ARCADIA

A novel reconfigurable by design highly distributed applications development paradigm over programmable infrastructure

Deliverable D4.1

Description of the Supported Metadata Annotations

Editor(s):	Panagiotis Gouvas
Responsible Organization(s):	Ubitech Ltd
Version:	1.00
Status:	Final
Date:	13/06/2016
Dissemination level:	Public

Deliverable fact sheet

Grant Agreement No.:	645372
Project Acronym:	ARCADIA
Project Title:	A novel reconfigur**A**ble by design highly dist**R**ibuted appli**CA**tions **D**evelopment parad**I**gm over progr**A**mmable infrastructure
Project Website:	http://www.arcadia-framework.eu/
Start Date:	01/01/2015
Duration:	36 months

Title of Deliverable:	D4.1 Description of the Supported Metadata Annotations
Related WP:	WP4 – ARCADIA Development Toolkit
Due date according to contract:	31/03/2016

Editor(s):	Panagiotis Gouvas
Contributor(s):	Constantinos Vassilakis, Eleni Fotopoulou, Anastasios Zafeiropoulos (UBITECH) Alessandro Rossini (SINTEF) Nikos Koutsouris (WINGS) George Kioumourtzis (ADITESS)
Reviewer(s):	Nikos Koutsouris (WINGS) George Kioumourtzis (ADITESS)
Approved by:	All Partners

Abstract:	This deliverable provides a description of the metadata/annotations that are supported by the ARCADIA software development paradigm, based on the identified requirements in WP2.
Keyword(s):	*Microservice Development, Application Lifecycle Management, Annotations*

Partners

NUI Galway OÉ Gaillimh	Insight Centre for Data Analytics, National University of Ireland, Galway	Ireland
SINTEF	Stiftelsen SINTEF	Norway
TU berlin	Technische Universität Berlin	Germany
cnit	Consorzio Nazionale Interuniversitario per le Telecomunicazioni	Italy
Univerza v Ljubljani	Univerza v Ljubljani	Slovenia
UBITECH ubiquitous solutions	UBITECH	Greece
WINGS ICT Solutions	WINGS ICT Solutions Information & Communication Technologies EPE	Greece
GRUPPO Maggioli	MAGGIOLI SPA	Italy
aditess Advanced Integrated Technology Solutions & Services	ADITESS Advanced Integrated Technology Solutions and Services Ltd	Cyprus

Revision History

Version	Date	Editor(s)	Remark(s)
0.1	01/03/2016	Panagiotis Gouvas (UBITECH)	Defining ToC
0.2	20/04/2016	Panagiotis Gouvas (UBITECH) Anastasios Zafeiropoulos (UBITECH) Alessandro Rossini (SINTEF) Nikos Koutsouris (WINGS) George Kioumourtzis (ADITESS)	First version with inputs from partners
0.3	17/05/2016	Panagiotis Gouvas (UBITECH) Eleni Fotopoulou, Constantinos Vassilakis (UBITECH) Alessandro Rossini (SINTEF) Nikos Koutsouris (WINGS) George Kioumourtzis (ADITESS)	Additions to all chapters and consolidated version
0.4	30/05/2016	Nikos Koutsouris (WINGS) George Kioumourtzis (ADITESS)	Internal review
1.0	13/06/2016	Panagiotis Gouvas (UBITECH)	Final version taking into account internal review comments

Statement of originality:

This deliverable contains original unpublished work except where clearly indicated otherwise. Acknowledgement of previously published material and of the work of others has been made through appropriate citation, quotation or both.

Executive Summary

ARCADIA aims to provide a novel reconfigurable-by-design highly-distributed applications (a.k.a. HDAs) development paradigm over programmable cloud infrastructure. Such a novel development paradigm is needed to take advantage of the emerging programmability of the cloud infrastructure, and hence develop reconfigurable-by-design applications that support high performance, scalability, failure prevention and recovery, and in general self-adaptation to changes in the execution environment. The services that comprise an HDA should support a specific set of characteristics in order to take full advantage of the underlying programmability layer. More specifically, there are at least eight functional characteristics that should be met. According to these, any service that takes part in an HDA should:

a. expose its configuration parameters along with their metadata;
b. expose chainable interfaces which will be used by other native applications in order to create a service graph;
c. expose required interfaces which will be also used during the creation of a service graph;
d. expose quantitative metrics regarding the QoS of the native application;
e. encapsulate a lifecycle-management programmability layer which will be used during the placement of one service graph in the infrastructural resources;
f. be stateless in order to be horizontally scalable by design;
g. be reactive to runtime modification of offered resources in order to be vertically scalable by design;
h. be agnostic to physical storage, network and general purpose resources.

In the frame of ARCADIA the HDAs can be either Legacy applications, which are existing applications that are already available in an executable format and have to be orchestrated by the ARCADIA Orchestrator; or Native ARCADIA applications, which benefit from the full set of capabilities of the framework or Hybrid applications that consist of application tiers from both cases above. The purpose of the ARCADIA annotation framework -as described in the current manuscript- is to make use of the extensibility features of modern programming languages and come up with specific sets of annotations that will be used during runtime in order to automate at a big extend the business logic that relates to the aforementioned characteristics. Therefore, ARCADIA makes use of JAVA's extensibility mechanisms namely; "JSR-175: A Metadata Facility for the Java Programming Language"[1] and "JSR-269: Pluggable Annotation Processing API"[2] to provide a set of annotations that automatically invoke runtime handlers related to component

[1] https://jcp.org/en/jsr/detail?id=175

[2] https://jcp.org/en/jsr/detail?id=269

management, configuration management, management of performance metrics, management of lifecycle and management of dependencies.

Table of Contents

List of Figures

1 Introduction

1.1 Purpose and Scope

This document provides a description of the ARCADIA annotation framework that will be used for the development of Highly Distributed Applications. The annotations are formally an elegant feature of modern programming languages (e.g. Java and C#) that allow developers to 'decorate' their programs with specific tags (i.e. the annotations). This decoration provides to the developed program specific functionality which is either design-time (i.e. during compilation) or run-time (i.e. during execution). This functionality may span to different aspects such as automated code generation, automated injection of business logic, design-time validation etc.

As already mentioned, ARCADIA aims to provide a novel reconfigurable-by-design and highly-distributed applications development paradigm over programmable cloud infrastructure. Such a novel development paradigm is needed to take advantage of the emerging programmability of the cloud infrastructure, and hence develop reconfigurable-by-design applications that support high performance, scalability, failure prevention and recovery, and in general self-adaptation to changes in the execution environment. To this end, the annotation framework is the cornerstone framework that is utilised in order to implement the development paradigm.

The purpose of this deliverable is to elaborate, on the one hand, on the exact annotations that have been designed and on the other hand, on the business logic that is bound to these annotations. The business logic is tailored to the functional requirements of HDAs. Therefore, Chapter 2 is devoted on the analysis of the exact characteristics that HDA have and how these characteristics can be addressed using an annotation framework.

Chapter 3 provides the core analysis of the annotations. The annotations are functionally grouped in five categories according to the nature of the business logic that is bound to the usage of the annotation. These categories include component management, configuration management, management of performance metrics, management of lifecycle, and management of dependencies. For each of these categories the exact business logic is discussed. We discuss in Chapter 4, the usage of the annotation framework through the ARCADIA IDE, which is under development. Finally, Chapter 5 concludes the deliverable.

2 The Role of the ARCADIA Annotations

ARCADIA aims to provide a programming paradigm for Highly Distributed Applications (a.k.a. HDAs). Such applications are the ones that are built in order to run on multi-cloud infrastructure. The structure of an HDA enables high flexibility in reusing and composingbasic services such as databases, web front-end, authentication modules, network functions and the like. To this end, according to the architectural deliverable [2], the following application types can be defined: **a) Legacy applications** that are existing applications that are already available in an executable format and have to be orchestrated by the ARCADIA Orchestrator; **b) Native ARCADIA applications** that benefit from the full set of capabilities of the framework and **c) Hybrid applications** that consist of application tiers from both the cases above.

The purpose of this deliverable is to elaborate on the ARCADIA annotations that are used in order to create native ARCADIA applications. The other two types of HDAs may be functional equivalent to the native ARCADIA applications if someone undertakes the task of creating thin wrappers that emulate the functionalities that is automatically offered by the native apps. Therefore, the reasonable questions that are raised are the following: **a)** what are the functional characteristics of an ARCADIA native application? **b)** what was the reason behind the architectural choice of code-level annotations? The following sections will shed light regarding both of these questions.

2.1 Native ARCADIA Applications and their Lifecycle

As already discussed, a Highly Distributed Application (HDA) is defined as a reconfigurable-by-design distributed scalable structured system of software entities constructed to illustrate a network service when implemented to run over a programmable cloud infrastructure. A HDA is a multi-tier cloud application consisting of application's tiers chained with other software entities illustrating network functions applied to the network traffic towards/from and between application's tiers. The term 'reconfigurable-by-design' may be considered as an extension of the term 'context aware adaptable' [5]. A HDA not only is designed to be context-aware, able to adapt its processes to the context but also is able to reconfigure its structure accordingly, share its context and enable programmability through exposing a programming interface.

Thus, a HDA may expand or shrink by supporting horizontal scaling (out and in) for each software entity in the service chain while may reform (change its structure) by including or excluding software components from the chain and/or change routing among them as needed. A HDA may expose its context and state while it may provide a programmable interface for communication and externally being adapted and configured or re-used as a component of other HDAs. Each Application tier of a HDA is a distinct application-specific logical operation layer. Each other software entity involved in the Application's chain is a Virtual Function (VF) specific logical operation layer. Each Application tier and other involved software

entity provide/expose to each other a Binding Interface (hereinafter BI) letting each other have access to provided functionalities and supporting communication.

While developing an Application Tier, it is necessary to have knowledge of the BI of every other software component whose functionalities are required to be utilized within this Application Tier. Recursively, a component of a HDA chain may also be a chain itself; an already developed HDA or a VF chain exposing required functionalities. There is no need to communicate with every component of the nested chain but with a Service Binding Interface (SBI) providing for communication and access to exposed functionalities.

The developer of an application tier should annotate its code with required qualitative and quantitative characteristics according to the context model. Annotations can be included within the source code and be properly translated before building the executable, as well as externally accompany the executable and be properly translated by other components of the architecture during executable's placement and runtime.

An indicative HDA is depicted in Figure 1(a) that corresponds to a graph containing a set of tiers along with a set of functions implemented in the form of Virtual Functions (VFs). It should be noted that in ARCADIA we are adopting the term Virtual Functions (VFs) instead of the term Virtual Networking Function (VNF) that is denoted in ETSI Network Function Virtualization (NFV) [6] since we do not only refer to networking functions but to generic functions. Each element in the graph has to be accompanied with a set of quantitative characteristics (e.g. set of metrics that can be monitored) and constraints (e.g. resource capacity constraints, dependencies). In Figure 1(b) an application tier (T4) of the example is shown to horizontally scale while the chain reconfigures itself by expanding and reforming, adding as well a Load Balancer component as now required.

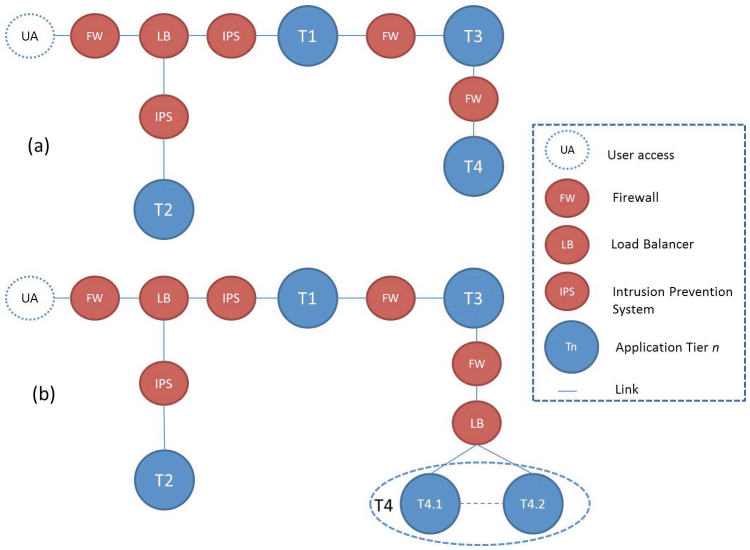

Figure 1 - (a) HDA Indicative Breakdown, (b) HDA horizontal scaling.

From an upper view, the lifecycle of an HDA starts from the development phase, followed by the deployment phase and operation phase and ends with its termination. Each phase requires several functional components of the architecture to work together to provide for and build an HDA, deploy it assigning resources from an infrastructure, run it while meeting objectives -developer wise and/or service provider wise- at all times and assure proper release of resources when terminates its operation. Deployment, Operation and Termination are supported by the Smart Controller as the intermediate between the applications and the infrastructure, while development is supported by several repositories providing easy access to reusable components and the defined context model providing access to the set of annotations and descriptions.

Taking under consideration all the above it is time **to define the 8 functional characteristics** of an ARCADIA Native application. Any ARCADIA Native application should:

i. expose its **configuration parameters** along with their metadata (e.g. what the acceptable values?, can the parameter change during the execution?);
j. expose **chainable interfaces** which will be used by other native applications in order to create a service graph;
k. expose **required interfaces** which will be also used during the creation of a service graph;
l. expose **quantitative metrics** regarding the QoS of the native application;
m. encapsulate a **lifecycle-management programmability** layer which will be used during the placement of one service graph in the infrastructural resources;
n. be **stateless** in order to be **horizontally scalable by design;**
o. be **reactive to runtime modification of offered resources** in order to be **vertically scalable by design;**
p. be **agnostic to physical storage, network and general purpose resources.**

The analysis of these principles super-exceed the scope of the current deliverable. The reader should consult D4.2 [4] where these characteristics are analysed. Towards this context, the role of the annotation framework is extremely concrete. The annotations will handle:

- the automatic extraction and setting of configuration parameters according to the runtime characteristics of a native application
- the automatic registration of the BI to central registry
- the run-time selection and chaining of applications based on the BI category
- the basic lifecycle management i.e. (Deploy, Start, Stop, Undeploy)
- the automatic interaction with the ARCADIA VFs that are responsible for scaling

2.2 Why an Annotation Framework?

Modern programming languages (such as Java and C#) offer an extremely useful mechanism named 'annotations' that can be exploited for several purposes. **Annotations are a form of metadata that provide data about a program that is not part of the program itself.** In other words, a specific set of design-time metadata can be used at the source-code level which will drive the normative schema creation. As already stated in D2.3 [2], from the software engineering perspective, annotations are practically a special interface. Such an interface may be accompanied by several constraints such as the parts of the code that can be annotated (it is called @Target in Java), the parts of the code that will process the annotation etc. An indicative annotations declaration using Java is presented in Table *2-1*.

Table 2-1: Indicative declaration of an annotation type

```
@Target({ElementType.METHOD })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface ArcadiaService{
   String controllerURI() default "htttp://controller.arcadia.eu";
}
```

According to this declaration, only Java methods can be annotated with the @ArcadiaService annotation (@Target({ElementType.METHOD })). Furthermore, the compilation procedure will **ignore** the annotation; yet the compiler **will keep the annotation at the binary** level since the it will be processed during runtime (@Retention(RetentionPolicy.RUNTIME)). Furthermore, if the annotation method is **overridden**, the annotation will be automatically propagated to the overridden method. In addition, annotation may be accompanied by **properties that can be set during declaration**. In the aforementioned case, the property 'controllerURI' is used to denote the URI of the hypothetical ARCADIA Smart Controller.

Annotations can be used in multiple ways. Each framework selects one handling technique in order to process annotations. In general, there are **three strategies** for **annotations' handling**. More specifically these strategies include:

a. **Source Generation Strategy**: This annotation processing option works by reading the source code and generating new source code or modifying existing source code, and non-source code (XML, documentation, etc.). The generators typically rely on container or other programming convention

and they work with any retention policy. Indicative frameworks that belong to this category are the Annotation Processing Tool[3] (APT), XDoclet[4] etc.

b. **Bytecode Transformation Strategy:** These annotation handlers parse the class files with Annotations and emit modified classes and newly generated classes. They can also generate non-class artifacts (like XML configuration files). Bytecode transformers can be run offline (**compile time**), at **load-time**, or **dynamically at run-time** (using JVMTI[5] API). They work with class or runtime retention policy. Indicative bytecode transformer examples include AspectJ[6], Spring, Hibernate, CGLib[7], etc.

c. **Runtime Reflection Strategy:** This option uses Reflection API to programmatically inspect the objects at runtime. It typically relies on the container or other programming convention and requires **runtime retention policy**. The most prominent testing frameworks like JUnit[8] and TestNG[9] use runtime reflection for processing the annotations.

From the three strategies that are presented above, the first one will not be utilized at all. However, the second and the third will be used. More specifically, the **'byte code transformation strategy'** will be used in order to **automate the procedure of normative schema generation** regarding the facets of **metadata**, **requirements** and **configuration**. To this end, specific type of annotations that will be processed during compilation will generate representative schema instances which are in line with the ARCADIA Context Model (see Deliverable D2.2 [1]). Finally, the '**runtime reflection strategy**' will be utilized in order to **dynamically implement specific behaviors** that relate to **programming interface binding**, **measurement of performance metrics** and **governance**.

ARCADIA makes use of JAVA's extensibility mechanisms namely; "JSR-175: A Metadata Facility for the Java Programming Language"[10] and "JSR-269:Pluggable Annotation Processing API"[11] to provide a set of annotations that will be described below in order to offer several functionalities.

[3] http://docs.oracle.com/javase/7/docs/technotes/guides/apt

[4] http://xdoclet.sourceforge.net/xdoclet/index.html

[5] http://docs.oracle.com/javase/7/docs/platform/jvmti/jvmti.html

[6] https://eclipse.org/aspectj

[7] https://github.com/cglib/cglib

[8] http://junit.org

[9] http://testng.org/doc/index.html

[10] https://jcp.org/en/jsr/detail?id=175

[11] https://jcp.org/en/jsr/detail?id=269

3 ARCADIA Annotations

Currently, there are 14 annotations that are defined, which serve also a functional purpose. An overview of these annotations is presented on Figure 2[12]. The annotations are functionally grouped in five categories according to the nature of the business logic that is bound to the usage of the annotation. These categories include component management, configuration management, management of performance metrics, management of lifecycle and management of dependencies.

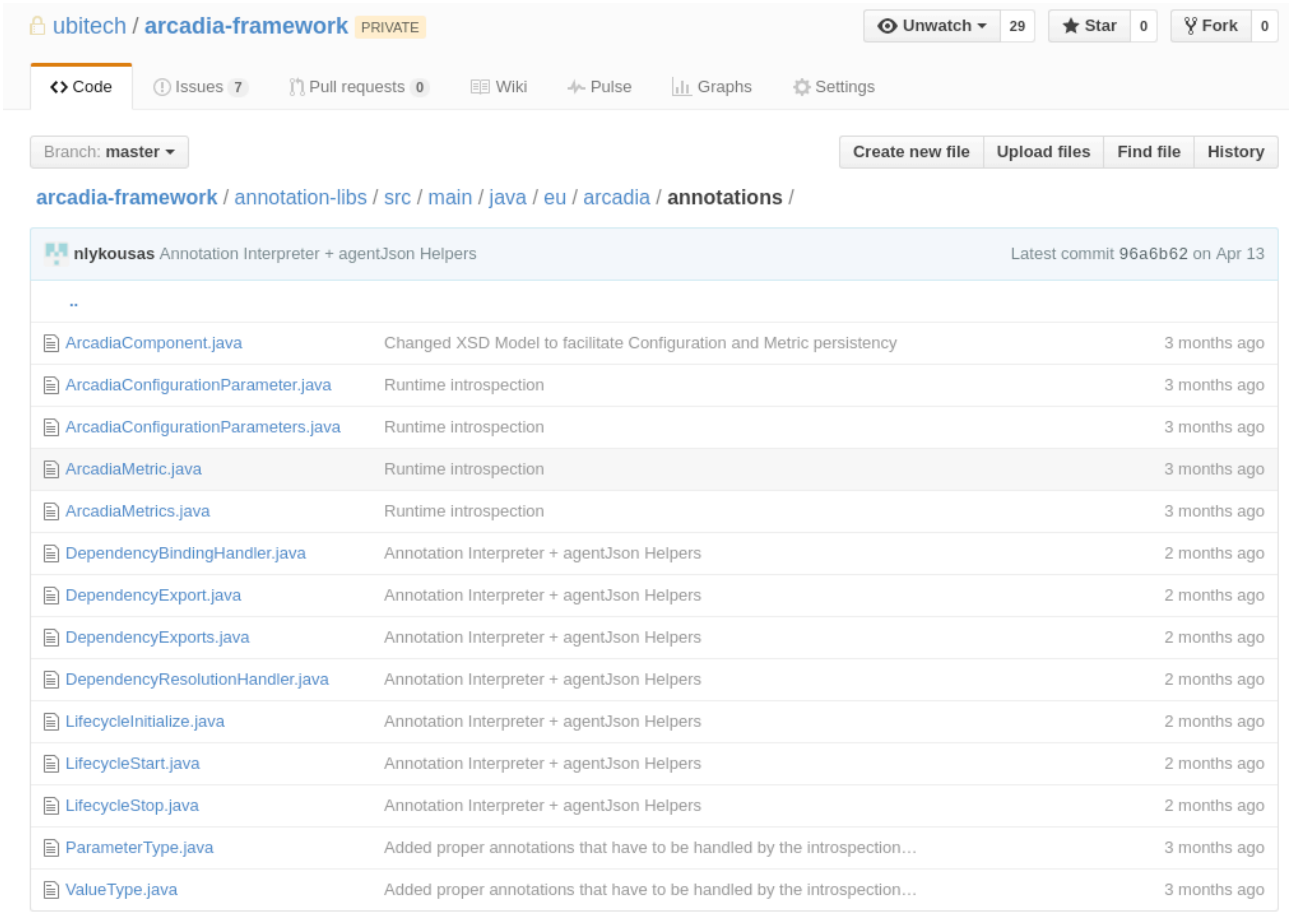

Figure 2 - Overview of ARCADIA Annotations

In the next sections we will elaborate each group separately.

3.1 Component Management

The first annotation that a developer can use is the ***@ArcadiaComponent*** annotation. The definition of the annotation is depicted on Figure 3.As it is depicted (from the *ElementType.Type*) the specific annotation is a class-level annotation i.e. it can decorate any Java class. Therefore, any class that is annotated as component will be handled. The handling that is bound to the annotation refers to the automatic generation of the formal component artefact and its registration to the ARCADIA Repository. More

[12] https://github.com/ubitech/arcadia-framework/tree/master/annotation-libs/src/main/java/eu/arcadia/annotations

specifically, as already analysed in D2.3[2] the creation of an HDA application involves the composition of several components that are already registered in a repository. The meta-model of the “chainable” component is depicted on Figure 4.

```java
package eu.arcadia.annotations;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Inherited;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * Validation expectations: Ide plugin should validate that ONLY one annotation exists in the entire environment
 *
 * @author Panagiotis Gouvas (pgouvas@ubitech.eu)
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface ArcadiaComponent {
    String componentname();
    String componentversion() default "0.1.0";
}
```

Figure 3 - ArcadiaComponent Annotation

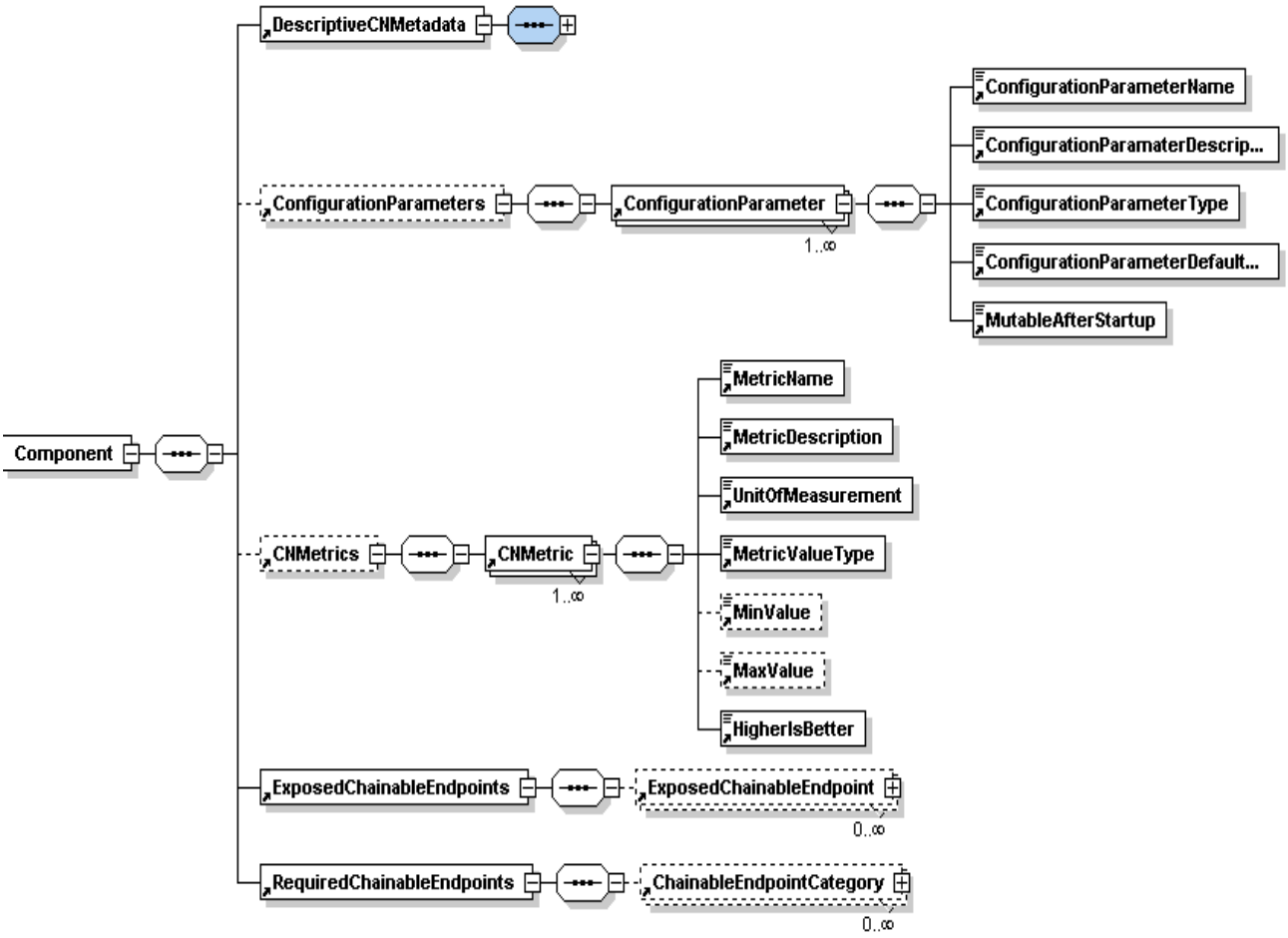

Figure 4 - Serialization Model for a specific component

As it is depicted in Figure 4 the formal metamodel is expressive enough to capture the "chainable" profile of a component. However as it will be shown below, the various elements that comprise the model (e.g. ExposedChainableEndpoints) are indicate through different annotations.

3.2 Configuration Management

Another annotation that a developer can use is the ***@ArcadiaConfigurationParameter***. The definition of the annotation is depicted on Figure 5. As it is shown, this annotation is also a class-level annotation. Through this annotation a developer can declare a configuration parameter which will automatically enrich the component model that is registered in the ARCADIA repository. The annotation supports 5 arguments i.e. '*name*', '*description*', '*parameterType*', '*defaultvalue*' and '*mutableafterstartup*'. '*Name*' is used to declare the method-name that exposes a configuration parameter. It should be clarified that using the principle of reflection the methods *setName* and *getName* are automatically exposed as we will explain later. The '*description*' provides a detailed explanation of the configuration parameter. The '*parameterType*' denotes if a parameter is single-value or multi-value. Finally, while the '*defaultvalue*' is self-explanatory the '*mutableafterstartup*' denotes whether or not a configuration parameter can be altered after the component initialization.

```java
package eu.arcadia.annotations;

import java.lang.annotation.*;


// internal task reflection to xsd so as introspection engine to produce valid xml
// expectations: a) a rest enpoint should exist that get the values of all parameters
// open issue: a) how to implement the rest controller that allows the setting of a paramater
//             b) how developer is goind to bind the setParamter_callback on his/her code (setter getter)


/**
 *
 * @author vmadmin
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@Repeatable(ArcadiaConfigurationParameters.class)
public @interface ArcadiaConfigurationParameter {
    String name();
    String description() default "";
    ParameterType parametertype();
    String defaultvalue();
    boolean mutableafterstartup() default false;
}
```

Figure 5– Configuration ParameterAnnotation

The enrichment of the component model is not the only business logic that is bound to the specific annotation. The most crucial aspect is the automatic generation of a thin REST layer that is responsible to expose the configuration parameters offering a 'proxy' for both setters and getters. This REST layer is consumed only by the ARCADIA Smart Controller. As discussed in D3.1 [3] any service graph is related to two types of policies. The first policy is the deployment policy and the second is the runtime policy. Part of the runtime policy is the reconfiguration of the component based on the monitoring streams that are generated. A service provider may trigger the change of a mutable configuration parameter during runtime in the frame of a policy execution. This change will be performed based on the interaction of the Smart Controller with the aforementioned REST layer.

3.3 Component's Metrics

The third family of annotations that are provided to the developers are the ***@ArcadiaMetric*** and the ***@ArcadiaMetrics***. Both of these annotations are depicted on Figure 6 and Figure 7, respectively.

```java
/**
 *
 * @author vmadmin
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@Repeatable(ArcadiaMetrics.class)
public @interface ArcadiaMetric {
    String name();
    String description() default "";
    String unitofmeasurement();
    ValueType valuetype();
    String minvalue();
    String maxvalue();
    boolean higherisbetter() default false;

}
```

Figure 6– ArcadiaMetric Annotation

```java
package eu.arcadia.annotations;


import java.lang.annotation.*;

/**
 * Created by nikos on 2/3/2016.
 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Inherited
@Documented
public @interface ArcadiaMetrics {
    ArcadiaMetric[] value();
}
```

Figure 7– ArcadiaMetrics Annotation

Since the *@ArcadiaMetrics* annotation is just an easy way to declare multiple *@ArcadiaMetric* we will emphasize our analysis in the latter. *@ArcadiaMetric* is a class-level annotation and is used to declare methods that return specific user-defined measurements. An indicative usage of the annotation would be the following:

@ArcadiaMetric(***name***="averageProcessingTime",***description*** = "URL hashing algorithm performance",***unitofmeasurement*** = "msec",***valuetype*** = ValueType.SingleValue,maxvalue = "6000",***minvalue*** = "1",***higherisbetter*** = false)

As it is depicted the *@ArcadiaMetric* annotation requires six arguments. The 'name' denotes the method that implements the actual stream extraction. Therefore, the indicative "averageProcessingTime" infers that there is a getAverageProcessingTime() method that can be invoked. The 'unitofmeasurement' provides the appropriate unit that can be used during quantification while the min and max denote the respective bounds. The Boolean flag 'higherisbetter' indicate whether the increase of a measurement is considered positive or not.

The business logic that is bound to the specific annotation is the automatic generation of a REST service that exposes the monitoring stream based on the requests of a Smart Controller. This is completely analogous to the business logic that has been developed for the @ArcadiaConfiguration; yet there is a significant difference. In the current case only a getter is meaningful since the runtime policies that are enforced require the handling of multiple parallel streams that refer to either VM-metrics of user-defined metrics. The streams that refer to the VM-metrics are embedded in the VM while the streams that refer to the user-defined metrics are automatically generated through the runtime interpretation of the annotation.

3.4 Lifecycle Management

A crucial aspect regarding the ARCADIA components is the application lifecycle management (hereinafter ALM). The ALM refers to the entire service graph as along as to components that participate in a service graph. Since a service graph is a directed acyclic graph the components entail several dependencies among them. Therefore, the deployment of one component follows a strict bootstrapping protocol according to which the component has to be firstly initialized (i.e. its execution environment is available), then its dependencies have to be resolved and finally the main execution context should start.

```
1  package eu.arcadia.annotations;
2
3  import java.lang.annotation.*;
4
5  /**
6   * Created by nikos on 29/3/2016.
7   */
8  @Target(ElementType.METHOD)
9  @Retention(RetentionPolicy.RUNTIME)
10 @Inherited
11 @Documented
12 public @interface LifecycleInitialize {
13 }
```

Figure 8– LifecycleInitialize Annotation

The implementation of the bootstrapping protocol is under the responsibility of the Smart Controller. However, the Smart Controller does not have the knowledge of the secrete internal states of the component. Therefore, specific annotations are required to denote the context initialization and the start/stop. These annotations are the *@LifecycleInitialize* (Figure 8), @LifecycleStart (Figure 9) and @LifecycleStop (Figure 10), respectively.

```java
package eu.arcadia.annotations;

import java.lang.annotation.*;

/**
 * Created by nikos on 29/3/2016.
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface LifecycleStart {
}
```

Figure 9– LifecycleStart Annotation

```java
package eu.arcadia.annotations;

import java.lang.annotation.*;

/**
 * Created by nikos on 29/3/2016.
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface LifecycleStop {
}
```

Figure 10– LifecycleStop Annotation

As it can be noticed, these annotations, contrary to the ones that we have examined up to now, are not class-level but method-level. This is denoted by the ElementType.METHOD tag. This is deliberate since these annotations decorate existing methods in the code that will be invoked when the Smart Controller dictates. In order for the invocation to be performed a pass-through Rest call will be performed from the Controller to the component. This is completely analogous to what is happening in the case of the configuration management and the case of export of the monitoring streams.

3.5 Dependency Management

The final set of annotations that can be used relate to the exposure and the usage of component's interfaces. As already analysed thoroughly in D2.2 [1] each component may expose one or more interfaces after its initialization. On the other hand, the same component may require one or more interfaces from third-party components before the actual initialization. Handling both the advertisement of an interface and the chaining requirement is performed through two distinct annotations i.e. the ***@DependencyExport*** and the ***@DependencyResolutionHandler***.

```java
package eu.arcadia.annotations;

import java.lang.annotation.*;

/**
 * Created by nikos on 31/3/2016.
 */
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@Repeatable(DependencyExports.class)
public @interface DependencyExport {
    String CEPCID();
    boolean allowsMultipleTenants() default true;
}
```

Figure 11– DependencyExport Annotation

@DependencyExport(class-level annotation) publishes to a high-available microservice repository the published interface while the *@ResolutionHandler* decorates a method which will handle the output of a lookup service that aims to identify a required service.

```java
package eu.arcadia.annotations;

import java.lang.annotation.*;

/**
 * Created by nikos on 6/4/2016.
 */
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface DependencyResolutionHandler {
    String CEPCID();
}
```

Figure 12– DependencyResolutionHandler Annotation

It should be noted that in Annex I, a completely annotated component is appended.

4 Usage of Annotations

ARCADIA is designed to help developers create and deploy applications quicker and more efficiently. It automates many tasks such as code and annotation validation, application deployment and application monitoring. The development of components will be performed using the ARCADIA web-based IDE environment. Through this environment, developers can use the aforementioned annotations that are validated by the smart controller and are automatically interpreted to components. This stands true only in the case of native applications.

4.1 ARCADIA web-based IDE environment

As it is explained in D4.2[4] the IDE environment relies on a state-of-the-art web-based IDE called Eclipse Che[13] which gives the possibility to the developers to develop components in a collaborative way. Eclipse Che is a general-purpose web-based IDE. Che is based on Docker[14], GWT[15], Orion[16] and RESTful APIs and offers both client and server side capabilities. The client-side of Che is based on the Orion editor, and it offers most of the features expected from a classic IDE such as automatic code completion, error marking, JAVA "intelligence" and documentation. On the other hand, the server side is responsible for managing the code repository, compiling and executing programs and managing runtimes. Through a specific ARCADIA plugin that is being developed the interaction of the IDE with the Smart Controller will be automated.

The primary task of the ARCADIA IDE plugin is to assist developers during component development. Through the plug-in, developers can view all available ARCADIA annotations, including some Javadoc-based documentation, authenticate with the ARCADIA platform and trigger component validation, through the ARCADIA server-side plug-in.

Specifically, when a developer starts writing an ARCADIA annotation, the plug-in offers auto-complete suggestions where developers can just choose which annotation they want to use. In addition, they can read the specifications of each annotation, like the required fields, naming conventions and examples. Moreover, if they choose to use one of the provided component templates, the plug-in will manage any ARCADIA requirement like maven dependencies. Using the web-based IDE, developers can develop components that adhere to the ARCADIA component metamodel. This compliancy is achieved through the usage of specific annotations that allow developers to declare components, register configuration parameters, define chainable endpoints etc.

The usage of annotations is performed in a seamless way through the development environment. Figure 13 depicts the development of an actual component. As it is depicted the developer is using several annotations such as *@ArcadiaComponent*, *@ArcadiaMetric*, *@ArcadiaExport* etc. The definition of these annotations during development is assisted by the IDE per se while the functional usage of these annotations take place after the submission of the executable to the Smart Controller.

[13] https://eclipse.org/che/
[14] https://www.docker.com/
[15] http://www.gwtproject.org/
[16] https://orionhub.org/

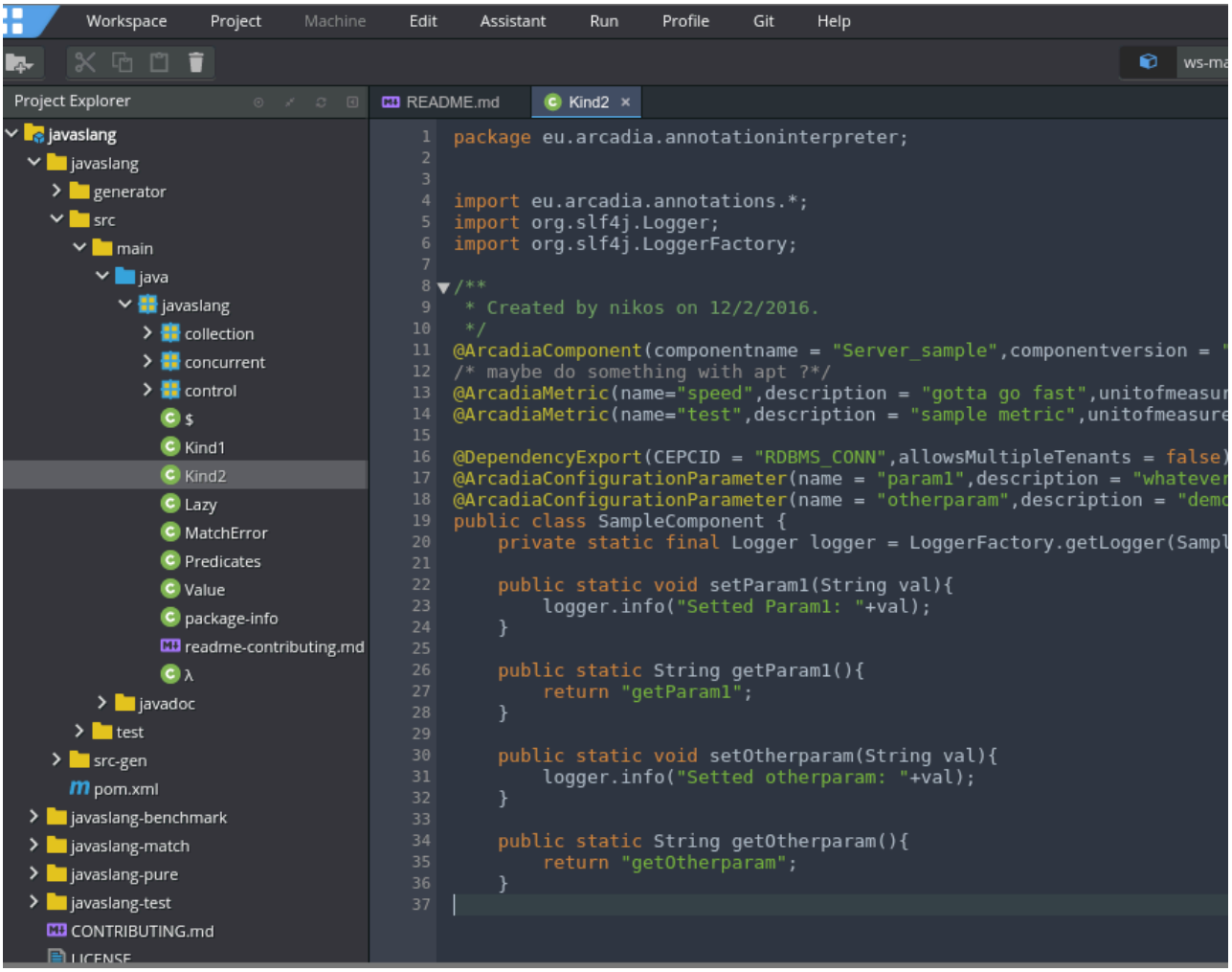

Figure 13 – Development of ARCADIA component through web-based IDE

4.2 Validation of Annotations &Generation of Thin layer

After the submission of the component to the Smart Controller, the system performs a set of validations in order to check the logical validity of the provided annotations. It should be noted that the structural validity of the annotations is **not checked** since if the project compiles correctly structural validity is assured. However, the logical validity refers to a set of aspects such as the uniqueness of the component name, the existence of the real method-hooks that correspond to the various getters (e.g. getMetricX), the avoidance of conflicting versions, the existence of chainable endpoints etc. All these are a small part of the logical validation which is performed using Bytecode introspection techniques.

The maven module that performs the actual introspection is called “annotationinterpreter”. Figure 14 depicts the exact location in the projects’ source code repository where this module exists. The source code repository is located at the following url: https://github.com/ubitech/arcadia-framework

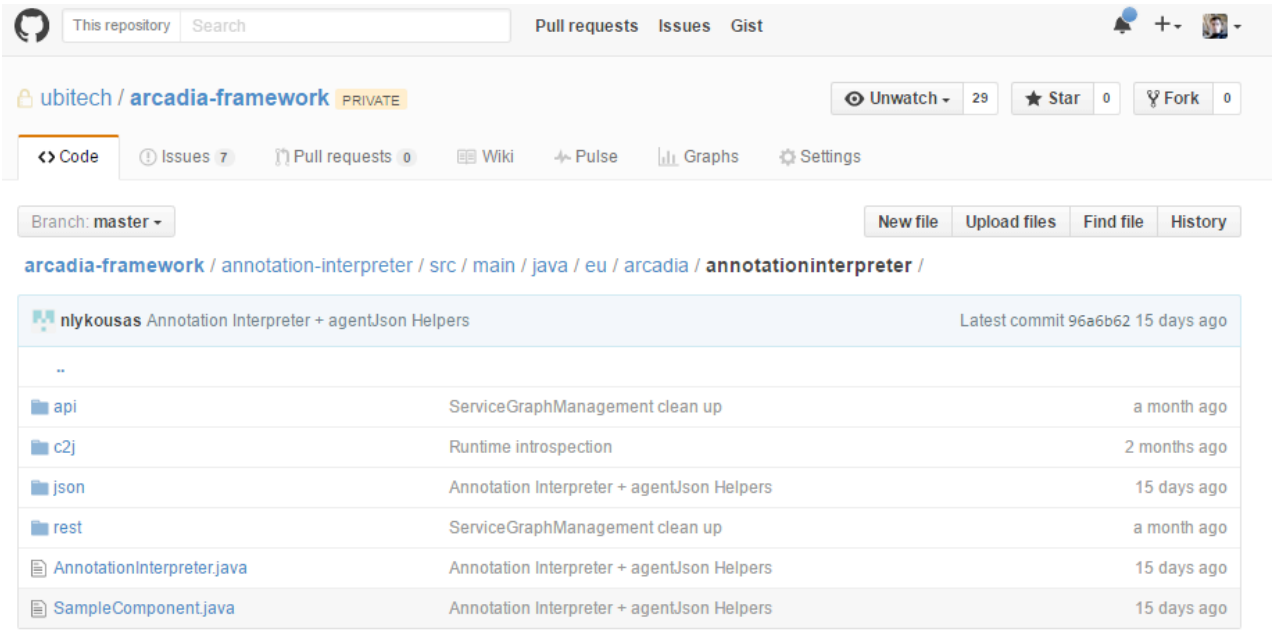

Figure 14– Maven module that performs the Annotation introspection

When the component passes the validation phase, the executable along all metadata that accompany the executable are stored in a structured format in the Smart Controller's persistency engine. A specific parser that is embedded in the annotationinterpreter undertakes the task to transform the arguments of the annotations (e.g. ArcadiaComponent(name="xxx")) to a serialized format that adheres to the ARCADIA Context Model[2].

The next step after the validation is the creation of the thin REST layer which is attached to each component. The thin REST layer is developed in the frame of the 'agent' maven module[17]. This agent is initialized by a configuration file that is generated automatically after the valid introspection. The snippet below provides such configuration for the real service that is provided in Annex I.

```
{
  "className": "eu.arcadia.annotationinterpreter.sample.URLShortener",
  "CNID": "d1e39865-02be-4b57-9896-6cde075d8f79",
  "lifecycleInitMethod": "init",
  "lifecycleStartMethod": "start",
  "lifecycleStopMethod": "stop",
  "dependencyResolutionHandlers": {},
  "dependencyBindingHandlers": {
    "MONGODB_CONNECTION": "bindMongoDB"
  },
  "metrics": [
    "hashedURLs",
    "averageProcessingTime"
```

[17] https://github.com/ubitech/arcadia-framework/tree/master/agent

```
  ],
  "configurationParameters": [
    "hashLength"
  ]
}
```

When the component is activated the agent bootstraps. Upon bootstrapping a REST interface is exposed to the SmartController. Such a REST interface for the service is provided below.

```
/conf/set/hashLength [POST]
{
  "value": "7"
}
/conf/get/hashLength [GET]
{
  "value": "7"
}
Metrics:
/metric/get/hashedURLs [GET]
{
  "value": "1"
}
/metric/get/averageProcessingTime [GET]
{
  "value": "1"
}
```

5 Conclusions

The purpose of this deliverable was to elaborate on the code-level annotations that have been designed and on the business logic that is bound to these annotations. Annotations per se are features of modern programming languages that allow developers to 'decorate' their programs. This decoration provides to the developed program specific functionality which is either design-time (i.e. during compilation) or run-time (i.e. during execution). To this end, ARCADIA made use of JAVA's extensibility mechanisms namely; "JSR-175: A Metadata Facility for the Java Programming Language" and "JSR-269:Pluggable Annotation Processing API" to provide a set of annotations that accelerates the development of HDA applications.

ARCADIA annotations are functionally grouped in five categories according to the nature of the business logic that is bound to the usage of the annotation. These categories include component management, configuration management, management of performance metrics, management of lifecycle, and management of dependencies. For each of these categories the exact business logic is discussed.

Regarding the component management, a proper annotation undertakes the task to create the appropriate artefact and store it in the ARCADIA repository in order to be used later on during the creation of a service graph. Regarding, the configuration management aspects, the annotations are used in order to create a fully functional REST thin-layer on-top of the component that undertake the task of getting or setting the configuration parameters. It should be clarified that getting is only meaningful in case of mutable parameters. As far as performance metrics are concerned, a proper annotation is responsible to expose developer-defined measurements per each metric. These measurements are propagated to the Smart Controller that is responsible for the policy enforcement.

Finally, regarding lifecycle management, ARCADIA annotations implement pass-through functions that are required by the Smart Controller in the frame of a service graph deployment. A service graph deployment entails a specific boot sequence protocol that requires uniform interaction with all parts of the graph. An analogous REST thin-layer that is auto-generated provides the required uniform interaction. The same stands true regarding the exposure and the binding of the interfaces that a component may perform. ARCADIA annotations can be used in any IDE environment. However, in the frame of the project a specific web-based IDE will be provided that will perform several types of validation on the ARCADIA components.

References

[1] Arcadia project, D2.2 - ARCADIA Context Model, Available Online: http://www.arcadia-framework.eu/wp/documentation/deliverables/

[2] Arcadia project, D2.3 - ARCADIA Architecture, Available Online: http://www.arcadia-framework.eu/wp/documentation/deliverables/

[3] Arcadia project, D3.1 - Smart Controller Reference Implementation

[4] Arcadia project, D4.2 - Description of the Applications' Lifecycle Management Support, Available Online: http://www.arcadia-framework.eu/wp/documentation/deliverables/

[5] M. Dalmau, P. Roose, S. Laplace, "Context Aware Adaptable Applications - A global approach," IJCSI International Journal of Computer Science Issues, Vol. 1, 2009

[6] ETSI, Network Function Virtualization, Online: http://www.etsi.org/technologies-clusters/technologies/nfv

Annex I – Sample Annotated Component

```
package eu.arcadia.sample;


import eu.arcadia.annotations.*;

import org.springframework.boot.SpringApplication;


@ArcadiaComponent(componentname = "URL Shortener",componentversion = "1.1.0",componentdescription =
"Make a long URL short, easy to remember and to share.",tags={"url-shortener","server"})

@ArcadiaMetric(name="hashedURLs",description = "Total number of shrinked URLs",unitofmeasurement =
"integer",valuetype = ValueType.SingleValue,maxvalue = "2147483648",minvalue = "0",higherisbetter =
false)

@ArcadiaMetric(name="averageProcessingTime",description = "URL hashing algorithm
performance",unitofmeasurement = "msec",valuetype = ValueType.SingleValue,maxvalue = "6000",minvalue
= "1",higherisbetter = false)

@DependencyExport(CEPCID = "URLSHORTENER_REST_API",allowsMultipleTenants = true)

@ArcadiaConfigurationParameter(name = "hashLength",description = "Defines the length of the hash
representing the shrinked URL",parametertype = ParameterType.SingleValue,defaultvalue = "7",
mutableafterstartup = false)

public class URLShortener {


    /*

    ArcadiaConfigurationParameter setter/getter

    */

    public static void setHashLength(String length){

        //application logic

        System.setProperty("URLShortener.hashLength",length);

    }


    public static String getHashLength(){

        //application logic

        return  System.getProperty("URLShortener.hashLength");

    }


    /*

    ArcadiaMetrics (getters only)

    */

    public static String getHashedURLs(){

        //application logic

        return "1";

    }

    public static String getAverageProcessingTime(){

        //application logic

        return "1";

    }


    /*
```

```
    Component Lifecycle Management
    */
    @LifecycleInitialize
    public static void init(){
        System.setProperty("server.port",Application.defaultPort.toString());
        System.setProperty("URLShortener.hashLength",Application.defaultHashLength.toString());
    }

    @LifecycleStart
    public static void start(){
        new SpringApplication(Application.class).run();
    }

    @LifecycleStop
    public static void stop(){
    }

    /*
    DependencyExport-related methods (for URLSHORTENER_REST_API)
    */
    public static String getUri(){
        return Application.defaultUri;
    }

    public static String getPort(){
        return Application.defaultPort.toString();
    }

    /*
    Handle the bidning of the required MongoDB component (set application-specific properties etc.)
    */
    @DependencyBindingHandler(CEPCID = "MONGODB_CONNECTION")
    public void  bindMongoDB(String ecepid, String json){

    }
}
```