Analyzing Web Frameworks through an Object-Oriented Lens: A Comprehensive research on Django and Spring Boot

Jayant Duneja Jayant.Duneja@colorado.edu

Tilak Singh Tilak.Singh@colorado.edu

Usage of OOPs in Web Development

1. Modular Design
2. Encapsulation
3. Inheritance
4. Polymorphism
5. Code Reusability
6. Ease of Maintenance
7. Abstraction
8. Collaborative Development
9. Scalability

OOPS pillars used in Spring and Django framework

Encapsulation in Django:

```python
from django.db import models

class Product(models.Model):
    name = models.CharField(max_length=100)
    price = models.DecimalField(max_digits=5, decimal_places=2)

    def __str__(self):
        return self.name
```

Encapsulation in SpringBoot:

```
# Application configuration properties
myapp.message=Hello, World!
```

```java
@Component
@ConfigurationProperties("myapp")
public class MyAppProperties {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}
```

OOPS pillars used in Spring and Django framework

Abstraction : Spring Boot

```
app.datasource.url=jdbc:mysql://localhost/test
app.datasource.username=dbuser
app.datasource.password=dbpass
app.datasource.pool-size=30
```

```
@Bean
@ConfigurationProperties("app.datasource")
public DataSource dataSource() {
        return DataSourceBuilder.create().build();
}
```

Abstraction : Django

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('blog/', include('blog.urls')),  # Include the blog app's URLs
]
```

```
from django.urls import path
from . import views

urlpatterns = [
    path('posts/', views.post_list, name='post_list'),
]
```

```
from django.shortcuts import render
from .models import BlogPost

def post_list(request):
    # Fetch blog posts from the database
    blog_posts = BlogPost.objects.all()

    # Render the list of posts using a template
    return render(request, 'blog/post_list.html', {'blog_posts': blog_posts})
```

OOPS pillars used in Spring and Django framework

Example of Inheritance in Spring Boot

```
<bean id="baseBean" class="com.example.BaseBean">
    <!-- Base bean configuration -->
    <property name="property1" value="value1"/>
</bean>

<bean id="childBean" class="com.example.ChildBean" parent="baseBea
n">
    <!-- Child bean configuration -->
    <property name="property2" value="value2"/>
</bean>
```

```
@Configuration
public class BaseConfig {
    @Bean
    public BaseBean baseBean() {
        BaseBean bean = new BaseBean();
        bean.setProperty1("value1");
        return bean;
    }
}

@Configuration
public class ChildConfig extends BaseConfig {
    @Bean
    public ChildBean childBean() {
        ChildBean bean = new ChildBean();
        bean.setProperty2("value2");
        return bean;
    }
}
```

Example of Inheritance in Django

```
from django.db import models

class Animal(models.Model):
    name = models.CharField(max_length=100)
    species = models.CharField(max_length=100)

    def __str__(self):
        return self.name

class Dog(Animal):
    breed = models.CharField(max_length=100)
    color = models.CharField(max_length=100)

class Cat(Animal):
    coat_color = models.CharField(max_length=100)
    eye_color = models.CharField(max_length=100)
```

OOPS pillars used in Spring and Django framework

Polymorphism: Spring Boot

```
    @GetMapping("/{id}")
    public ResponseEntity<Book> getBookById(@PathVariable Long id)
{
        // Logic to retrieve book by ID
        Book book = bookService.getBookById(id);
        return ResponseEntity.ok(book);
    }

    @PostMapping("/{id}")
    public ResponseEntity<String> updateBook(@PathVariable Long id,
@RequestBody Book updatedBook) {
        // Logic to update the book
        bookService.updateBook(id, updatedBook);
        return ResponseEntity.ok("Book updated successfully");
    }
```

Polymorphism: Django

```
from django.views import View
from django.http import HttpResponse

class BaseView(View):
    def get(self, request):
        return HttpResponse("BaseView - HTTP GET")

class CustomView(BaseView):
    def get(self, request):
        return HttpResponse("CustomView - HTTP GET")
```

```
from django.urls import path
from .views import CustomView

urlpatterns = [
    path('custom/', CustomView.as_view(), name='custom-view'),
]
```

Comparative Analysis of OOP Principles and Design Patterns

Spring Boot

- MVC
- Inversion of Control
- Data Access Object (DAO)
- Repository pattern

Django

- MVC
- Command Pattern
- Observer Pattern
- Template Pattern

Spring Boot Design Patterns:

MVC

- The Model represents the application's data and business logic.
- The View is responsible for presenting the data to the user and receiving user input.
- The Controller handles user input, processes it (possibly involving the Model), and selects the appropriate view to render.

IoC

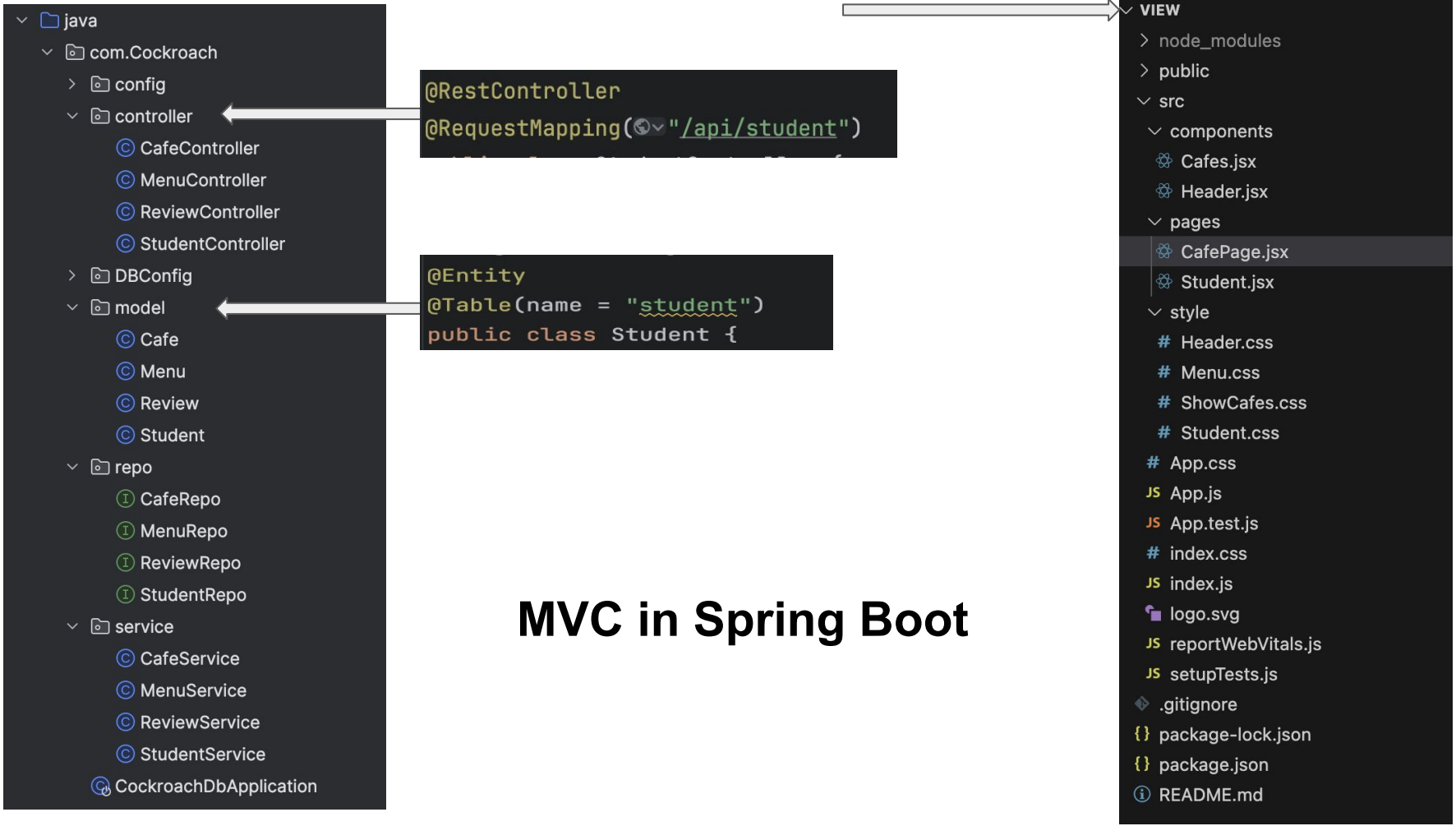
- IoC is a design pattern that changes the control flow in application components. It establishes a framework to manage dependencies between components, also known as dependency injection.

DAO

- DAO (Data Access Object) is an abstraction that provides a simplified interface for interacting with databases, reducing the need for direct and low-level database queries in application code

Repository Pattern

- The Repository Pattern is a design pattern that abstracts and centralizes data access logic, providing a unified interface for managing storage, retrieval, and search operations in software applications.



MVC in Spring Boot

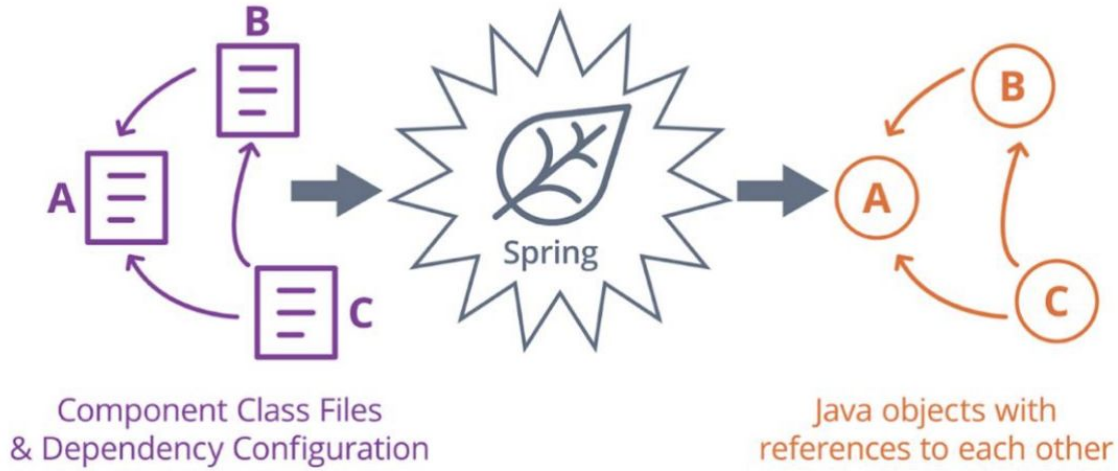


Figure 16. Spring IoC Configuration

Ref.- Implementation and Analysis of Software Development in Spring Boot

```java
package com.example.dao;

import com.example.model.Student;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import java.util.List;

public class StudentDao {

    @PersistenceContext
    private EntityManager entityManager;

    public List<Student> getAllStudents() {
        // Implementation to retrieve all students from the database using JPA or Hibernate
        return entityManager.createQuery("SELECT s FROM Student s", Student.class).getResultList();
    }

    public Student getStudentById(int studentId) {
        // Implementation to retrieve a student by ID from the database
        return entityManager.find(Student.class, studentId);
    }

    public void saveStudent(Student student) {
        // Implementation to save a student to the database
        entityManager.persist(student);
    }

    public void updateStudent(Student student) {
        // Implementation to update a student in the database
        entityManager.merge(student);
    }

    public void deleteStudent(int studentId) {
        // Implementation to delete a student from the database
        Student student = getStudentById(studentId);
        if (student != null) {
            entityManager.remove(student);
        }
    }
}
```

```java
@Repository
public interface MenuRepo extends JpaRepository<Menu, Long> {
    Tilak Singh
    @Query(value = "SELECT * FROM defaultdb.menu", nativeQuery = true)
    List<Menu> findAll();

    1 usage  Tilak Singh
    @Query(value = "SELECT * FROM defaultdb.menu WHERE cafe_id = ?1", nativeQuery = true)
    List<Menu> findByCafe_Cafe_id(Long cafeId);
}
```

Django Design Patterns:

- MVC :
  - While Django is often associated with the MVC (Model-View-Controller) pattern, it actually follows a slightly different architecture called Model-Template-View (MTV). In this design, Django separates concerns between database interfacing (Model), request processing (View), and final presentation (Template).
- Command Pattern:
  - In Django, the HttpRequest object encapsulates a request in an object, aligning with the principles of the Command pattern.
- Observer Pattern:
  - Signals are essentially messages announcing that a certain event has occurred. When one object changes state, all its listeners are notified and updated automatically using Signals
- Template Pattern
  - Steps of an algorithm can be redefined by subclassing without changing the algorithm's structure using class based generic views

Model

```python
from django.db import models

class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

    def __str__(self):
        return self.title
```

View

```python
from django.shortcuts import render
from .models import Book

def book_list(request):
    books = Book.objects.all()
    return render(request, 'book_list.html', {'books': books})
```

Template

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Book List</title>
</head>
<body>
    <h1>Book List</h1>
    <ul>
        {% for book in books %}
            <li>{{ book.title }} by {{ book.author }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Using HttpRequest and HttpResponse in Django

```python
from django.http import HttpResponse

def process_request(request):
    # Accessing HttpRequest properties
    method = request.method
    path = request.path
    params = request.GET  # Query parameters

    # Performing command logic
    result = f"Processing {method} request for {path} with parameters: {params}"

    # Generating HttpResponse as the response
    return HttpResponse(result)
```

Observer Pattern using Signals in Django

```python
from django.db.models.signals import post_save
from django.dispatch import receiver
from django.db import models

# Subject (Model)
class Book(models.Model):
    title = models.CharField(max_length=100)
    author = models.CharField(max_length=50)

# Observer (Signal Listener)
@receiver(post_save, sender=Book)
def book_saved(sender, instance, **kwargs):
    print(f"Book '{instance.title}' by {instance.author} has been saved.")

# Usage
book = Book(title="The Observer Pattern", author="John Doe")
book.save()  # Triggers the post_save signal
```

Template Pattern using Class Based Views in Django

```python
from django.views import View
from django.shortcuts import render
from django.http import HttpResponse

class GenericListView(View):
    template_name = 'generic_list.html'

    def get_queryset(self):
        # Default implementation to be overridden by subclasses
        return []

    def get_context_data(self, **kwargs):
        # Default implementation to be overridden by subclasses
        return {}

    def get(self, request, *args, **kwargs):
        queryset = self.get_queryset()
        context = self.get_context_data(object_list=queryset)
        return render(request, self.template_name, context)

# Subclassing to customize behavior
class BookListView(GenericListView):
    template_name = 'book_list.html'

    def get_queryset(self):
        # Custom implementation to fetch books
        return Book.objects.all()

    def get_context_data(self, **kwargs):
        # Custom implementation to add extra context for books
        context = super().get_context_data(**kwargs)
        context['title'] = 'Book List'
        return context
```

TEST DRIVEN DEVELOPMENT

TDD

- TDD is a software development practice that focuses on creating unit test cases before developing the actual code.
- It helps in maintaining clean and modularized code, making it easy to maintain and expand.
- It also helps us write minimal code for the system.
- TDD reduces bugs in the system, providing a systematic approach to development.

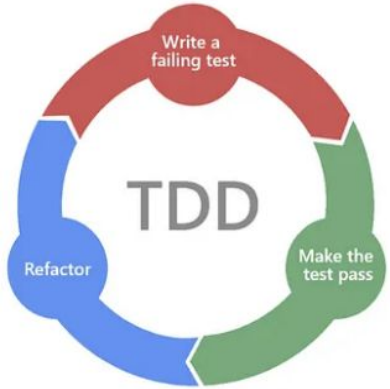
Uncle Bob's 3 Rules of TDD

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than sufficient to fail.
- You are not allowed to write any more production code than sufficient to pass the failing unit test

TDD in Spring Boot

```java
public class TestCases {
    @Test
    void testEmberKnightResonance(){
        EmberKnight testEmberKnight = new EmberKnight("");
        Room room= new ElementalRoom(Element.FIRE, 1, 1);
        ConcreteSubject testSubject = new ConcreteSubject();
        testEmberKnight.handleElementalEffects(room, testSubject);
        Assertions.assertEquals(testEmberKnight.getBaseCombatRoll(), 2);
    }
    @Test
    void testMistWalkerDiscord(){
        MistWalker testMistWalker = new MistWalker("");
        Room room = new ElementalRoom(Element.EARTH, 1, 1);
        ConcreteSubject testSubject = new ConcreteSubject();
        float prevDodgeChance = testMistWalker.getDodgeChance();
        testMistWalker.handleElementalEffects(room, testSubject);
        Assertions.assertEquals(testMistWalker.getDodgeChance() - prevDodgeChance, -25);
    }
    @Test
    void testMistWalkerExpertise(){
        MistWalker testMistWalker = new MistWalker("");
        testMistWalker.UpdateExpertise(1);
        Assertions.assertEquals(testMistWalker.getCombatExpertiseBonus(), 2);
    }

    @Test
    void testGameBoardAdventurers(){
        GameBoard gameBoard = new GameBoard("EK", "Test_Adventurer", "");
        List<Adventurer> current_adventurers = gameBoard.getRoom(Constants.STARTING_ROOM_ID).getAdventurers();
        Assertions.assertEquals(current_adventurers.size(), 1);
        Assertions.assertEquals(current_adventurers.get(0).getAcronym().acronym, "EK");
        Assertions.assertEquals(current_adventurers.get(0).getDisplayName(), "Test_Adventurer");
    }
    @Test
    void testGameBoardCreaturesEmpty(){
        GameBoard gameBoard = new GameBoard("EK", "Test_Adventurer", "");
        List<Creature> current_creatures = gameBoard.getRemainingCreatures();
        Assertions.assertEquals(current_creatures.size(), 0);
    }
    @Test
    void testGameBoardCreaturesNonEmpty(){
        GameBoard gameBoard = new GameBoard("EK", "Test_Adventurer", "A,F,T,Z");
        List<Creature> current_creatures = gameBoard.getRemainingCreatures();
        Assertions.assertEquals(current_creatures.size(), 4);
    }
}
```



TDD in Django:

```python
from django.test import TestCase, RequestFactory
from unittest.mock import patch
from .views import external_data_view

class ExternalDataViewTest(TestCase):
    def setUp(self):
        self.factory = RequestFactory()

    @patch('requests.get')  # Mocking the requests.get function
    def test_external_data_view(self, mock_requests_get):
        # Set up the mock response from the external API
        mock_response = self._create_mock_response(status_code=200, json_data={'key': 'value'})
        mock_requests_get.return_value = mock_response

        # Create a request to the view
        request = self.factory.get('/external-data/')
        response = external_data_view(request)

        # Check that the view renders the template with the mocked data
        self.assertContains(response, 'value')

    def _create_mock_response(self, status_code, json_data=None):
        """Helper method to create a mock HTTP response."""
        mock_response = MagicMock()
        mock_response.status_code = status_code
        mock_response.json.return_value = json_data
        return mock_response
```