An Empirical Study about Software Architecture Configuration Practices with the Java Spring Framework

Quentin Perez, Alexandre Le Borgne, Christelle Urtado, Sylvain Vauttier

An Empirical Study about Software Architecture Configuration Practices with the Java Spring Framework

Quentin Perez, Alexandre Le Borgne, Christelle Urtado, and Sylvain Vauttier

LGI2P, IMT Mines Ales, Univ Montpellier, Ales, France

{Quentin.Perez, Alexandre.Le-Borgne, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

Abstract

Software architecture modeling plays a key role in software development and, beyond, in software quality. The Spring framework is widely used in industry to deploy software. This paper evaluates whether Spring fosters good practices for architecture definition. It describes the results of an empirical study, based on a corpus of open-source Spring projects. Analysis shows that a strong (70%) majority of projects mixes all Spring architecture definition features. This can be considered as a pragmatic use of a very flexible tool. However, few good practice documentation and tool assistance exist to prevent hazardous architecture constructions. The paper highlights these situations and concludes on recommendations to assist developers.

Keywords: Software architecture, architecture deployment, architecture configuration, empirical software engineering, Spring framework, GitHub open-source project.

1 Introduction

Architecture design is a critical issue that impacts software engineering [5]. Architectures are the natural consequence of modularity: They compose software from elementary components that can easily be developed, tested, maintained or reused.

Spring is a popular industrial framework designed for architecture development in Java. As established by a survey involving 2044 developers[1], it is the most used framework for web-service development. Spring has evolved over time with technologies (*e.g.*, adoption of Java annotations) or application needs (*e.g.*, automation of deployment). Spring now provides developers with multiple features, that complement one another and sometimes overlap, for the architecture definition.

The remainder of this paper is organized as follows. Section 2 motivates this work. Section 3 first describes the features provided by the Spring framework then compares them with respect to expected qualities. Section 4 describes the empirical study on developers' practices. First , the methodology used for GitHub data extraction is presented. Then, the statistical analysis is developed before identifying threats to the study's validity and presenting the implemented tool that supports our study. To conclude, Section 5 presents related works and Section 6 draws perspectives for this work.

2 Motivations

Architecture models are used in nearly all steps of the software lifecycle, from its early design, as an abstract, ideal solution to meet users requirements, to its actual deployment and execution. Academic research has proposed many architecture description languages (ADLs) to support architecture conceptual design [11] whereas industry has proposed frameworks for runtime architecture deployment and management [4]. The ideal technology should both be flexible and easy to use for software developers and help document the architecture, increase component reusability and maintainability and manage change for software architects.

Defining a software architecture amounts to describe its components and their connections (the links that support their interactions). In the case of Java software, this amounts to define the constituent objects and the reference bindings to be created at runtime. When no specific archi-

[1] https://zeroturnaround.com/rebellabs/java-tools-and-technologies-landscape-2016/

DOI reference number: 10.18293/SEKE2019-202

tecture management feature is available, architecture construction is classically hard-coded in the main procedure. Otherwise, an architecture deployment descriptor can be defined through a framework to set the architecture up. To compare their modeling capabilities, we have established a set of expected qualities:

Explicitness. Explicit architecture models are defined with dedicated elements, clearly separated from source code.

Declarativity. Declarative architecture models are defined by abstract elements, that specify the expected instantiated structures, not the instantiation code.

Encapsulation. Encapsulated architecture model definitions are not scattered across source code but gathered into modules.

Assistance. Assisted architecture design is supported by tools that verify consistency, use of good practices or architectural styles, and control evolution to prevent architecture drift or erosion [13].

The purpose of this study is to compare the different kinds of architecture descriptors provided by Spring and to run an empirical analysis to determine whether industrial practice is influenced by architecture model qualities.

3 Spring Features and their Qualities

Since Spring 4 (2013), three closely related architecture definition features are offered to developers:

XML descriptors. Architectures are defined by several XML descriptors that are parsed and interpreted at runtime by the Spring container. Architecture components are declared by the *<bean>* tag, which must define an identifier (*id*) and the instantiated class (*class*). Listing 1 defines a small home automation architecture composed of an *Orchestrator* object connected to a *Lamp* and a *Clock*. Connections are defined by binding bean properties to bean references, as declared by *property* tags. Bean properties correspond to component dependencies (actually instance attributes). These component dependencies are supplied by the Spring container using the declared beans in the XML descriptor (dependency injection).

Configuration classes. Architectures can alternatively be defined by specific Java classes, identified by the *@Configuration* annotation (*see* Listing 2). Configuration classes are automatically detected by the Spring container and executed to build the architecture. Beans are declared by methods annotated by *@Bean*. This enables to program all the necessary pre- or post- bean instantiation treatments required to manage complex settings. Connections are handled by passing bean references to bean constructors, as for the *myOrchestrator* bean in Listing 2, or to bean property setters. In Listing 2, the *clock1* and *lamp1* methods are thus used to retrieve the bean references passed to the *myOrchestrator* constructor. Being genuine Java, configuration classes leverage IDE tools and compile-time verification, as type-safe bean connections and scoped identifiers.

Self-annotated classes. Architecture definition is integrated to the code of the supporting classes thanks to annotations. The *@Component* annotation identifies the classes that will be automatically instantiated by the container to create architecture beans (*see* Listings 3 and 4)[2]. Similarly, the *@Autowired* annotation, which can be associated to attributes, setters or constructors, identifies dependencies (*i.e.*, connections) that will automatically be supplied by the container, using corresponding existing beans (retrieved by name or type). On the one hand, self-annotated classes are the most declarative way to define architectures. On the other hand, architecture definitions are scattered through and mixed with source code.

Moreover, Spring also supports any combination of the aforementioned architecture definition features.

Feature Quality Analysis. XML descriptors and configuration classes enable explicit and encapsulated architecture definitions. Regarding modularity, configuration classes leverage the object-orientation of Java. Considering these three qualities, configuration classes are the best choice and self-annotated classes the worst. This analysis is coherent with technical literature that recommends to limit the use of self-annotated classes to small projects [8, 14, 15]. Besides, self-annotated classes define architectures only with singleton classes. Rather than a limitation, this constraint is intended to enforce strong cohesion between class and architecture structures. Two antagonist approaches of architecture definition are thus supported: architectures that are orthogonal (generic and flexible) or integrated (to avoid architectural drifts) to the code of classes. Self-annotated classes therefore are recommended for projects that are subject to frequent changes [1, 2, 14]. When it comes to choosing features for architecture definition, technical literature only provides scarce guidance, often considering this choice as a matter of developers' tastes [16, 17, 15]. Furthermore, to our knowledge, the qualities of architecture definition mixing features have not yet been studied. We expect that these feature combinations result from rational decisions of experienced developers that use the most adapted feature to different parts of architectures.

4 Empirical Study of Developers' Practices when Using Spring Framework

Data Extraction. A corpus of 524 projects has been extracted from GitHub. In order to consider only significant, quality projects, we applied the selection criteria proposed

[2]Specialized annotations, like *@Service*, *@Repository* and *@Controller*, have been derived from *@Component* to identify the roles of beans in specific architecture kinds, like web-service architectures.

```xml
<beans
xmlns="http://www.springframework.org/schema/beans">
  <bean class="my.smartHome.Clock" id="clock1"/>
  <bean class="my.smartHome.Lamp" id="lamp1"/>
  <bean class="my.smartHome.Orchestrator" id="myOrchestrator">
        <property name="time" ref="clock1" />
        <property name="light" ref="lamp1"  />
  </bean>
</beans>
```

Listing 1: XML descriptor bean configuration

```java
@Configuration
public class BeansConfiguration{
  @Bean
  public Clock clock1(){return new Clock();}
  @Bean
  public Lamp lamp1(){return new AdjustableLamp();}
  @Bean
  public Orchestrator myOrchestrator()
    {return new Orchestrator(clock1(),lamp1());}
}
```

Listing 2: Annotation-based bean configuration

```java
// Clock.java file extract
@Component
public class Clock implements Time {/*...*/}
// Lamp.java file extract
@Component
public class Lamp implements Light {/*...*/}
```

Listing 3: Annotated components

```java
// Orchestrator.java file extract
@Component
public class Orchestrator{
  @Autowired
  private Time time;
  @Autowired
  private Light light;
}
```

Listing 4: Autowired configuration

in Jarczyk *et al.* [7], like the numbers of forks and stars. We thus extracted the last commit of projects with 100 stars or more, forked at least 10 times, written in Java, containing the "Spring" keyword and created after 2010-01-01 (*i.e.*, after Spring release 3). For reproducibility purposes, our metadata is available online[3].

Empirical Analysis. To understand the state-of-practice, we first analyzed which Spring features where used in the studied corpus. Surprisingly, a majority of Spring projects mixes architecture definition features ($\simeq$ 69.3%) and, despite their qualities, configurations classes are only used in a minority of projects ($\simeq$ 6.5%). They are challenged by XML descriptors ($\simeq$ 12.6%) and self-annotated classes ($\simeq$ 11.6%). Apart from routine, in the case of XML descriptors, which are the oldest proposed feature, declarativity may thus be a key quality in developers' decisions. Figure 1 presents the distribution of all the combination of architecture definition features, depending on the size of the projects measured with the Source Lines Of Code (SLOC) metric. As expected, self-annotated classes alone are only used in small projects. Surprise comes again from configuration classes, that are used alone in only rather small projects. Explicit and encapsulated architecture descriptions do not appear to be a primary concern. Again, declarative features are rather used in bigger projects and even the biggest ones. More interestingly, the biggest project seems to require the support of all the features together.

To confirm the intuition that project size has an impact on used features, we evaluate two hypotheses using a non-parametric statistical Kruskal-Wallis test:

Null hypothesis	**Alternative hypothesis**
$\mathcal{H}_0$: No influence of architecture definition features on project size.	$\mathcal{H}_1$: Influence of architecture definition features on project size.

[3] https://github.com/DedalArmy/MISORTIMA/tree/data-study-spring-deploy-features

Defining risk $\alpha = 5\%$, the result of the test is $H \simeq 93.68$ with a *p-value* of $\simeq 5.196^{-18}$. As *p-value* $\leq \alpha$, the null hypothesis $\mathcal{H}_0$ is rejected with a 5% risk. This demonstrates that architecture definition features have a significant influence on project size. As the choice of architecture definition features by the developer obviously does not determine the size of the project, we can infer that the correlation we measure is the reciprocal relation of the actual situation: the choice of architecture definition features is influenced by the size of the project. It would be interesting to study whether the choice of the technique is done *a priori* or evolves, depending on the size of the project.

Finally, we also analyzed isolatedly the use of self-annotated classes on the corpus of 524 Spring projects, depending of their size, using a chi-square test. This test rejects the hypothesis of a relation between project size and use of self-annotated classes, confirming the intuitive analysis of Figure 1. Usage of self-annotated classes thus seems definitely motivated by declarative convenience rather than sound modeling capabilities.

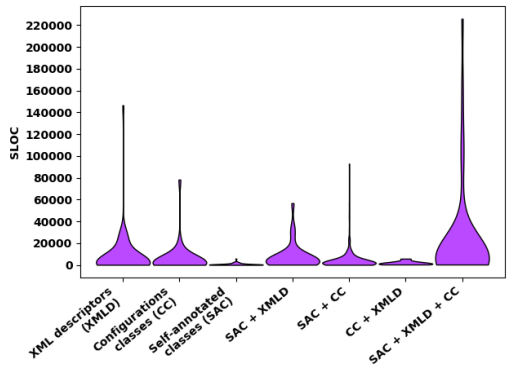

Figure 1: Distribution of architecture definition features regarding SLOC

Threats to Validity. One of the major issues regarding validity of our study is its generalization. This paper focuses exclusively on the Spring framework although Spring is only one among many architecture definition frameworks. This focus might bias the study. It would thus be interesting to explore developers' practices while they use other frameworks in order to compare results with those of this paper. Moreover, by using data provided and mined through the GitHub API we are confronted to the threats already identified by Kalliamvakou *et al.* [9].

5 Related Works

Several works on the deployment of component architecture have already been carried out. Parrish *et al.* [12] modeled the deployment of component architectures in a formal way. It makes it possible to describe several deployment strategies while ensuring deployment safety in terms of installation and component compatibility. Dearle [3] compared the different methods for architecture deployment and showed that dependency injection, as implemented in Spring, is a desirable mechanism for component lifecycle management. Another study has shown that Spring reduces the developers' workload, in particular by extra flexibility, as compared to Java EE [6]. To our knowledge, there is a lack of research on architecture definition combinations and different practices in terms of deployment configuration even if a strong technical literature exists on the subject [8, 14, 15, 16, 17].

6 Conclusion

The empirical analysis we have run on a corpus of 524 projects extracted from the hosting GitHub service about the usage of the architectural definition features provided by the Spring framework leaves opened questions. It shows that usage is strongly related to project size and thus results from rational developer decisions. However, usage seems to be motivated by rather practical than quality concerns, as shown by the predominant use of combined features including self-annotated classes in any size of projects. A first perspective is to study more precisely how Spring features are combined according to project size or domain. We also plan to compare features from other technologies (languages) and frameworks.

Previous work has shown that it is possible to recover an explicit architecture documentation by mining Spring XML configuration files and compiled source code [10]. However, the relevance of results depend on the quality of the architecture modeled in the code. The goal of the on-going work presented in this paper is complementary: fostering the best architectural qualities when source code is used as a standalone model in agile processes.

A more practical perspective is to pursue the development effort to try and better assist developers in their architecture deployment activities by visualization and development of assistance tools.

References

[1] J. Carnell. *Spring Microservices in Action*. Manning, 2017.

[2] I. Cosmina, R. Harrop, C. Schaefer, and C. Ho. *Pro Spring 5: An in-depth guide to the Spring framework and its tools*. Apress, 5th edition, 2017.

[3] A. Dearle. Software deployment, past, present and future. In L. C. Briand and A. L. Wolf, editors, *FOSE workshop at 29th ICSE*, pages 269–284, Minneapolis, USA, May 2007. IEEE.

[4] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35(4):573–591, April 2009.

[5] D. Garlan. Software architecture: A roadmap. In *FOSE track at 20th ICSE*, pages 91–101, Limerick, Ireland, June 2000. ACM.

[6] P. Gupta and M. C. Govil. Spring Web MVC framework for rapid open source J2EE application development: a case study. *IJEST*, 2(6):1684–1689, 2010.

[7] O. Jarczyk, B. Gruszka, S. Jaroszewicz, L. Bukowski, and A. Wierzbicki. GitHub projects. quality analysis of open-source software. In L. M. Aiello and D. McFarland, editors, *6th international SocInfo conference*, volume 8851 of *LNCS*, pages 80–94, Barcelona, Spain, Nov. 2014. Springer.

[8] T. M. Jog. *Learning Spring 5.0*. Packt, 2017.

[9] E. Kalliamvakou, G. Gousios, K. Blincoe, L. Singer, D. M. German, and D. Damian. The promises and perils of mining GitHub. In *11th Working MSR conference*, pages 92–101, Hyderabad, India, May 2014. ACM.

[10] A. Le Borgne, D. Delahaye, M. Huchard, C. Urtado, and S. Vauttier. Recovering three-level architectures from the code of open-source java Spring projects. In X. He, editor, *30th international SEKE conference*, pages 199–202, San Francisco, USA, July 2018.

[11] A. Mokni, C. Urtado, S. Vauttier, M. Huchard, and Z. Huaxi Yulin. A formal approach for managing component-based architecture evolution. *Science of Computer Programming*, 127:24–49, Oct. 2016.

[12] A. Parrish, B. Dixon, and D. Cordes. A conceptual foundation for component-based software deployment. *JSS*, 57(3):193–200, July 2001.

[13] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, Oct. 1992.

[14] D. Rajput. *Spring 5 design patterns: Master efficient application development with patterns such as proxy, singleton, the template method, and more*. Packt, 2017.

[15] A. Sarin and J. Sharma. *Getting Started with Spring Framework: A Hands-On Guide to Begin Developing Applications Using Spring Framework*. CreateSpace, 3rd edition, 2016.

[16] C. Walls. *Spring in action*. Manning, 4th edition, Nov. 2014.

[17] N. S. Williams. *Professional Java for Web Applications*. Wiley & sons, 2014.