RMI)



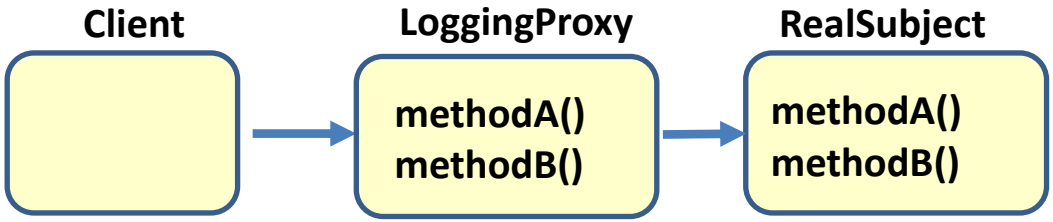
Where are proxies used: Hibernate

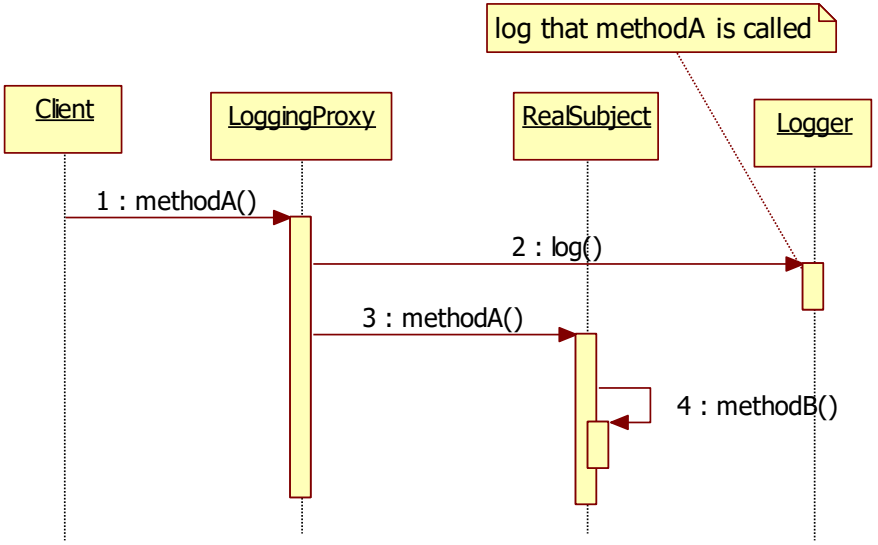
- Hibernate is an Object Relational Mapper (ORM) framework used for persisting objects
- It uses lazy loading using proxies



Issue with a proxy

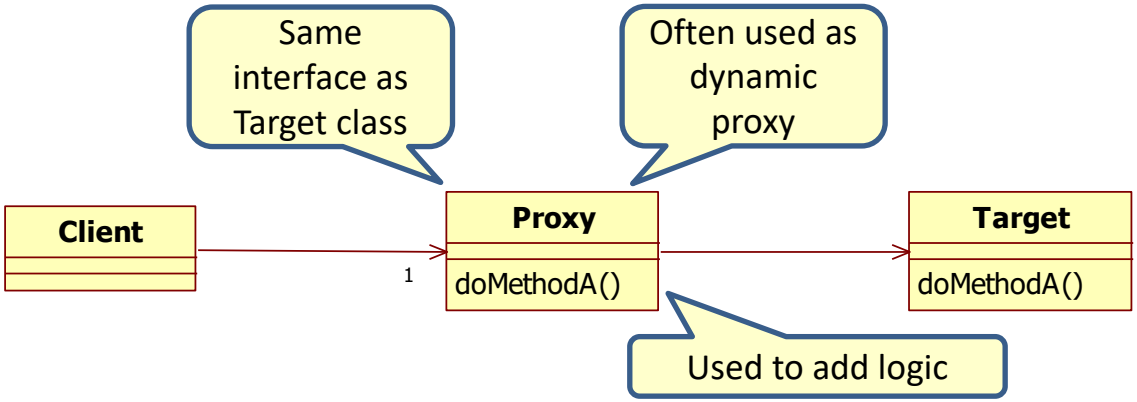
- If a method of the real subject calls a method of itself, this will not go through the proxy.



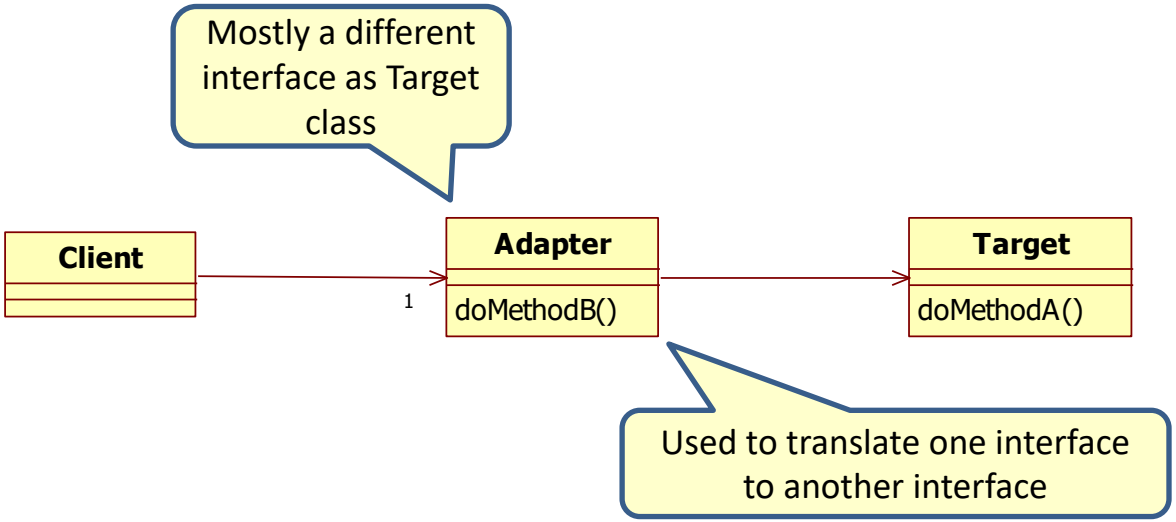


Adapter and Proxy: wrapper

- Proxy



- Adapter

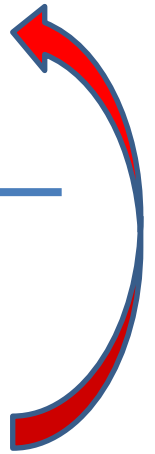


Main point

- 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.

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.