CS 525 - ASD

Advanced Software Development

MS.CS Program

Department of Computer Science

Rene de Jong, MsC.

Maharishi University
OF MANAGEMENT

CS 525 - ASD

Advanced Software Development

Maharishi University
OF MANAGEMENT

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

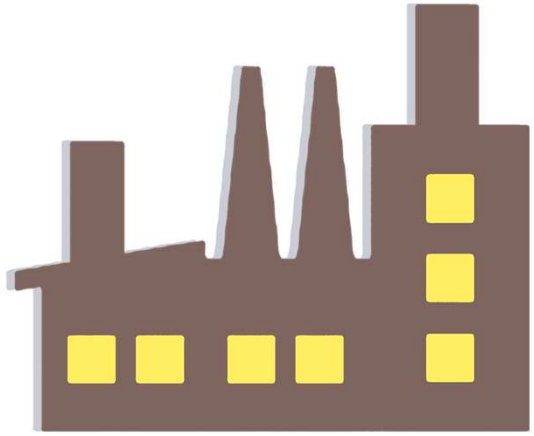
Midterm

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

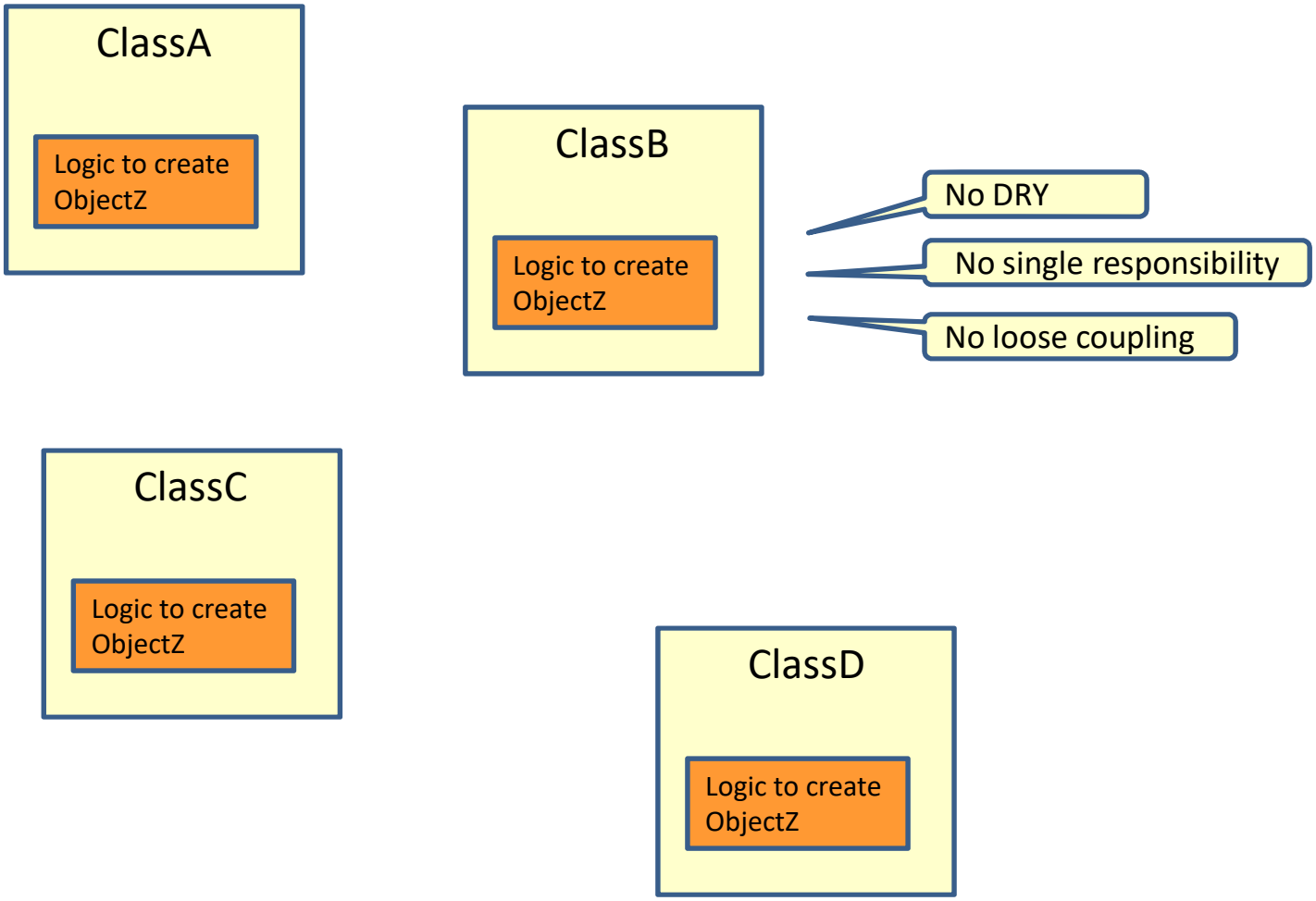
Final

Factory pattern

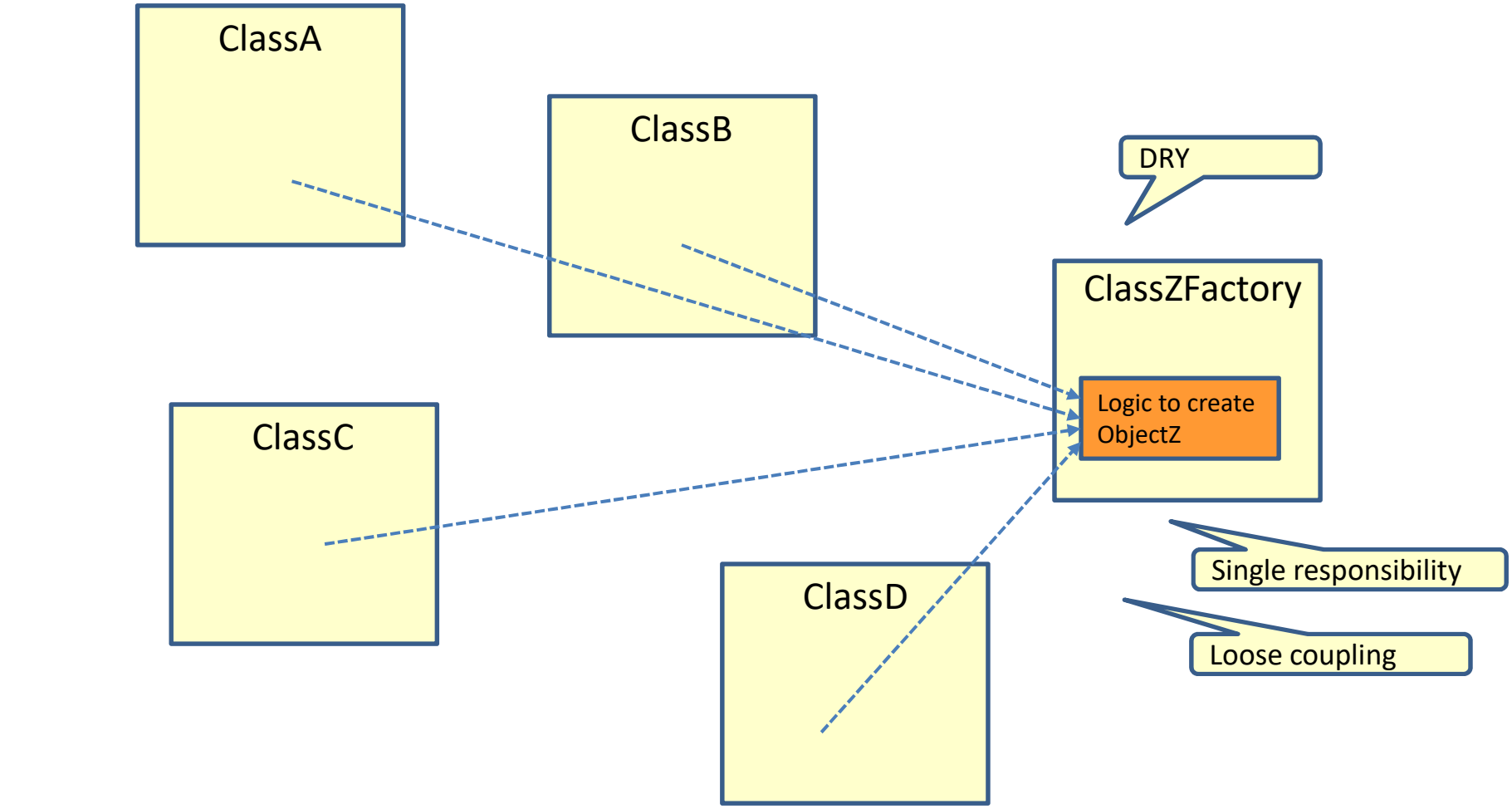
- A factory creates objects
  - Encapsulation of the logic to create objects



Without a factory

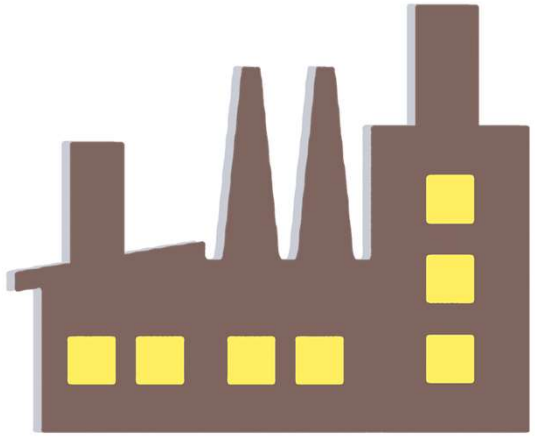


With a factory



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;
  }
}
```

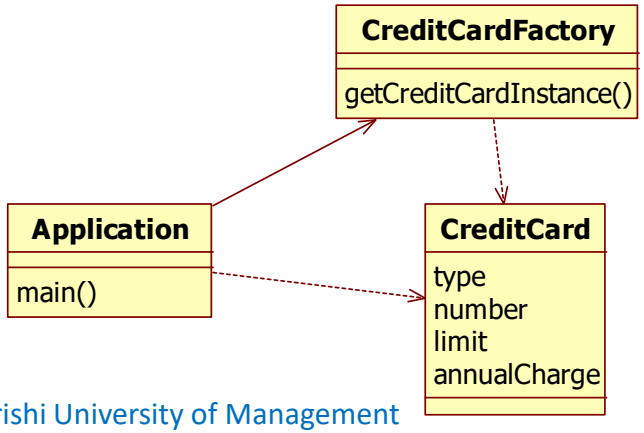
```
public class Application {

  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 = min;
        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

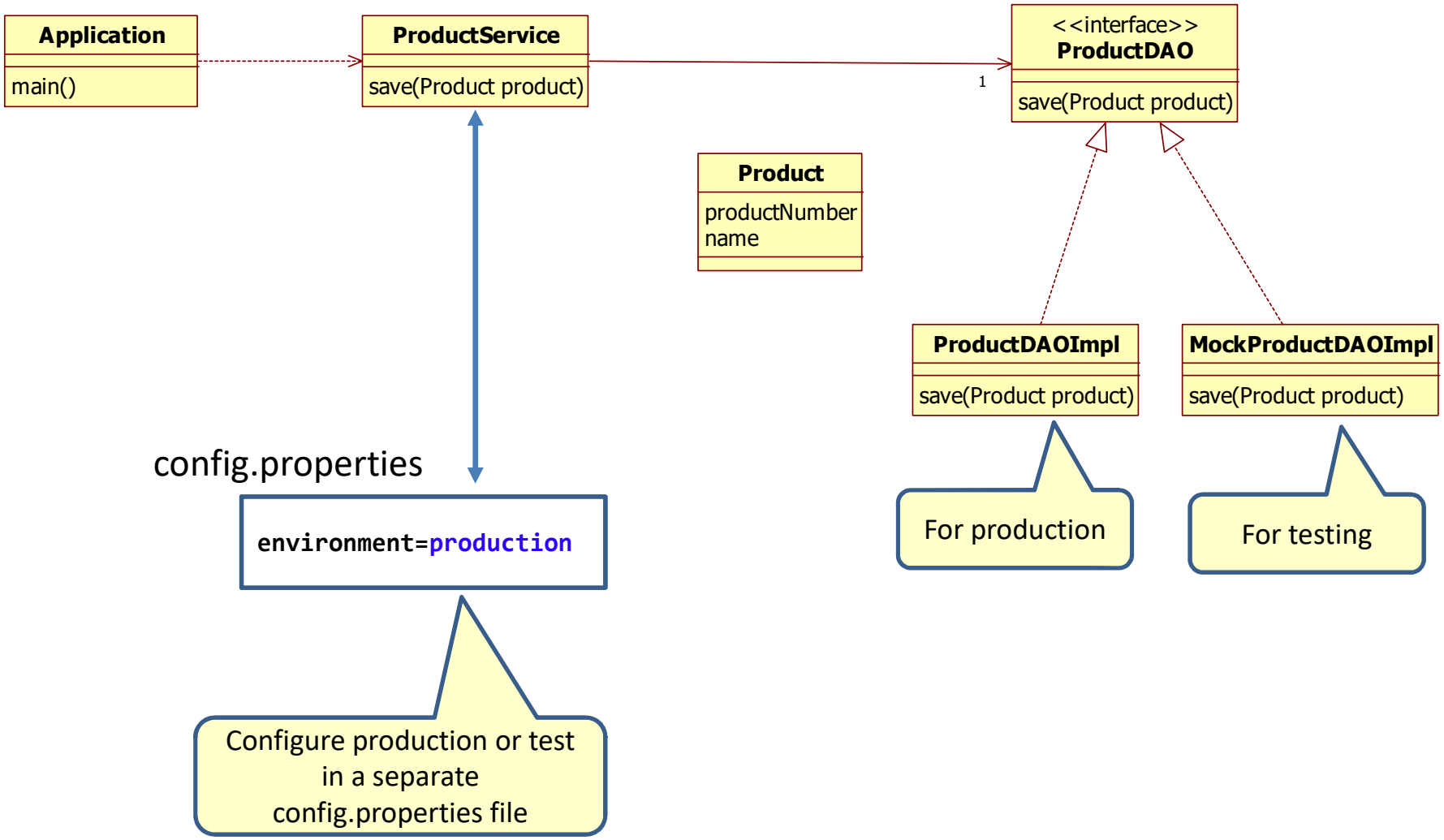
java.util

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.

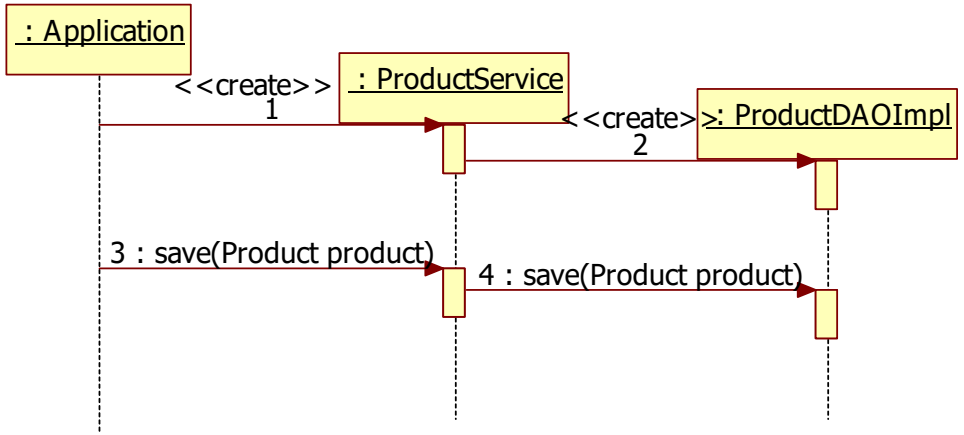
Example application



Example application

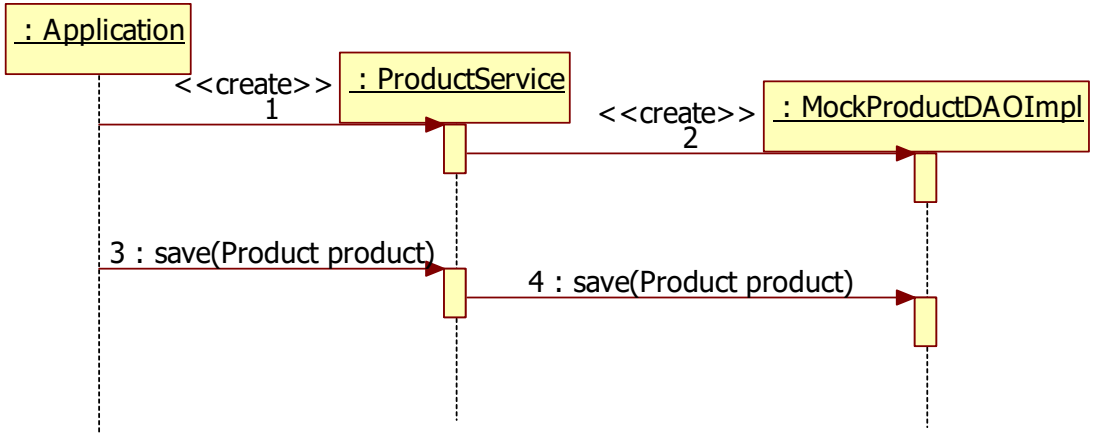
config.properties

```
environment=production
```



config.properties

```
environment=test
```



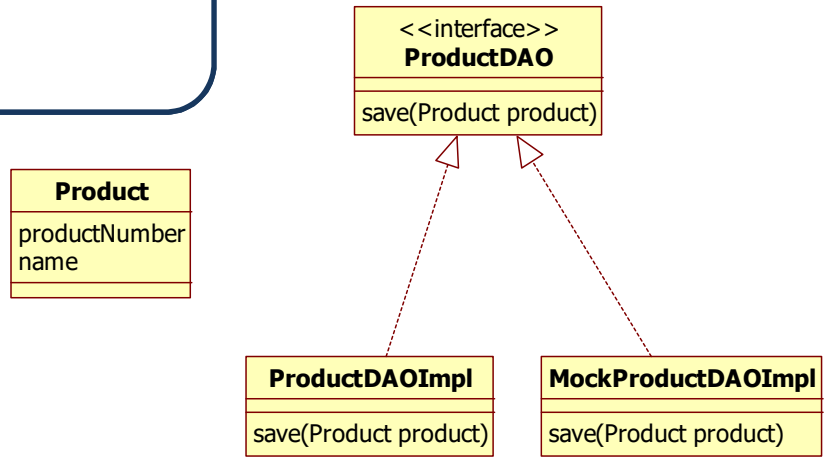
Product and DAO

```
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);
  }
}
```

```
ProductDAOImpl saves product
```

config.properties

```
environment=production
```

```
MockProductDAOImpl saves product
```

config.properties

```
environment=test
```

What is the problem?

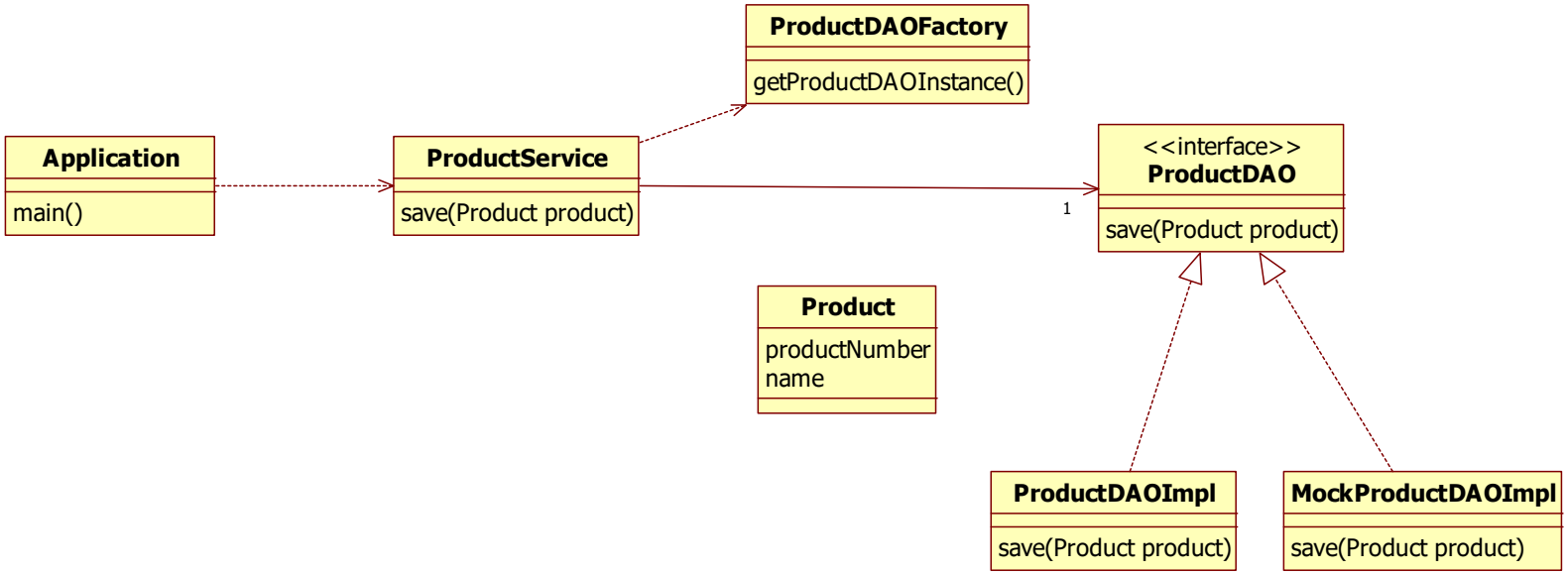
```
public class ProductService {
  ProductDAO productDAO;

  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);
  }
}
```

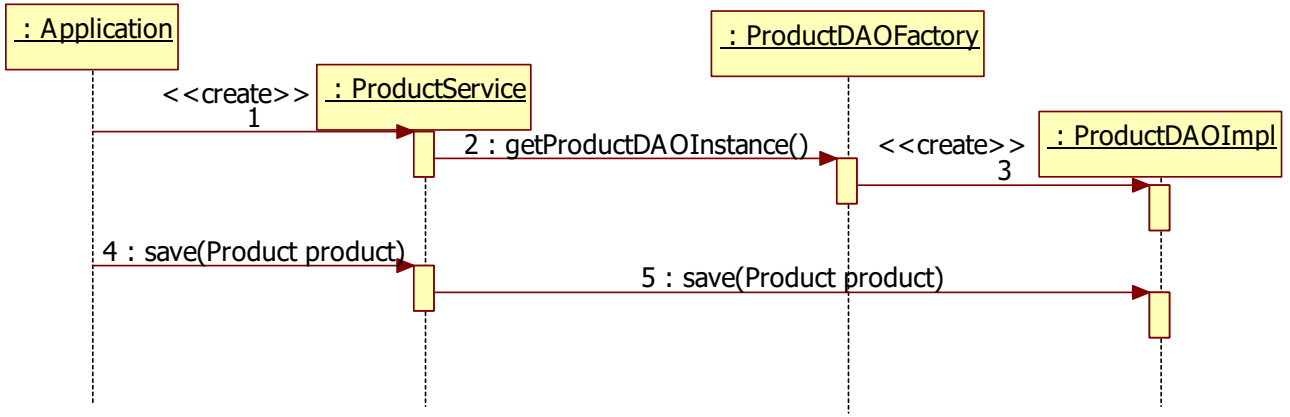
ProductService contains complex logic about creating the ProductDAO

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;
  }
}
```

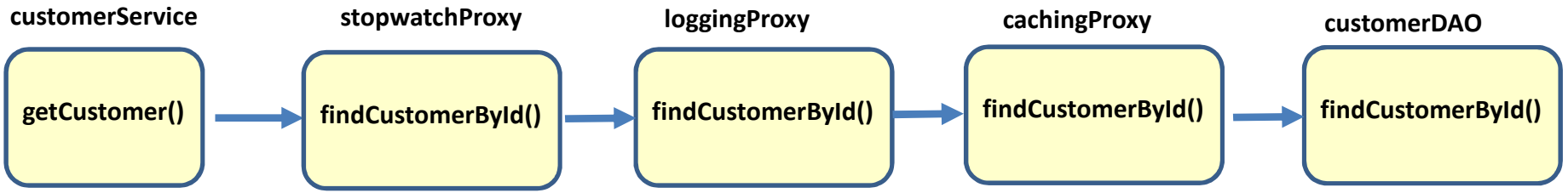
Encapsulate the logic to create objects

```
public class ProductService {
  ProductDAO productDAO;

  public ProductService() {
    productDAO=ProductDAOFactory.getProductDAOInstance();
  }

  public void save(Product product) {
    productDAO.save(product);
  }
}
```

Creating a dynamic proxy

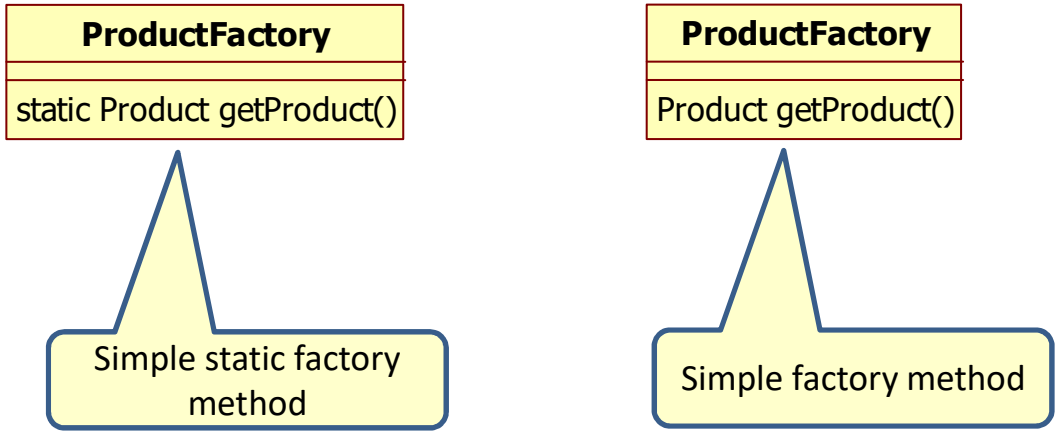
```
public class CustomerService {
  CustomerDAO customerDAO = new CustomerDAOImpl();
  ClassLoader classLoader = CustomerDAO.class.getClassLoader();
  CustomerDAO cachingProxy =
    (CustomerDAO) Proxy.newProxyInstance(classLoader,
                                         new Class[] { CustomerDAO.class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

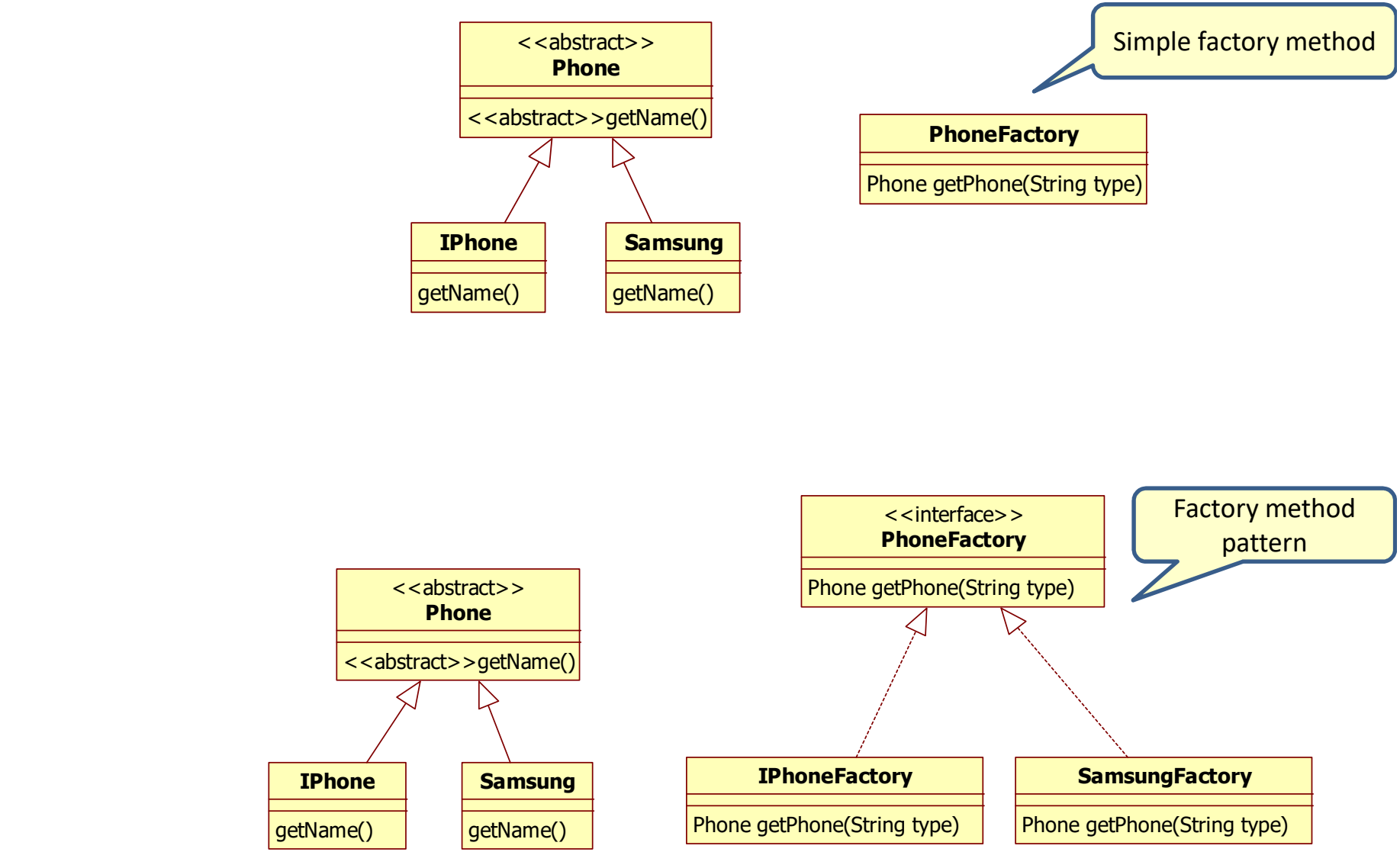


FACTORY METHOD PATTERN

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

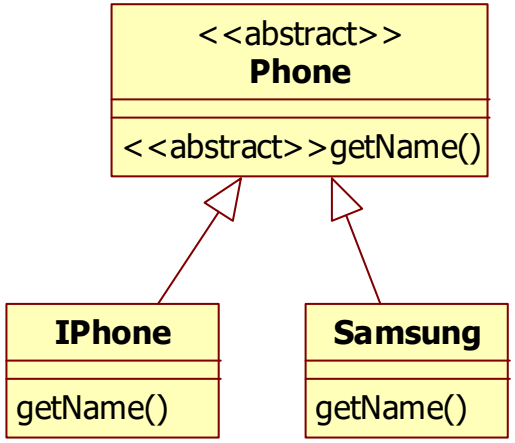


The phones

```
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";
  }
}
```

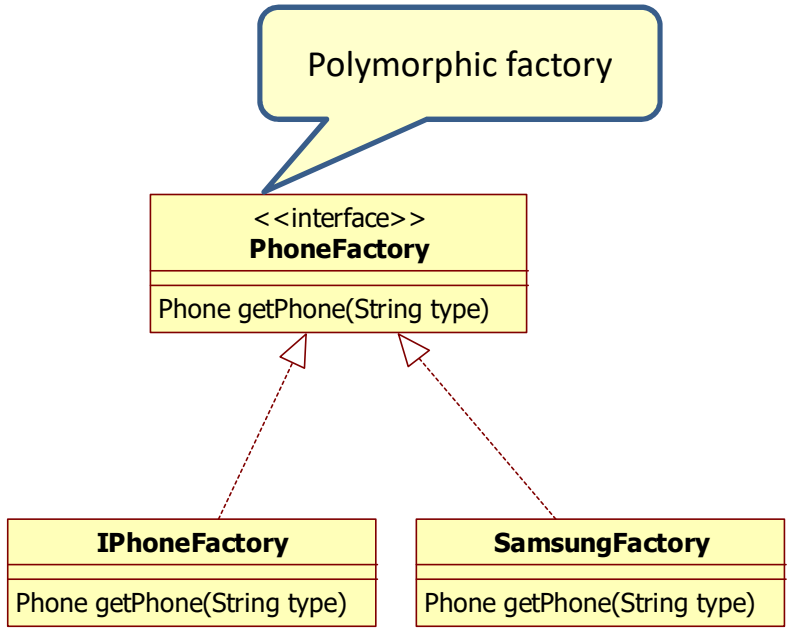


The phone factories

```
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();
  }
}
```

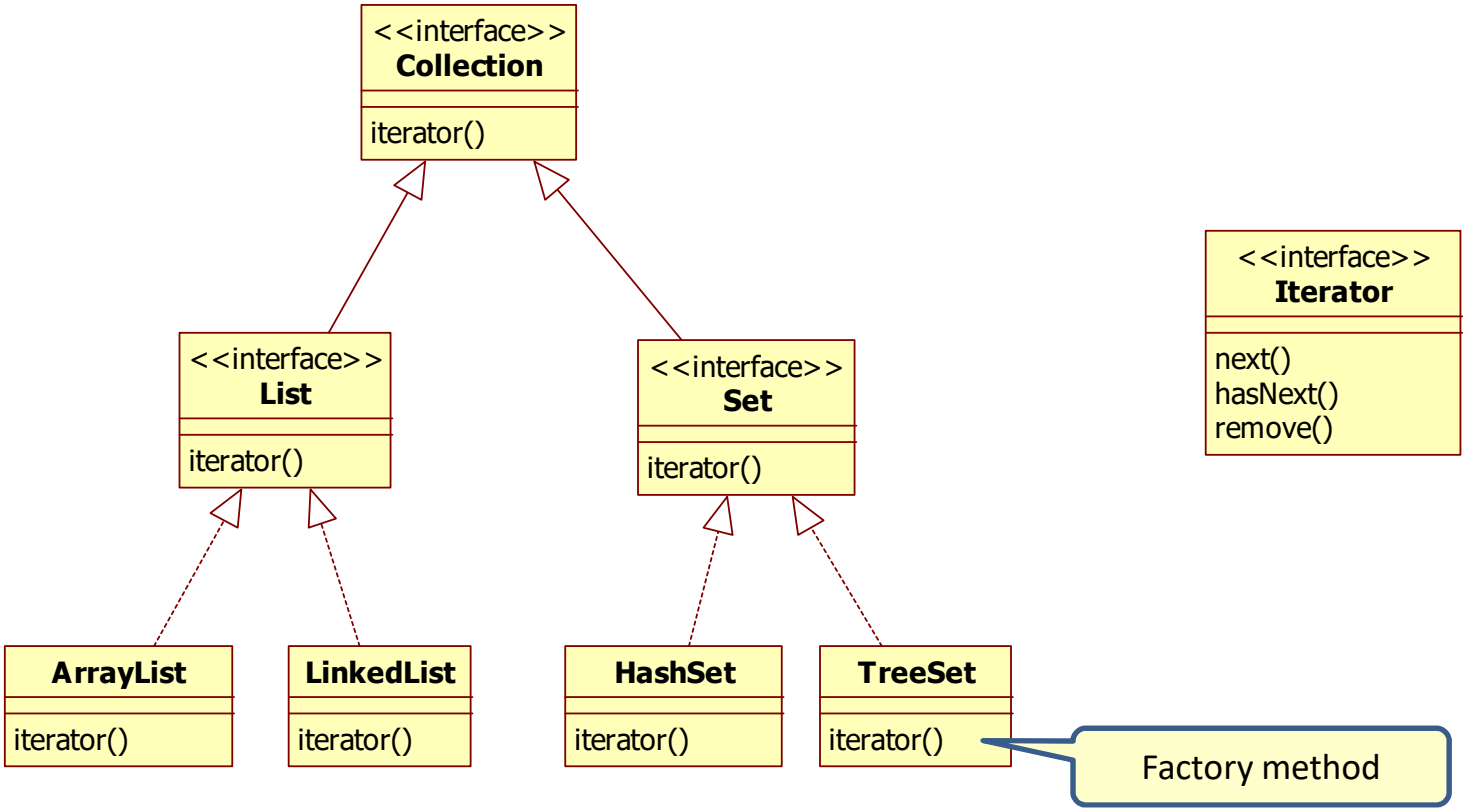
Flexibility: You can set (inject) any PhoneFactory

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

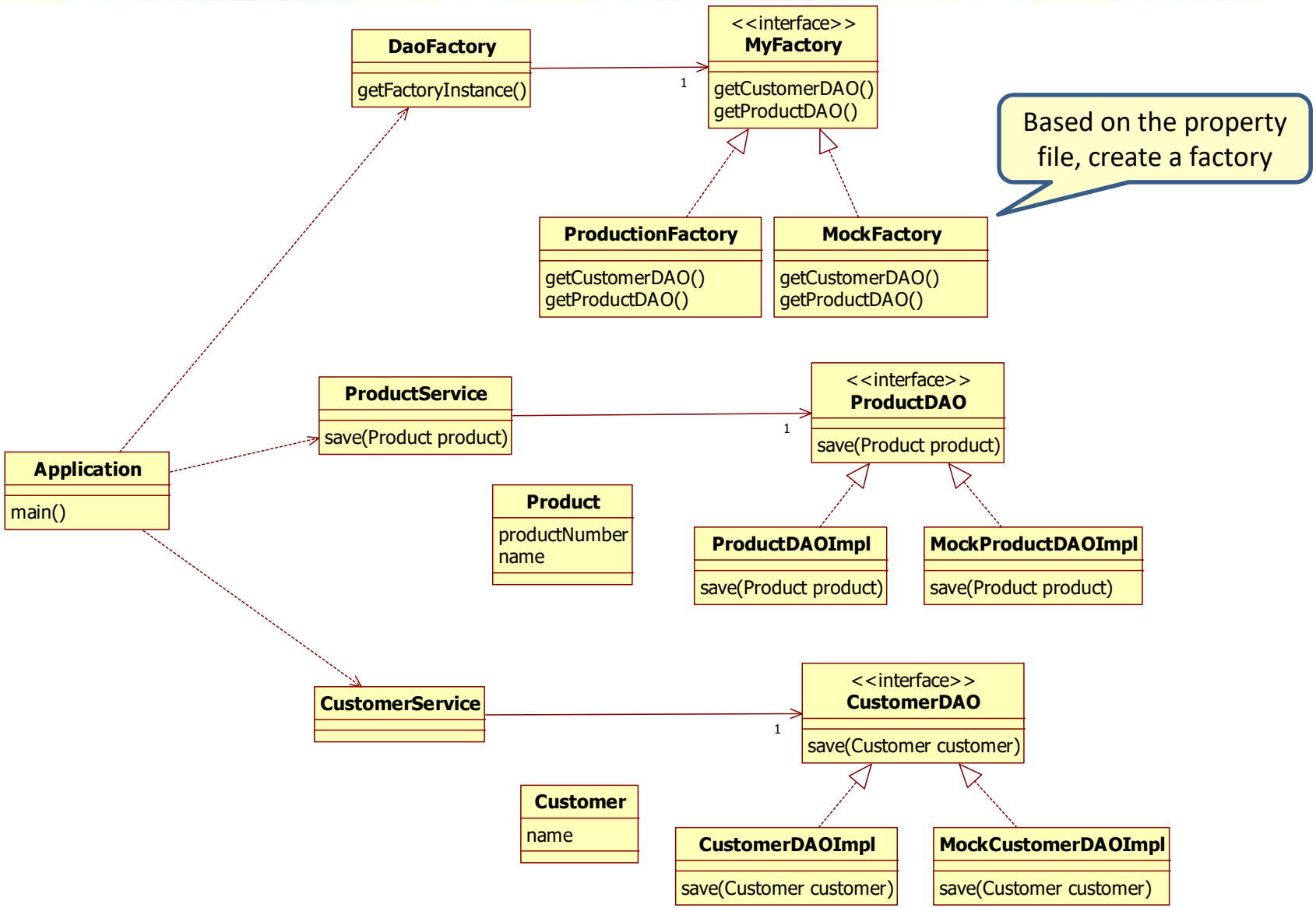


ABSTRACT FACTORY PATTERN

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

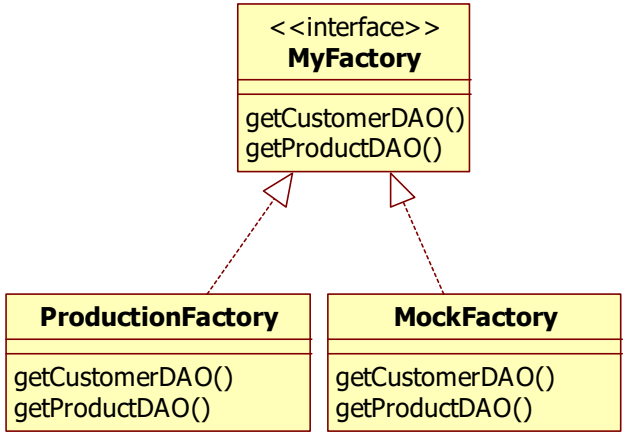
```
public class DaoFactory {
  private MyFactory factory;

  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();
  }
}
```



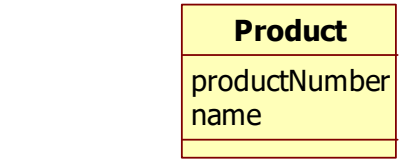
Product and DAO

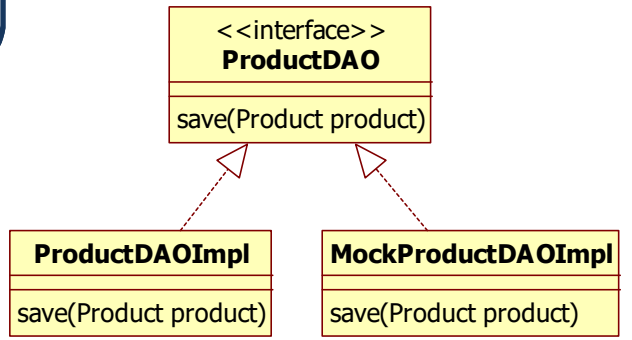
```
public interface ProductDAO {
  void save(Product product);
}
```

```
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;

  ....
}
```





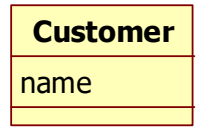
Customer and DAO

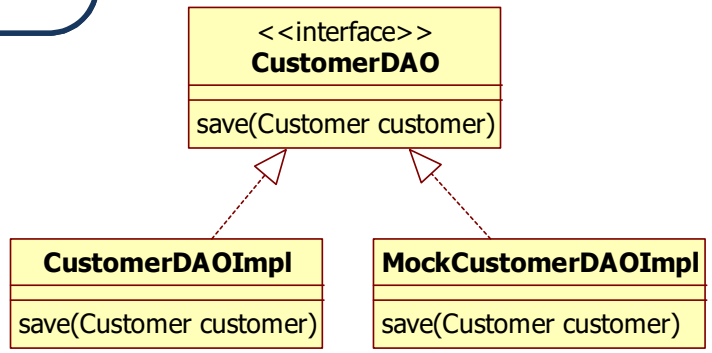
```
public interface CustomerDAO {
  void save(Customer customer);
}
```

```
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;

  ....
}
```





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.