國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science & Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

從 Spring 中的 XML 命名空間到 Java 註解的自動轉換

Auto-Transform XML Namespaces to Java Annotations in Spring

許容縈

Jung-Ching Hsu

指導教授: 李允中 博士

Advisor: Jonathan Lee Ph.D.

中華民國 113 年 7 月

July, 2024

doi:10.6342/NTU202404078

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

從 Spring 中的 XML 命名空間到 Java 註解的自動轉換

Auto-Transform XML Namespaces to Java Annotations in Spring

本論文係許容綮君（學號 R11922134）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 113 年 7 月 29 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 29 July 2024 have examined a Master's thesis entitled above presented by HSU, JUNG-CHING (student ID: R11922134) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

李允中		
（指導教授 Advisor）		

系主任/所長 Director: 陳祝嵩

Acknowledgements

首先，我要感謝我的指導教授李允中博士這兩年來的教導，指導研究方向，探討研究主題，並教導我如何拆解並分析問題，進而提供解決方案，在此謹向恩師致上最誠摯的感謝；再者，我要感謝台灣大學軟體工程研究室的同屆成員，朱紹瑜、梁乃匀、蔡智冠、鍾昀諺、王堃宇、以及學弟妹、助理等，在一些共同合作的項目中互相砥礪、討論，並給予協助，讓我得以完成此篇論文。

doi:10.6342/NTU202404078

摘要

在大多數的 Java 專案中，經常會使用 Spring 框架來加速軟體開發的速度。然而，在傳統的 Spring 專案中，通常使用 XML 作為配置文件。然而，使用 XML 配置文件會遇到很多問題，例如文件可能會變得非常冗長和複雜，尤其是當配置項目很多時。此外，XML 配置文件靈活性不足，配置內容需要提前定義，無法動態配置。

因此，我們希望能夠找到一套流程和方法，自動將 XML 配置文件轉換成 Java 配置文件。本研究主要提出一個 XML 轉換到註解的自動轉換流程。過程中，我們會對 Spring 專案的所有配置文件進行分析，並為每個命名空間訂製轉換規則，依據這些規則將 XML 轉換成 Java 配置文件。最後，我們提供一個驗證機制，用以確認轉換規則的正確性。

為了驗證我們的轉換規則，我們以一個大型的專案作為範例進行佐證。

關鍵字： Spring 框架、依賴注入、自動轉換、設計模式

doi:10.6342/NTU202404078

Abstract

In most Java projects, the Spring framework is often used to accelerate software development. However, in traditional Spring projects, XML files are typically used as the configuration files. Nonetheless, using XML configuration files encounters many issues, such as files becoming very lengthy and complex, especially when there are many configuration items. Additionally, XML configuration files lack flexibility; the configuration content needs to be predefined and cannot be dynamically configured.

Therefore, we aim to find a process to automatically transform XML configuration files into Java configuration files. This study proposes an automatic transform process from XML to annotations. During the process, we analyze all configuration files in the Spring project and compile conversion rules for each namespace. Based on these conversion rules, we convert XML configuration files into Java configuration files. Finally, we provide a validation mechanism to ensure the correctness of Java configuration files.

To validate our conversion rules, we use an large-scale project as an example to sup-

doi:10.6342/NTU202404078

port our process.

Keywords: Spring framework, DI, auto-transformer, Design Pattern

doi:10.6342/NTU202404078

Contents

doi:10.6342/NTU202404078

doi:10.6342/NTU202404078

doi:10.6342/NTU202404078

doi:10.6342/NTU202404078

List of Figures

doi:10.6342/NTU202404078

List of Tables

doi:10.6342/NTU202404078

Chapter 1

Introduction

In software development, the Spring framework significantly enhances the development efficiency of enterprise applications with its rich features, simplifying the development process. However, traditional XML-based configurations have gradually revealed several limitations and challenges. Firstly, XML configuration files often become excessive and difficult to maintain, especially as the project grows in size and complexity. Developers need to spend considerable time managing and updating these configuration files, which can lead to errors and inconsistencies. Secondly, XML configurations must be fully defined before the application starts. This restriction limits the dynamic adjustment of configurations during runtime.

This study aims to design an automated conversion process that transforms XML configuration files into Java configuration files. We conduct a comprehensive analysis of all configuration files in the project and compile corresponding conversion rules for twelve XML namespaces. Subsequently, we verify the Java configuration files through a verification mechanism. Finally, we explore its potential applications in modern software development using case studies.

doi:10.6342/NTU202404078

The structure of this paper is as follows: Chapter 2 introduces the relevant tools and concepts used in the study; Chapter 3 discusses related research; Chapter 4 provides an overview of the entire automated conversion process and introduces the system architecture; Chapter 5 analyzes XML files; Chapter 6 describes how to find all Spring Beans in the project and build their dependencies; Chapter 7 compiles conversion rules for each namespace; Chapter 8 explains how to generate Java configuration files; Chapter 9 introduces the verification process; Chapter 10 applies our conversion process using a case study; and Chapter 11 summarizes our research and suggests directions for future studies.

doi:10.6342/NTU202404078

Chapter 2

Background Work

The Spring Framework is an open-source Java framework used for developing enterprise applications. It encompasses a broad range of Spring projects designed to address various aspects of different domains. The core design principles of Spring include modularity, loose coupling, flexibility, and scalability, enabling developers to create well-structured and maintainable applications.

The Spring Framework offers several key features:

Inversion of Control The Spring IoC Container (BeanFactory) is responsible for instantiating, configuring, and assembling objects known as beans. This process fundamentally reverses the control over dependency creation from the objects themselves. The Spring IoC Container supports two types of configuration metadata: 1. XML-based configuration 2. Java-based and Annotation-based configuration

Dependency Injection（DI） Dependency Injection is a process in which an object's dependencies (i.e., other objects it interacts with) are provided through constructors, factory methods, or property setters. Spring supports two types of dependency

doi:10.6342/NTU202404078

injection: constructor injection and setter injection.

Spring ecosystem The Spring ecosystem consists of a variety of specialized Spring projects tailored for different domains. These projects also provide their own namespaces, allowing for configuration via XML.

The Java Code Parser is a tool used for analyzing Java source code. Huang [6] used the Java Reflection API and JDK Compiler to parse Java source code and generate class models, which were subsequently used by Yu [11] to store the parsed information in a service component database. This study will utilize this tool to analyze Java code and work with the class models and method models.

doi:10.6342/NTU202404078

Chapter 3

Related Work

3.1 Dependency Injection

Razina and Janzen [9] investigated whether dependency injection enhances software maintainability by analyzing 20 project sets with and without dependency injection. Metrics such as Coupling between Objects (CBO), Response for Class (RFC), and Lack of Cohesion (LCOM) were used. The results indicated that projects with over 10% dependency injection exhibited lower coupling.

Laigner et al. [7] identified and analyzed anti-patterns related to Dependency Injection (DI) in Java software projects. They evaluated the prevalence of these anti-patterns in selected Java projects and conducted an expert survey to assess the relevance of their catalog. The identified anti-patterns include Control Freak, Constructor Over-Injection, Service Locator Misuse, Singleton Abuse, and Circular Dependency.

doi:10.6342/NTU202404078

3.2 Java Annotation

Rocha and Valente [10] conducted an empirical study on the use of annotations in Java, analyzing 106 open-source Java systems and evaluating over 160,000 annotations. Key findings include the prevalence of the 'annotation-hell' phenomenon, the use of both pre-defined and external framework annotations (particularly for persistence and testing), a high percentage of annotations applied to methods, and the popularity of JPA (Java Persistence API) annotations. Additionally, despite Java not officially supporting annotations in anonymous classes, programmers still employ them.

Yu et al. [12] presented a large-scale empirical study on Java annotations across 1,094 open-source projects on GitHub. This study examines the usage, evolution, and impact of annotations, revealing that annotations are widely utilized, with a median of 1,707 annotations per project. Some developers repeatedly use annotations for complex conditions, which could lead to overuse. The study highlights the need for tools to detect and warn against annotation overuse and offers insights for improving Java annotation engineering.

The above studies demonstrated the widespread use of Java annotations but warned us to avoid misuse and overuse. Regardless, applying annotations to the configuration in Spring Boot is commonly treated appropriately.

3.3 Design Pattern in Software Migration

Balalaie et al. [4] introduced migration patterns for transitioning traditional architectures to cloud-native microservices. These patterns, derived from industrial projects,

doi:10.6342/NTU202404078

assist in rearchitecting systems for cloud migration. By selecting and combining these patterns, organizations can develop effective migration strategies. Empirical evaluations confirm the patterns' effectiveness in facilitating architectural refactoring.

Raj and Ravichandra [8] discussed the migration from SOA to microservices architecture, focusing on design patterns to address challenges such as decomposing SOA services, determining the size of microservices, and detecting anomalies. They proposed three patterns and evaluated them using a standard web-based application. Their work underscores the importance of design patterns in guiding the migration process.

doi:10.6342/NTU202404078

Chapter 4

The Process of Auto-Transformer

In this study, to automatically convert XML configuration files of a project into Java configuration files, we developed a transformation process consisting of six steps:

1. Parse the existing Java source code to obtain detailed information about classes, methods, and other elements.
2. Parse XML configuration files to extract XML components and their dependencies.
3. Build a container stored `BeanDefinition` based on the XML configuration files.
4. Follow conversion rules to convert XML configuration files into Java configuration files.
5. Generate Java configuration files.
6. Verify the generated Java configuration files through verification mechanisms to ensure that the project executes correctly after the replacement.

We have assembled the entire system according to the above steps, as shown in Figure 4.1. Our system consists of six modules, each responsible for one step of the process. The

doi:10.6342/NTU202404078

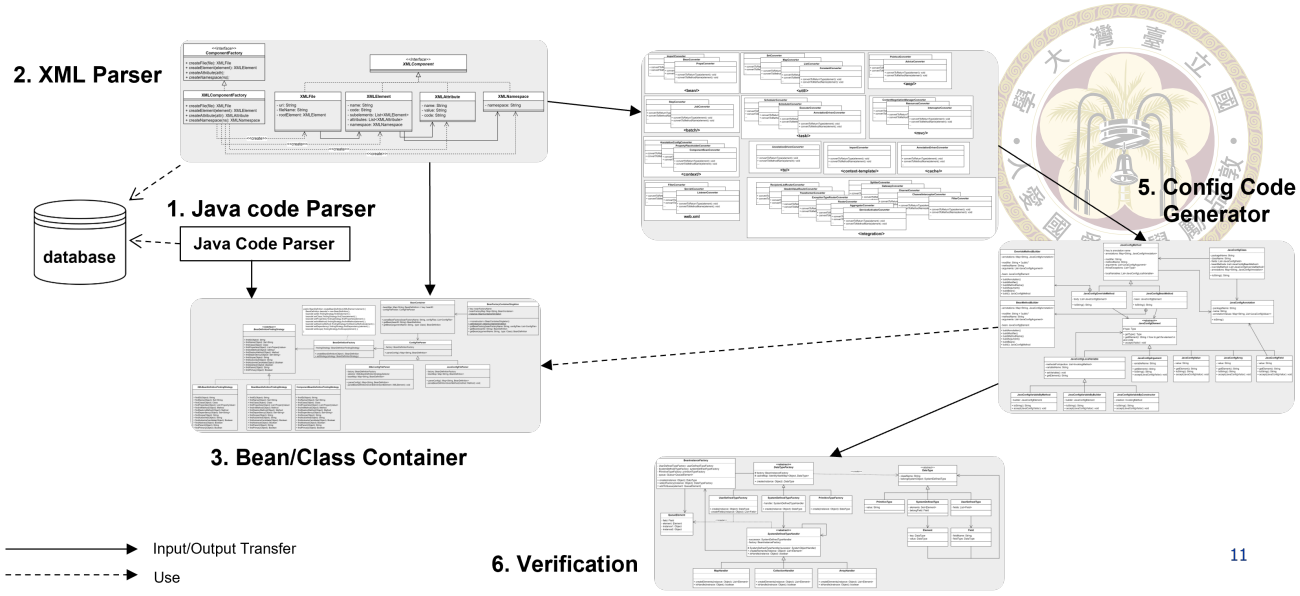


Figure 4.1: System Architecture - XML Transformer

first module uses the Java code parser [5] to analyze Java source code, extracting information such as return types and parameter types for each method. The class models will be used to build the `ClassContainerSingleton`, which provides the class and method information during the conversion of XML configuration files.

doi:10.6342/NTU202404078

Chapter 5

Parse XML file

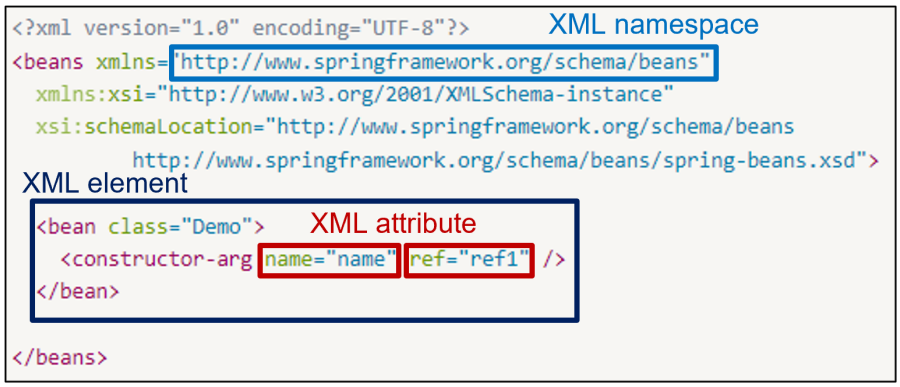


Figure 5.1: Example - XML Component

First, we parse the XML files into the required data types for subsequent transformation. We use the dom4j library [1] to decompose XML files into various XML components, including XML elements, XML attributes, and XML namespaces. These XML components and their dependencies are then stored in a database.

For example, as illustrated in Figure 5.1, an XML file contains a single root XML element, with each XML element belonging to a specific XML namespace and containing zero or more child XML elements and XML attributes. To build the dependencies between XML components, we utilize the Abstract Factory pattern to create XML components, as shown in Figure 5.2. This pattern ensures that when creating an XML file, the dependent

doi:10.6342/NTU202404078

XML elements are generated simultaneously, and when creating an XML element, the associated XML attributes, XML elements, and XML namespaces are also created.

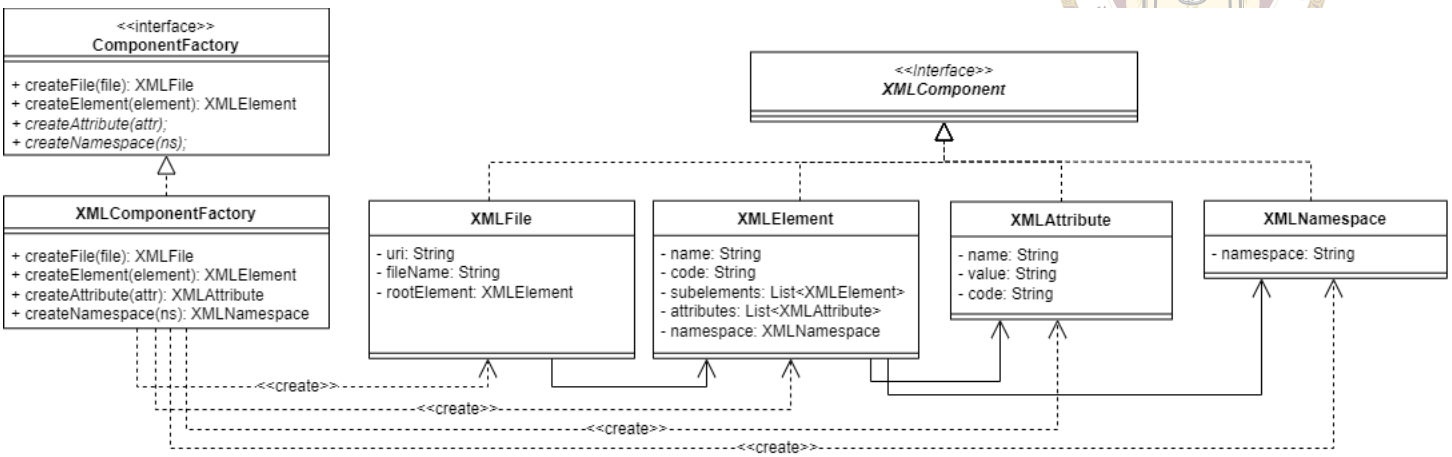


Figure 5.2: Class Diagram - XMLComponentFactory

doi:10.6342/NTU202404078

Chapter 6

Build Bean Container

When converting an XML element that references other beans, it is essential to have detailed information about the types and attributes of these beans. To address this need, we have implemented a `BeanContainerSingleton` to maintain metadata for all beans within the project, which will be utilized during the conversion process.

In a Spring project, obtaining information about all beans requires an understanding of the IoC container. The IoC container is responsible for instantiating and managing beans, with `BeanFactory` being the fundamental interface defining the IoC container in Spring. Therefore, we need to analyze how many instances of `BeanFactory` exist within the Spring project and identify the beans managed by each `BeanFactory`.

Therefore, we devised a process to construct the Bean container:

1. Identify all `BeanFactory` instances within the project.
2. Determine all configuration files utilized by each `BeanFactory`.
3. Traverse these configuration files and create bean definitions.

doi:10.6342/NTU202404078

6.1 Identify all `BeanFactory` in the project

We classify `BeanFactory` instances into two categories:

- `BeanFactory` instances generated by Spring
- `BeanFactory` instances generated by the source code

For the first category, we can determine their configuration through `web.xml`. For instance, `web.xml` defines servlets and uses `<init-param>` to set contextConfigLocation, specifying the location of configuration files. Similarly, `<context-param>` can also set contextConfigLocation to indicate which configuration files the Spring root `BeanFactory` should use. This information can be directly analyzed from `web.xml`.

For the second category of `BeanFactory`, which appears in the source code. We need to analyze which classes instantiate `BeanFactory` and which configuration files are passed during instantiating. Therefore, we have established the following process:

1. Traverse all methods to identify statements that instantiate `BeanFactory`.
2. Identify the types of parameters passed to `BeanFactory` (e.g., fields, variables, methods).
3. Analyze how to inject the types of parameters (e.g., method injection, constructor injection).
4. find the potential values (e.g., configuration paths) for parameters passed to `BeanFactory`

After identifying all statements that instantiate the `BeanFactory`, we need to determine the types of the passed arguments. These arguments could be values, fields, local vari-

doi:10.6342/NTU202404078

ables, method calls, or various other expressions. Therefore, we utilize the Chain-of-Responsibility pattern to handle different expressions and determine their possible values. This approach allows us to easily handle new expressions in the future by simply creating a new handler, thus adhering to the Open-Closed Principle. Refer to Figure 6.1 for the class diagram.

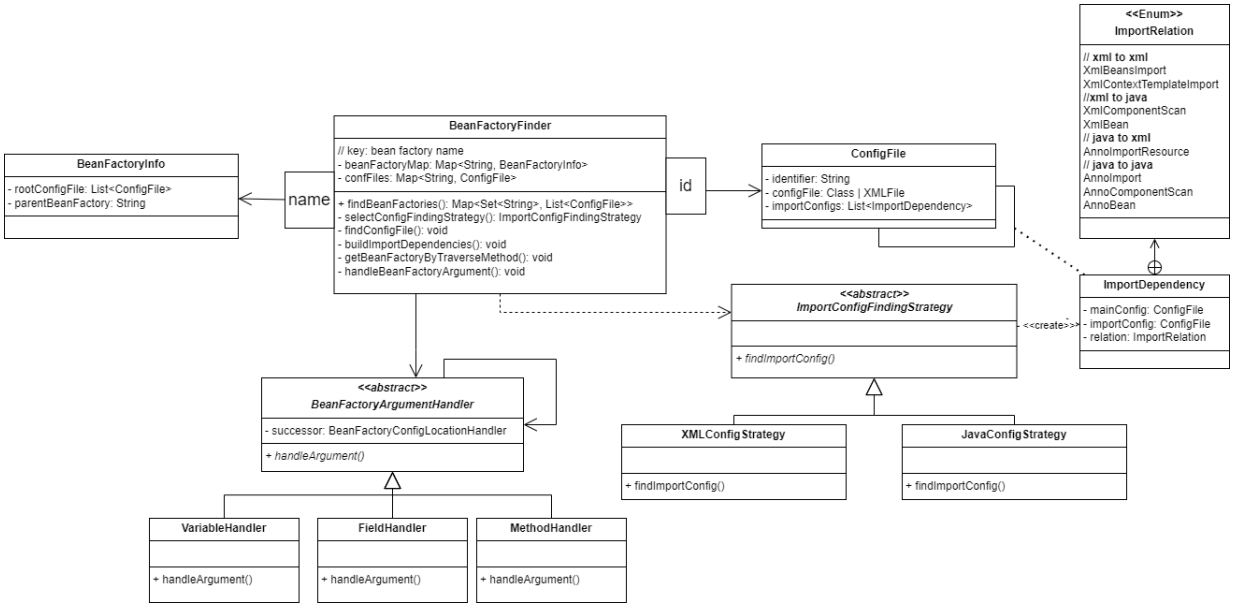


Figure 6.1: Class Diagram - BeanFactoryFinder

6.2 Identify all configurations files used by each BeanFactory

Once we obtain the actual values of the arguments passed during the instantiation of each `BeanFactory` (i.e., the locations of the configuration files), we need to identify all configuration files that are imported together with them. There are two types of configuration files: 1. XML-based configuration files; 2. Java-based configuration files (classes annotated with `@Configuration`). Based on the type of configuration file, we categorize the dependencies between them into four categories:

doi:10.6342/NTU202404078

1. XML configuration file imports XML configuration file
    - `<bean:import/>`
    - `<context-template:import/>`
2. XML configuration file imports Java configuration file
    - `<context:component-scan/>`
    - `<bean:bean/>`, if the bean class is annotated with `@Configuration`
3. Java configuration file imports XML configuration file
    - `@ImportResource`
4. Java configuration file imports Java configuration file
    - `@Import`
    - `@ComponentScan`

After establishing the aforementioned dependencies for each configuration file, we can use the configuration file passed to the `BeanFactory` to identify all the configuration files that this `BeanFactory` will utilize.

6.3 Parse configuration files and extract bean definitions

The final step is to analyze the configuration files we have identified to extract complete information about the beans. We construct our own `BeanDefinition` based on Spring's `BeanDefinition`, which includes attributes such as id, class, init-method, destroy-method, scope, abstract, primary, and dependencies (all other beans used). We employ

doi:10.6342/NTU202404078

the strategy pattern to obtain the `BeanDefinition` for beans declared in different types of configuration files. We handle three types of beans: 1. Methods annotated with `@Bean`; 2. Beans defined in XML configurations; 3. Classes annotated with `@Component` (or `@Service`, `@Repository`, `@Controller`).

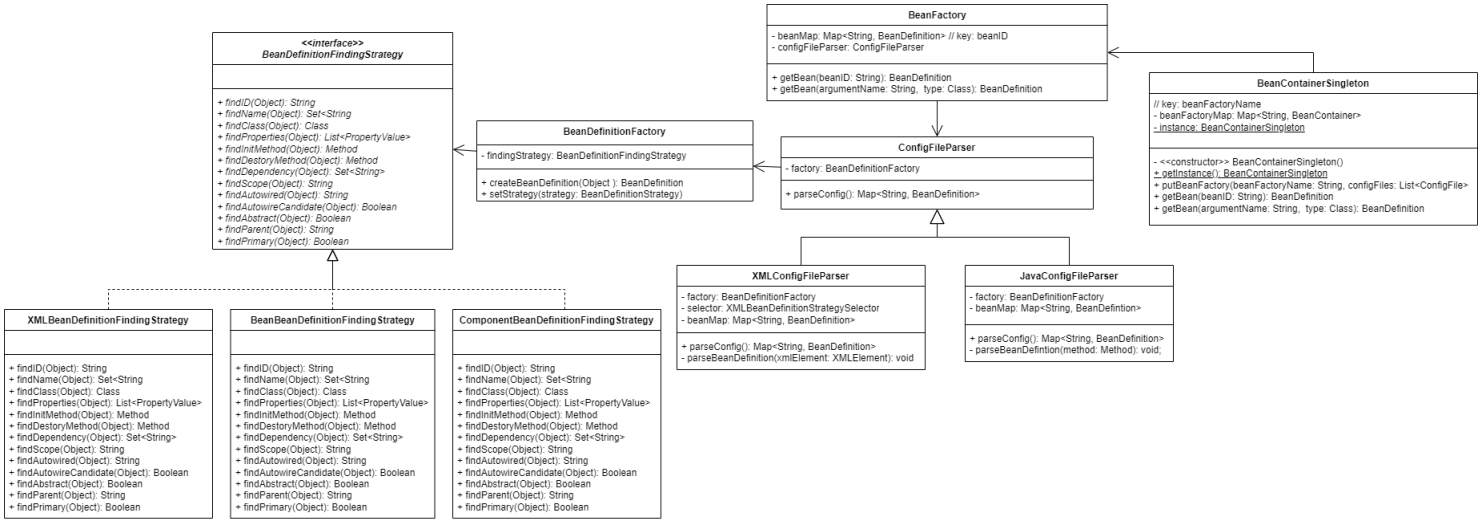


Figure 6.2: Class Diagram - BeanContainerSingleton

We will store the retrieved `BeanDefinition` in a `BeanContainer` and use the Singleton pattern to create this container (as shown in Figure 6.2). This approach serves two purposes: 1. Ensuring that only one `BeanContainer` exists while processing a single project; 2. Acting as a global variable, allowing us to access it directly via static methods without needing to pass the object around.

doi:10.6342/NTU202404078

Chapter 7

Conversion rules

Next, we define conversion rules for each element in every namespace. Each namespace represents XML configuration provided by Spring projects for handling various application-layer domains. We need to address the following 12 namespaces, 11 of which are provided by the Spring project, including `<beans/>`, `<util/>`, `<aop/>`, `<mvc/>`, `<batch/>`, `<security/>`, `<context/>`, `<task/>`, `<int/>`, `<tx/>`, and `<cache/>`. The remaining one, `<context-template/>`, is provided by a project from GitHub [3].

Compiling conversion rules will impact our subsequent verification process, as our verification must ensure that the beans produced by XML configuration files match those produced by Java configuration files. Therefore, many conversion rules must consider more than just the Spring documentation. So, how do we compile conversion rules for each namespace?

1. The documentation provided by Spring.
2. Execute the XML configuration files and examine the created beans.
3. Trace the source code that parses these XML elements.

doi:10.6342/NTU202404078

After compiling the conversion rules, we have identified the following common conversion rules:

- One XML file →One Java file and one class annotated with `@Configuration`
- One namespace →Zero or more class annotations
- One XML element →One class annotation or one method

Finally, based on the conversion rules for the elements in each namespace, we can summarize them in the diagram shown in Figure 7.1.

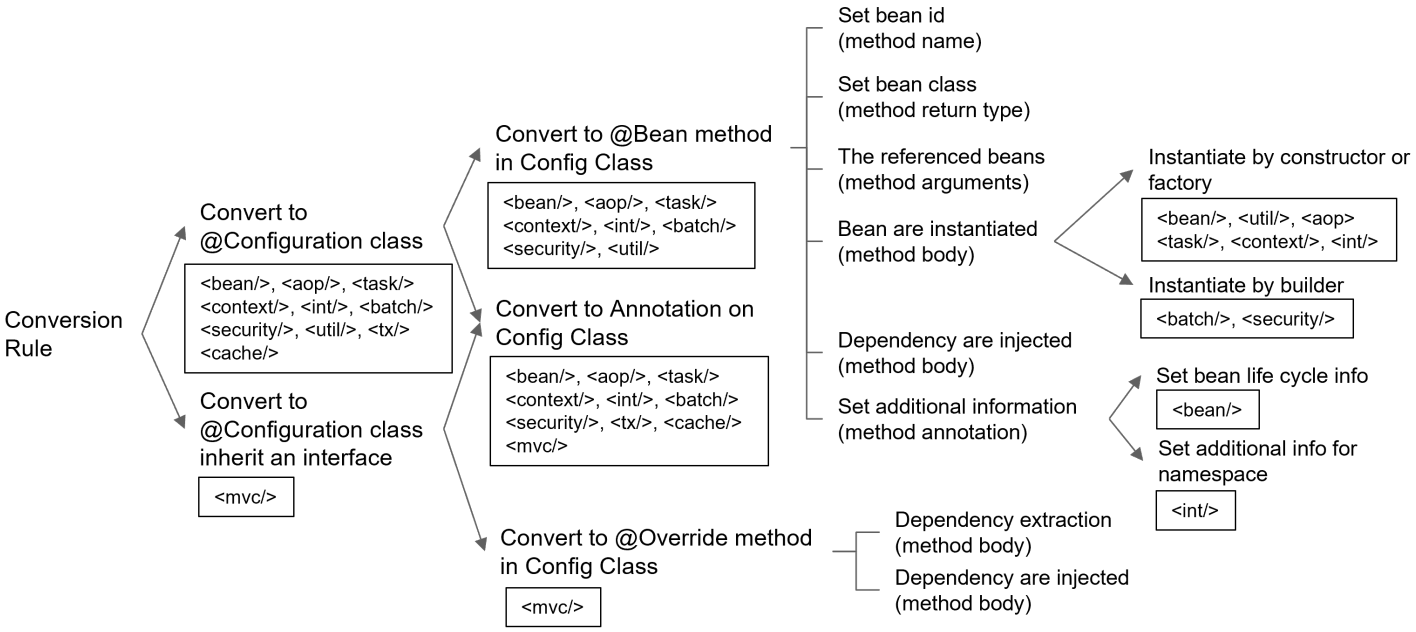


Figure 7.1: Summary Conversion Rule

7.1 Conversion Rule for XML file

1. Java File: Create a java file for the corresponding XML file, and name the java file as “XML's filename + ‘Config’ ”.

doi:10.6342/NTU202404078

2. Java Class: Create a Java class with the same name as the Java file, and annotate this class with `@Configuration`.

7.2 Conversion Rule for `<beans/>` Namespace

This namespace is provided by the Spring Framework project and is used to define and configure beans managed by Spring, supporting dependency injection and scope attribute.

7.2.1 For `<bean/>`

3. Java Method: For each `<bean/>` in the XML file, create a method annotated with `@Bean` in the class established in the second step.

   (3.1) Method Annotations

      i. Configure `@Bean` based on `<bean: init-method/destroy-method/ scope>`

      ii. Configure `@DependsOn` based on `<bean: depends-on>`

      iii. Configure `@Primary` based on `<bean: primary>`

   (3.2) Method Name: Use the `<bean: id>` attribute as the method name.

   (3.3) Method Return Type: Use the `<bean: class>` attribute.

   (3.4) Method Parameters: Use beans specified in `<constructor-arg: ref>`, `<property: ref>`, and `<entry: value-ref>` attributes as method parameters, and check the parameters used by the constructor of the bean class to add any missing beans as method parameters.

doi:10.6342/NTU202404078

(3.5) Method Body: Instantiate the bean

i. Use constructor injection to instantiate the bean based on `<constructor-arg/>`

ii. Use setter injection to inject the bean based on `<property/>`

7.2.2 For `<import/>`

3. Class Annotation: Add the `@Import` annotation to the class created in the second step. In the parameters of `@Import`, set it to the Java configuration class corresponding to the XML file specified in the `<import: resource>` attribute.

7.3 Conversion Rule for `<util/>` Namespace

This namespace is provided by the Spring Framework project and is used to provide utility tools, allowing the use of various collection.

7.3.1 For `<map/>`, `<set/>`, `<list/>`

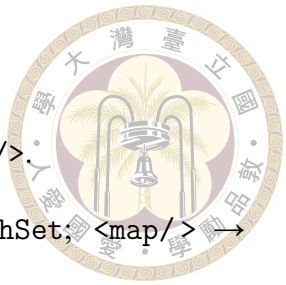
3. Java Method: For each collection element in the XML file, create a method annotated with `@Bean` in the class established in the second step.

(3.1) Method Name: Use the `<util: id>` attribute.

(3.2) Method Return Type: `<list/>`: Return type is `List`; `<set/>`: Return type is `Set`; `<map/>`: Return type is `Map`.

(3.3) Method Parameters: Beans specified in the `<ref/>`, `<entry: key-ref>`, and `<entry: value-ref>` attributes.

doi:10.6342/NTU202404078



(3.4) Method Body: Instantiate the bean

i. Instantiate the collection based on the type of `<util/>`.

`<list/>` → `ArrayList`; `<set/>` → `LinkedHashSet`; `<map/>` → `LinkedHashMap`.

ii. Inject `<value/>`, `<ref/>`, and `<entry/>` into the collection.

7.4 Conversion Rule for `<aop/>` Namespace

This namespace is provided by the Spring Framework project and is used to configure aspect-oriented programming (AOP), supporting the definition of aspects, advice, and pointcuts.

7.4.1 For `<aop/>` Namespace

3. Class Annotation: Add the `@EnableAspectJAutoProxy` annotation to the class created in the second step.

7.4.2 For `<pointcut/>`

3. Java Method: Create a method annotated with `@Bean` in the class established in the second step.

(3.1) Method Name: Use the `<pointcut: id>` attribute as the method name.

(3.2) Method Return Type: `AspectJExpressionPointcut` class.

(3.3) Method Body: Instantiate the bean

i. Instantiate the `AspectJExpressionPointcut` class.

doi:10.6342/NTU202404078

ii. Use setter injection to inject the pointcut expression into the `AspectJExpressionPointcut` instance.

7.4.3 For Advice Elements

3. Java Method: For each advice element in the XML file (`<before/>`, `<after/>`, `<after-returning/>`, `<after-throwing/>`, or `<around/>`), create a method annotated with `@Bean` in the class established in the second step.

(3.1) Method Annotation: Set `org.springframework.aop.aspectj.AspectJPointcutAdvisor#{index}` in `@Bean`

- Index is assigned based on the order of bean creation, to match XML and Java-generated beans during validation.

(3.2) Method Return Type: `AspectJPointcutAdvisor` class.

(3.3) Method Parameters: `<aop: pointcut-ref>` attribute and `BeanFactory`.

(3.4) Method Body: Instantiate the advice bean

i. Use `beanFactory` as a bridge to obtain the ref bean.

A. Instantiate the `SimpleBeanFactoryAwareAspectInstanceFactory` class.

B. Inject the `<aspect: ref>` attribute as the bean name and `BeanFactory` into the factory instance.

ii. Instantiate a subclass of `AbstractAspectJAdvice` based on the advice element type.

- `<before/>` → `AspectJMethodBeforeAdvice`
- `<after/>` → `AspectJAfterAdvice`

doi:10.6342/NTU202404078

- `<after-returning/>` → `AspectJAfterReturningAdvice`
- `<after-throwing/>` → `AspectJAfterThrowingAdvice`
- `<around/>` → `AspectJAroundAdvice`

iii. Use constructor injection to inject the method, pointcut, and factory into the subclass.

iv. Instantiate the `AspectJPointcutAdvisor` class.

v. Use constructor injection to inject the instantiated subclass into the `AspectJPointcutAdvisor` instance.

7.5 Conversion Rule for `<mvc/>` Namespace

This namespace is provided by the Spring Web MVC project. It is used to configure Spring MVC applications, supporting controllers, view resolvers, and other web components.

7.5.1 For `<mvc/>` Namespace

3. Class Annotation: Add the `@EnableWebMvc` annotation to the class created in the second step.

4. Implementing Class: Implement the `WebMvcConfigurer` interface in the class created in the second step.

doi:10.6342/NTU202404078

7.5.2 For <annotation-driven/>

3. Java Method: For each <mvc/> element, override the configureContentNegotiation() method in WebMvcConfigurer.

   (3.1) Method Parameter: ContentNegotiationConfigurer

   (3.2) Method Body: Inject dependencies

      i. Use applicationContext.getBean() to obtain the dependent beans.

      ii. Use the defaultContentTypeStrategy() method to inject the obtained beans into ContentNegotiationConfigurer.

7.5.3 For <resources/>

3. Java Method: For each <mvc/> element, override the addResourceHandlers() method in WebMvcConfigurer.

   (3.1) Method Parameter: ResourceHandlerRegistry

   (3.2) Method Body: Inject dependencies

      i. Use the addResourceHandler() method of ResourceHandlerRegistry to inject <resources: mapping>.

      ii. Use the addResourceLocations() method of ResourceHandlerRegistry to inject <resources: location>.

7.5.4 For <interceptors/>

3. Java Method: For each <mvc/> element, override the addInterceptors() method in WebMvcConfigurer.

doi:10.6342/NTU202404078

(3.1) Method Parameter: `InterceptorRegistry`

(3.2) Method Body: Inject dependencies

i. Use the `addInterceptor()` method of `InterceptorRegistry` to inject interceptors.

ii. Use the `addPathPatterns()` method of `InterceptorRegistry` to inject `<mapping: path>`.

7.6 Conversion Rule for `<batch/>` Namespace

This namespace is provided by the Spring Batch project. It is used for designing and executing batch processing tasks, including task scheduling and execution.

7.6.1 For `<batch/>` Namespace

3. Class Annotation: Add the `@EnableBatchProcessing` annotation to the class created in the second step.

7.6.2 For `<job/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

(3.1) Method Name: Use the `<job: id>` attribute as the method name.

(3.2) Method Return Type: `Job`

(3.3) Method Parameters: `<listener: ref>`, `<step: ref>`, `<step: next>`, `<job: job-repository>` attributes and `JobBuilderFactory`

doi:10.6342/NTU202404078

(3.4) Method Body: Create the bean instance.

i. Call the `get()` method of `JobBuilderFactory` to inject `<job: id>`.

ii. Call the `repository()` method of `JobBuilder` to inject `<job: job-repository>`.

iii. Call the `start()` and `next()` methods of `SimpleJobBuilder` to inject `<step: ref>`.

iv. Call the `listener()` method of `SimpleJobBuilder` to inject `<listeners/>`.

v. Call the `build()` method of `SimpleJobBuilder` to return the bean.

7.6.3 For `<step/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

(3.1) Method Name: Use the `<step: id>` attribute as the method name.

(3.2) Method Return Type: `Step`

(3.3) Method Parameters: `<tasklet: transaction-manager>`, `<chunk: reader/writer/processor>`, `<listener: ref>` attributes

(3.4) Method Body: Create the bean instance.

i. Call the `get()` method of `StepBuilderFactory` to inject `<step: id>`.

ii. Call the `chunk()` method of `StepBuilder` to inject `<chunk: commit-interval>`.

iii. Call the `reader()`, `writer()`, and `processor()` methods of `SimpleStepBuilder` to inject `<chunk: reader/ writer/`

doi:10.6342/NTU202404078

processor>.

iv. Call the `listener()` method of `SimpleStepBuilder` to inject `<listeners/>`.

v. Call the `build()` method of `SimpleStepBuilder` to return the bean.

7.7 Conversion Rule for `<security/>` Namespace

This namespace is provided by the Spring Security project. It is used to configure Spring Security, offering authentication and authorization functionalities.

7.7.1 For `<security/>` Namespace

3. Class Annotation: Add the `@EnableWebSecurity` annotation to the class created in the second step.

7.7.2 For `<http/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

   (3.1) Method Annotation: Set `org.springframework.security.web.SecurityFilterChain#{index}` in the `@Bean`.

   (3.2) Method Name: `http#{index}`

   (3.3) Method Return Type: `SecurityFilterChain`

   (3.4) Method Parameters: `<custom-filter: ref>`, `<http: authentication-manager-ref>`, `<http: entry-point-ref>` attributes and `HttpSecurity`

doi:10.6342/NTU202404078

(3.5) Method Body: Create the bean instance.

i. Call the `securityMatcher()` method of `HttpSecurity` to inject `<http: pattern>`.

ii. Call the `authenticationEntryPoint()` method of `HttpSecurity` to inject `<http: entry-point-ref>`.

iii. Call the `authenticationManager()` method of `HttpSecurity` to inject `<http: authentication-manager-ref>`.

iv. Call the `addFilterAt()` method of `HttpSecurity` to inject `<custom-filter: ref>`.

v. Call the `build()` method of `HttpSecurity` to return the bean.

7.7.3 For `<authentication-manager/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

(3.1) Method Annotation: Set `org.springframework.security.authentication.AuthenticationManager#{index}` in the `@Bean`.

(3.2) Method Name: `authentication#{index}`

(3.3) Method Return Type: `AuthenticationManager`

(3.4) Method Parameters: `<authentication-provider: ref>` attribute and `HttpSecurity`

(3.5) Method Body: Create the bean instance.

i. Call the `getSharedObject()` method of `HttpSecurity` to obtain `AuthenticationManagerBuilder`.

doi:10.6342/NTU202404078

ii. Call the `authenticationProvider()` method of `AuthenticationManagerBuilder` to inject `<authentication-provider: ref>`.

iii. Call the `build()` method of `AuthenticationManagerBuilder` to return the bean.

7.7.4 For `<global-method-security/>`

3. Class Annotation: Add the `@EnableMethodSecurity` annotation to the class created in the second step.

7.8 Conversion Rule for `<context/>` Namespace

This namespace is provided by the Spring Framework project. It is used to configure the Spring application context, including resource loading and event handling.

7.8.1 For `<component-scan/>`

3. Class Annotation: Set the `@ComponentScan` class annotation according to the `<component-scan: base-package>` attribute, and add it to the class created in the second step.

7.8.2 For `<property-placeholder/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

doi:10.6342/NTU202404078

(3.1) Method Annotation: Set `org.springframework.context.support.PropertySources PlaceholderConfigurer#{index}` in the `@Bean`.

(3.2) Method Name: `property#{index}`

(3.3) Method Return Type: `PropertySourcesPlaceholderConfigurer`

(3.4) Method Body: Create the bean instance.

i. Instantiate the `PropertySourcesPlaceholderConfigurer` class.

ii. Use setter injection to inject the `<property-placeholder: ignore- unresolvable>` attribute into the `PropertySourcesPlaceholderConfigurer` instance.

7.9 Conversion Rule for `<task/>` Namespace

This namespace is provided by the Spring Framework project. It is used to manage and schedule asynchronous task execution.

7.9.1 For `<executor/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

(3.1) Method Name: Use the `<executor: id>` attribute as the method name.

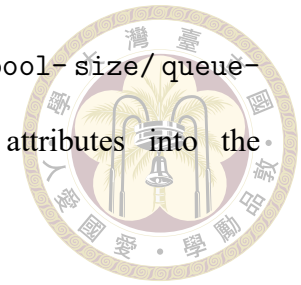
(3.2) Method Return Type: `ThreadPoolTaskExecutor`

(3.3) Method Body: Create the bean instance.

i. Instantiate the `ThreadPoolTaskExecutor` class.

doi:10.6342/NTU202404078

ii. Use setter injection to inject the `<executor: pool-size/ queue-capacity/ keep-alive/ rejection-policy>` attributes into the `ThreadPoolTaskExecutor` instance.



7.9.2 For `<scheduler/>`

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

   (3.1) Method Name: Use the `<scheduler: id>` attribute as the method name.

   (3.2) Method Return Type: `ThreadPoolTaskScheduler`

   (3.3) Method Body: Create the bean instance.

      i. Instantiate the `ThreadPoolTaskScheduler` class.

      ii. Use setter injection to inject the `<executor: pool-size>` attribute into the `ThreadPoolTaskScheduler` instance.

7.9.3 For `<annotation-driven/>`

3. Class Annotation: Add the `@EnableAsync` annotation to the class created in the second step.

4. Class Annotation: Add the `@EnableScheduling` annotation to the class created in the second step.

doi:10.6342/NTU202404078

7.10 Conversion Rule for <int/> Namespace

This namespace is provided by the Spring Integration project. It is used to design and configure Spring Integration flows, supporting message and event-driven architecture.

7.10.1 For <int/> namespace

3. Class Annotation: Add the `@EnableIntegration` annotation to the class created in the second step.

7.10.2 For <channel/>

3. Java Method: Create a method annotated with `@Bean` in the class created in the second step.

   (3.1) Method Name: Use the `<channel: id>` attribute as the method name.

   (3.2) Method Return Type:

   - If it contains the `<queue/>` sub-element: Return type is `QueueChannel`.
   - If it contains the `<dispatcher/>` sub-element: Return type is `ExecutorChannel`.
   - Otherwise, return type is `DirectChannel`.

   (3.3) Method Parameters: `<interceptors/>` sub-elements and `<dispatcher: task-executor>` attribute.

   (3.4) Method Body: Create the bean instance.

   i. Instantiate the class corresponding to the method return type in step (3.2).

doi:10.6342/NTU202404078

ii. Use constructor injection to inject the `task-executor` into the `ExecutorChannel` instance.

iii. Call the `addInterceptor()` method to add interceptors to the instance created in step (i).

7.10.3 For `<gateway/>`

3. Java Method: Create a method annotated with `@Bean` and `@Gateway` in the class created in the second step.

(3.1) Method Annotation: Set the `<gateway: default-payload-expression>` attribute in the `@Gateway` annotation.

(3.2) Method Name: Use the `<gateway: id>` attribute as the method name.

(3.3) Method Return Type: `GatewayProxyFactoryBean`

(3.4) Method Body: Create the gateway bean instance.

i. If it contains `<int:method/>` sub-elements:

A. Instantiate the `LinkedHashMap<String, GatewayMethodMetadata>` class.

B. For each `<method/ >` sub-element, instantiate the `GatewayMethodMetadata` class.

C. Use the `setRequestChannelName()` method to inject the `<method: request-channel>` into the `GatewayMethodMetadata` instance.

D. Use the `setReplyChannelName()` method to inject the `<method: reply-channel>` into the `GatewayMethodMetadata` instance.

doi:10.6342/NTU202404078

E. Use the `<method: name>` attribute as the key and the `GatewayMethodMetadata` instance as the value, placing them into the `LinkedHashMap` instance.

ii. Instantiate the `GatewayProxyFactoryBean` class.

iii. Use constructor injection to inject the `<gateway: service-interface>` into the `GatewayProxyFactoryBean` instance.

iv. Use setter injection to inject the `<gateway: error- channel/ default-request-channel/default-reply-channel>` and the previously created `LinkedHashMap` into the `GatewayProxyFactoryBean` instance.

7.10.4 For `<service-activator/>`

3. Create a method annotated with `@Bean`, `@EndpointId`, and `@ServiceActivator`.

(3.1) Method Name: `serviceActivator{index}`

(3.2) Method Annotations:

(3.1) Use `"org.springframework.integration.config.ServiceActivator FactoryBean#{index}"` in the `@Bean` annotation.

(3.2) Use `"org.springframework.integration.config.ConsumerEndpoint FactoryBean#{index}"` in the `@EndpointId` annotation.

(3.3) Set the `<service-activator: input-channel/auto-startup>` attributes and `<int:poller/>` sub-elements in the `@ServiceActivator` annotation.

(3.3) Method Return Type: `MessageHandler`

doi:10.6342/NTU202404078

(3.4) Method Parameters: `<service-activator: ref>` attribute.

(3.5) Method Body: Create the service activator bean instance.

i. Instantiate:

A. Scenario 1: If it contains the `<service- activator: expression>` attribute:

- Instantiate the `ExpressionEvaluatingMessageProcessor` class.
- Inject the expression into the `ExpressionEvaluatingMessageProcessor` instance.
- Instantiate the `ServiceActivatingHandler` class.
- Use constructor injection to inject the `ExpressionEvaluatingMessageProcessor` instance into the `ServiceActivatingHandler` instance.

B. Scenario 2: If it contains the `<service-activator: ref>` attribute:

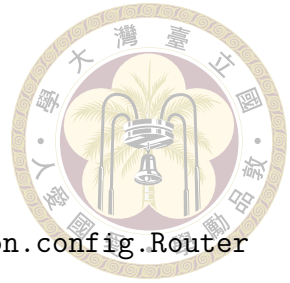
- Instantiate the `ServiceActivatingHandler` class.
- Use constructor injection to inject the `ref` and `method` into the `ServiceActivatingHandler` instance.

ii. Use setter injection to inject the `output- channel` and `primaryExpression` into the `ServiceActivatingHandler` instance.

7.10.5 For `<router/>`

3. Create a method annotated with `@Bean`, `@EndpointId`, and `@Router`

doi:10.6342/NTU202404078



(3.1) Method name: `router{index}`

(3.2) Method annotations:

(3.1) Use `org.springframework.integration.config.RouterFactoryBean#{index}` in `@Bean`

(3.2) Use `org.springframework.integration.config.ConsumerEndpointFactoryBean#{index}` in `@EndpointId`

(3.3) Configure the `<router: input- channel>` attribute and `<int:poller/>` sub-element in `@Router`

(3.3) Return type: `MessageHandler` class

(3.4) Parameters: `<router: ref/>` attribute

(3.5) Method body: Create a router bean instance

i. Instantiation:

- Case 1: If `<router: expression>` attribute is present:

A. Instantiate `ExpressionEvaluatingRouter` class

B. Inject the expression into the `ExpressionEvaluatingRouter` instance

- Case 2: If `<router: ref>` attribute is present:

A. Instantiate `MethodInvokingRouter` class

B. Use constructor injection to inject ref and method into the `MethodInvokingRouter` instance

ii. Use setter injection to inject componentName into the router bean instance

iii. Use setter injection to inject `<int:mapping/>` sub-elements into the

doi:10.6342/NTU202404078

router bean instance

7.10.6 For <filter/>

3. Create a method annotated with @Bean, @EndpointId, and @Filter

   (3.1) Method name: filter{index}

   (3.2) Method annotations:

      (3.1) Use org.springframework.integration.config.FilterFactoryBean#{index} in @Bean

      (3.2) Use org.springframework.integration.config.ConsumerEndpointFactoryBean#{index} in @EndpointId

      (3.3) Configure the <filter: input-channel> attribute in @Filter

   (3.3) Return type: MessageHandler class

   (3.4) Parameters: <filter: discard-channel> attribute

   (3.5) Method body: Create a filter bean instance

      i. Instantiate ExpressionEvaluatingSelector class

      ii. Use constructor injection to inject the expression into the ExpressionEvaluatingSelector instance

      iii. Instantiate MessageFilter class

      iv. Use constructor injection to inject the selector into the MessageFilter instance

      v. Use setter injection to inject <filter: discard-channel/output-channel> and componentName into the MessageFilter instance

doi:10.6342/NTU202404078

7.10.7 For <splitter/>

3. Create a method annotated with @Bean, @EndpointId, and @Splitter

   (3.1) Method name: splitter{index}

   (3.2) Method annotations:

      (3.1) Use org.springframework.integration.config.SplitterFactoryBean#{index} in @Bean

      (3.2) Use org.springframework.integration.config.ConsumerEndpointFactoryBean#{index} in @EndpointId

      (3.3) Configure the <splitter: input-channel> attribute in @Splitter

   (3.3) Return type: MessageHandler class

   (3.4) Parameters: <splitter: ref> attribute

   (3.5) Method body: Create a splitter bean instance

      i. Instantiate MethodInvokingSplitter class

      ii. Use constructor injection to inject ref and method into the MethodInvokingSplitter instance

      iii. Use setter injection to inject <splitter: output-channel> attribute and componentName into the MethodInvokingSplitter instance

7.10.8 For <transformer/>

3. Create a method annotated with @Bean, @EndpointId, and @Transformer

   (3.1) Method name: transformer{index}

doi:10.6342/NTU202404078

(3.2) Method annotations:

(3.1) Use `org.springframework.integration.config.Transformer FactoryBean#{index}` in `@Bean`

(3.2) Use `org.springframework.integration.config.ConsumerEndpoint FactoryBean#{index}` in `@EndpointId`

(3.3) Configure the `<transformer: input- channel>` attribute in `@Transformer`

(3.3) Return type: `MessageHandler` class

(3.4) Parameters: `<transformer: ref>` attribute

(3.5) Method body: Create a bean instance

i. Case 1: If `<transformer: expression>` attribute is present:

A. Instantiate `ExpressionEvaluatingTransformer` class

B. Use constructor injection to inject the expression into the `ExpressionEvaluatingTransformer` instance

ii. Case 2: If `<transformer: ref>` attribute is present:

A. Instantiate `MethodInvokingTransformer` class

B. Use constructor injection to inject ref and method into the `MethodInvokingTransformer` instance

iii. Instantiate `MessageTransformingHandler` class

iv. Use constructor injection to inject the transformer into the `MessageTransformingHandler` instance

v. Use setter injection to inject `<transformer: output- channel>`, component name, and primary expression into the

doi:10.6342/NTU202404078

`MessageTransformingHandler` instance

7.11 Conversion Rule for `<tx/>` Namespace

This namespace is provided by the Spring Framework project. It is used to configure declarative transaction management, supporting transaction boundaries and propagation behavior.

7.11.1 For `<annotation-driven/>`

3. Class annotation: Add the `@EnableTransactionManagement` class annotation to the class created in step two

7.12 Conversion Rule for `<cache/>` Namespace

This namespace is provided by the Spring Framework project. It is used to configure Spring's cache management, supporting cache annotations and automatic cache handling.

7.12.1 For `<annotation-driven/>`

3. Class annotation: Add the `@EnableCaching` class annotation to the class created in step two

doi:10.6342/NTU202404078

Figure 7.2: Example - Generated Files for `<context-template>`

7.13 Conversion Rule for `<context-template/>`

This namespace is special as it is not provided by the Spring project but rather comes from a GitHub project. It is used to configure an XML template, which can then be utilized with `<context-template import/>` to substitute variables into this XML template, allowing the creation of many similar but distinct beans from the same template.

Since we cannot create a Java configuration template, a new Java configuration must be created each time `<context-template import/>` is used. Additionally, a properties file is created to store the variables used in the template, as illustrated in Figure 7.2.

7.13.1 For `<import/>`

3. Class Annotation: Add the `@Import` annotation to the class created in step two. In the `@Import` annotation's parameters, include all Java configuration classes corresponding to the XML files specified in the `<import: resource>` attribute.

doi:10.6342/NTU202404078

7.13.2 For <variable/>

1. Properties File: For each set of <variable/> elements, create a properties file named "XMLFileName#{Index}" (where the index denotes the order of elements in the XML file).

2. For each <variable/>, create an entry in the properties file from step 1 with the format: "PropertiesFileName.<variable: name>=<variable: value>".

7.13.3 For imported XML file

1. Java File: Create a Java file for the XML file specified in the <import: resource> attribute, and name it "ImportedXMLFileName_PropertiesFileName_Config".

2. Java Class: Create a Java class with the same name as the Java file, and add @Configuration and @PropertySource annotations. In the @PropertySource annotation, specify the path to the properties file created in 7.13.2.

3. Java Method: Establish a method with the @Bean annotation in the class created in step 2.

   (3.1) Method Annotation: Use the id attribute of the <bean/> element as the parameter for @Bean, prefixed with "PropertiesFileName.".

   (3.2) Method Name: 'method' + #{Index}

   (3.3) Method Return Type: The type specified by the <bean: class> attribute.

   (3.4) Method Parameters: Include BeanFactory and beans referenced by <constructor-arg: ref>, <property: ref>, and <entry: value-ref> attributes as parameters.

doi:10.6342/NTU202404078

- If any of the attributes above contain placeholders, change the parameter type to 'java.lang.String', annotate the parameter with `@Value`, and use the properties value prefixed with "PropertiesFileName.".

(3.5) Method Body: Instantiate the bean

i. Use the BeanFactory from step (3.4) to retrieve dependent beans with placeholder ids.

ii. Instantiate the bean class and inject `<constructor-arg/>`.

iii. Inject `<property/>` using setter methods.

7.14 Implementation

After compiling the conversion rules, we implement conversion rules using the Template Method Pattern. We define concrete methods in the superclass and execute the conversion steps within these methods. If certain steps in the rules require different implementations for various elements, these steps are declared as abstract methods. The specific details of these steps are implemented by the converters (subclass) for each element. See Figure 7.3 for illustration.

doi:10.6342/NTU202404078

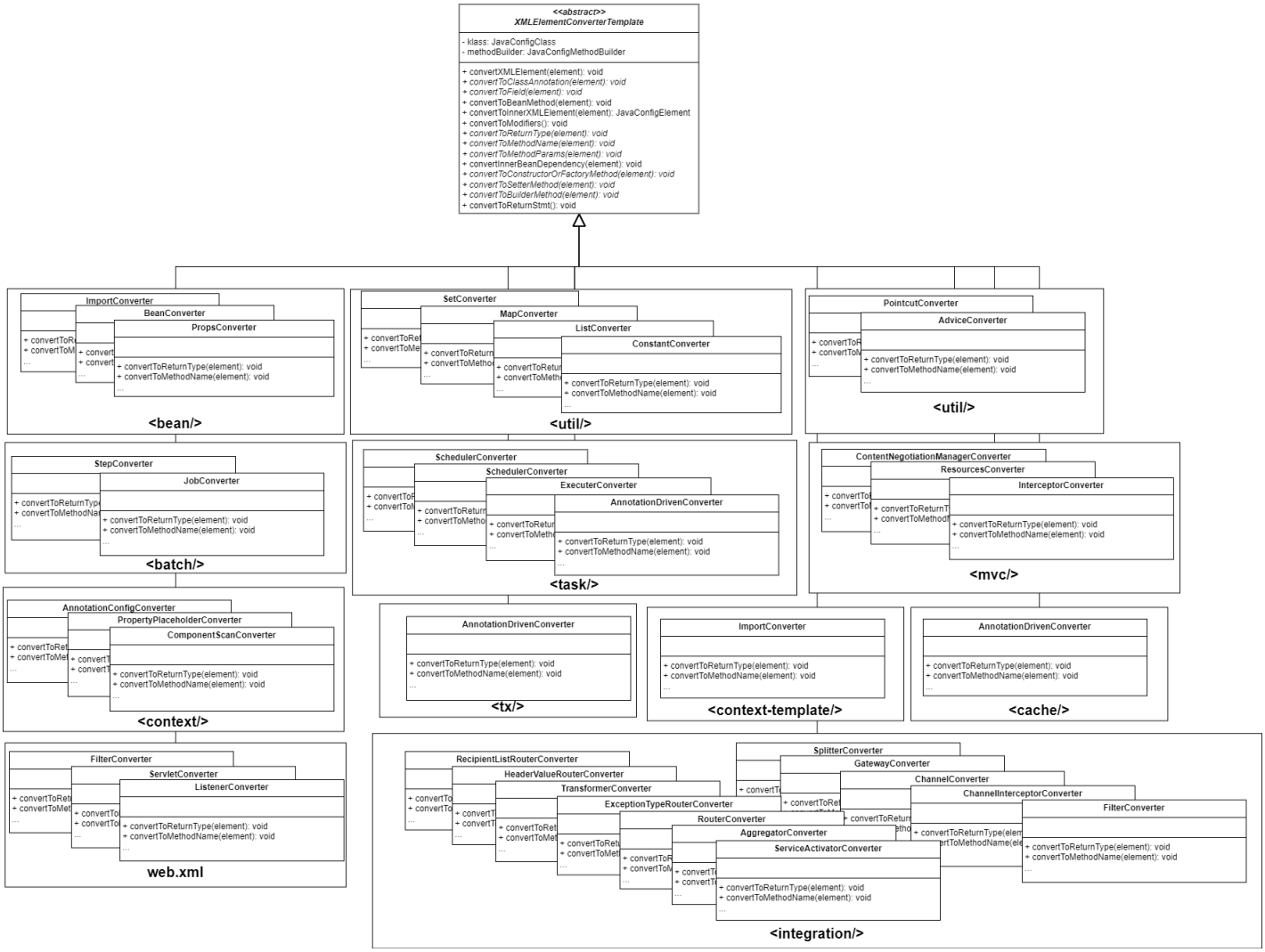


Figure 7.3: Class Diagram - XMLElementConverterTemplate

doi:10.6342/NTU202404078

Chapter 8

Generating Java Configuration

This chapter describes how to generate our Java code. As discussed previously, our focus is on creating a Java Configuration Class. This class will contain methods annotated with `@Bean` and methods annotated with `@Override`. The primary purpose of these methods is to instantiate beans and inject dependencies. Based on our observations of Java code, we categorize the method bodies into three types:

1. Creating instances using constructors or factory methods, and injecting dependencies via setter methods.
2. Using the builder pattern to inject dependencies and create instances.
3. Directly returning primitive values or static fields.

We also classify common elements that may be injected into instances:

1. Local variables (inner beans)
2. Method parameters (referenced beans)

doi:10.6342/NTU202404078

3. Values
    - Constants
    - `java.lang.String`
    - `java.lang.Class`
    - Static fields
    - Enum

Finally, we model the Java configuration code based on the above classifications. Since the generated objects form a tree structure, we use the Composite Pattern for implementation and ensure that each object implements `toString()` method to convert it to Java code. Given that Java is a strongly typed language, each element must have a specified type. To handle this, we abstract a class named `JavaConfigElement` which includes two abstract methods: `getType()` and `getElement()`. When generating code with `toString()`, `getType()` provides the type of the element, and `getElement()` provides the element name (variable name or constant). See Figure 8.1.

doi:10.6342/NTU202404078

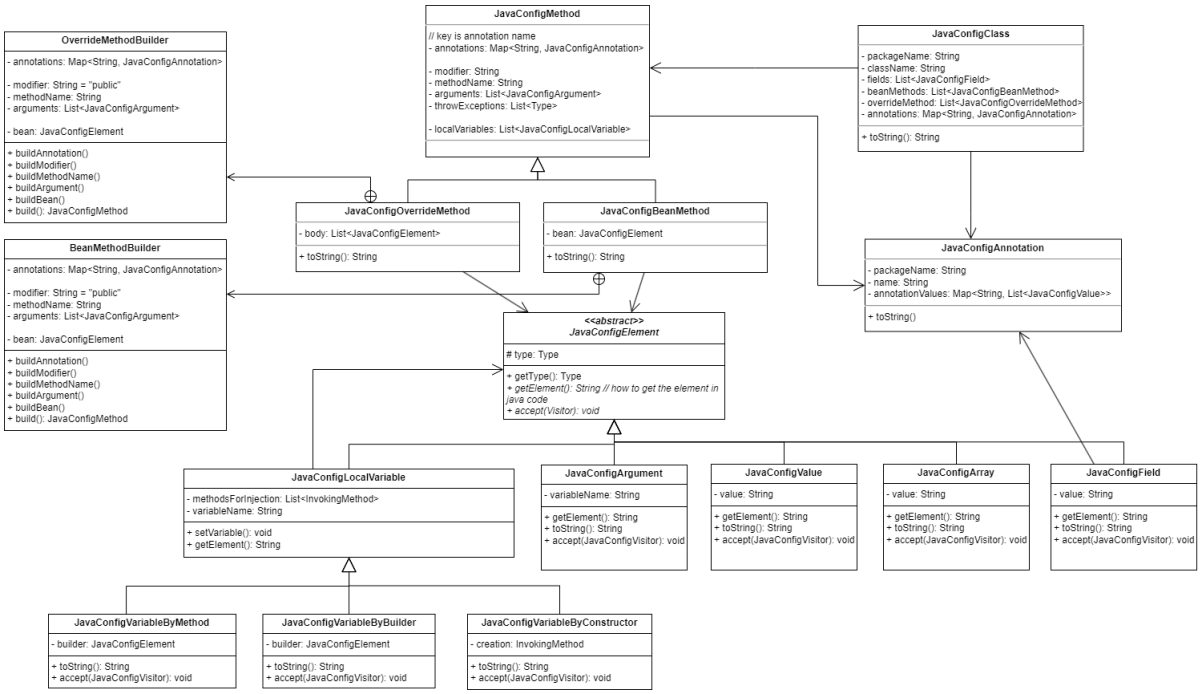


Figure 8.1: Class Diagram - Java Configuration Code

doi:10.6342/NTU202404078

Chapter 9

Verification

Finally, we need to ensure the correctness of the generated Java Configuration. If the beans produced by the generated configuration are equivalent to those in the original project, their behavior should be consistent as well. Therefore, we use the original XML-based project as the benchmark for verification, evaluating the generated beans. The verification process is divided into two parts:

1. Verifying bean definitions
2. Verifying bean instances

9.1 Verify Bean Definition

After parsing the configuration metadata, Spring generates `BeanDefinition`, which is then used to create bean instances. In Spring, `BeanDefinition` includes the following information, detailed in Table 9.1 [2]: 9.1 所示 [2]：

- Class name

doi:10.6342/NTU202404078

Property	Explained in...
Class	Instantiating Beans
Name	Naming Beans
Scope	Bean Scopes
Constructor arguments	Dependency Injection
Properties	Dependency Injection
Autowiring mode	Autowiring Collaborators
Lazy initialization mode	Lazy-initialized Beans
Initialization method	Initialization Callbacks
Destruction method	Destruction Callbacks

Table 9.1: bean definition

- Configuration for bean behaviors (scope, lifecycle, callbacks, etc.)
- References to other beans

However, our testing reveals that there are some differences between `BeanDefinition` generated from XML configuration and Java configuration. For example, the class information differs. XML configuration uses the value specified in the <bean: class> attribute, while Java configuration does not store class information directly but stores it as a factory bean (the configuration class we generate) and a factory method (the methods we generate). Additionally, constructor arguments and properties information are not directly available in the Java configuration; they are encapsulated within method bodies. This information will be verified in the bean instance verification phase.

The process to verify bean definitions is as follows:

1. Identify beans in both projects and store their `BeanDefinition` in a database.
2. Identify beans with the same ID in both XML and Java configurations.
3. Compare the `BeanDefinition` properties of beans with the same ID in the same `BeanFactory`.

doi:10.6342/NTU202404078

9.2 Verify Bean Instance

Bean instances are created by the BeanFactory based on bean definitions. We can retrieve these instances from the BeanFactory, each bean may belong to a different class. Therefore, we first convert the bean instances intounified data structure and serialize these objects into files. Subsequently, we deserialize the files and compare the beans with same bean id.

We categorize bean instances into three types of data structures, designed using the Factory Pattern. Due to deep recursion, a queue-based implementation was used, as illustrated in Figures 9.1 and 9.2:

- **UserDefinedDataType**: Custom-defined classes
- **SystemDefinedDataType**: Containers such as collections, maps, and arrays
- **PrimitiveDataType**: Primitive types or strings

The verification process for bean instances is as follows:

1. Convert bean instances into the three data types: UserDefinedDataType, SystemDefinedDataType, and PrimitiveDataType.
2. Serialize, compress, and save these data types to files.
3. Repeat steps 1 and 2 for all bean instances in XML files from two different time points, performing the process twice.
4. Repeat steps 1 and 2 for all bean instances from Java files.

doi:10.6342/NTU202404078

5. Decompress and deserialize the three files obtained, which include two XML files and one Java file.

6. Compare the bean instances generated from the same XML file at two different time points:

    - Identify any differences during the comparison and determine the iteration count where differences are detected, known as the cut-off point.

7. Use the cut-off point to stop the comparison and verify whether the bean instances in XML and Java are identical.

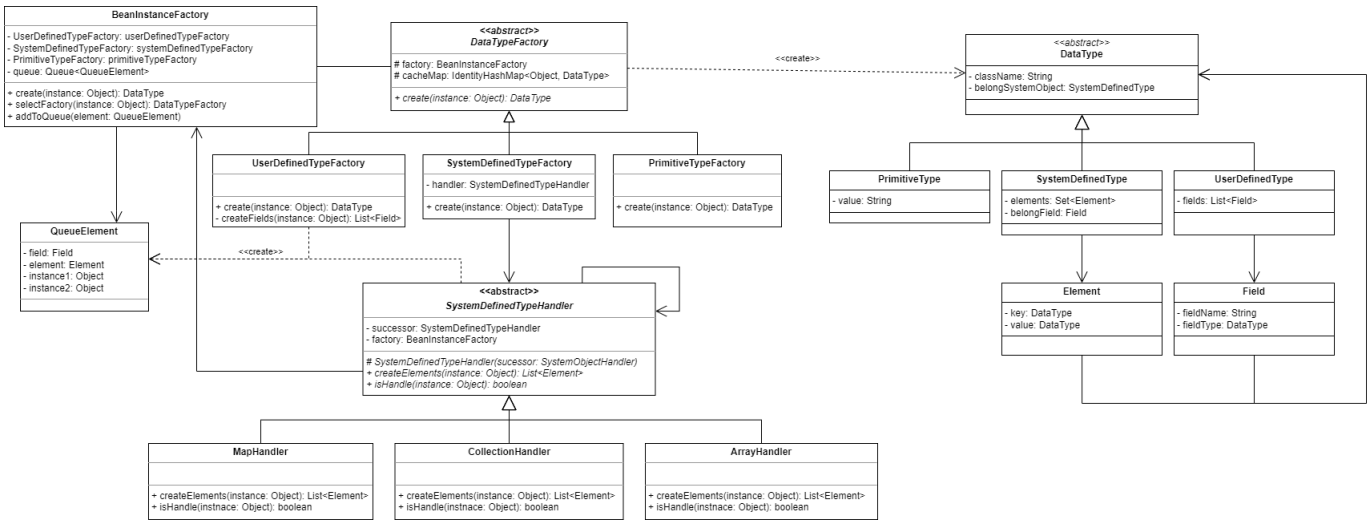


Figure 9.1: Class Diagram - Bean Instance Factory

doi:10.6342/NTU202404078

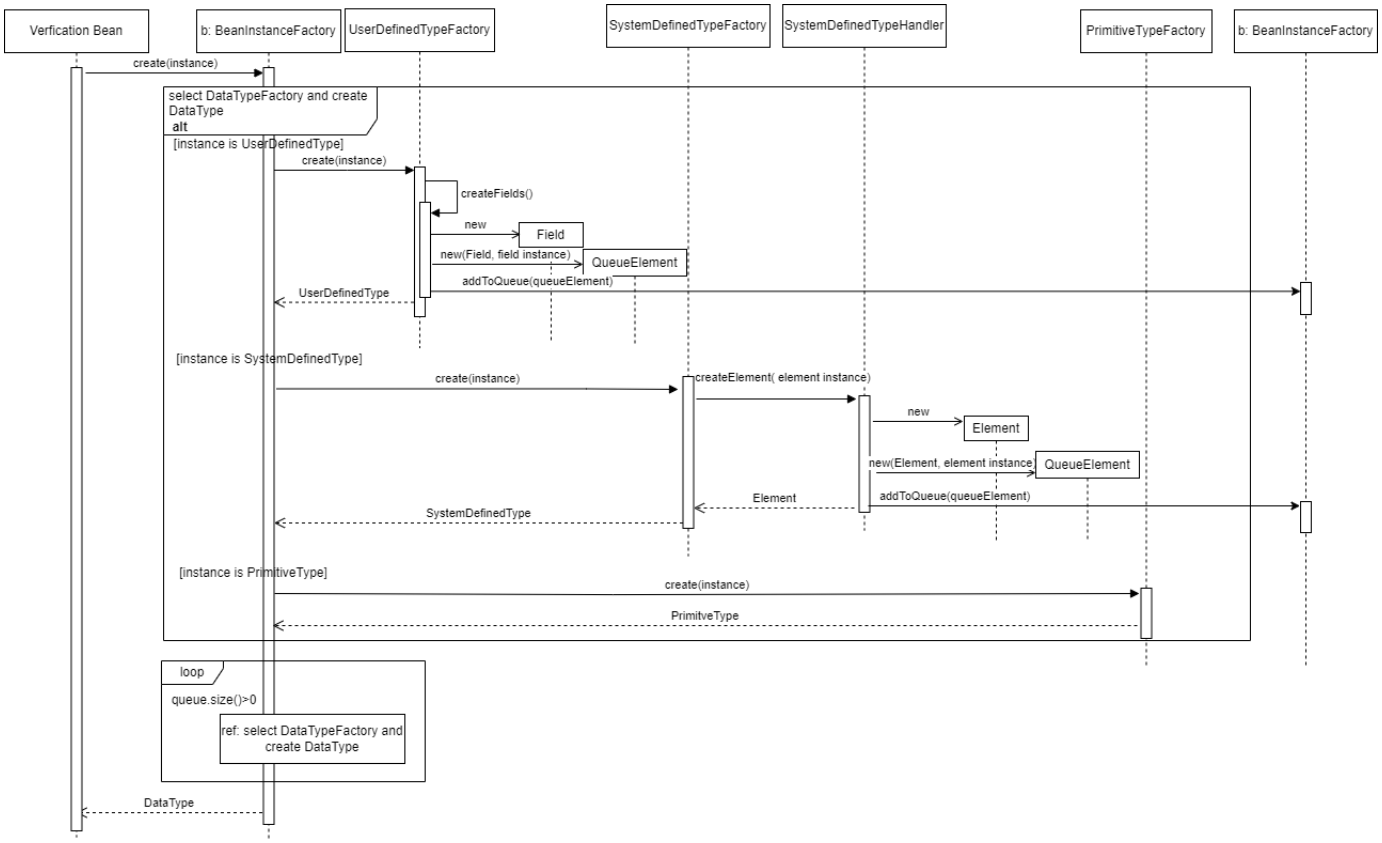


Figure 9.2: Sequence Diagram - BeanInstanceFactory

doi:10.6342/NTU202404078

Chapter 10

Case Study

We analyzed a project consisting of 6,440 Java files, totaling 542,017 lines of code, and utilized 274 XML files to declare approximately 20,000 beans. These XML files involve 12 namespaces, as detailed in Section 7. The project includes 4 Spring-created BeanFactories and 7 BeanFactories generated from the source code.

The conversion process was divided into three steps:

1. Conversion: Generate Java configuration files.
2. Replacement: Replace XML files with Java files.
3. Verification: Execute our verification program to ensure the correctness of each bean.

10.1 Issue

During the replacement and verification processes, we encountered an issue: the Spring AOP (Aspect-Oriented Programming) mechanism did not function successfully.

doi:10.6342/NTU202404078

10.1.1 Problem Description

Before addressing this issue, it is essential to understand the Spring AOP mechanism. Spring AOP aims to allow users to add specific logic to aspects by registering advisor beans. These advisor beans consist of: 1. pointcuts (aspects) and 2. advices (business logic). After registering the advisor beans with the `BeanFactory`, Spring checks each bean in the `BeanFactory` to see if it matches the pointcut criteria. If so, it weaves the advice into the bean. Weaving involves wrapping the bean and advice into a proxy.

Spring provides a bean called `AutoProxyCreator` that automatically performs this operation. The execution timing of `AutoProxyCreator` is related to the lifecycle stages of bean creation in Spring, as shown in Figure 10.1.

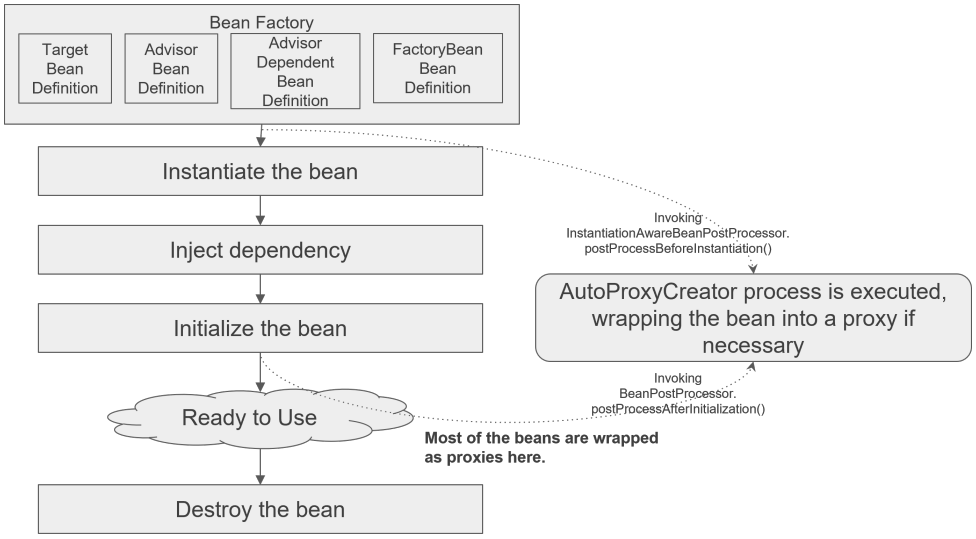


Figure 10.1: The process of creating beans wrapped by proxies

The issue we discovered was that the transaction advisor bean failed to wrap beans with `@Transactional` annotations into proxies.

doi:10.6342/NTU202404078

10.1.2 Root Cause

The root cause was related to converting the advisor beans from XML configuration to Java configuration. This conversion caused `AutoProxyCreator` to generate all `FactoryBean` beans prematurely. Because the method of locating dependent beans changed from direct specification (XML) to type-based (Java), all `FactoryBean` beans were generated to inspect all bean types. During the generation of these `FactoryBean` beans, some advisor beans were also being generated, which led `AutoProxyCreator` to skip using these advisor beans. As a result, these `FactoryBean` beans were not wrapped into proxies. The detailed steps of `AutoProxyCreator` are illustrated in Figures 10.2 and 10.3.

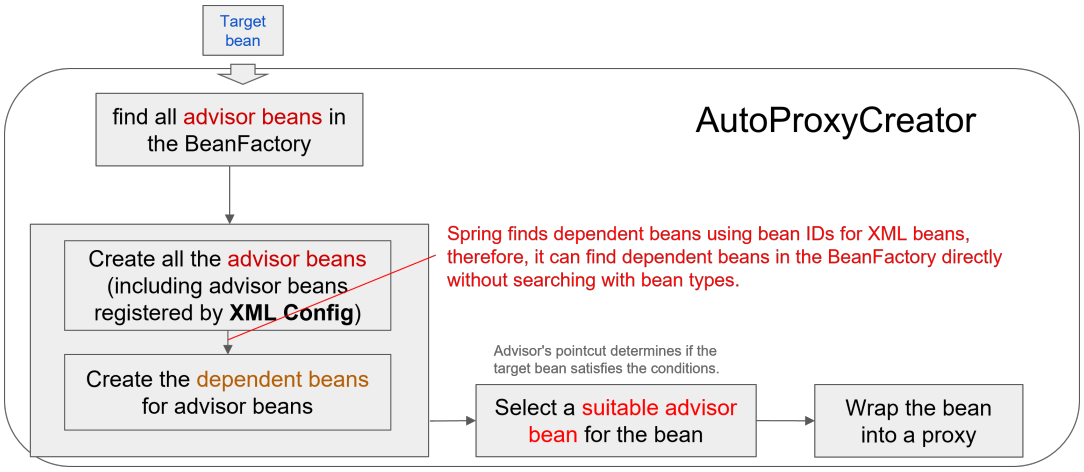


Figure 10.2: The process of AutoProxyCreator with XML advisor bean

10.2 Findings

Finally, we gathered several insights from analyzing the project, summarized as follows:

- The number of BeanFactories in the project.

doi:10.6342/NTU202404078

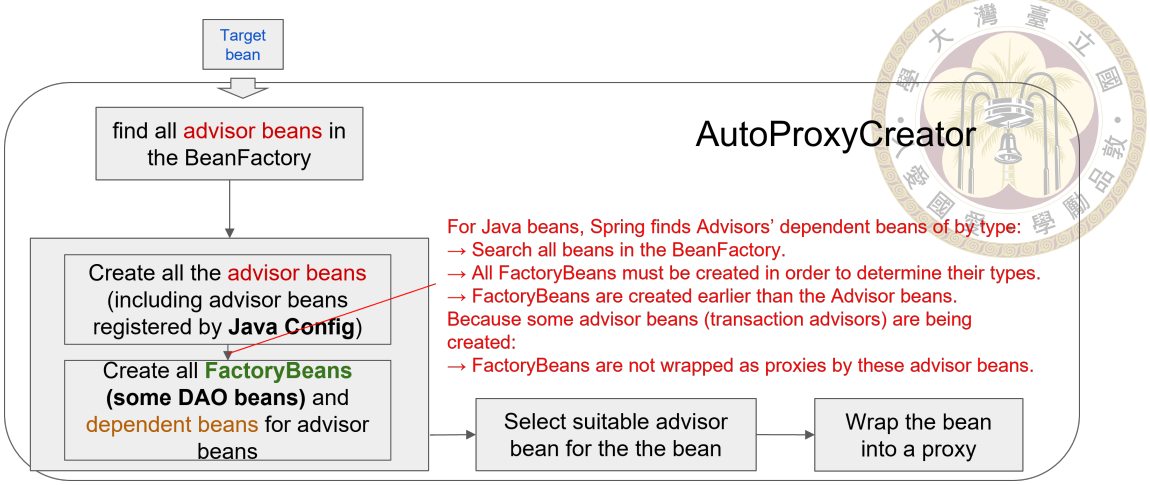


Figure 10.3: The process of AutoProxyCreator with Java advisor bean

- Whether any duplicate beans were overwritten within the same `BeanFactory`.
- Whether any bean IDs contained leading or trailing spaces.
- Which configuration files were not utilized by any `BeanFactory`.
- Which configuration files were used by multiple BeanFactories.

doi:10.6342/NTU202404078

Chapter 11

Conclusion

11.1 Summary

In this study, we proposed an automated process for converting XML configuration into annotation-based configuration. Our contributions include:

- Analyze and examine Spring projects to identify all BeanFactories and their contained beans.
- Compile conversion rules for 12 different namespaces.
- Provide a verification mechanism to verify the correctness of our conversion rules.

11.2 Future work

While we have successfully converted XML configuration into annotation-based configuration, there are several key areas for future research to further enhance the results of this study:

doi:10.6342/NTU202404078

- Compile conversion rules for additional namespaces to handle more Spring project.
- Automate the transformation of Spring framework projects into Spring Boot projects to leverage the additional benefits provided by Spring Boot

doi:10.6342/NTU202404078

References

[1] dom4j. https://dom4j.github.io/.

[2] Spring bean overview. https://docs.spring.io/spring-framework/reference/core/beans/definition.html.

[3] spring-context-template. https://github.com/MarkusBernhardt/spring-context-template.

[4] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. Tamburri, and T. Lynn. Microservices migration patterns. Software: Practice and Experience, 48, 07 2018.

[5] L.-S. Chen. From monolithic to microservice: A dependency decoupling approach. Master's thesis, National Taiwan University, 2023.

[6] H.-Y. Huang. Atomic service generation from java-based open source software. Master's thesis, National Taiwan University, 2019.

[7] R. Laigner, M. Kalinowski, L. Carvalho, D. Mendonça, and A. Garcia. Towards a catalog of java dependency injection anti-patterns. In Proceedings of the XXXIII Brazilian Symposium on Software Engineering, SBES '19, page 104 – 113, New York, NY, USA, 2019. Association for Computing Machinery.

doi:10.6342/NTU202404078

[8] V. Raj and R. Sadam. Patterns for migration of soa based applications to microservices architecture. Journal of Web Engineering, 07 2021.

[9] E. Razina and D. Janzen. Effects of dependency injection on maintainability. In Proceedings of the 11th IASTED International Conference on Software Engineering and Applications, SEA '07, page 7–12, USA, 2007. ACTA Press.

[10] H. Rocha and M. T. Valente. How annotations are used in java: An empirical study. In International Conference on Software Engineering and Knowledge Engineering, 2011.

[11] J.-J. Yu. Construct service components from open source java projects. Master's thesis, National Taiwan University, 2021.

[12] Z. Yu, C. Bai, L. Seinturier, and M. Monperrus. Characterizing the usage, evolution and impact of java annotations in practice. IEEE Transactions on Software Engineering, 47(5):969–986, 2021.

doi:10.6342/NTU202404078