Recovering Three-Level Architectures from the Code of Open-Source Java Spring Projects

Alexandre Le Borgne, David Delahaye, Marianne Huchard, Christelle Urtado, Sylvain Vauttier

Recovering Three-Level Architectures from the Code of Open-Source Java Spring Projects

Alexandre Le Borgne[1], David Delahaye[2], Marianne Huchard[2], Christelle Urtado[1], and Sylvain Vauttier[1]

[1]LGI2P, IMT Mines Ales & Montpellier University, Ales, France
{Alexandre.Le-Borgne, Christelle.Urtado, Sylvain.Vauttier}@mines-ales.fr

[2]Montpellier University, CNRS, LIRMM, Montpellier, France
{David.Delahaye, Marianne.Huchard}@lirmm.fr

Abstract

Despite the well-admitted benefits of keeping design decisions as a documentation all along the lifecycle of software, many software projects have lost this information. In order to use design information to guide software maintenance and evolution, this paper proposes to retro-engineer architecture descriptions from source code. The originality of this work is to target a three-leveled architecture description language which represents software specification, configuration and deployment. Retro-engineering these three levels will provide a more precise source of guidance for the maintenance of software. Targeted projects are open-source Java projects that use Spring to describe the implemented "architecture".

Keywords: Component-Based Software Engineering, Model-driven engineering, Architecture retro-engineering from code, Architecture evolution, Component reuse, Architecture reuse.

I. Introduction

As software systems constantly become more complex, retrieving design decisions has become an increasingly important problematics when conceptual documentation is missing. However, despite numerous researches in the field of software architecture reconstruction, few work was dedicated to extract raw ("as implemented") component-based description. It is important, in the first place, to understand design decisions to recover architectures as they are implemented and to not perform any improvement (re-engineering tasks) altogether. Moreover, it is important to represent the software at three abstraction levels in order to trace design decisions through the whole development process. To do so, we use the Dedal [12], [8] architecture description language (ADL) developed by our team. This paper proposes to reconstruct component-based architectures from Java Spring [6] projects.

The remainder of this paper is organized as follows. Section II presents the background of our approach. The core of the paper is developed in Section III where component-based architecture reconstruction from Java Spring projects is explained. Section IV details the existing work in software architecture reconstruction and Section V concludes on perspectives.

II. Background

A. Dedal, a Three-Level Architecture Description Language

Dedal [12], [8] is a three-level architecture description language (ADL) designed to give a representation of the entire life cycle of architectures and a support to manage their evolution. Design is represented by the ***Specification*** level which is composed of abstract component types. Those types are called *roles* which means that they define the functionalities present in the components of the future software. Implementation choices are captured by the ***Configuration*** level. This architecture level is composed of concrete component classes which are realizations of

DOI reference number: 10.18293/SEKE2018-140

the roles. Deployment is described in the ***Assembly*** level. This level is composed of a set of component instances that define how to tailor software for specific execution contexts.

B. Spring

Java Spring framework [6] is widely used in industry to manage the deployment of software architectures. It provides standardized architecture deployment capabilities thanks to a container that is able to handle explicit architecture descriptors. Architecture descriptors are defined as XML files or directly embedded in the code as annotations. They are based on the concept of beans, which define the objects that the container must instantiate and connect in order to set the initial architecture up. For instance, the deployment descriptor of Figure 1a is composed of four beans: *lampDesk*, *lampSitting*, *clock1* and *orchestrator1*.

Connections between beans are handled by the container using dependency injection to preserve decoupling. Beans only declare reference attributes corresponding to their dependencies with other beans. These dependencies are then resolved at runtime thanks to the connections defined in the deployment descriptor. For instance, the deployement descriptor in Figure 1a defines three connections between the orchestrator, the lamps and the clock beans. Those connections are defined by the *property* tag which corresponds to the injection of dependencies. For instance, $< property\ name = "clock"\ ref = "clock1" / >$ sets the $clock1$ bean as the $clock$ property of the $orchestrator1$ bean.

As compared to raw code, Spring projects provide some explicit architectural descriptions. However, these descriptions do not capture design decisions and thus cannot be considered as abstractions of the software architecture.

III. Extracting Component-Based Software Architectures

In order to ease the extraction of information from the Spring descriptor, using a model-driven approach, a small domain specific language named SpringDSL has been developed. It consists of an implementation of the Spring XML descriptor grammar in EMF[1]. XText[2] has been used to automatically generate the corresponding EMF metamodel since we could not find an already developed metamodel. This metamodel enables thus to parse Spring XML descriptors and get all their content as concept instances.

This section discusses how each concept of Dedal is extracted from the source code and a Spring deployment description.

A. Extracting Components

Considering component extraction as a model transformation between SpringDSL and Dedal, only a simple mapping is required to extract the assembly and also a small part of the configuration models. To do so, the eclipse QVTo[3] language is used since it defines model to

[1]https://www.eclipse.org/modeling/emf/[Last seen 03-14-2018].
[2]https://www.eclipse.org/Xtext/ [Last seen 03-14-2018].
[3]https://projects.eclipse.org/projects/modeling.mmt.qvt-oml [Last seen 03-14-2018].

```
<bean class="AdjustableLamp" id="lampDesk" />
<bean class="AdjustableLamp" id="lampSitting" />
<bean class="Clock" id="clock1" />
<bean class="Orchestrator" id="orchestrator1" >
    <property name="lamps">
        <set><ref bean="lampDesk" />
            <ref bean="lampSitting" /></set></property>
    <property name="clock" ref="clock1" />
</bean>
```

(a) HAS Spring description

```
public class Clock implements Time {
    public void run() {...}
}
public class AdjustableLamp extends Light {
    ...
    public void switchState(State s) {...}
    public void adjustInensity(int value) {...}
}
public class Orchestrator extends HomeOrchestratorImpl {
    protected void run(){...}
}
```

(b) HAS classes instantiated in Figure 1a

```
public interface Time {
    public void run();
}
public abstract class Light {
    ...
    public abstract void switchState(State s);
}
public abstract class HomeOrchestrator {
    Light[] lights;
    Time clock;
    public abstract void run();
}
public class HomeOrchestratorImpl
                        extends HomeOrchestrator {
    protected void run(){...}
    public HomeOrchestratorImpl (Light[] lights,
                        Time clock) {...}
    public Light[] getLights() {...}
    public void setLights(Light[] lights) {...}
    public Time getClock() {...}
    public void setClock(Time clock) {...}
}
```

(c) HAS most abstract classes

Figure 1: HAS implementation

```
component_role HomeOrchestrator_role
    id "_C-nKzSH3EeicKLj-dMtreg" required_interfaces
        interface ILight_HomeOrchestrator_role
            interface_direction required
            implementation ILight_Type;
        interface ITime_HomeOrchestrator_role
            interface_direction required
            implementation Time;
```

Figure 2: Extracted component class: $Orchestrator$

model transformations through the concept of mapping. As a first step, beans are mapped to the component instances of the architecture **Assembly**, using the *id* of the beans as the name of the components and the *class* attribute as the instantiated component class. Thus the bean tag describing the $Orchestrator$ instance $orchestrator1$ in Figure 1a is mapped as the **component_instance** of Figure 3 named $orchestrator1$ that is an **instance_of** the **primitive_component_class** $Orchestrator$. If the component class does not exist in the **Configuration** yet, then it is created.

Figures 1b and 1c present an extract from the Java code of the Home Automation Software (HAS) example. Code introspection enables to extract complementary information required to build higher level architecture models and more detailed component definitions.

For generating the component roles of the architecture **Specification**, the type hierarchy of the beans classes is analyzed, in order to extract the most generic, thus reusable, architecture model as possible. The main idea is to retrieve the abstract superclasses that are realized by the bean class corresponding to a component class. To extract the component role, the type hierarchy is traversed and the role which is picked is the most generic component role which still holds all the required interfaces that are present in the corresponding component class and which preserves the connections which exist in the **Configuration**. Figure 2 is the **component_role** $HomeOrchestrator\_role$ that is realized by the **primitive_component_class** $Orchestrator$.

B. Extracting Interfaces

Two types of interfaces are distinguished: (i) **provided** and (ii) **required** interfaces. All the methods that are provided by the beans classes must be provided into respective component interfaces. However, in order to not provide only one large interface per component, the interfaces are cut according the type hierarchy of classes. In other words, each implemented interface is mapped as a component interface and if a class does not implement an interface, a "conceptual" interface is extracted, which is composed of the public methods of the beans class, except for getters / setters that are used whether to initialize properties or to manage connections. For extracting required interfaces,

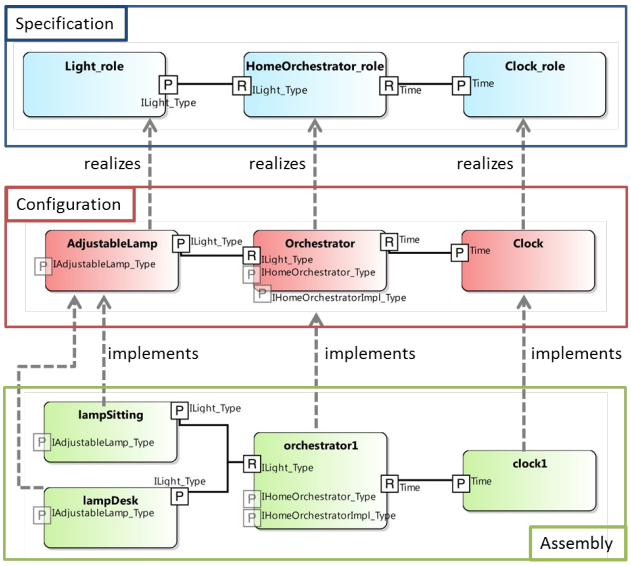

Figure 3: HAS generated Dedal three-level architecture

```
class_connection "_C9ZCsCH3EeicKLj-dMtreg"
    client Orchestrator.ILight_Orchestrator
    server AdjustableLamp.ILight_AdjustableLamp;
```

Figure 4: Extracted class connection

the reference attributes declared in the Spring descriptor are used to manage the binding of the beans. The type and the name of the attribute are used to generate the type and the name of the corresponding interface.

The $HomeOrchestrator\_role$ extracted interfaces are described in Figure 2 with their names, direction and implemented types.

C. Extracting Connections

The XML description (Figure 1a) makes it possible to start the extraction of connections between components. Indeed, thanks to the dependency injection, clients and servers of connections are identifiable. For instance, in the current example, the *orchestrator1* bean contains a property which refers to *clock1*, so it is possible to map a new connection between *orchestrator1* and *clock1*. This connection is propagated to the **Configuration** level by creating a connection between the two instantiated component classes. Following the same principle, the connection between the realized component roles is created.

Then the interfaces that are implied in connections must be set. To do so, we search among two connected components which are their matching interfaces. For matching two interfaces, their types must be equal and their direction complementary (provided with required). Thus, for instance in our case, $Clock$ provides the $ITime$ interface of type $Time$ and, $Orchestrator$ requires an interface of

the same type (Figure 3). Then those two interfaces match and the connection presented in Figure 4 can be set. Then it is propagated following the instantiate relation between component class interfaces and component instance interface (Figure 3). Finally, connection between roles are set in the same way as the connections between component classes.

Figure 3 is the visual representation of the three-level Dedal architecture which is composed of four component instances into the **Assembly** (that correspond to the beans of the Spring description), the component classes of the **Configuration** that are instantiated by the component instances and also the component roles into the **Specification** which are realized by component classes. The connections between components are also represented.

IV. Related Work

This section narrows the studied approaches to the ones which extract component-based architecture descriptions and, if possible, from object-oriented code. Moreover, retro-engineering approaches which consist in simply abstracting the software artifacts for retrieving raw design decisions are differentiated from re-engineering ones which intend to re-organize the extracted information and / or the software artifacts.

In their work, Ducasse *et al.* [3] defined a taxonomy for categorizing software architecture reconstruction approaches. Following this taxonomy, the **goals** of the discussed approach are twofold. The first goal is to improve component reuse, by extracting component-based architecture descriptions, such as MAP [10], PuLSE / SAVE [7] and ROMANTIC [1], [9] approaches, but targeting the Dedal [12], [8] ADL. The second goal is to provide the foundations for managing conformance checking (Bauhaus [4], [2], DiscoTect [11], PuLSE / SAVE [7]), evolution, co-evolution (PuLSE / SAVE [7], Huang *et al.* [5]) and maintenance using the formal rules that have previously [8] been defined in Dedal.

However none of the studied methods intends to extract raw information of how the software is implemented. Moreover, all the discussed approaches only deal with two levels of abstraction (*i.e.*, implementation and architecture) that may not correspond to the same paradigms (code vs component-based architecture description). Indeed three component-based architecture descriptions are essential for maintaining, evolving, tracking software life-cycle since it gives a more global and direct understanding to the architect which can get an overview of the code structure by managing components. Moreover, even the approaches which seem to fit with the discussed one, either recover architecture in a semi automatic manner from execution trace of software (*i.e.*, DiscoTect [11]) or do not reconstruct raw architecture such as ROMANTIC [1], [9] approach which performs re-engineering of the deployed architecture by clustering classes into bigger semantic components that encapsulate classes.

This is why a retro-engineering approach is proposed that builds three-level component-based architecture description from structural artifacts.

V. Conclusion and Future Work

This paper introduces an approach for software architecture reconstruction, using three levels of architecture models (**Assembly**, **Configuration**, **Specification**). An aspect of future work will be to improve and refine the extraction of the **Specification** for making it more abstract.

Real Spring projects have already been identified in open-source repositories in order to perform large scale experimentations on evolution and reuse. Getting projects from open-source repositories will also allow the implementation of versioning mechanisms.

References

[1] Z. Alshara, A. D. Seriai, C. Tibermacine, H. L. Bouziane, C. Dony, and A. Shatnawi. Materializing architecture recovered from object-oriented source code in component-based languages. In *10th ECSA Proc.*, volume 9839 of *LNCS*, pages 309–325, Copenhagen, Denmark, Nov. / Dec. 2016. Springer.

[2] A. Christl, R. Koschke, and M. A. Storey. Equipping the reflexion method with automated clustering. In *12th WCRE Proc.*, pages 10–98, Pittsburgh, USA, Nov. 2005.

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

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE TSE*, 29(3):210–224, 2003.

[5] G. Huang, H. Mei, and F. Q. Yang. Runtime recovery and manipulation of software architecture of component-based systems. *Automated Software Engineering*, 13(2):257–281, April 2006.

[6] R. Johnson, J. Hoeller, K. Donald, C. Sampaleanu, R. Harrop, et al. The spring framework – reference documentation. *Interface*, 21:27, 2004.

[7] J. Knodel, M. Lindvall, D. Muthig, and M. Naab. Static evaluation of software architectures. In *10th CSMR Proc.*, pages 279–294, Bari, Italy, March 2006. IEEE.

[8] A. Mokni, C. Urtado, S. Vauttier, M. Huchard, and H. Y. Zhang. A formal approach for managing component-based architecture evolution. *SCP*, 127:24–49, 2016.

[9] A. Shatnawi, A. D. Seriai, H. Sahraoui, and Z. Alshara. Reverse engineering reusable software components from object-oriented APIs. *JSS*, 131:442–460, 2017.

[10] C. Stoermer and L. O'Brien. Map–mining architectures for product line evaluations. In *IEEE / IFIP WICSA Proc.*, pages 35–44, Amsterdam, The Netherlands, Aug. 2001.

[11] H. Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman. DiscoTect: a system for discovering architectures from running systems. In *26th ICSE Proc.*, pages 470–479, Edinburgh, UK, May 2004.

[12] H. Y. Zhang, C. Urtado, and S.Vauttier. Architecture-centric component-based development needs a three-level ADL. In *4th ECSA Proc.*, volume 6285 of *LNCS*, pages 295–310, Copenhagen, Denmark, Aug. 2010. Springer.