PA 165 – Enterprise Java

Component Design

30th Sept 2015

Content

- Well Designed Components
- Component Lifecycle and Dependency management
- Component Testing
- Basic Enterprise Patterns

Well Designed Components

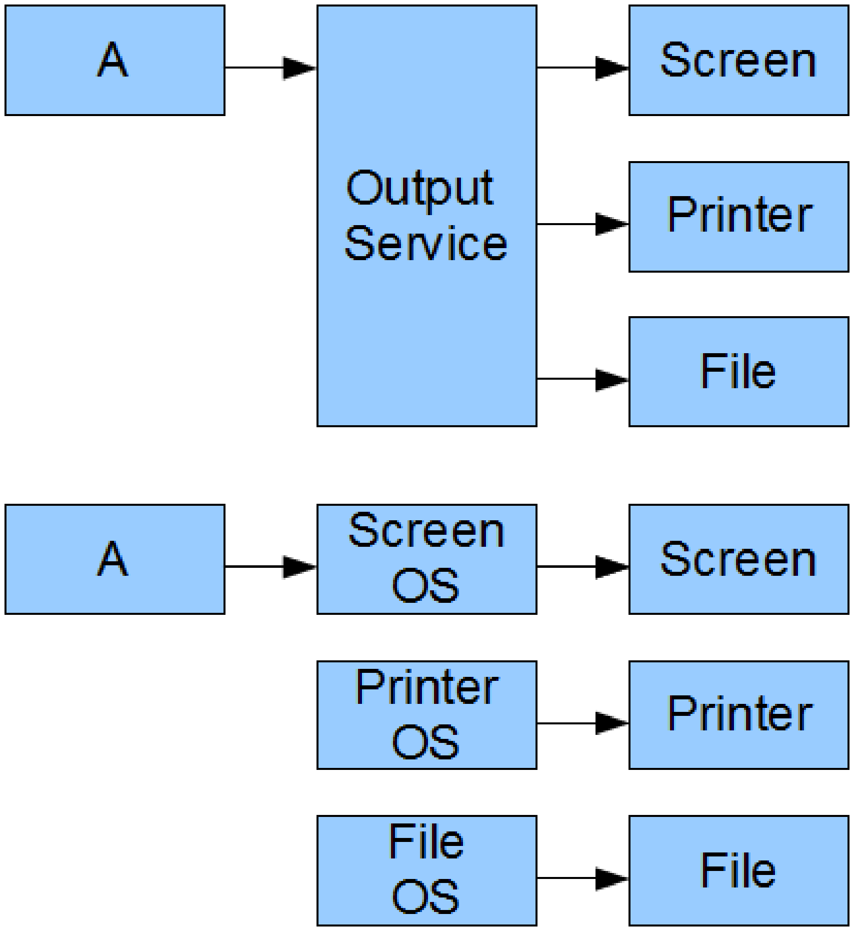
Well Designed Components

- Simple (single component = single task)
- Loosely Coupled (minimum dependencies)
- Well Defined Contract (well documented)
- Well Tested (unit tests)

Simple Components

- Single component should have just single responsibility (single task)
- Such compoent is easy to maintain, easy to test, more flexible and less dependant on other components
- When making components loosely coupled, start with making them simple (more responsibility means more dependancies).

Example



- A is transitively dependant on Printer and File even it needs just to write to screen.

Loosely Coupled Components

- Amount of dependancies between components is minimized
- Components don't depend on specific implementations, but on general interfaces
- Transitive dependencies (can be reduced by separating interface and implementation)
- Big problem are especially circular dependencies
- Too much dependencies indicates bad design

Simple Example

```
// Tight Coupled
public interface ProductService {

    void addProducts(ArrayList<Product> products);
    LinkedList<Product> getAllProducts();

}

// Loosely Coupled
public interface ProductService {

    void addProducts(List<Product> products);
    List<Product> getAllProducts();

}
```

More Complex Examples

- Table from PV168 components quiz
- PropertiesService

External Dependencies

- Try to make component independant on specific technologies, frameworks or libraries (at least at the API level).
- Use general types, exceptions and annotations instead of the proprietary ones (e.g. use @Inject instead of @Autowired).
- Avoid allways dependencies of API on another layer technology (e.g. using classes from frontend frameworks in business API)

Bad Examples

```
public interface ClientEventService {
    void saveLoginEvent(String userName, HttpServletRequest request);
}
public interface ReportService {
    List<javax.persistence.Tuple> getDailyReportData();
}
public void ContractService {
    void create(Contract contract) throws SQLException;
}
```

Well Defined Contract

- What should be defined
  - Component behaviour in all possible situations (especially in non-standard and erroneous ones)
  - Entry conditions
  - Thread safety
- Described with javadoc

Exceptions

- Use general exceptions for contract violation
  - IllegalArgumentException (also for null argument)
  - IllegalStateException
  - UnsupportedOperationException
- Consider to use checked/unchecked exception
  - Unchecked exception for contract violation or programmer mistake (could occure anywhere)
  - Checked exception for situations which should be caller aware of

Well Defined Contract Example

```
package net.homecredit.biometrics.fingerprints.properties;

/**
 * This interface represents some property value. PropertyValue can be static or
 * dynamic, according to the implementation. Static value is implemented as
 * immutable class and its {@link #get()} method is always returning the same
 * value. Dynamic value can be backed by some properties store and
 * {@link #get()} method is always returning the actual value in the properties
 * store.
 *
 * @author petr.adamek@embedit.cz
 * @param <T> property value type
 */
public interface PropertyValue<T> {

    /**
     * Returns current value of associated property. Value can be {@code null} if
     * appropriate {@link PropertyDefinition} allows that.
     *
     * <p>If the PropertyValue is dynamic, current value is loaded as string from
     * underlying properties store and converted to property value type with
     * appropriate {@link PropertyConvertor}. If the conversion fails
     * due illegal string representation of property value,
     * {@link IllegalPropertyValueException} is thrown. If the value is {@code null}
     * although {@code null} values are not allowed,
     * {@link IllegalPropertyValueException} is thrown as well. If the property
     * is not available anymore (e.g. because it has been deleted from underlying
     * properties store), {@link PropertyNotFoundException} is thrown. </p>
     *
     * <p>{@link IllegalPropertyValueException} and
     * {@link PropertyNotFoundException} should be never thrown for
     * static PropertyValue.</p>
     *
     * @throws IllegalPropertyValueException when string representation of property
     * value is invalid and it can't be converted to appropriate type or when
     * the value is {@code null} although {@code null} values are not allowed
     * for given property.
     * @throws PropertyNotFoundException when the property is not available anymore.
     * @return current value of the property
     */
    T get() throws IllegalPropertyValueException, PropertyNotFoundException;

    /**
     * Methods is returning current property value as string.
     * Calling this method is equivalent of {@code String.valueOf(this.get())}.
     * Result can be different than the string representation of property
     * value used for storing into underlying properties repository!
     *
     * @return current property value as string
     */
    @Override
    String toString();

    /**
     * Returns true if this PropertyValue is dynamic.
     *
     * @return true if this PropertyValue is dynamic, false otherwise.
     */
    boolean isDynamic();

}
```

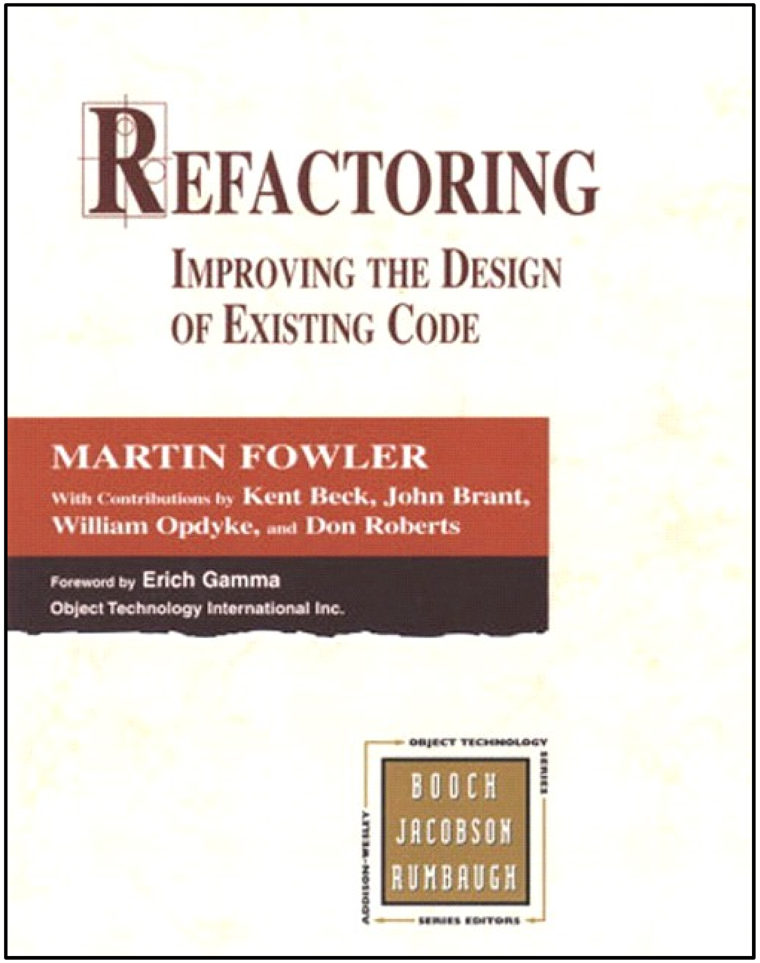
Reusabilty

- Avoid duplicit code, prefer to reuse components
- If the code is not exactly the same, but just similar, make the component more general and reusable (Design Patterns, Effective java)
- But be careful with if-statements – Replace Conditional with Polymorphism (https://sourcemaking.com/refactoring/replace-conditional-with-polymorphism)
- On other hand, avoid making universal components with currently not needed functionality

Refactoring

- Changing code structure without changing functionality
- Two hats principle (don't change the structure and functionality at the same time)
- *Catalog of Refactorings* is useful also for writing new code.

Resources



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma

Object Technology International Inc.

OBJECT TECHNOLOGY SERIES

BOOCH JACOBSON RUMBAUGH

ADDISON-WESLEY

SERIES EDITORS

Refactoring: Improving the Design of Existing Code

Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

http://amazon.com/dp/0201485672/

Best Practices

- Make components as simple as possible, follow single responsibility principle
- Separate interfaces from implementation, minimize dependencies
- Try to make component independent on specific technology (at least at API level).
- Define and document well the contract
- Don't hesitate to refactor component
- Read *Effective Java* and *Refactoring: Improving the Design of Existing Code*.

Component Lifecycle and Dependency management

Inversion of Control

- Component is not responsible for required resources, it expects that all resources will be provided by user of the component
- Hollywood principle – Don't call us, we call you
- Helps to reduce dependencies

Example

```
public class ContractServiceImpl implements ContractService {

    private final ContractDao contractDao;

    public ContractServiceImpl() {
        this.contractDao = new ContractDaoImpl();
    }

}

public class ContractServiceImpl implements ContractService {

    private final ContractDao contractDao;

    public ContractServiceImpl(ContractDao contractDao) {
        this.contractDao = contractDao;
    }

}
```

Dependency Injection

- Implementation of IoC principle
- Resources can be injected with
  - Field
  - Property
  - Constructor
- JSR 330: Dependency Injection for Java (@Inject, etc.)
- Qualifiers

Example

```
public class ContractServiceImpl implements ContractService {

    // Field injection
    @Inject
    private ContractDao contractDaoA;

    // Constructor injection
    private final ContractDao contractDaoB;

    @Inject
    public ContractServiceImpl(ContractDao contractDao) {
        this.contractDaoB = contractDao;
    }

    // Property injection
    private final ContractDao contractDaoC;

    @Inject
    public void setContractDao(ContractDao contractDao) {
        this.contractDaoC = contractDao;
    }
}
```

Java Naming and Directory API

- JNDI is standard way how to retrieve resources
- Allows to separate application from system configuration (database, external services, JMS, etc.)

JNDI Resources

```
public class ContractDaoImpl implements ContractDao {

    private final DataSource dataSource;

    public ContractServiceImpl() {
        Context context = new InitialContext().lookup("java:/comp/env");
        this.dataSource = (DataSource) context.lookup("jdbc/contractDb");
    }

}

public class ContractDaoImpl implements ContractDao {

    @Resource("jdbc/contractDb")
    private final DataSource dataSource;

}
```

Example

```
public class ProductDaoImpl implements ProductDao {

    private final EmbeddedDataSource dataSource;

    public ProductDaoImpl() {
        this.dataSource = new EmbeddedDataSource();
        this.dataSource.setDatabaseName("");


    }

}

public class ContractServiceImpl implements ContractService {

    private final ContractDao contractDao;

    public ContractServiceImpl(ContractDao contractDao) {
        this.contractDao = contractDao;
    }

}
```

Lifecycle Management

- Lifecycle management (creating and destroying components) is usually handled by some container, which is also providing Dependency Injection and Resource management.
- EJB container, Web container, CDDI container, Spring container, etc.
- Configured with annotations, xml files, JavaConfig, etc.

Component Testing

Unit Tests

- Test of isolated component
- Deterministic (always the same initial conditions, no random data or current time)
- Isolated (no interference with other tests)
- Border values and non-standard situations should be tested
- Test should be easy-to-understand

Mock Objects

- Tested component is isolated from the environment, behaviour of other objects is simulated with Mock objects
- Easy when interfaces are separated from implementation
- Various libraries
  - Mockito (http://mockito.org)
  - JMock
  - EasyMock

Demo

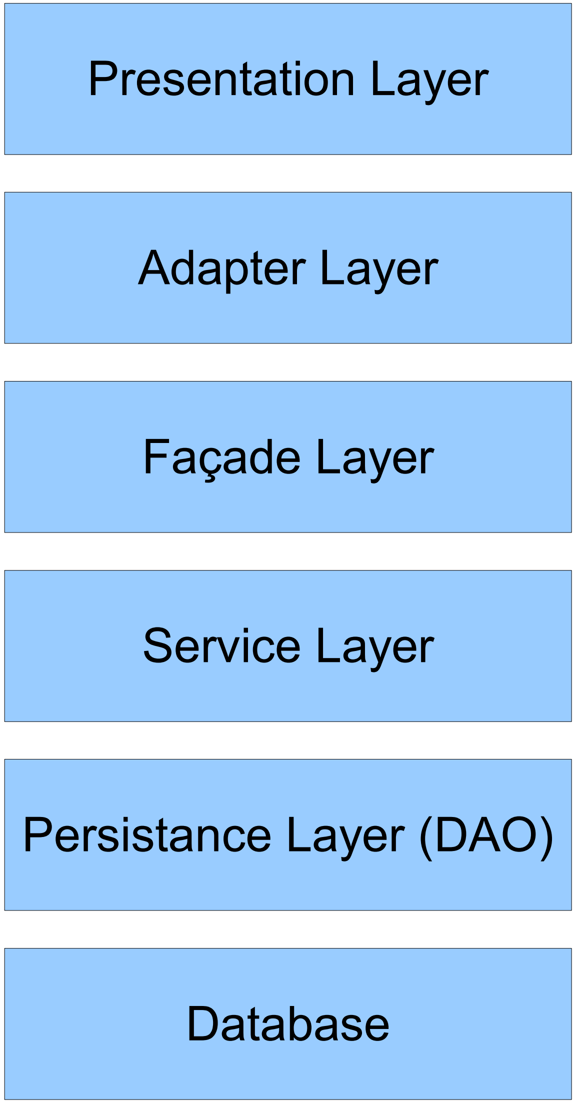
- Exception handling
- Hamcrest
- Mockito
    - Junit Integration
    - Verify
    - When
    - Argument matcher
    - Argument capture

Gold Rule

- When the component is well designed (single responsibility, loosely coupled), it is easy to write unit tests.
- If it is hard to write unit test, the component has the most probably bad design.

Basic Enterprise Patterns

Application Architecture Example



Presentation Layer

Adapter Layer

Façade Layer

Service Layer

Persistance Layer (DAO)

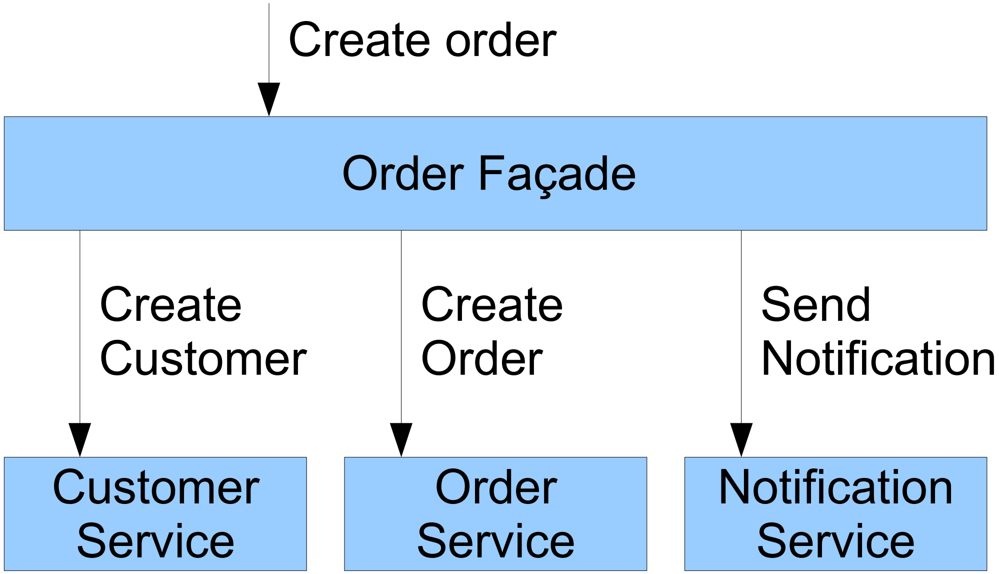
Database

Data Transfer Object

- Encapsulates data about some entity or entities for transport between layers
- Purpose
  - Remove dependency on entities (e.g. in service layer API)
  - Different scope (subset of attributes or agregated information from multiple entities)
- Can be created at any Layer

Service Façade

- Encapsulates complex business logic and expose it to the client as simple coarse-grained API
- Service orchestration



Adapter

- Adapts some interface to another one.
- Can be used
  - To make some component compatible with another interface
  - To convert method parameters types (e.g. Entities to DTO and vice versa)

Questions

?