Service Architecture Model
Draft 0.2

Paweł Cesar Sanjuan Szklarz
paweld2@gmail.com

June 6, 2012

Abstract

Service Architecture Model (SAM) is a mathematical model for definition of service oriented architectures. SAM Architecture products are defined using code and precisely defined relations. Validation of architecture is based on a compilation process. It is also possible to automatically verify implementations compliance with service specifications. A development and verification process for service implementation is also defined. A key idea behind SAM is the use of a canonical execution protocol, intuitively related to "methods execution recording".

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivs 3.0 Unported License.

Copyright ©2011 Paweł Cesar Sanjuan Szklarz

Contents

List of Figures

1 Introduction

The documentation is ordered as follows. This sections contains a intuitive overview of Service Architecture Model. Section 2 present a precise definition of presented concepts. Abstract Domain Dependency Model defined on section 3 is necessary to define service injection. Section 4 present a execution model for services of a SAM Architecture.

1.1 Architecture Overview

A SAM **Architecture** defines a **Service Taxonomy**. Service Taxonomy is a set of separate **Categories** grouping **Service Specifications**.

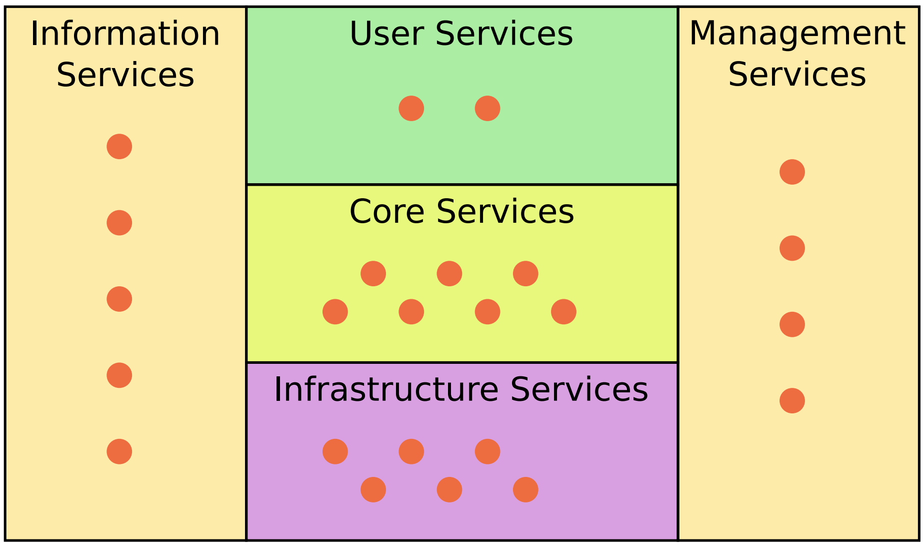

Figure 1: Service Taxonomy example

Figure 1 shows a service taxonomy consisting of five categories. Each category has different Service Specifications indicated by dots.

A **Service Specification** is a set of **Interfaces**. A Interface intuitively should be treated as a normal Java interface. No class members are allowed on Interfaces. Service Specifications are disjoint.

For one Service Specification it is possible to have many different Service Implementations. A **Service Implementation** or Implementation is a **Binding Provider** (see section 3) and all necessary classes to instantiate the Service Specification Interfaces. For simplicity, one can assume that it is a set of classes implementing the Service Specification Interfaces. See 1.3 for more details about Service Implementations.

Implementations are executed on a **Service Execution Environment**. One Implementation executed is called a **Service Instance** or Instance for short, and it has a unique **Instance Identificator** (IID). It is possible to execute many Service Instance of the same Service Specification simultaneously. Each Service Instance can be a instantiation of a different Implementation. See section 1.6 for a overview of Service Execution Environment and section 4 for reference information.

To use external services, the Implementation developer must introduce to the code references to interfaces belonging to some external services. **Dependency Injection Model** is used to inject references to Service Instances of appropriate type. The choose of external Service Instance is made completely on the Service Execution Environment as a configuration aspect. The exact definition of Dependency Injection Model is given in section 3.

SAM imposes constraints on the external services used in Implementations defined be a **Category Accessibility Relation**: category C_2 is accessible to category C_1 if we allow injection of interface of services from category C_2 in the implementation of services of C_1. As an example, assume that category accessibility relation for architecture given in figure 1 is as follow:

$$\begin{array}{rcl}
\text{User Services} & \hookleftarrow & \text{Core Services} \\
\text{Core Services} & \hookleftarrow & \text{Core Services} \\
\text{Core Services} & \hookleftarrow & \text{Infrastructure Services} \\
\text{Core Services} & \hookleftarrow & \text{Management Services} \\
\text{Core Services} & \hookleftarrow & \text{Information Services}
\end{array}$$

This mean that a Service Implementations of Service Specification belonging to Category “User Services” can inject only references to Service Instances belonging Category “Core Service”.

1.2 Service Specification

We present on more detailed overview of Service Specification definition and some concrete examples.

Lets see the definition of “UserData” Service Specification belonging to Category “User Service”. It is defined be two interface’s as shown on listing 1,2.

Listing 1: Service Specification Interface API - UserData

```
package eu.pmsoft.sam.service.user;

public interface UserDataSimpleAPI {

 boolean simpleMethod();

 UserDataComplexAPI complexInteractionAPI(Integer basicType);

}
```

Method `simpleMethod` is just a simple example without arguments and return type `boolean`. Method `complexInteractionAPI` is more interesting, it has a simple type argument and returns a reference to interface `UserDataComplexAPI` defined below. On SAM Interface methods signature it is allowed to use only primitive types and interface types declared on the architecture.

A important relation between Interfaces is the **Signature Relation** resulting from the types used on methods signatures. As a examples we say that Interface `UserDataSimpleAPI` uses Type related to `UserDataComplexAPI` because it appears as a return type of method `complexInteractionAPI`. Please note that Signature Relation is defined between Interfaces and Types resulting from Interfaces.

Listing 2: Service Specification Interface API - UserData

```
package eu.pmsoft.sam.service.user;

import eu.pmsoft.sam.ds.DomainData;
import eu.pmsoft.sam.ds.DomainTypeExample;

public interface UserDataComplexAPI {

 void clientInformation(int data);

 boolean finalizeComplexClientInteraction();

 DomainTypeExample getDomainData(DomainData data);
}
```

Interface `UserDataComplexAPI` is very similar to previous one. It defines three methods with some parameters and result types. Note that there are two new interfaces `DomainData` and `DomainTypeExample` not belonging to the Service Specification. A set of interfaces used on a Service Specification is called a **Domain Specification**. It is required for Domain Specifications to be mutually disjoint.

Signature Relation implies a **Domain Dependency Relation** between Domain Specifications: If a type derived from Domain Specification d_2 is used on some method signature of a Interface from Domain Specification d_1, then we say that d_1 depends on d_2.

We treat Service Specification as a Domain Specification, but do not allow to use Types from Service Specifications on other Domain Specifications. We also require that the Domain Dependency Relation be acyclic. A possible graph related to a Domain Dependency Relation is shown on figure 2.

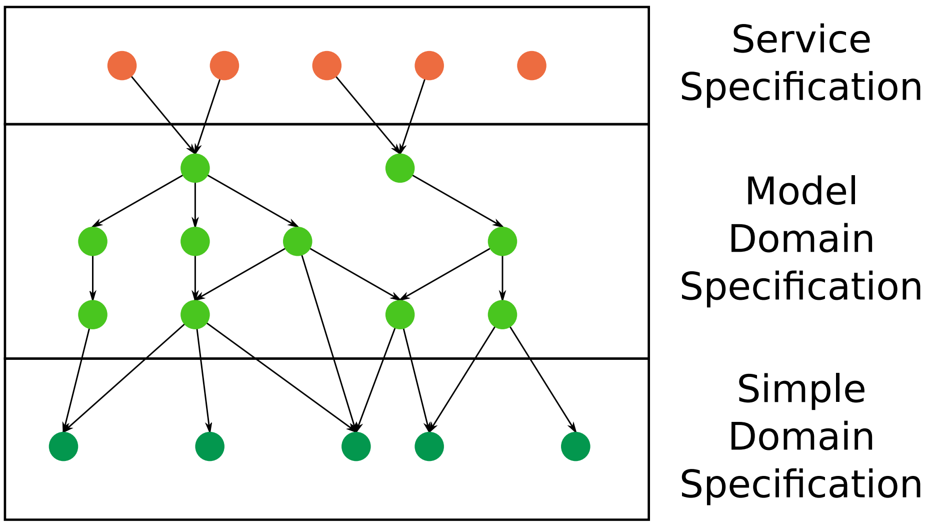

Figure 2: Domain Dependency Relation example

A Domain Specification without dependencies is called a Simple Domain Specification. If it has dependencies, then is called Model Domain Specification. As mentioned earlier, a Service Specification can't be used as a dependency.

Lets define a Service Specification “CoreServiceExample” containing interface 3 to be used in the following examples.

Listing 3: CoreServiceExample Service Specification

```
package eu.pmsoft.sam.service.core;


public interface CoreServiceExample {

 void resetProcess();

 void putData(int value);

 boolean isProcessStatusOk();

}
```

1.3 Service Implementation and Service Instance Injection

A simple implementation ONE of Service Specification “CoreServiceExample” is class `ImplementationCoreServiceExampleOne` implementing interface `CoreServiceExample` , see listing 4.

Listing 4: Implementation ONE of “CoreServiceExample”

```
package eu.pmsoft.sam.impl.core;

import javax.inject.Inject;

import eu.pmsoft.sam.binding.BindingAnnotationExample;
import eu.pmsoft.sam.service.core.CoreServiceExample;

public class ImplementationCoreServiceExampleOne implements
    CoreServiceExample {

 @Inject
 private CoreServiceExample serviceInjectionPoint;

 @Inject @BindingAnnotationExample
 private CoreServiceExample
     serviceInjectionPointWithBindingAnnotation;

 private Integer counter;
 private Integer limit = 100;

 public void resetProcess() {
  counter = 0;
 }

 public void putData(int value) {
  counter += value;
 }

 public boolean isProcessStatusOk() {
  return counter < limit;
 }
}
```

Note fields of Type `CoreServiceExample`. These are examples of **Injection Points**, places where it is possible to receive injections, it is references to external Service Instance. Field `public CoreServiceExample serviceInjectionPointWithBindingAnnotation ;` has also a Binding Annotation. **Binding Annotations** instruct the Service Execution Environment on how to interconnect Service Instances. Each Injection Point has associated a **Key** defined as a pair Type-Binding Annotation. Binding Annotation may be empty.

A Implementation TWO of "CoreServiceExample" is created with classes `ImplCoreTwo,ImplCoreTwoBinding` and a non trivial Binding Provider following scheme:

$$\begin{aligned} \text{CoreServiceExample} \times 0 &\longmapsto \text{ImplCoreTwo} \\ \text{CoreServiceExample} \times \text{@BindingAnnotationExample} &\longmapsto \text{ImplCoreTwoBinding} \end{aligned}$$

Implementation TWO provides a different class implementing `CoreServiceExample` according to used Key.

Assume that on the Service Execution Environment are created two Instances of Service Specification "CoreServiceExample" using previously presented Implementations ONE and TWO, with IID "01" and "02" respectively.

Service Execution Environment could be configured to inject Instance "02" to Instance "01" as shown in figure 3.

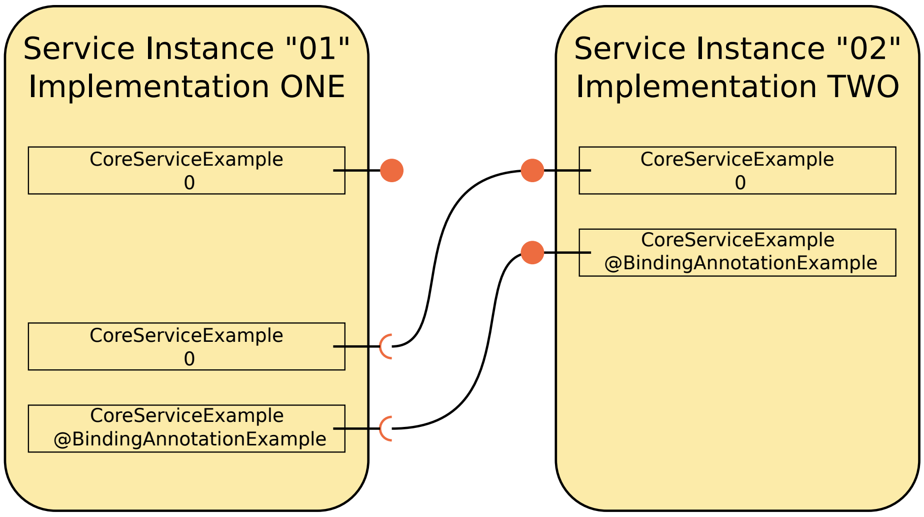

Figure 3: Service Instance injection

Other possible scenario is to inject Instance "01" on Instance "01" as shown on figure 4. Implementation ONE don't define any binding for key CoreServiceExample × @BindingAnnotationExample, so the simplest key CoreServiceExample × 0 is used because it also match the annotated Injection Point.

For details on Dependency Injection Model and Injection Process see section 3.

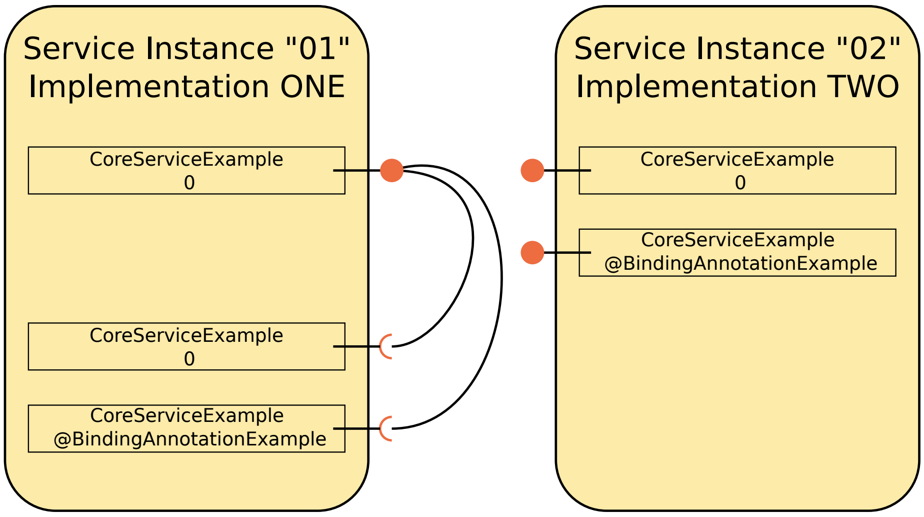

Figure 4: Self-Reference injection

1.4 Service interaction and Canonical Execution Protocol

Existing service interaction methods and technologies are based on a function execution abstraction, a service is understood as a external set of functions available through remote calls. This approach leads to the following logical scheme during architecture and software development:

- Prepare service request data
- Execution: Send service request, wait for response
- Interpret response

Such a pattern is associated with high costs related to calls on external service. This leads to creation of increasingly complex data structures to define the parameters and results of service methods, or "external functions".

SAM introduce a completely new approach to service calls and service interaction. To illustrate this, lets use a Implementation of "UserData" Service Specification that contain class `ImplementationOneSimple` shown on listing 5.

Listing 5: Class of UserData Implementation

```
package eu.pmsoft.sam.impl.user.one;

import javax.inject.Inject;

import eu.pmsoft.sam.service.core.CoreServiceExample;
import eu.pmsoft.sam.service.user.UserDataComplexAPI;
import eu.pmsoft.sam.service.user.UserDataSimpleAPI;

public class ImplementationOneSimple implements UserDataSimpleAPI {

 private Integer internalCounter = 0;

 @Inject
 private CoreServiceExample coreService;
```

```
    public boolean simpleMethod() {
     coreService.resetProcess();
     for (int i = 0; i < internalCounter; i++) {
      coreService.putData(i);
     }
     return coreService.isProcessStatusOk();
    }

    public UserDataComplexAPI complexInteractionAPI(Integer basicType)
        {
     internalCounter += basicType;
     return new ImplementationOneComplex(internalCounter);
    }
   }
```

Method `public boolean simpleMethod()` is implemented using external service "CoreServiceExample". Note that executing this method may require a large number of calls to interface `CoreServiceExample`, but how many external requests are needed? only one!. See listing 3 for the definition of interface `CoreServiceExample` and note that only the `isProcessStatusOk` method has a return type different that `void`. SAM define a **Canonical Execution Protocol** to serialize calls to Service Specification Interfaces. Execution of method `public boolean simpleMethod()` serialized with the canonical protocol produce a service request containing:

```
i_0.resetProcess()|i_0.putData(i_1)|...
...i_0.putData(i_n)|i_0.isProcessStatusOk()r_0|
PAYLOAD
i_0=KEY<CoreServiceExample X 0>
i_1="IntegerSerialization"
...
i_n="IntegerSerialization"
r_0=Type<boolean>
```

This is the exact information necessary for a external Service Instance implementing "CoreServiceExample" to repeat calls to interface `CoreServiceExample` and return the final result.

Generally, an external call is required when the return type of a Interface method is a class type. In the case that a return type is a interface type, it is possible to continue "recording" executions to that interface without any external request. See method `changeUserName` on listing 6 and resulting canonical service request.

Listing 6: Calls to external service with nested interface executions

```
   package eu.pmsoft.sam.examples;

   import javax.inject.Inject;

   import eu.pmsoft.sam.binding.BindingAnnotationExample;

   public class ServiceCode  {

    @Inject @BindingAnnotationExample
    private ExternalServiceAPI externalService;
```

```
  public boolean changeUserName(String username,Integer userId){
   ExternalUserModel userModel = externalService.userModel(userId);
   boolean changeOk = userModel.setUserName(username);
   return changeOk;
  }
 }
```

```
i_0.sendData(i_1)r_0|r_0.sendMoreData(i_2)r_1|
PAYLOAD
i_0=KEY<ExternalServiceInterface X @BindingAnnotationExample>
r_0=InterfaceType<NestedExternalInterface>
i_1="IntegerSerialization"
i_2="StringSerialization"
r_1=Type<Boolean>
```

In this case also, only one external request is necessary to realize service interaction.

The reason for which Domain Specifications consist only of interfaces is to maximize the possible sequences of calls without exchange of requests between Service Instances.

1.5 Service Implementation development

A Service Implementation that don't inject any external Service is called a **Prototype Implementation** or Prototype. During development of any Service Implementation, it is very convenient to have Prototypes for each used Service Specification. If a Prototype implementation of each used Service Specification is available, then a local execution environment can be easily created.

Having Prototypes injected it is possible to execute internal tests. A SAM Architecture can additionally provide predefined tests for a given Service Specification called Test Specification. **Test Specification** is a set of test classes containing injection points only for one given Service Specification. Note that Test Specifications can be developed independently of any Service Implementations.The use of Canonical Execution Protocol for service interaction makes possible to create Test Specification on base of previously recorded services interaction. In case that a complex bug is found on some Implementation, a test case could be recorded and then executed on all existing Implementations.

Using Prototypes and Test Specifications on development process leads to Service implementations with defined injection as in figure 5.

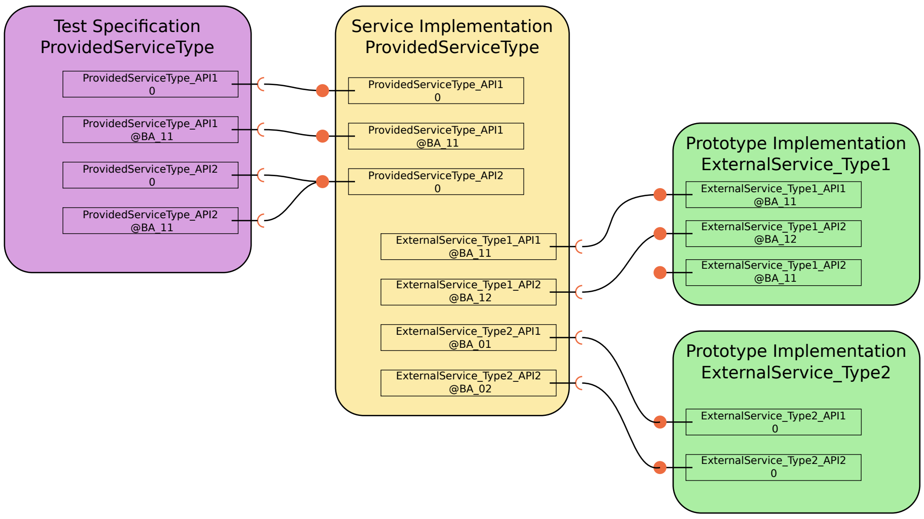
Test Specification
ProvidedServiceType
ProvidedServiceType_API1 0
ProvidedServiceType_API1 @BA_11
ProvidedServiceType_API2 0
ProvidedServiceType_API2 @BA_11
Service Implementation
ProvidedServiceType
ProvidedServiceType_API1 0
ProvidedServiceType_API1 @BA_11
ProvidedServiceType_API2 0
ExternalService_Type1_API1 @BA_11
ExternalService_Type1_API2 @BA_12
ExternalService_Type2_API1 @BA_01
ExternalService_Type2_API2 @BA_02
Prototype Implementation
ExternalService_Type1
ExternalService_Type1_API1 @BA_11
ExternalService_Type1_API2 @BA_12
ExternalService_Type1_API2 @BA_11
Prototype Implementation
ExternalService_Type2
ExternalService_Type2_API1 0
ExternalService_Type2_API2 0

Figure 5: Service Implementation injection for development

1.6 Service Execution Environment

Execution of a Service Implementation to create Service Instances is done by injecting dependencies for some existing Service Instance. Injection Process defined on section 3.2 support such execution process, allowing also the creation of cyclic injections between Service Instance. Each Service Instance gets a unique Instance ID (IID).

Injection Configuration defining Service Instance injection is a mapping

$$config : IID \times ServiceSpecification \longrightarrow IID$$

maintained by a **Service Execution Environment**.

Note that Service Implementations don't have any control over the injection process, they don't even have knowledge of IID of injected Service Instances. All the control over Service Instance injection is transferred to the Service Execution Environment.

Given two Injection Configurations, it is possible to merge then in a new mapping. This ensure Service Execution Environment scalability.

Two Service Execution Environments not trusting each other, can share limited information about running Services Instances and enable service interaction under the supervision of special security policies.

2 Service Architecture Model

This section define precisely all the Service Architecture Model concepts and assumptions.

To remark the use of SAM concepts on the text Start Case is used for concept names.

Defined sets and relations are related to one Architecture defined using Service Architecture Model.

2.1 Types and Interface

A object oriented approach is taken to define Types and Interfaces.

Definition 2.1 (Basic Type) *A predefined set of primitive types.*

A concrete list of accessible Basic Types is related to a given implementation of SAM. Bindings to specific programming languages may define mappings between provided Basic Types on SAM implementation and language types.

Definition 2.2 (Type) *Basic Type and Interface Types uniquely defined by a Interface.*

Definition 2.3 (Interface) *Interface is a set of methods. Method signature is defined by a*

- *method Name*
- *return Type*
- *list of arguments (list of Types)*

An Interface should be uniquely identified by a name.

The set of Interfaces is denoted by $\mathcal{I}$. The set of Types is denoted by $\mathcal{T}$. The set of Basic Types is denoted by $\mathcal{B} \subset \mathcal{T}$.

The mapping between Interface and corresponding Interface Type is denoted by function t

$$t : \mathcal{I} \to \mathcal{T} \tag{1}$$

and implies also a function $\tilde{t}$ between Interfaces sets to Types sets defined as:

$$\tilde{t} : 2^{\mathcal{I}} \to 2^{\mathcal{T}} \quad \tilde{t}(\{i_1, \ldots, i_\alpha\}) := \{t(i_1), \ldots, t(i_\alpha)\} \tag{2}$$

Definition 2.4 (Signature Relation) *Interface 'i' is signature related to Type 't' if a method signature of 'i' contains 't'. This relation is denoted by set '$\mathring{s}$':*

$$\mathring{s} \subset \mathcal{I} \times \mathcal{T} \tag{3}$$

Definition 2.5 (Signature Maps) *The Signature Map is the function s*

$$s : \mathcal{I} \to 2^{\mathcal{T}} \tag{4}$$

given by the statement:

$$(i, t) \in \mathring{s} \Rightarrow t \in s(i)$$

A set version of the Signature Map is defined by function $\tilde{s}$

$$\tilde{s} : 2^{\mathcal{I}} \to 2^{\mathcal{T}} \tag{5}$$

defined as

$$\tilde{s}(\{i_1, \ldots, i_\alpha\}) := s(i_1) \cup \cdots \cup s(i_\alpha) \tag{6}$$

Assumptions:

1. For a given Architecture, Signature methods of Interfaces use only Basic Types or Types defined by Interfaces belonging to the given Architecture.

2.2 Domain Specification Model

Definition 2.6 (Domain Specification) *A named set of Interfaces.*

Assumptions:

1. Domain Specifications on Architecture are mutually disjoint.

The set of Domain Specifications is denoted by $\mathcal{D}$. Because for $d \in \mathcal{D}$ we have $d \subset 2^{\mathcal{I}}$, function $\tilde{t}$ and $\tilde{s}$ are well defined on $\mathcal{D}$.

Definition 2.7 (Domain Dependency Relation) *Domain Specification d_1 **depends on** Domain Specification d_2 $[(d_1, d_2) \in \delta]$, if a Type related to a Interface belonging to d_2 is used on a method signature of a Interface belonging to d_1*

$$\delta \subset \mathcal{D} \times \mathcal{D} \tag{7}$$
$$\tilde{t}(d_2) \cap \tilde{s}(d_1) \neq \emptyset \Rightarrow (d_1, d_2) \in \delta$$

Assumptions:

1. Domain Dependency Relation δ is acyclic.

Simple Domain Specification is a Domain Interfaces that use in methods signature only Basic Types and own Types, precisely:

Definition 2.8 (Simple Domain Specification) *Domain Specification d such that*

$$\tilde{s}(d) \subset \mathcal{B} \cup \tilde{t}(d) \tag{8}$$

Definition implies that Simple Domain Specification don't depend on any Domain Specification.

Definition 2.9 (Model Domain Specification) *Domain Specification that depends on other Domain Specification.*

2.3 Service Specification

Definition 2.10 (Service Specification) *A Model Domain Specification.*

Assumptions:

1. Services are not used as Domain Dependencies: if s is a Service Specification, then

$$\forall d \in \mathcal{D} \quad (d, s) \notin \delta \tag{9}$$

The set of Service Specification is denoted by $\mathcal{S}$, we have $\mathcal{S} \subset \mathcal{D}$.

2.4 Taxonomy and Category

Definition 2.11 (Taxonomy) *Equivalence relation of Services Specifications*

$$\tau \subset \mathcal{S} \times \mathcal{S} \tag{10}$$

Definition 2.12 (Category) *Equivalence class of Taxonomy.*

The set of Categories is denoted by Γ. Taxonomy projection is denoted by

$$cat : \mathcal{S} \to \Gamma \tag{11}$$

2.5 Service Implementation

Service Implementation definition is based on Dependency Injection Model presented on section 3.

Definition 2.13 (Architecture Annotations) *A set of Binding Annotations.*

Definition 2.14 (Service Implementation) *A Binding Provider on Architecture Annotations for all the Types related to a Service Specification.*

The set of Service Implementations is denoted by $\mathcal{P}$. Function *spec*

$$spec : \mathcal{P} \to \mathcal{S} \tag{12}$$

assigns Service Implementation to realized Services Specifications.

Definition 2.15 (Injection Relation) *Service Implementation p **injects** Service Specification s $[(p, s) \in \kappa]$ if the following chain can be build:*

$$\mathcal{P} \ni p \xrightarrow[\text{contain Class}]{} c \xrightarrow[\text{with Injection Point}]{} ip \xrightarrow[\text{matching Key}]{} k \xrightarrow[\text{of Type}]{} a$$

$$a \xrightarrow[\text{related to Interface}]{t(i)=a} i \xrightarrow[\text{belonging to Service Specification}]{i \in s} s \in \mathcal{S}$$

$$\kappa \subset \mathcal{P} \times \mathcal{S} \tag{13}$$

Initial four relations on definition of Injection Relation are defined in Dependency Injection Model section 3. The last two relations are defined by logical condition over the arrows.

Definition 2.16 (Set of Injected Service Specifications) *of Service Implementation p is*

$$\begin{aligned} injected &: \mathcal{P} \to 2^{\mathcal{S}} \\ injected(p) &:= \{s : (p, s) \in \kappa\} \subset \mathcal{S} \end{aligned} \tag{14}$$

2.6 Category Accessibility Relation

Definition 2.17 (Category Accessibility Relation) *A predefined relation*

$$\pi \subset \Gamma \times \Gamma \tag{15}$$

Category c_2 **is accessible** to Category c_1 if $(c_1, c_2) \in \pi$.

As determined by the next assumption, Category Accessibility Relation π restrict injection of Services on Implementations.

Assumptions:

1. $\forall s_x \in \mathit{injected}(p) \quad (\mathit{cat}(\mathit{spec}(p)), \mathit{cat}(s_x)) \in \pi$.

This mean that if the following conditions are satisfied

- Implementation p implements Service s $[\mathit{spec}(p) = s]$
- Implementation p injects Service s_x $[s_x \in \mathit{injected}(p) \equiv (p, s_x) \in \kappa\ ]$
- Service s belongs to Category C $[s \in C]$
- Service s_x belongs to Category C_x $[s_x \in C_x]$

then it is required that Category C_x is accessible to Category C $[(C, C_x) \in \pi]$.

2.7 Prototype and Test Specifications

For development of Service Implementations it is necessary to have access to external services for local execution and testing.

Definition 2.18 (Prototype Implementation) *is a Service Implementation that don't inject any Service Specification.*

Prototypes can be executed without any other resource, as they don't inject any service.

Definition 2.19 (Test Specification) *is a Binding Provider that injects only one Service Specification.*

Test Specifications can inject a Service Implementation and execute test using the Service Specification Interfaces. Test Specifications may implement a Service Specification to control the execution of tests and provide results in a specific format, but this is a per Architecture choice.

Test Specifications may be treated as formal specifications for a Service Specification.

3 Dependency Injection Model

Dependency Injection Model define a Injection Process to instantiate instances of classes with injected dependencies. This is a self-containing abstract model independent of Service Architecture Model.

Dependency Injection Model idea is based on Guice dependency injection framework abstraction model, but it cover only explicitly defined bindings. All injection frameworks know to the author are compatible with this model, only specific configuration details change. For details Guice Abstract Model see `http://code.google.com/p/google-guice/wiki/ExtendingGuice#Core_Abstractions`.

The used subset of the Guice Abstract Model is selected to provide only external injection configurations. In this way, the injection process can be controlled from a external execution environment. This feature is used on the definition of the Service Execution Environment.

Use of Guice library is NOT necessary to implement the Dependency Injection Model nor Service Architecture Model.

This definition is provided in the context of the Java language, but any semantically equivalent implementation in other languages are possible.

3.1 Binding Provider

Definition 3.1 (Binding Annotation) *A Java annotation instance.*

Definition 3.2 (Injection Point) *A constructor argument, field or method argument of a Java Class that can receive injections, plus an optional binding annotation.*

Usually Injection Point are annotated with the `javax.inject.Inject` annotation.

Definition 3.3 (Key) *A pair containing a Type and a Binding Annotation or empty set ∅.*

$$(\mathit{Type} \quad , \quad \mathit{Binding\ Annotation})$$
$$(\mathit{Type} \quad , \quad \emptyset)$$

A Key is represented by a object of type `com.google.inject.Key<T>`.

Definition 3.4 (BindingProvider interface) `BindingProvider` *interface defined by listing 7.*

Listing 7: BindingProvider Interface

```java
package eu.pmsoft.inject;

import javax.inject.Provider;

import com.google.inject.Key;

public interface BindingProvider {

 <T> Provider<T> getProvider(Key<T> key);

}
```

Definition 3.5 (Binding Provider) *on a set of Annotation Types B for a set of Types T is a implementation of* `BindingProvider` *interface that retrieve a instance of type* `javax.inject.Provider<T>` *for each possible generated* `com.google.inject.Key<T>` *using (T,B).*

Type `javax.inject.Provider<T>` has method `T get();` used to retrieve a instance of Type `T` as marked in Key.

3.2 Injection Process

Given a Binding Provider, the process of creation of instances by a Key is given by

Definition 3.6 (Injection Process) *The algorithm to retrieve an instance of Type T for Key k of type $< T, \beta >$ using a Binding Provider* `binding` *defined as follow:*

1. *retrieve provider for key k and execute internal Provider logic*

   ```
   javax.inject.Provider<T> p = binding.getProvider(k);
   ```

   ```
   T = p.get();
   ```

2. *Provider p can return a already existing instance of Type T and finish.*

3. *If this is a recursive execution of Injection Process that generate a cyclic dependency on Provider p, then return a Proxy instance that will delay further steps of Injection Process until first method call to that Proxy. Proxy instance executes ones the Injection Process and fix the retrieved instance to redirect all methods calls to it.*

4. *Provider choose a class constructor to create a new instance on base of Key information $< T, \beta >$.*

   Internal Provider logic must produce a instance with the same references on injection points as given by the following steps

5. *Find all* ***constructor argument*** *Injection Point on the constructor that Provider p will use to instantiate the instance*

6. *Find all* ***method argument*** *Injection Points on the class that Provider p will use to instantiate the instance*

7. *Find all* ***field*** *Injection Points on the class that Provider p will use to instantiate the instance*

8. *For each Injection Point from 5), look for matching Keys and run the Injection Process*

9. *Use retrieved instances from 8) as arguments to execute the selected constructor*

10. *For each Injection Point from 6), look for matching Keys and run the Injection Process*

11. *For each Injection Point from 7), look for matching Keys and run the Injection Process*

12. *Use retrieved instances from 10) as arguments to related Injection Point methods*

13. *Use retrieved instances from 11) to set related Injection Point field reference*

Provider behavior can be externally controlled.

3.3 Binding Replacement

When deploying a Service Implementation on a Service Execution Environment, it is necessary to change a Prototype implementation of a service with some other implementations. This section define the Binding Replace operation to make that change.

Definition 3.7 (Binding Replacement) *is a operation that generate a new Binding Provider on base of Binding Provider $\tilde{b}$ and a pair consisting on Binding Provider b and set of Keys K.*

$$BReplace(\tilde{b}, (b, K)) \tag{16}$$

Binding Provider $BReplace(b, (b, K))$ retrieve Providers using logic:

$$BReplace(\tilde{b}, (b, K)).get(key) = \begin{cases} \tilde{b}.get(key) & if \quad key \notin K \\ b.get(key) & if \quad key \in K \end{cases} \tag{17}$$

For a set of pairs $\{(b_i, K_i)\}$ of Binding Provider and set of Keys, such that K_i are mutually disjoin define

$$BReplace(\tilde{b}, \{(b_i, K_i)\}) = \begin{cases} \tilde{b}.get(key) & if \quad key \notin \bigcup K_i \\ b_i.get(key) & if \quad key \in K_i \end{cases} \tag{18}$$

4 Service Execution Environment

Creation of Service Instances and management of Service injection is fully controlled by a Service Execution Environment. This section defines a model for configuration of a Service Execution Environment. Concrete implementations may use internally a different approach, nevertheless Service Execution Environment interaction as defined on this section require exchange of information according to the presented model.

4.1 Configuration Model

Service Instance is a execution of a Service Implementation, it is a instantiation of the Binding Provider given by the Service Implementation. It is possible to create many Service Instance using the same Service Implementation, also different Service Implementations may be used to create Service Instance on the same Service Execution Environment. Each Service Instance has a unique Service Instance ID (IID).

Note that instantiation of the Binding Provider do not require instantiation of classes implementing the Service Specification Interfaces. Only after the first method execution, the injection process must executed.

The set of Service Instances ID on a Service Execution Environment is denoted by Ω.

Define function *provider* such that retrieve the Service Implementation used to create a Service Instance.

$$provider : \Omega \to \mathcal{P} \tag{19}$$

Remark that for a Service Implementation, function *spec* give the Service Specification implemented and function *injected* give the Set of Injected Service Specifications.

$$\begin{aligned} spec &: \mathcal{P} \to \mathcal{S} \\ injected &: \mathcal{P} \to 2^{\mathcal{S}} \end{aligned}$$

Definition 4.1 (Injection Configuration) *is a mapping*

$$config : \Omega \times \mathcal{S} \to \Omega \tag{20}$$

such that

$$\begin{aligned} &\forall \alpha \in \Omega \quad \forall i \in injected(provider(\alpha)) \\ &\qquad spec(provider(config(\alpha, i))) = i \end{aligned} \tag{21}$$

Condition on Injection Configuration ensure that given Service Instance ID by mapping *config* is a good candidate for replacing injected Service Specifications on a Service Implementation.

4.2 Service Instance creation

To create a Service Instance for a given Service Implementation each Injection Point is bind to the Service Instance marked be the Injection Configuration.

Definition 4.2 (Service Instance creation algorithm) *To create a Service Instance for Service Implementation $p \in \mathcal{P}$:*

1. *create a new IID α and extend Injection Configuration $config$.*
2. *for each injected service*

   $$i \in injected(p)$$

   create the pair (b_i, K_i) (Binding Provider,Key set) given be (Service Implementation with IID = $config(\alpha, i)$, maximum set of Keys generated with Service Specification i).
3. *create a new Binding Provider*

   $$\tilde{p} = BReplace(p, \{(b_i, K_i)\})$$
4. *the new Service Instance is the execution of Binding Provider $\tilde{p}$.*

Dynamic change of Injection Configuration, it is change of injected Service Instance during execution are not defined. It may be possible, but require additional assumptions on internal Service Instance execution.

4.3 Environment interaction

Given two Injection Configurations over the same Architecture

$$config_1 : \Omega_1 \times \mathcal{S} \to \Omega_1$$
$$config_2 : \Omega_2 \times \mathcal{S} \to \Omega_2$$

it is possible to define a merged one as

$$\begin{aligned} config &: \Omega_1 \cup \Omega_2 \times \mathcal{S} \to \Omega_1 \cup \Omega_2 \\ config(\alpha, s) &= \begin{cases} config_1(\alpha, s) & \text{if} \quad \alpha \in \Omega_1 \\ config_2(\alpha, s) & \text{if} \quad \alpha \in \Omega_2 \end{cases} \end{aligned}$$

Using the merged Injection Configurations, it is possible to create new Service Instance with injections belonging to the set $\Omega_1 \cup \Omega_2$.

Given two separate Service Execution Environments, it is possible to interconnect Service Instances using a merged Injection Configuration as above. This is a key feature to allow extension of the Service Execution Environment to a distributed environment.

4.4 Canonical Execution Protocol

Service Instance interaction in a Service Execution Environment is defined by exchange of messages using the following Canonical Execution Grammar.

Definition 4.3 (Canonical Execution Grammar)

$\langle serviceExecution \rangle$ ::= $\langle execution \rangle$ 'PAYLOAD' $\langle payload \rangle$

⟨*execution*⟩ ::= ⟨*methodCall*⟩*

⟨*methodCall*⟩ ::= ⟨*objectid*⟩ METHOD_SIGNATURE '(' ⟨*objectid*⟩* ')' ⟨*objectid*⟩?

⟨*objectid*⟩ ::= 'r_' INTEGER | 'i_' INTEGER

⟨*payload*⟩ ::= ⟨*objectPayload*⟩*

⟨*objectPayload*⟩ ::= ⟨*objectid*⟩ '=' ⟨*objectData*⟩

⟨*objectData*⟩ ::= ⟨*bindingData*⟩ | ⟨*serializationData*⟩ | ⟨*typeData*⟩

⟨*bindingData*⟩ ::= 'KEY⟨*' TYPE ',' BINDING_ANNOTATION '*⟩'

⟨*serializationData*⟩ ::= OBJECT_SERIALIZATION

⟨*typeData*⟩ ::= 'TYPE⟨*' TYPE '*⟩'

METHOD_SIGNATURE, INTEGER, TYPE, BINDING_ANNOTATION and OBJECT_SERIALIZATION are literal categories.

Calls to methods from Interfaces of Service Specifications can be serialized to a **Canonical Request** using the Canonical Execution Grammar. The following rules apply:

1. Create one ⟨*serviceExecution*⟩ production to record methods calls.
2. For each method call create a ⟨*methodCall*⟩ [see 4)] and add it to the list on ⟨*execution*⟩ of ⟨*serviceExecution*⟩
3. If called object reference has no related ⟨*objectid*⟩, then

   Create a new ⟨*objectid*⟩ using 'i_' and increasing integer value for literal INTEGER

   Create a new ⟨*objectPayload*⟩ for this object.

   Remember Object reference relation to generated ⟨*objectid*⟩.

   Add ⟨*objectPayload*⟩ to ⟨*payload*⟩ of ⟨*serviceExecution*⟩
4. ⟨*methodCall*⟩ is created using

   ⟨*objectid*⟩ related to object reference called

   Unique METHOD_SIGNATURE literal as a hash function of the executed method signature

   '('

   ⟨*objectid*⟩ for each object reference argument

   ')'

   If method has void/VOID return Type, then no final ⟨*objectid*⟩ element is added. Otherwise:

   Create a new ⟨*objectid*⟩ using 'r_' and increasing integer value for literal INTEGER

   Create a new ⟨*objectPayload*⟩ production with alternative ⟨*typeData*⟩.

5. If object reference on 3) is a injected object reference of a Interface of a Service Specification, then use $\langle bindingData \rangle$ alternative, otherwise use $\langle serializationData \rangle$
6. To produce $\langle bindingData \rangle$ use Type and Binding Annotation of Key associated to the injected object reference.
7. To produce $\langle serializationData \rangle$ use a unique serialization method for objects.

Service Execution Environment is capable to repeat method's calls on base of Canonical Requests: for a Canonical Request produced on Service Instance with IID i

1. for a given $\langle bindingData \rangle$, deserialize Key k and find relative Service Specification s for TYPE.
2. find Service Instance ID to call using the Injection Configuration

$$e = config(i, s) \tag{22}$$

3. On Service Instance with IID e call the BindingProvider with Key k to get object o
4. Repeat method call on object o.

Note that it is possible to pass object references returned from one service as arguments to other service. In this case the Service Execution Environment must pass a object reference to a Proxy and serialize methods call to it as methods call to a injected object. This process can produce service interaction between Service Instances not directly interconnected.

5 Model Concepts

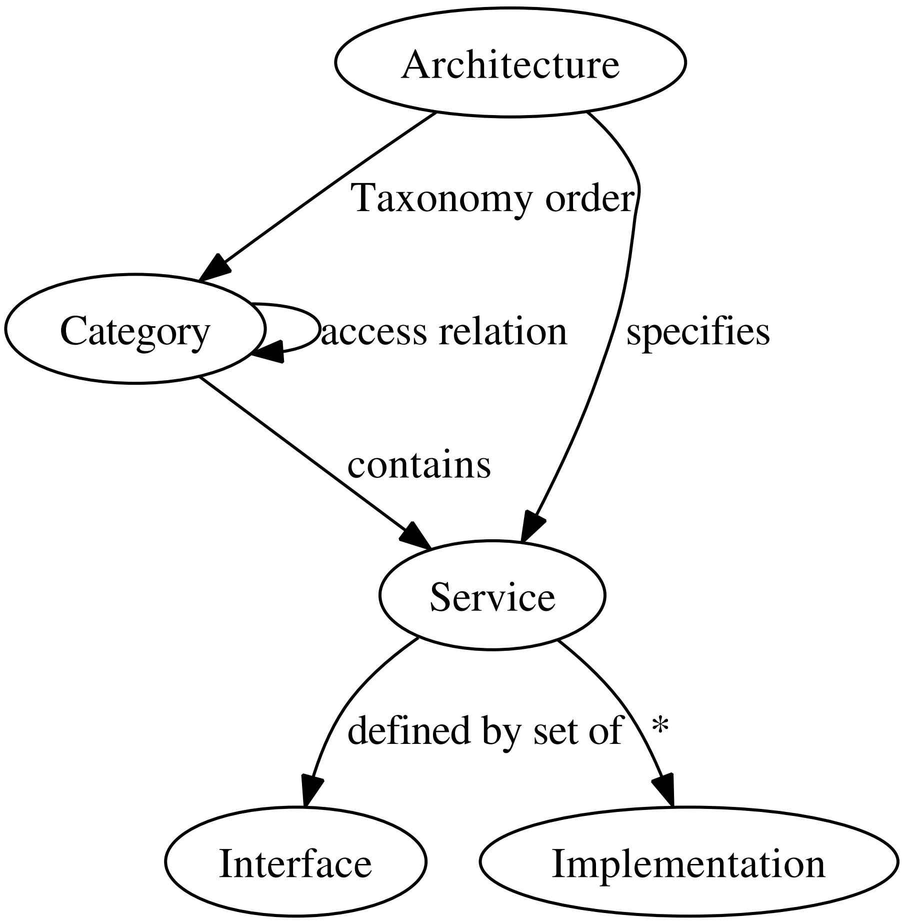

Figure 6: Model Concepts relations