Draft Technical Standard

Service-Oriented Architecture Ontology

Contents

Preface

The Open Group

The Open Group is a vendor-neutral and technology-neutral consortium, whose vision of Boundaryless Information Flow™ will enable access to integrated information within and between enterprises based on open standards and global interoperability. The Open Group works with customers, suppliers, consortia, and other standards bodies. Its role is to capture, understand, and address current and emerging requirements, establish policies, and share best practices; to facilitate interoperability, develop consensus, and evolve and integrate specifications and Open Source technologies; to offer a comprehensive set of services to enhance the operational efficiency of consortia; and to operate the industry's premier certification service, including UNIX® certification.

Further information on The Open Group is available at www.opengroup.org.

The Open Group has over 15 years' experience in developing and operating certification programs and has extensive experience developing and facilitating industry adoption of test suites used to validate conformance to an open standard or specification.

More information is available at www.opengroup.org/certification.

The Open Group publishes a wide range of technical documentation, the main part of which is focused on development of Technical and Product Standards and Guides, but which also includes white papers, technical studies, branding and testing documentation, and business titles. Full details and a catalog are available at www.opengroup.org/bookstore.

As with all *live* documents, Technical Standards and Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards-compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Readers should note that updates – in the form of Corrigenda – may apply to any publication. This information is published at www.opengroup.org/corrigenda.

This Document

This document is the Draft Technical Standard for Service-Oriented Architecture Ontology. It has been developed by The Open Group.

Trademarks

Boundaryless Information Flow™ and TOGAF™ are trademarks and Making Standards Work®, The Open Group®, UNIX®, and the "X" device are registered trademarks of The Open Group in the United States and other countries.

The Open Group acknowledges that there may be other brand, company, and product names used in this document that may be covered by trademark protection and advises the reader to verify them independently.

Referenced Documents

The following documents are referenced in this Technical Standard:

- **[BPEL]** Web Services Business Process Execution Language. The Organization for the Advancement of Structured Information Standards, available from http://www.oasis-open.org
- **[BPMN]** Business Process Modeling Notation. Version 1.1. The Object Management Group, available from http://www.omg.org
- **[COBIT]** Control Objectives for Information and related Technology (COBIT) Version 4.1, The IT Governance Institute, available from http://www.isaca.org
- **[IEEE 1471]**Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471-2000, the IEEE, http://www.ieee.org
- **[OASIS RM]**

  OASIS Reference Model for Service-Oriented Architecture Version 1.0, The Organization for the Advancement of Structured Information Standards, available from http://www.oasis-open.org
- **[OECD]** The OECD Corporate Governance Principles 2004, the Organization for Economic Cooperation and Development, http://www.oecd.org
- **[OWL]** OWL Web Ontology Language Reference: W3C Recommendation 10 Feb 2004. The World-Wide Web Consortium, http://www.w3.org
- **[OWL-S]** Semantic Mark-Up for Web Services: W3C Member Submission 22 November 2004. The World-Wide Web Consortium, http://www.w3.org
- **[TOGAF]** The Open Group Architecture Framework Version 8.1.1, The Open Group, http://www.opengroup.org
- **[WSMO]** Web Services Moedeling Ontology. Developed as part of the European Semantic Systems Initiative. Available from http://www.wsmo.org

1 Introduction

1.1 Objective

The purpose of this Technical Standard is to contribute to the Open Group mission of Boundaryless Information Flow, by developing and fostering common understanding of SOA in order to improve alignment between the business and information technology communities, and facilitate SOA adoption.

It does this in two specific ways.

1. It defines the concepts, terminology and semantics of SOA in both business and technical terms, in order to:
    - Create a foundation for further work in domain-specific areas,
    - Enable communications between business and technical people,
    - Enhance the understanding of SOA concepts in the business and technical communities, and
    - Provide a means to state problems and opportunities clearly and unambiguously to promote mutual understanding.
2. It potentially contributes to model-driven SOA implementation.

The ontology is designed for use by:

- Business people, to give them a deeper understanding of SOA, and its use in the enterprise;
- Architects, as metadata for architectural artifacts; and
- Architecture methodologists, as a component of SOA metamodels.

1.2 Overview

This Technical Standard defines a formal ontology for Service Oriented Architecture.

Service-Oriented Architecture (SOA) is an architectural style that supports service orientation**:** a way of thinking in terms of services and service-based development and the outcomes of services.

The ontology is written in the Web Ontology Language (OWL) defined by the World-Wide Web Consortium (see [OWL]). It contains classes and properties corresponding to the important

concepts of SOA. The formal OWL definitions are supplemented by textual explanations of the concepts, with graphic illustrations of the relations between them, and examples of their use.

OWL has three increasingly expressive sublanguages: OWL-Lite, OWL-DL, and OWL-Full. This ontology uses OWL-DL, the sublanguage that provides the greatest expressiveness possible while retaining computational completeness and decidability.

This Chapter provides an introduction to the whole document. Chapter 2: *Services – Basic Definitions* describes the basic concepts associated with services. Chapter 3: *Services as Business Activities* describes the concepts associated with activities, and in particular business activities, and describes how they relate to services. Chapter 4: *Design and Implementation* describes concepts related to the implementation of services, such as *choreography*, and *orchestration*. Chapter 5: *Architecture and Governance* describes concepts related to the development and management of service-oriented architectures, and to the governance of their development, implementation and operation. The Appendix contains the formal OWL definitions of the ontology, collected together.

1.3 Applications

The ontology was developed in order to aid understanding, and potentially be a basis for model-driven implementation.

To aid understanding, it can simply be read. To be a basis for model-driven implementation, it should be applied to particular usage domains. And application to example usage domains will aid understanding.

The ontology is applied to a particular usage domain by adding to it instances which are things in that domain. This is sometimes referred to as “populating the ontology”. In addition, an application can add definitions of new classes and properties, can import other ontologies, and can import the ontology into other ontologies.

This technical standard uses example applications to illustrate the ontology. One of these, the car-wash example, is used consistently throughout to illustrate the main concepts. Other examples are used ad-hoc in individual sections to illustrate particular points.

The ontology defines the relations between terms, but does not prescribe exactly how they should be applied. The sections of this Technical Standard that use the car-wash example are describing one way in which the ontology could be applied in a practical situation. Different applications of it to the same situation would nevertheless be possible. For instance, Joe's bucket becoming empty is not regarded as an event in the car-wash example as described in this standard, but it would be quite possible to treat this as an event. The precise interpretation of the terms of the ontology in particular practical situations is a matter for users of the ontology, and is not constrained by the ontology definition. The use of a term by an application does not imply that the term is used accurately or truthfully.

While this Technical Standard does not prescribe exactly how the ontology should be applied, it does clearly suggest particular ways in which it should be applied to enterprise services and

enterprise IT architecture. This should be regarded as guidance for the user rather than as a matter of formal conformance.

1.4 Conformance

This Technical Standard does not define what it means for a system to conform to it.

A conforming application:

- Must conform to the OWL standard [OWL];
- Must include the whole of the ontology contained in Appendix A of this Technical Standard;
- Must add one or more instances from a particular domain or domains;
- Can add other OWL constructs, including class and property definitions;
- Can import other ontologies;
- Must be consistent; and
- Can be imported into other ontologies.

1.5 Terminology

Can
: Describes a permissible optional feature or behavior available to the user or application. The feature or behavior is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.

Implementation-dependent
: (Same meaning as "implementation-defined".) Describes a value or behavior that is not defined by this document but is selected by an implementer. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence of the value or behavior. An application that relies on such a value or behavior cannot be assured to be portable across conforming implementations. The implementer shall document such a value or behavior so that it can be used correctly by an application.

Legacy
: Describes a feature or behavior that is being retained for compatibility with older applications, but which has limitations which make it inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality.

May
: Describes a feature or behavior that is optional for an implementation that conforms to this document. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. To avoid ambiguity, the opposite of "may" is expressed as "need not", instead of "may not".

Must	Describes a feature or behavior that is mandatory for an application or user. An implementation that conforms to this document shall support this feature or behavior.
Shall	Describes a feature or behavior that is mandatory for an implementation that conforms to this document. An application can rely on the existence of the feature or behavior.
Should	For an implementation that conforms to this document, describes a feature or behavior that is recommended but not mandatory. An application should not rely on the existence of the feature or behavior. An application that relies on such a feature or behavior cannot be assured to be portable across conforming implementations. For an application, describes a feature or behavior that is recommended programming practice for optimum portability.
Undefined	Describes the nature of a value or behavior not defined by this document that results from use of an invalid program construct or invalid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.
Unspecified	Describes the nature of a value or behavior not specified by this document that results from use of a valid program construct or valid data input. The value or behavior may vary among implementations that conform to this document. An application should not rely on the existence or validity of the value or behavior. An application that relies on any particular value or behavior cannot be assured to be portable across conforming implementations.
Will	Same meaning as "shall"; "shall" is the preferred term.

1.6 Typographic Conventions

Bold font is used for OWL class, property, and instance names where they appear in section text. ***Bold Italic*** font is used to distinguish them in section headings.

Italic strings are used for emphasis and to identify the first instance of a word requiring definition.

OWL definitions and syntax are shown in fixed-width font.

1.7 Diagrammatic Symbolism

This specification describes a formal OWL ontology. For purposes of exposition, it includes diagrams that illustrate the classes and properties of the ontology. These diagrams use a set of conventions for representing formal OWL constructs, which are explained in this section.

Note that these diagrams make many simplifications. They omit some classes and properties – particularly the inverse properties of those shown. The OWL definitions contained in this specification constitute the authoritative definition of the ontology.

1.7.1 Class

Owl classes are represented by rectangular boxes, labeled with the class names, similar to the example in Figure 1.

Figure 1: Example Diagrammatic Representation of a Class

Class names are nouns with their first letters in uppercase. Where a class name includes multiple words, they are shown in the diagrams with the words separated by spaces, for example: **Information Item**. The spaces are omitted in the names used in OWL, as in `InformationItem`.

The word **Anything** is used in the diagrams to represent the OWL construct `Thing`.

1.7.2 Subclass

A subclass relation between classes is shown by nesting the boxes that represent them, as illustrated in Figure 2, which shows **Service** as a subclass of **Activity**.

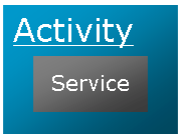

Figure 2: Example Diagrammatic Representation of Subclass Relationship

1.7.3 Property

A property is represented by an arrow, as illustrated in Figure 3.

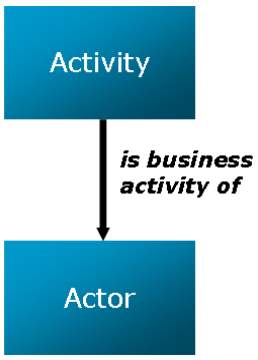

Figure 3: Example Diagrammatic Representation of Property

This shows that **Activity** and **Actor** are classes and that the **is business activity of** property provides a connection between them.

Note that it does not imply that the domain of **is business activity of** is **Activity**, or that its range is **Actor**. It happens in this case that **is business activity of** has domain **Activity** and range **Actor**, but the symbolism is used in some other cases to show how a property connects classes other than its domain and range. For example, Figure 5 shows that the **has effect** property provides a connection between the **Service** and **Effect** classes, but **Service** is not the domain of **has effect**.

Further, there is no implication that for every instance of **Activity** there is an instance of **Actor** that it **is business activity of**, or that for every instance of **Actor** there is an instance of **Activity** that **is business activity of** it.

The diagrammatic representation does not in fact imply any formal statement about the classes and property that it illustrates. It simply indicates that there are many instances of **Activity** for which **is business activity of** has a value and that its values for those instances are instances of **Actor**. Its meaning is intuitively quite clear, although the formal explanation turns out to be rather complex.

In the diagrams, property names are verbs and are all in lowercase. Where a property name includes multiple words, they are shown in the diagrams with the words separated by spaces, for example: **is business activity of**. The spaces are omitted in the names used in OWL, but the initial letters of the second and subsequent words are in uppercase, as in `isBusinessActivityOf`.

The arrow points from the domain of the property to its range, and the name is chosen such that the subject of the verb is in the domain of the property and the object of the verb is in its range. For example, an **Activity is business activity of** an **Actor**.

1.7.4 Cardinality Constraint

A cardinality constraint can be indicated by an annotation to an arrow representing a property, as illustrated in Figure 4.

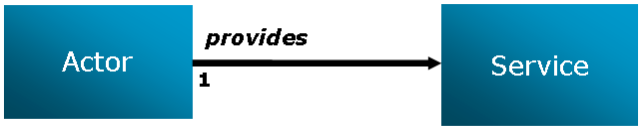

Figure 4: Example Diagrammatic Representation of a Cardinality Constraint

This indicates that, for every instance of the **Service** class, there is exactly one instance of the **Actor** class that **provides** that **Service**. (If the figure 1 had been at the other end of the arrow then it would indicate that every **Actor provides** exactly one **Service**.)

1.8 Future Directions

It is anticipated that this will be a living document that will be updated as the industry evolves and SOA concepts are refined.

Also, this ontology can be used as a core for domain-specific ontologies that apply to the use of SOA in particular sectors of commerce and industry. The Open Group does not currently plan to develop such ontologies, but encourages other organizations to do so to meet their needs.

2 Services – Basic Definitions

2.1 Introduction

Service is the core concept of this ontology. It is a concept that is well-understood in practice, but is not easy to define. The ontology assumes the following definition, which was developed by The Open Group's SOA Work Group.

> A *service* is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit; provide weather data, consolidate drilling reports). It is self-contained, may be composed of other services, and is a "black box" to its consumers.

In the ontology:

- The class **Service** is defined as a subclass of the **Activity** class;
- Outcomes of services are instances of the class **Effect**, which is related to **Service** by the **is effect of** property;
- A service composition is an instance of the **Composition** class;
- A **Description** can **describe** a service as a self-contained "black box";
- A service consumer is an instance of the class **Actor**, which is related to **Service** by the **consumes** property.

This chapter describes the basic concepts of the ontology:

- Services;
- Providers and Consumers;
- Effects;
- Information; and
- Systems and Composition.

2.2 Services

2.2.1 Overview

Service is the most fundamental concept of this ontology. This section describes the **Service** class.

Services are activities. The **Activity** class and related concepts are described in Chapter 3.

2.2.2 Example

Joe has a one-man business. He stands on a street corner with a sponge, a bucket of water, and a sign saying "Car Wash $5". A customer drives up to him and asks him to wash the car. Joe asks the customer for five dollars. The customer gives him five dollars. Joe washes the car, then says, "That's all done now," and the customer drives away.

Joe carries out a repeatable business *activity*. He *provides* a car-wash *service*. It has an *effect* that is a specified outcome: the customer's car is cleaned, which is a *change* to the car. It is self-contained – it is not linked to any other activity. It is a "black box" to its consumers: the customer does not care whether Joe washes the car by hand or uses a car-wash machine, so long as the car is cleaned. Joe and the customer are *actors*. The customer *consumes* the service.

2.2.3 The *Service* Class

```
<owl:Class rdf:ID="Service">
   <rdfs:subClassOf rdf:resource="#Activity"/>
</owl:Class>
```

A *service* is a logical representation of a repeatable business activity. The class **Service** is defined here as a subclass of the **Activity** class.

A service has a *provider*, may have multiple *consumers*, and produces one or more *effects* (which have value to the consumers). The providers and consumers are *actors*. The classes and properties corresponding to these concepts are illustrated in Figure 5. The **Actor** class and the **provides** and **consumes** properties are defined in Section 2.3. The **Effect** class and the **has effect** property are defined in Section 2.4.

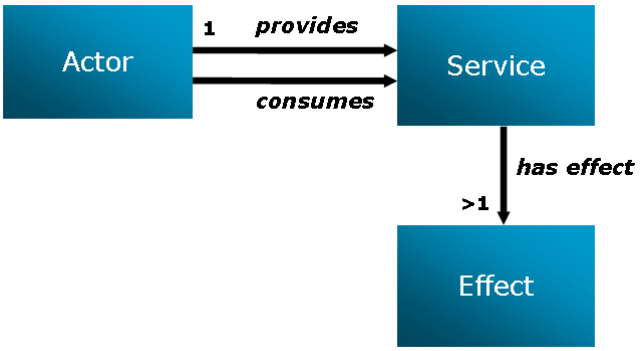

Figure 5: Service

2.3 Providers and Consumers

2.3.1 Overview

An idea that is central to the concept of service is that a service has a provider, and may have multiple consumers. In this ontology, providers and consumers are *actors*. This section defines the **Actor** class and the **provides** and **consumes** properties.

2.3.2 The *Actor* Class

```
<owl:Class rdf:ID="Actor"/>
```

An *actor* is someone or something that does something.

This is a general concept. The ontology includes various kinds of actor other than service providers and customers.

An actor can be a person or an organization or a piece of technology. (See sections 4.6.2, 4.6.5, and 4.6.3 for definitions of subclasses for these kinds of actor.)

This classification is not exhaustive. For example, an animal might be an actor. But only these kinds of actor are described in this ontology.

In modeling, an actor typically represents a role, or class, rather than an individual: for example "Barber", rather than "Sweeney Todd". In this ontology, the **Actor** class includes actors in both of these senses – both "Barber" and "Sweeney Todd".

In the car-wash example, Joe and the customer are *actors*. "Customer" is a role, while "Joe" is an individual. The example could have been described entirely in terms of roles, by characterizing Joe as "car washer", or it could have been described in terms of individuals, if a particular customer had been identified by name.

(See section 4.3 for discussion of how an actor that represents a role or class is an abstraction, and how a real actor such as Joe can be a realization of such an abstraction.)

2.3.3 The *provides* and *is provided by* Properties

```
<owl:ObjectProperty rdf:ID="provides">
   <rdfs:domain rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isProvidedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isProvidedBy">
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#provides"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Service">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#isProvidedBy"/>
         <owl:cardinality
```

```
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

A *provider* of an activity is an actor that takes responsibility for it being carried out. Every service has a provider.

The ontology does not include a class for service providers, but focuses on the **provides** property and its inverse **is provided by**. A *provider* is an instance of the domain of the **provides** property.

A service has a single provider - hence the cardinality restriction on **Service**.

"Provides" is not just a transient relation. It includes "provides at this instant", "has provided", and "may in future provide".

In the car-wash example, Joe *provides* the car-wash service.

2.3.4 The *consumes* and *is consumed by* Properties

```
<owl:ObjectProperty rdf:ID="consumes">
   <rdfs:domain rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isConsumedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isConsumedBy">
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#consumes"/>
</owl:ObjectProperty>
```

A *consumer* of a service (or some other thing) is an actor that uses it. A service can have one or more consumers.

The ontology does not include a class for service consumers, but focuses on the **consumes** property and its inverse **is consumed by**. A "consumer" is an instance of the domain of the **consumes** property.

"Consumes" is not just a transient relation. It includes "consumes at this instant", "has consumed", and "may in future consume".

In the car-wash example, the customer *consumes* the car-wash service.

A consumer may pass information items to a service, which may affect its operation. This is described under *Interfaces* in Chapter 3.

2.4 Effects

2.4.1 Overview

A service has *effects*. These comprise the outcome of the service, and are how it delivers value to its consumers. This section describes the **Effect** class and the **is effect of** property. Two kinds of

effect are described by the ontology: changes and events. The **Change** class and the **is change to property** are described in this section. The **Event** class is described in Section 3.2.5.

2.4.2 The *Effect* Class

```
<owl:Class rdf:ID="Effect">
   <owl:disjointWith rdf:resource="#Service"/>
   <owl:disjointWith rdf:resource="#Actor"/>
</owl:Class>
```

Effects are represented in the ontology by the **Effect** class.

Neither **Change** nor **Event** is a subclass of **Effect** – there can be changes or events that are not effects of anything. Also, the ontology does not preclude the possibility of other kinds of effect. **Effect** is therefore not defined to be disjoint with **Change** and **Event**, but it is defined as being disjoint with **Service** and **Actor**, and it will be defined as being disjoint with other classes as they are introduced.

In the car-wash example, the car-wash service has the *effect* of making the customer's car clean.

2.4.3 The *has effect* and *is effect of* Properties

```
<owl:ObjectProperty rdf:ID="isEffectOf">
   <rdfs:domain rdf:resource="#Effect"/>
   <owl:inverseOf rdf:resource="#hasEffect"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasEffect">
   <rdfs:range rdf:resource="#Effect"/>
   <owl:inverseOf rdf:resource="#isEffectOf"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Service">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasEffect"/>
         <owl:minCardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
           >1</owl:minCardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

The property **has effect** and its inverse **is effect of** capture the relationship between an effect and whatever it is that produces that effect.

Effects can be produced by many things; they are not necessarily produced by services. The ontology does not specify the domain of **has effect** and the range of **is effect of**.

The cardinality restriction on **has effect** states that every service has at least one effect. It is an essential characteristic of services, which this restriction captures, that they have effects that deliver value to their consumers.

2.4.4 The *Change* Class

```
<owl:Class rdf:ID="Change">
   <owl:disjointWith rdf:resource="#Service"/>
   <owl:disjointWith rdf:resource="#Actor"/>
</owl:Class>
```

A *change* to some thing conveys the idea that the thing is different beforehand and afterwards The ontology includes the **Change** class, corresponding to the change concept, but does not include classes or properties corresponding to the notions of "before" and "after". (This is for conciseness, and because the other concepts represented in the ontology do not depend on these notions.)

In the car-wash example, the effect of Joe's car-wash service is that the customer's car is clean. This is a *change* to the car.

The **Effect** and **Change** classes, and the **is change to** property (which is described in the next section), are illustrated in Figure 6.

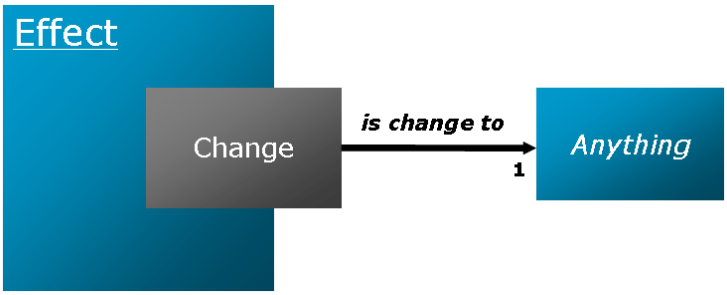

Figure 6: Change

2.4.5 The *is change to* and *is changed by* Properties

```
<owl:ObjectProperty rdf:ID="isChangeTo">
   <rdfs:domain rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isChangedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isChangedBy">
   <rdfs:range rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isChangeTo"/>
</owl:ObjectProperty>
```

```
<owl:Class rdf:about="#Change">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#isChangeTo"/>
         <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

The **is change to** property, and its inverse **is changed by**, capture the relation between a change and the thing that it changes.

In this ontology, as Figure 6 illustrates, a change is a change to exactly one thing. This is captured by the restriction on **Change**. Anything can be the subject of a change.

2.5 Information

2.5.1 Overview

This section describes the basic classes and properties relevant to information. These are the **Information Item**, **Information Type**, and **Description** classes, and the **has type** and **describes** properties.

2.5.2 The *Information Item* Class

```
<owl:Class rdf:ID="InformationItem">
   <owl:disjointWith rdf:resource="#Service"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
</owl:Class>
```

The concept of information is a very general one. An *information item* is a thing that is known about some other thing. This ontology includes the **Information Item** class, whose instances are such pieces of information.

Information Item is disjoint with **Service**, **Actor**, **Effect** and **Change**; an information item may be about a service, an actor, an effect or a change, but it is different from all of these things.

This ontology does not cover the general concept of that information being "about" something, although it does cover the specific cases of information type and description (see sections 2.5.3, 2.5.4, 2.5.5 and 2.5.6 below). Nor does it attempt to differentiate "information" from "data".

An information item may be simple or it may be composed of other information items.

In the car-wash example, "Car Wash" and "$5" are *information items*. "Car Wash $5" is also an *information item*; it is a more complex item that is composed of the two simple ones.

An information item may have one or more *information types*. Figure 7 illustrates the **Information Type** class and the **is type of** property, which are described in the following sections.

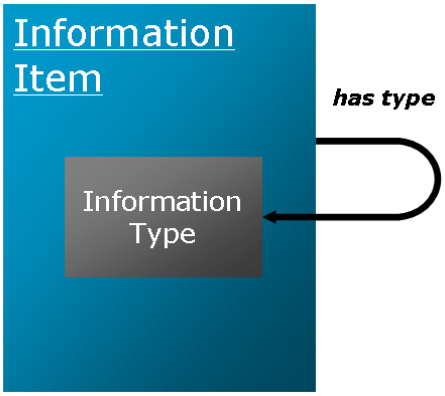

Figure 7: Information Type

2.5.3 The *Information Type* Class

```
<owl:Class rdf:ID="InformationType">
   <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>
```

A *type* is the defining characteristic of a class of things. An *information type* is the defining characteristic of a class of information items. It is thus an information item about another information item – what is often called "meta-information" or "metadata". This concept is represented by the **Information Type** class.

Because an information type is a special kind of information item, **Information Type** is defined as a subclass of **Information Item**. (This implies that it is disjoint with all the classes with which **Information Item** is disjoint.)

In the car-wash example, $5 is the price of the car-wash service. The *information item* "$5" has *information type* "price".

2.5.4 The *has type* and *is type of* Properties

```
<owl:ObjectProperty rdf:ID="hasType">
   <owl:inverseOf rdf:resource="#isTypeOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isTypeOf">
   <owl:inverseOf rdf:resource="#hasType"/>
</owl:ObjectProperty>
```

```
<owl:Class rdf:about="#InformationItem">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasType"/>
         <owl:allValuesFrom rdf:resource="#InformationType"/>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

Where a class has a defining characteristic, each member of the class *has a type* that is that characteristic. The **has type** property and its inverse **is type of** correspond to the relation between a thing and the type (or types) that it has.

In the case of an information item, the type will be an information type. The restriction on the **Information Type** class captures this.

In the car-wash example, "5" *has type* "price".

2.5.5 The *Description* Class

```
<owl:Class rdf:ID="Description">
   <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>
```

A *description* is an information item that is represented in words, possibly accompanied by supporting material such as graphics. The **Description** class corresponds to the concept of a description as a particular kind of information item that applies to something in particular – the thing that it describes. It is not just a set of words that could apply to many things.

Anything can have a description.

A description may be expressed in plain text (perhaps with graphics) or may be in a formal language. OWL may be used as the formal language of a description. An OWL description may use the constructs of this ontology.

In the car-wash example, the sign saying "Car Wash $5" is a *description* of Joe's car-wash service. It is intended to apply to Joe's car-wash service, and not to anything else (even though there may in fact be other car-wash services that charge $5).

The **Description** class and the **describes** property (which is described in the next section) are illustrated in Figure 8.

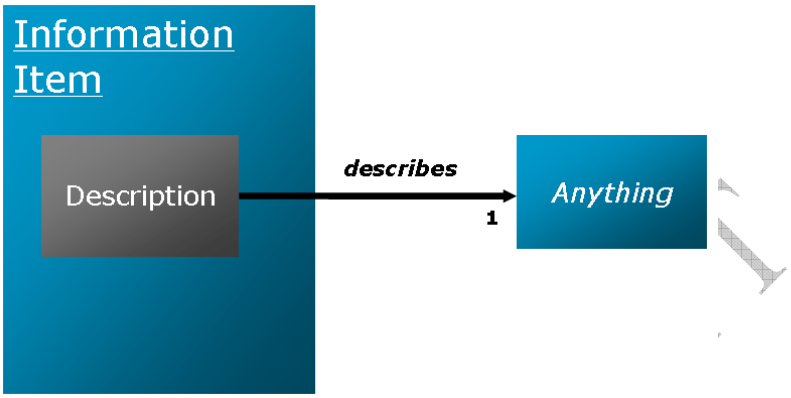

Figure 8: Description

2.5.6 The *describes* and *is described by* Properties

```
<owl:ObjectProperty rdf:ID="describes">
   <rdfs:domain rdf:resource="#Description"/>
   <owl:inverseOf rdf:resource="#isDescribedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isDescribedBy">
   <rdfs:range rdf:resource="#Description"/>
   <owl:inverseOf rdf:resource="#describes"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Description">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#describes"/>
         <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

The **describes** property, and its inverse **is described by**, capture the relation between a description and the thing that it describes.

The restriction on the **Description** class captures the notion that a description describes exactly one thing.

In the car-wash example, the “Car Wash $5” sign *describes* Joe’s car-wash service.

2.6 Systems and Composition

2.6.1 Overview

This section describes the basic classes and properties relevant to systems and composition: the **System** and **Composition** classes, and the **has component** and **produces** properties.

2.6.2 The *System* Class

```
<owl:Class rdf:ID="System">
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
</owl:Class>
```

A *system* is an organized collection of other, simpler things. This concept is captured by the **System** class, which is illustrated in Figure 9.

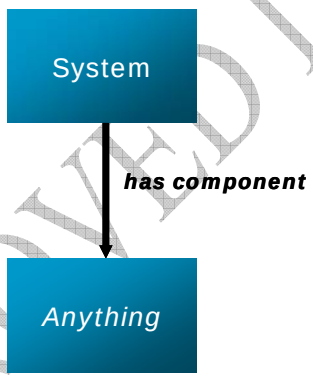

Figure 9: Composition

The **System** class is defined as disjoint with the *Effect*, *Change*, and *Information Item* classes. Instances of these classes are considered not to be collections of other things, although they can be composed of other things (see Section 2.6.4).

2.6.3 The *has component* and *is component of* Properties

```
<owl:ObjectProperty rdf:ID="hasComponent">
   <owl:inverseOf rdf:resource="#isComponentOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isComponentOf">
   <owl:inverseOf rdf:resource="#hasComponent"/>
</owl:ObjectProperty>
```

Each of the things in the organized collection that is a system *is a component of* that system, and the system *has as a component* each of those things. The **has component** property, and its inverse **is component of**, capture the relation between a system and its components.

2.6.4 The *Composition* Class

```
<owl:Class rdf:ID="Composition">
   <rdfs:subClassOf rdf:resource="#System"/>
</owl:Class>
```

A *composition* is a collection of things that are put together to form a single thing. This concept is captured by the **Composition** class.

A composition is a structured collection of things, and is therefore a system, whose components are the things in the collection. The **Composition** class is therefore a subclass of the **System** class.

A particularly important case of composition is *service composition*.

Joe has a friend Muhammad who has a business in which he carries out a complete car-valet service, which is an example of service composition. Muhammad's business process is a little more complicated than Joe's. The customer drives up to Muhammad and ask for his car to be valeted. Muhammad asks him for $50. The customer gives him $50. Muhammad sends his friend Lin to the street corner to fetch Joe. Joe washes the car. Muhammad also sends Lin to fetch two other friends: Masha, who vacuums the car, and Juan, who shampoos the upholstery. Muhammad then says, "That's all done now," and the customer drives away. This is a *composition* of four services: car washing, by Joe; vacuuming, by Masha; shampooing, by Juan; and direction, by Muhammad.

2.6.5 The *produces* and *is produced by* Properties

```
<owl:FunctionalProperty rdf:ID="produces">
   <owl:inverseOf rdf:resource="#isProducedBy"/>
</owl:FunctionalProperty>

<owl:ObjectProperty rdf:ID="isProducedBy">
   <owl:inverseOf rdf:resource="#produces"/>
</owl:ObjectProperty>
```

A composition is a collection of things that are put together to form a single thing. Putting together the things in the composition *produces* the single thing, and that thing *is produced by* the composition. The **produces** property, and its inverse **is produced by**, capture the relation between a composition and the thing that results from combining its components.

A composition results in a unique single thing, and *produces* is a functional property. Note, however, that *is produced by* is not functional. A thing can be formed by composition in more than one way. In particular, a service is a "black box"; how it is produced is transparent to its consumers, and it may be produced in several different ways.

The phrase "*is composed of*" is used as shorthand for "is produced by a composition whose components are". (A formal **is composed of** property is not defined.) .

Muhammad's car-valet service *is produced by* a composition whose components are four other services, as described above. That composition *produces* the car-valet service. As shorthand, we say that "The car-valet service is composed of four other services."

3 Services as Business Activities

3.1 Introduction

Service is originally a business concept. It has been adopted by technologists as a paradigm for the definition of software modules that support business activities – and it is also used for software modules that form part of the enterprise infrastructure and do not directly support its business activities.

This chapter covers the original concept of service as a business activity, and related business concepts. Because these concepts have been adopted by technologists, the definitions in this chapter also apply to service-oriented software implementation. This will be covered in Chapter 4.

The chapter first describes what an activity is, and introduces concepts of actors participating in activities, and of events to which activities respond. It then looks at what distinguishes business activities from other activities. An important characteristic of services as distinct from other activities is that they have clearly-understood interfaces; the chapter describes concepts relating to interfaces and to their associated events and information exchanges. Finally, the chapter describes the important business concepts related to the ideas of contract and policy.

3.2 Activities

3.2.1 Overview

A service is a kind of activity. Services are distinguished from activities in general because a service has:

- a simple, well-defined interface, so that it is a self-contained "black box", and can be used as a unit from which other services or activities are composed;
- a specified outcome;
- a provider that is responsible for its performance; and
- consumers that use it.

However, many of the properties of the **Service** class are inherited from its parent **Activity** class.

The concept of *activity* described in this ontology is a very broad one – it is that of a system of actions that are performed by actors in response to events. It is not distinguished from "task" on the one hand, or "process" on the other. It includes simple activities consisting of single actions, and complex activities composed of many actions and simpler activities.

This section describes the **Activity** and **Event** classes and the **responds to** property that connects them, and the **is component of**, **takes part in**, and **is occurrence of** properties that capture relations between the **Activity** and **Event** classes and the **Actor**, **Action**, and **Change** classes described in Chapter 2.

3.2.2 Example

Joe's friend Muhammad is an amateur boxer in his spare time. Boxing is an activity in which Muhammad takes part. (Joe takes part also; he sits in Muhammad's corner, with his sponge.) It consists of actions performed in response to events. For example, when the bell rings at the start of the fight, Muhammad responds by leaving his corner. (Following which, the actions and events get very physical!)

Boxing is not a service: it does not have a well-defined interface; it does not have a specified outcome; it does not have a provider; it does not have consumers. There are activities related to boxing that are services. For example, putting on a boxing match is a service: there is a provider (the promoter), consumers (the paying customers), a well-defined interface (the customer buys a ticket and gets to watch), and a specified outcome (the boxer that does best according to the rules is declared the winner). But the activity of boxing in general is not a service.

3.2.3 The *Activity* Class

```
<owl:Class rdf:ID="Activity">
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#Actor"/>
</owl:Class>
```

The **Activity** class corresponds to the concept of activity described above.

Activities are distinct from effects, changes, information items, and actors. The **Activity** class is disjoint from the **Effect**, **Change**, **Information Item** and **Actor** classes.

It was stated in Chapter 2 that a service can have effects. Any activity can have effects. If an effect is an effect of an action, and the action is a component of an Activity, then the effect is also an effect of the Activity

Joe's car-wash service, described in Chapter 2, is one example of an activity. Boxing, as described above, is another example.

An activity represents a particular, described, pattern of behavior, such as "car wash", not an instance, such as "the washing of my car yesterday". Different patterns of behavior can be different activities or the same activity, at the discretion of whoever is populating the ontology. For example, "car wash" could include both "with wheel-scrub" and "without wheel-scrub" behavior patterns, or "car wash with wheel-scrub" and "car wash without wheel-scrub"could be separate activities – perhaps instances of a **Car Wash** subclass of **Activity**.

An activity includes actions and responds to events. The actions are performed by actors, and these actors take part in the activity. An action has effects (see Chapter 2), and a change can be one kind of effect. Another kind of effect can be an event (to which this or, more probably,

another activity might respond). One kind of event is the occurrence of a change. The relevant classes and properties are illustrated in Figure 10. Those that have not been covered already are described in the following subsections.

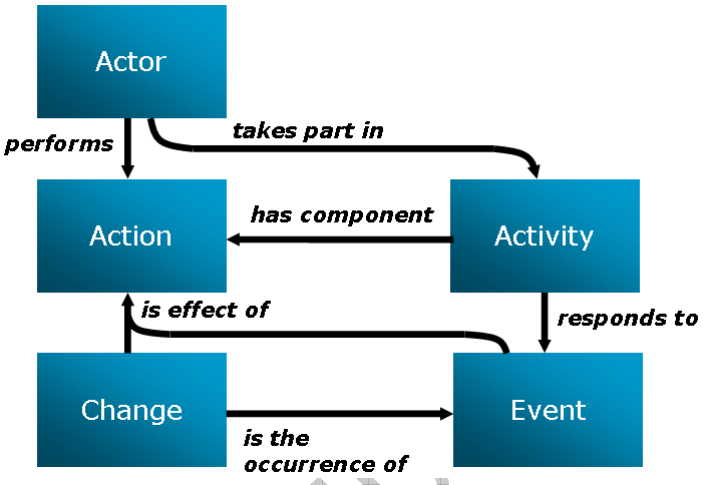

Figure 10: Activity

Note that the **has component** property, defined in section 2.6.5, captures the relation between an activity and an action of that activity.

Joe's car-wash activity has a number of component actions. Joe asks the customer for $5. He washes the car. He tells the customer that the job is finished. Washing the car might, at the discretion of whoever is populating the ontology, be regarded as a single action or as a collection of actions such as filling the bucket with water, adding detergent, sponging the car, re-filling the bucket with clean water, and rinsing the car.

3.2.4 The *takes part in* and *has participant* Properties

```
<owl:ObjectProperty rdf:ID="takesPartIn">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#hasParticipant"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasParticipant">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#takesPartIn"/>
</owl:ObjectProperty>
```

Actions are performed by actors. The **has participant** property and its inverse **takes part in** capture the relation between an activity and an actor that performs an action that is a component of that activity.

Muhammad *takes part in* the activity of boxing. Joe *takes part in* that activity, and also in his car-wash activity.

3.2.5 The *Event* Class

```
<owl:Class rdf:ID="Event">
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Activity"/>
</owl:Class>
```

An *event* is something that happens, to which an activity may respond.

An event is not necessarily associated with any particular activity or activities.

As described in 3.2.7 below, the occurrence of a change may be an event, but the concept of change is distinct from that of event, and the **Event** and **Change** classes are disjoint. **Event** is also disjoint from **Actor**, **Information Item**, **System**, and **Activity** (and therefore from **Service**). However, an event can be an effect of an action; the **Event** and **Effect** classes are not disjoint.

In the car-wash example, the arrival of a customer who asks Joe to wash his car is an event. Other events in that example are the customer giving Joe the money, and Joe saying "That's all done, now."

3.2.6 The *responds to* and *is responded to by* Properties

```
<owl:ObjectProperty rdf:ID="respondsTo">
   <rdfs:domain>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Activity"/>
            <owl:Class rdf:about="#Actor"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:domain>
   <rdfs:range rdf:resource="#Event"/>
   <owl:inverseOf rdf:resource="#isRespondedToBy"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="isRespondedToBy">
   <rdfs:domain rdf:resource="#Event"/>
   <rdfs:range>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Activity"/>
            <owl:Class rdf:about="#Actor"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:range>
   <owl:inverseOf rdf:resource="#respondsTo"/>
</owl:ObjectProperty>
```

When an actor taking part in an activity takes action because an event occurs, the actor and the activity *respond to* the event. The **responds to** property and its inverse **is responded to by** capture the relation between an activity and the events to which it responds.

When an activity responds to an event, it is a particular actor taking part in the activity that responds. This is captured by including **Actor**, as well as **Activity**, in the domain of **responds to** and the range of **is responded to by**.

More than one activity (or actor) can respond to the same event, and an activity (or actor) can respond to more than one event.

In the car-wash example, Joe *responds to* the customer arrival event by asking for $5. This is another event, to which the customer responds by giving Joe $5. Joe responds to this by performing the actions necessary to wash the car. He then says "That's all done, now," which is another event to which the customer responds by driving away. The car-wash activity, as well as the actor Joe, is said to *respond to* the arrival and payment events.

3.2.7 The *is the occurrence of* and *occurs as* Properties

```
<owl:ObjectProperty rdf:ID="isTheOccurrenceOf">
   <rdfs:domain rdf:resource="#Change"/>
   <rdfs:range rdf:resource="#Event"/>
   <owl:inverseOf rdf:resource="#occursAs"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="occursAs">
   <rdfs:domain rdf:resource="#Event"/>
   <rdfs:range rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isTheOccurrenceOf"/>
</owl:ObjectProperty>
```

A change to something *occurs at* a point or over a period of time. There are many activities in which action is taken when a change occurs. For example, in a process control activity, a process parameter may be monitored, and corrective action may be taken when it exceeds a limit – that is, when a change in the parameter value occurs. In this ontology, a change is not an event, but the occurrence of a change can be an event. This concept is captured by the **is the occurrence of** property and its inverse **occurs as**.

In the car-wash example, Joe stops work at 5 o'clock sharp. The event that prompts him to finish *is the occurrence of* a change in the time of day.

3.3 Business Activities

3.3.1 Overview

Service is a business concept that has been adopted as a technical paradigm. But what is a business service or, more generally, a business activity? What is it that distinguishes such activities from technical activities, or activities of other kinds?

There is no fundamental characteristic that distinguishes a business activity. An activity that is a business activity for one person or organization may not be a business activity for others. A business activity for a technical services provider may be a technical activity for its customers.

Because of this, the ontology does not include a class corresponding to *business activity*, but it includes the **is a business activity of** property corresponding to the relation between an actor and an activity that is a business activity for that actor.

3.3.2 Example

Joe's car-wash service *is a business activity of* Joe. Boxing was a business activity of World champion Muhammad Ali. However, Joe's friend Muhammad takes part in boxing, but this is not a business activity as far as he is concerned, it is a leisure activity.

3.3.3 The *is a business activity of* and *has business activity* Properties

```
<owl:ObjectProperty rdf:ID="isBusinessActivityOf">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#hasBusinessActivity"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasBusinessActivity">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#isBusinessActivityOf"/>
</owl:ObjectProperty>
```

The **is a business activity of** property illustrated in Figure 11, and its inverse **has business activity**, capture the relation between an actor and an activity in that the actor provides or takes part in and that is related to the actor's business mission.

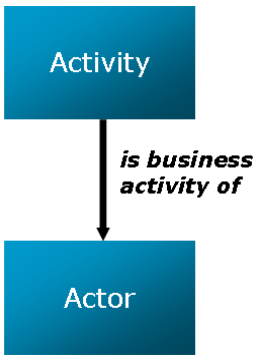

Figure 11: Business Activity

3.4 Interfaces

3.4.1 Overview

An important characteristic of services is that they have simple, well-defined interfaces. This makes it easy to interact with them, and enables other activities to use them as components.

This is a principle that applies to the business concept of service, and translates naturally to the technical service paradigm. As software can support any degree of complexity, SOA puts an emphasis on interface simplicity, with the insistence that services must be “loosely-coupled”.

This section describes the **Interface** class, the **is interface of** property that relates it to activities, the **is event at** property that relates it to events, and two properties that relate it to exchanged information, **is input at** and **is output at**.

3.4.2 Example

The interface to Joe’s car-wash service is simple and informal. The customer asks Joe to wash the car. There may be an exchange of additional information, such as the customer asking whether this includes scrubbing the wheels, and Joe saying, “That’s $2 extra.” The customer then gives Joe the money.

With an automated car wash, the interface is typically a little more formal. The customer selects from a set range of differently-priced options, and purchases a token, which is then inserted into the car-wash machine.

Some non-technical interfaces can be very formal. When Joe applies to the city for a permit to trade on the public highway, he has to visit the city offices and fill in some very extensive and complicated forms.

3.4.3 The *Interface* Class

```
<owl:Class rdf:ID="Interface">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
</owl:Class>
```

An *interface* is a protocol by which actors can interact with and exchange information with an activity. This concept is captured by the **Interface** class of the ontology. An activity may have any number of interfaces.

This concept is distinct from all those described earlier, and the **Interface** class is therefore disjoint with all classes defined earlier.

An interface can be composed of other interfaces. The initiation interface to Joe's car-wash service can be considered as being composed of two simpler interfaces: request for service, and payment.

As illustrated in Figure 12, an interface of an activity enables actors to interact with the activity by means of events, and to input information to the activity and receive information that is output by the activity. The classes and properties related to these concepts are described in the following subsections.

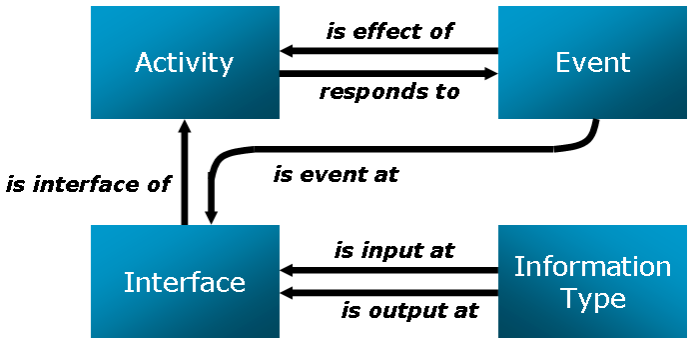

Figure 12: Interface

3.4.4 The *is interface of* and *has interface* Properties

```
<owl:ObjectProperty rdf:ID="isInterfaceOf">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#hasInterface"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInterface">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#isInterfaceOf"/>
</owl:ObjectProperty>
```

The **is interface of** property and its inverse **has interface** capture the relation between an interface and the activity that it enables actors to interact with.

3.4.5 The *is event at* and *includes event* Properties

```
<owl:ObjectProperty rdf:ID="isEventAt">
   <rdfs:domain rdf:resource="#Event"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#includesEvent"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="includesEvent">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#Event"/>
   <owl:inverseOf rdf:resource="#isEventAt"/>
</owl:ObjectProperty>
```

An interface of an activity has associated with it (but does not consist of) a set of events which the activity generates or to which it responds. The relation between an interface and such events is captured by the **is event at** property and its inverse **includes event**.

In the car-wash example, the arrival of a customer who asks Joe to wash his car *is an event at* the initiation interface, as is the customer giving Joe the money.

3.4.6 The *is input at*, *has input information*, *is output at*, and *has output information* Properties

```
<owl:ObjectProperty rdf:ID="isInputAt">
   <rdfs:domain rdf:resource="#InformationType"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#hasInputInformation"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="hasInputInformation">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#InformationType"/>
   <owl:inverseOf rdf:resource="#isInputAt"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isOutputAt">
   <rdfs:domain rdf:resource="#InformationType"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#hasOutputInformation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutputInformation">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#InformationType"/>
   <owl:inverseOf rdf:resource="#isOutputAt"/>
</owl:ObjectProperty>
```

An interface can enable an actor to give information to or receive information from an activity. The relation between an interface and the types of information that an actor can give the activity via that interface is captured by the **is input at** property and its inverse **has input information**. The relation between an interface and the types of information that an actor can receive from the activity via that interface is captured by the **is output at** property and its inverse **has output information**.

Note that the range of **is input at** – and of **is output at** – is **Information Type** rather than **Information Item**. The values of these properties are types of information, not specific information items.

In the car-wash example, when the customer asks Joe whether wheel scrub is included, and Joe tells him that it is included for an extra $2, the $2 is price information that *is output at* the initiation interface. When the customer says that he would like wheel scrub, this is option request information that *is input at* the interface.

3.5 Contracts and Policies

3.5.1 Overview

Contract and *policy* are two important business concepts that are part of the SOA technical paradigm. This section describes the constructs of the ontology that capture those concepts. These are the **Contract**, **Policy** and **Rule** classes, and the **applies to**, **has condition**, **is contract for**, **is party to**, **has policy**, and **is policy for** properties.

Following the OASIS SOA reference model [OASIS RM], this ontology makes the distinction that a *contract* is agreed between two or more parties, while a *policy* is owned by a single party.

Contracts and policies have conditions. These are rules. A rule can apply to anything. The corresponding classes and properties are illustrated in Figure 13.

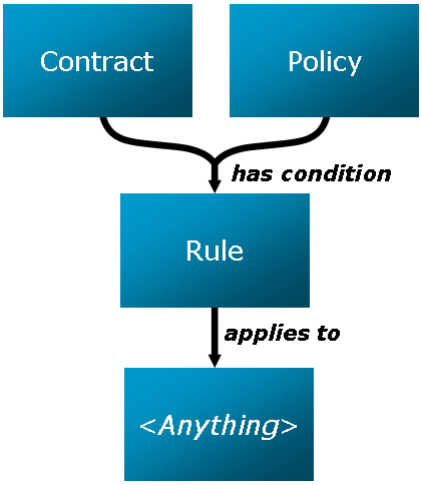

Figure 13: Contract and Policy

In particular, a rule can apply to a service. The cases where a contract or policy has conditions that apply to a service are of particular interest to this ontology.

3.5.2 Example

When a customer asks Joe to clean his car, and Joe takes his money, there is a *contract* made between Joe and the customer. In this case, the contract is unwritten and implicit. In other cases, a contract may be explicitly recognized as such, and captured in writing. For example, when Joe bought his house, he and the seller signed a written contract that was drawn up by lawyers.

Joe has a cash up-front *policy*. He does not give credit. He will not clean a car until the customer has paid him the $5. This is not something that he agrees with his customers, it is simply a course of action that he always follows when performing his service.

A contract for the sale of a house can be quite complex. For example, it may stipulate that[1]:

- The buyer pays 10% of the price to a neutral deposit holder when the contract is signed;
- The buyer pays the remaining 90% to the seller and the stakeholder gives the 10% to the seller on the agreed sale date provided that the seller then gives the house to the buyer;
- If the buyer does not give the remaining 90% to the seller on the agreed date then the deposit holder gives the 10% to the seller and the seller does not have to give the house to the buyer;
- If the seller does not give the house to the buyer on the agreed date then the deposit holder gives the 10% back to the buyer and the seller pays a penalty charge to the buyer.

[1] This is a simplified description of the procedure that is normally followed in England. In other countries, sale contracts will typically have different conditions.

These are rules that apply to the sale of the house and are conditions of the contract.

Joe's cash up-front policy has a single condition. This is the rule that the customer pays the money before Joe washes the car.

3.5.3 The *Rule* Class

```
<owl:Class rdf:ID="Rule">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
</owl:Class>
```

A *rule* is a statement describing how something should be. This concept is captured by the **Rule** class.

A rule may specify what should be done, for example, "Drive on the right," or what should not be done, for example, "No smoking."

This concept is distinct from all those described earlier, and the **Rule** class is therefore disjoint with all classes defined earlier.

In particular, it is disjoint with the **Information Item** class. A rule states what should be, not what is. Rules are sometimes broken.

3.5.4 The *applies to* and *is affected by* Properties

```
<owl:ObjectProperty rdf:ID="appliesTo">
   <rdfs:domain rdf:resource="#Rule"/>
   <owl:inverseOf rdf:resource="#isAffectedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isAffectedBy">
   <rdfs:range rdf:resource="#Rule"/>
   <owl:inverseOf rdf:resource="#appliesTo"/>
</owl:ObjectProperty>
```

A rule is a statement of how something should be. The **applies to** property, and its inverse **is affected by**, capture the relation between the rule and that something.

A rule can apply to anything. In particular, it can apply to:

- A service; or
- An effect of a service.

Joe's rule that his customers must give him the money before he cleans their cars *applies to* his car-wash service.

The implicit contract between Joe and his customers includes the condition that the car is clean when Joe has washed it. This is a rule that *applies to* an effect of Joe's service.

3.5.5 The *has condition* and *is condition of* Properties

```
<owl:ObjectProperty rdf:ID="hasCondition">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Contract"/>
        <owl:Class rdf:about="#Policy"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Rule"/>
  <owl:inverseOf rdf:resource="#isConditionOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isConditionOf">
  <rdfs:domain rdf:resource="#Rule"/>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Contract"/>
        <owl:Class rdf:about="#Policy"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
  <owl:inverseOf rdf:resource="#hasCondition"/>
</owl:ObjectProperty>
```

The statements of how things should be according to a contract or policy are the *conditions* of that contract or policy. The **has condition** property, and its inverse **is condition of**, capture the relation between a contract or property and a rule that is a condition of it.

The domain of the **has condition** property, and the range of the **is condition of** property, is the union of the **Contract** and **Policy** classes. These classes are defined in Sections 3.5.6 and 3.5.9.

"The buyer pays 10% of the price to a neutral deposit holder when the contract is signed" is a rule that is a condition of a contract for the sale of a house. "The customer pays the money before I wash the car" is rule that is a condition of Joe's cash up-front policy, and also of the implicit contract that he has with his customers.

3.5.6 The *Contract* Class

```
<owl:Class rdf:ID="Contract">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
</owl:Class>
```

A *contract* is an agreement between two or more actors - the *parties* to the contract. The term is most commonly used for a written agreement, or one that is enforceable at law, but it can be applied more widely. The **Contract** class captures this concept.

This concept is distinct from all those described earlier, and the **Contract** class is therefore disjoint with all classes defined earlier.

In the car-wash example, there is an implicit *contract* between Joe and each customer that Joe will clean the customer's car in exchange for $5.

This is an example of a situation of particular interest for this ontology, that where there is a contract between a service provider and its consumers for the performance of that service. The relevant classes and properties are illustrated in Figure 14. The **is contract for** and **is party to** properties are described in sections 3.5.7 and 3.5.8.

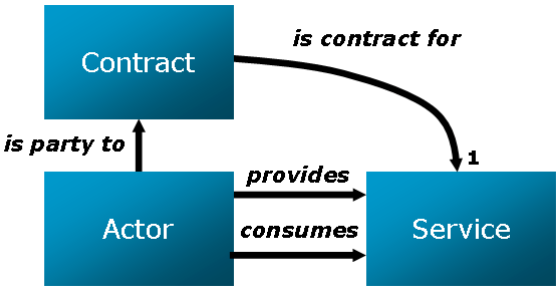

Figure 14: Contract for a Service

3.5.7 The *is contract for* and *is subject of contract* Properties

```
<owl:ObjectProperty rdf:ID="isContractFor">
  <rdfs:domain rdf:resource="#Contract"/>
  <owl:inverseOf rdf:resource="#isSubjectOfContract"/>
</owl:ObjectProperty>

<owl:FunctionalProperty rdf:about="#isContractFor" />
```

```
<owl:ObjectProperty rdf:ID="isSubjectOfContract">
   <rdfs:range rdf:resource="#Contract"/>
   <owl:inverseOf rdf:resource="#isContractFor"/>
</owl:ObjectProperty>
```

A contract is generally an agreement about some particular thing – its *subject*. This concept is captured by the **is contract for** property and its inverse **is subject of**.

In this ontology, a contract has only one subject. This is captured by **is contract for** being a functional property.

When Joe buys his house, the contract between him and the seller *is a contract for* the sale of the house.

The particular case of interest for this ontology is that of a contract between the provider and consumers of a service for the performance of that service. The implicit contract between Joe and his customers for the car-wash service falls into this category.

3.5.8 The *is party to* and *has party* Properties

```
<owl:ObjectProperty rdf:ID="isPartyTo">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Contract"/>
   <owl:inverseOf rdf:resource="#hasParty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasParty">
   <rdfs:domain rdf:resource="#Contract"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isPartyTo"/>
</owl:ObjectProperty>
```

The relation between an actor and a contract to which that actor is a party is captured by the **is party to** property and its inverse **has party**.

Each of Joe's customers *is party to* the contract for Joe's car-wash service, and so is Joe himself. Although there is no written agreement, once Joe has accepted the $5, he is obliged to clean the car, and the customer can take legal action for breach of contract if he does not do so.

3.5.9 The *Policy* Class

```
<owl:Class rdf:ID="Policy">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
</owl:Class>
```

A *policy* is a course of action that an actor may intend to follow or may intend that another actor should follow. This concept is captured by the **Policy** class.

This concept is distinct from all those described earlier, and the **Policy** class is therefore disjoint with all classes defined earlier.

Joe's cash up-front policy consists of waiting until he has been paid $5 before washing a customer's car. This is a course of action that he intends to follow. When he goes to city hall to collect his street trading permit, he is constrained by the city's "no smoking" policy: the city authority intends that Joe, and other visitors to city hall, should follow the course of action of not smoking while on their premises.

A situation of particular interest for this ontology is that where a service provider has a policy for the service. The relevant classes and properties are illustrated in Figure 15. The **is policy of** and **is policy for** properties are described in sections 3.5.10 and 3.5.11.

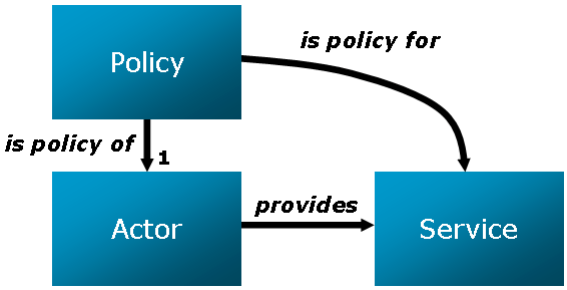

Figure 15: Policy

3.5.10 The *has policy* and *is policy of* Properties

```
<owl:ObjectProperty rdf:ID="hasPolicy">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Policy"/>
   <owl:inverseOf rdf:resource="#isPolicyOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isPolicyOf">
  <rdfs:domain rdf:resource="#Policy"/>
  <rdfs:range rdf:resource="#Actor"/>
  <owl:inverseOf rdf:resource="#hasPolicy"/>
</owl:ObjectProperty>

<owl:FunctionalProperty rdf:about="#isPolicyOf" />
```

The **has policy** property, and its inverse **is policy of**, capture the relation between an actor and a policy that the actor intends should be followed.

Joe *has policy* "cash up-front", and the city authority *has policy* "no smoking".

In this ontology, a policy is the policy of a single actor; if different actors have similar policies, then they are regarded as distinct policies. (So, if Squaresville and Plainsville both have no-smoking policies, they are regarded as separate policies – the Squaresville no-smoking policy, and the Plainsville no-smoking policy.) This is captured by **is policy of** being a functional property.

A particular case of interest for this ontology is that where the conditions of a policy apply to a service or other activity. This is described in Section 3.5.11.

The conditions of a policy can apply to things other than activities. Another case of interest is that where a policy has conditions that apply to a contract. For example, a company may have an equal pay policy whose conditions apply to the contracts that it has with its employees.

The implicit contract for Joe's car-wash service is affected by the rule that is the condition of his cash up-front policy. If Joe were to offer a written contract for his service, it might include terms such as "The customer shall pay all money due before washing of the car commences."

3.5.11 The *is policy for* and *is subject of policy* Properties

```
<owl:ObjectProperty rdf:ID="isPolicyFor">
   <rdfs:domain rdf:resource="#Policy"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#isSubjectOfPolicy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isSubjectOfPolicy">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Policy"/>
   <owl:inverseOf rdf:resource="#isPolicyFor"/>
</owl:ObjectProperty>
```

A policy that has conditions that apply to a service or other activity implies that the activity is performed in a particular way. The **is policy for** property and its inverse **is subject of policy** capture the relationship between such a policy and activity.

Joe's cash up-front policy *is a policy for* his car-wash service. The city no-smoking policy *is a policy for* all activities that take place in city hall.

While a particular case of interest for this ontology is that where the provider of a service has a policy for the service, a policy for a service is not necessarily owned by the provider of that service, or by a consumer. For example, government food and hygiene regulations (a policy that is law) cover restaurant services. In an enterprise, corporate policy may cover services provided by divisions or departments.

4 Design and Implementation

4.1 Introduction

Service is originally a business concept, and Chapter 3 describes services as business activities. But the concept of service has been adopted in the world of information technology, resulting in the software services that are the fundamental components of service-oriented solutions.

This chapter describes concepts related to how services and service-based systems are built, and defines the corresponding classes and properties of the ontology.

It does this by first describing some general concepts related to design and implementation, then describing the kinds of actor that take part in services, and finally describing some specific ways in which the concepts apply to the design and implementation of services in a service-oriented architecture.

The general concepts build on the basic service concepts and the concepts of services as business activities that were introduced in previous chapters. They do not assume an information technology context. They relate to:

- Requirements and solutions;
- Abstraction and Realization;
- Design; and
- Implementation.

The actors that take part in services in a service-oriented architecture include human actors, technology actors, and organization actors. It is at the point where the concept of technology actor is described that ideas that specifically relate to information technology and software are introduced.

The third part of the chapter generally assumes an information technology context, although some of its concepts are more general. The specific service design and implementation patterns that are described are:

- Software Services;
- Choreography; and Orchestration;
- Messaging;
- Discovery; and
- Virtualization.

4.2 Requirements and Solutions

4.2.1 Overview

The idea of designing solutions to satisfy requirements is fundamental to the creation of services to meet the business needs of an enterprise. It is an important part of architecture although, as will be seen in Chapter 5, architecture is more than just design.

This section describes classes and properties related to requirements and solutions. These are the **Requirement**, **Solution**, and **Solution Building Block** classes, and the **requires**, **satisfies**, and **entails** properties. They are illustrated in Figure 16.

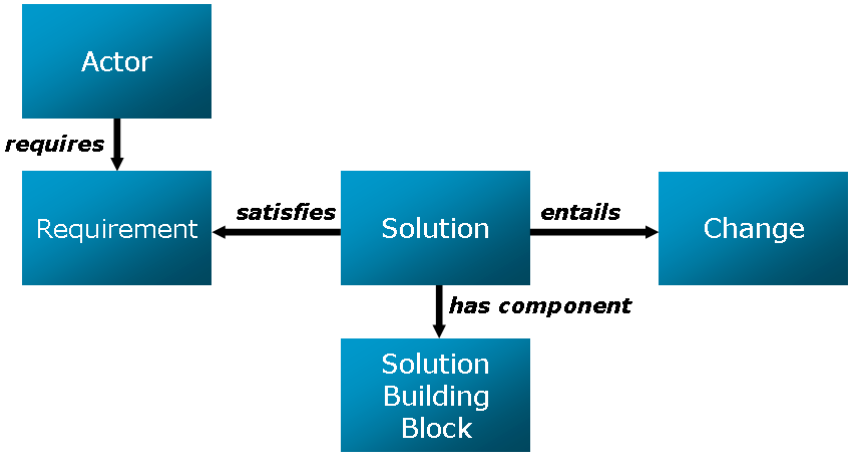

Figure 16: Requirements and Solutions

4.2.2 The *Requirement* Class

```
<owl:Class rdf:ID="Requirement">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
   <owl:disjointWith rdf:resource="#Policy"/>
</owl:Class>
```

A *requirement* is a desire that an actor has for something to have some particular characteristic or characteristics. This concept is captured by the **Requirement** class.

A requirement may be simple, or it may be complex, composed of other requirements.

A requirement is different from all the concepts that have been introduced so far, and the **Requirement** class is disjoint with the classes that correspond to those concepts. (Note that, although **Requirement** is disjoint with **System**, a requirement can be produced from a composition that is a system of simpler requirements.)

Joe has a *requirement.* He is tired of washing cars all day. It makes his arms ache. He wants something – his way of earning a living – to have some particular characteristic – it should be less tiring.

4.2.3 The *requires* and *is required by* Properties

```
<owl:ObjectProperty rdf:ID="requires">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Requirement"/>
   <owl:inverseOf rdf:resource="#isRequiredBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRequiredBy">
   <rdfs:domain rdf:resource="#Requirement"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#requires"/>
</owl:ObjectProperty>
```

The **requires** property, and its inverse **is required by**, capture the relation between the actor that has the desire for something to have some particular characteristic or characteristics and the requirement that is that desire.

Joe *requires* a way of making a living that involves less physical effort than washing cars by hand.

4.2.4 The *Solution* Class

```
<owl:Class rdf:ID="Solution">
</owl:Class>
```

A requirement is a desire for something to have particular characteristics. A *solution* is something that would give that thing those characteristics. Such a solution *satisfies* the requirement

Anything can satisfy a requirement of some kind. The **Solution** class is not disjoint with any of the other classes defined in this ontology.

Joe can think of two *solutions* that would satisfy his requirement for an easier way of making a living. One is to persuade his nephew Billy to do all the hard work of washing the cars, while he talks to the customers and takes the money. The other is to buy an automatic car-wash machine. Either of these solutions would mean he was less tired at the end of the day. They have different advantages and disadvantages. Persuading Billy would not cost anything, but it would only be a

short-term solution, as his nephew would soon realize that he could do better to set up on his own. Joe feels that the car-wash machine is the better long-term solution.

4.2.5 The *satisfies* and *is satisfied by* Properties

```
<owl:ObjectProperty rdf:ID="satisfies">
   <rdfs:domain rdf:resource="#Solution"/>
   <rdfs:range rdf:resource="#Requirement"/>
   <owl:inverseOf rdf:resource="#isSatisfiedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isSatisfiedBy">
   <rdfs:domain rdf:resource="#Requirement"/>
   <rdfs:range rdf:resource="#Solution"/>
   <owl:inverseOf rdf:resource="#satisfies"/>
</owl:ObjectProperty>
```

The **satisfies** property, and its inverse **is satisfied by**, capture the relation between a requirement and a solution that satisfies that requirement.

An automatic car wash is a solution that *satisfies* Joe's requirement for a less tiring way of making a living.

4.2.6 The *entails* and *is entailed by* Properties

```
<owl:ObjectProperty rdf:ID="entails">
   <rdfs:domain rdf:resource="#Solution"/>
   <rdfs:range rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isEntailedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isEntailedBy">
   <rdfs:domain rdf:resource="#Change"/>
   <rdfs:range rdf:resource="#Solution"/>
   <owl:inverseOf rdf:resource="#entails"/>
</owl:ObjectProperty>
```

Putting a solution in place generally means changing something, and often means changing a number of things: it *entails* changes to those things. The relation between solutions and changes that are made to put them in place is captured by the **entails** property and its inverse **is entailed by**.

Joe's automatic car-wash solution *entails* a change to his car-wash service, in which a car-wash machine replaces him as the car washer.

4.2.7 The *Solution Building Block* Class

```
<owl:Class rdf:ID="SolutionBuildingBlock">
  <rdfs:subClassOf rdf:resource="#Abstraction" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isComponentOf" />
      <owl:someValuesFrom rdf:resource="#Solution" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

A *solution building block* is an abstraction that is a component of a solution. There are many cases where a solution that satisfies a requirement or set of requirements is designed as a system. The components of that system are solution building blocks. Those building blocks, and the relations between them, are refined as the design proceeds.

This concept is captured by the **Solution Building Block** class, which is defined as a subclass of the **Abstraction** class, with a restriction on the **is component of** property that ensures that each solution building block is a component of some solution. (The **Abstraction** class is defined in Section 4.3.2.)

Almost anything can be a solution building block, including activities, actors, effects, changes, information items, compositions, events, interfaces, contracts, and policies. The **Solution Building Block** class is not defined as disjoint with any other class of this ontology.

In particular, a solution building block can have component solution building blocks; the **Solution Building Block** class is not disjoint with the **System** class.

A solution building block generally has to satisfy requirements that are derived from requirements for the overall solution. A solution building block is thus generally also a solution. The **Solution Building Block** and **Solution** classes are not disjoint.

4.3 Abstraction and Realization

4.3.1 Overview

This section describes concepts associated with the implementation of an abstract idea as a real thing. They are represented by the **Abstraction** and **Realization** classes, and the **is abstraction of** and **is realization of** properties. These classes andproperties are illustrated in Figure 17.

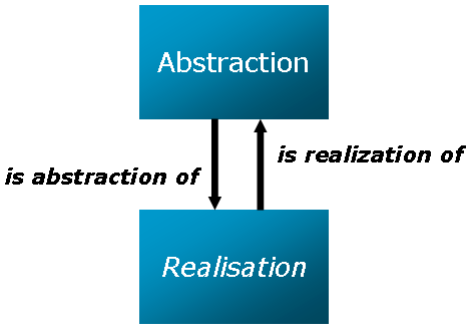

Figure 17: Abstraction and Realization

An *abstraction* is an idea of a class of things. Those things are *realizations* of the abstraction. The central idea captured in this ontology is that of something abstract being realized. The ontology does not divide the world into two categories: real and abstract. It describes abstractions that can be realized, and things that are realizations of those abstractions. But it also allows for the possibility that there are things that are not abstractions and that are not realizations of abstractions.

The ontology does not allow for abstractions of abstractions; such things may be possible, but they are not of sufficient practical use to be included.

The instances of many of the other concepts of this ontology can be abstractions or realizations. In particular, an actor can be an abstraction. The kind of actor that is a role or class used in modeling is an abstraction. Such an actor can be realized by an actor that is a real person, organization, or piece of technology.

An activity, and in particular a service, can also be an abstraction or a realization. Where the actors that take part in a service are abstractions, the service is also an abstraction. A service in which realizations of the abstract actors take part would be a realization of the abstract service.

4.3.2 The *Abstraction* Class

```
<owl:Class rdf:ID="Abstraction">
   <owl:disjointWith rdf:resource="#Requirement"/>
</owl:Class>
```

The concept of an abstraction is captured by the **Abstraction** OWL class. Note that the instances of **Abstraction** are ideas, not the classes that those ideas represent: making the instances of **Abstraction** classes would require OWL-FULL

Although *requirement* is an abstract concept, a requirement is not an abstraction of anything, and the **Requirement** and **Abstraction** classes are disjoint.

Instances of any of the other classes introduced so far can be abstractions. The **Abstraction** class is not defined as being disjoint with any of these classes.

When Joe thinks of his car-wash service in the abstract, the service itself, the car washer and the customer are all *abstractions*.

4.3.3 The *Realization* Class

```
<owl:Class rdf:ID="Realization">
   <owl:disjointWith rdf:resource="#Abstraction"/>
   <owl:disjointWith rdf:resource="#Requirement"/>
</owl:Class>
```

The **Realization** class captures the concept of something that is a realization of an abstraction.

This class is disjoint with **Abstraction** (the ontology does not include the idea of an abstraction of an abstraction).

It is also disjoint with the **Requirement** class: just as a requirement is not an abstraction of anything, neither is it a realization of anything.

Instances of any of the other classes introduced so far can be realizations, however, and the **Realization** class is not defined as being disjoint with any of these classes.

When Joe thinks of his actual car-wash service, the service, Joe himself, and the people who are his customers are all *realizations*.

4.3.4 The *is abstraction of* and *is realization of* Properties

```
<owl:ObjectProperty rdf:ID="isAbstractionOf">
   <rdfs:domain rdf:resource="#Abstraction"/>
   <rdfs:range rdf:resource="#Realization"/>
   <owl:inverseOf rdf:resource="#isRealizationOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRealizationOf">
   <rdfs:domain rdf:resource="#Realization"/>
   <rdfs:range rdf:resource="#Abstraction"/>
   <owl:inverseOf rdf:resource="#isAbstractionOf"/>
</owl:ObjectProperty>
```

The **is realization of** property, and its inverse **is abstraction of**, capture the relation between a real thing and an idea that is an abstraction of it.

Joe *is a realization of* the car-washer abstraction, and car washer *is an abstraction of* Joe. The car-wash machine that Joe wishes to buy is also *a realization of* the car-washer abstraction, and car washer is also *an abstraction of* this machine.

4.4 Design

4.4.1 Overview

Design is the process of creating an abstract solution, and such a solution is called “a design”. These concepts are captured by the **Design** and **Design Activity** classes that are described in this section.

4.4.2 Example

Joe’s uncle dies and leaves him a legacy, which will enable him to realize his dream of owning an automatic car wash. But Joe does not go out and buy the first vacant lot he sees and order the first car-wash machine in the catalog. If he did this, he might find that the machine would not fit on the lot, or that he could not give it water and power supplies, or that there were many other kinds of problem. Joe sets out to design his car-wash system.

“Automatic car wash” may be a solution that satisfies his initial statement of requirements but, as he thinks about it, he finds that he has further, more detailed, requirements for that solution.

Joe decides that his solution must have a number of components, including the car-wash machine itself, and the site to put it on. These components are his solution building blocks. His requirements for the solution imply requirements for each of the building blocks.

Joe starts by writing down descriptions of these requirements. Then he thinks about them and refines them. His first description of the requirement for a site for his car wash might simply be “place to put the car-wash machine”. He might add, on further thought, “must have easy street access”, “must have electricity and water supplies available”, “must be at least 50 feet square”, and so on. Similarly, he might add to the description of the requirement for the car-wash machine clauses about the features that he wants; “must do wheel-scrub”, and so on. He might also draw site plans, showing such things as the entry and exit points, the on-site roads, the car-wash machine, and a kiosk for an operator.

At the end of the design process, he has a complete abstract car-wash solution that he is ready to implement.

4.4.3 The *Design* Class

```
<owl:Class rdf:ID="Design">
    <rdfs:subClassOf rdf:resource="#Abstraction"/>
    <rdfs:subClassOf rdf:resource="#Solution"/>
</owl:Class>
```

A *design* is an abstraction that meets a requirement – which means that it is an abstraction that is a solution. This concept is captured by the **Design** class, which is defined as a subclass of the **Abstraction** and **Solution** classes. (It is not defined as the intersection of those classes, allowing for the possibility of a solution that is an abstraction but is not a design.)

The corresponding classes and properties are illustrated in Figure 18.

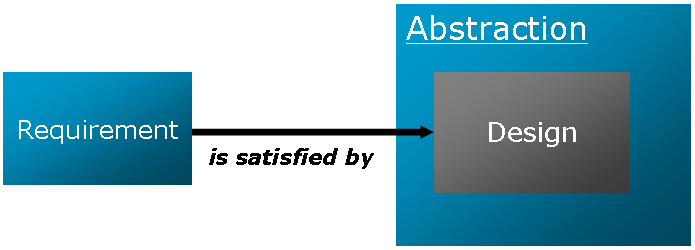

Figure 18: Design

Joe's abstract car-wash solution is a *design*.

4.4.4 The *Design Activity* Class

```
<owl:Class rdf:ID="DesignActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasEffect" />
      <owl:someValuesFrom>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#isChangeTo" />
          <owl:someValuesFrom rdf:resource="#Design" />
        </owl:Restriction>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

A *design activity* is an activity that creates or modifies a design. This concept is captured by the **Design Activity** class, which is defined as the intersection of the **Activity** class with the class of things that have effects that are changes to designs. The relevant classes and properties are illustrated in Figure 19.

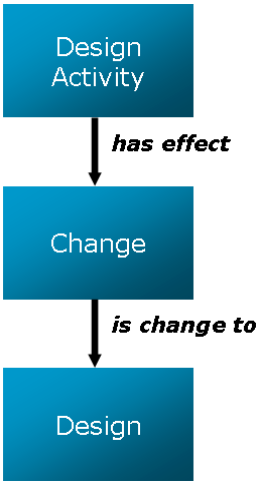

Figure 19: Design Activity

The process that Joe goes through to create his abstract car-wash solution is a *design activity*. The creation of the design is an effect of that activity, and creation is a kind of change.

4.5 Implementation

4.5.1 Overview

The process of realizing an abstract solution – a design – is referred to as "implementation", and the result of that process is called "an implementation". These concepts are captured by the **Implementation Activity** and **Implementation** classes, described in this section and illustrated in Figure 20.

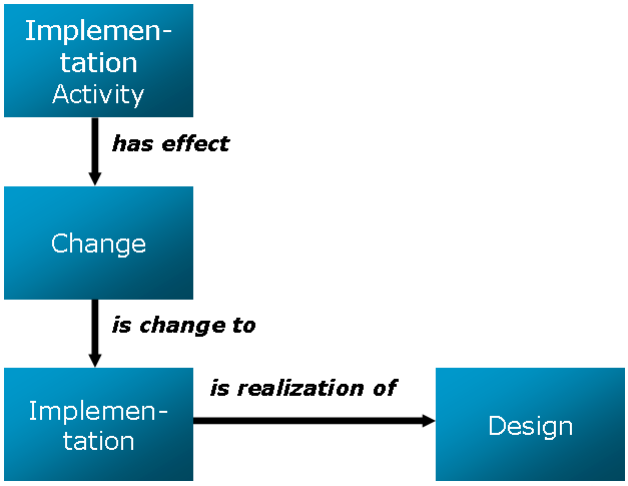

Figure 20: Implementation

4.5.2 Example

Once Joe is satisfied with his design, he purchases the site and the car-wash machine, and has the machine installed on the site in accordance with the plans. The site and the machine are realizations of his site and car-wash machine solution building blocks, and the complete installed system is a realization of his overall automatic car-wash system design.

4.5.3 The *Implementation* Class

```
<owl:Class rdf:ID="Implementation">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isRealizationOf" />
      <owl:someValuesFrom rdf:resource="#Design" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>
```

An *implementation* is a realization of a design. This concept is captured by the **Implementation** class, which is defined as the class of things that are realizations of designs.

The automatic car-wash system that Joe creates is an *implementation*. It is a realization of his automatic car-wash design.

4.5.4 The *Implementation Activity* Class

```
<owl:Class rdf:ID="ImplementationActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasEffect" />
      <owl:someValuesFrom>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#isChangeTo" />
          <owl:someValuesFrom rdf:resource="#Implementation" />
        </owl:Restriction>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

The **Implementation Activity** class captures the concept of implementation as a process that realizes a design. It is defined as the intersection of the **Activity** class with the class of things that have effects that are changes to implementations.

The building and installation of Joe's new automatic car-wash system is an *implementation activity*.

4.6 Kinds of Actor

4.6.1 Overview

The **Actor** class was introduced in section 2.3. In that section, it was stated that an actor can be a person or an organization or a piece of technology. This section describes classes for those kinds of actor. These are the **Human Actor**, **Organization Actor**, and **Technology Actor** subclasses of the **Actor** class. This section also describes **Software Actor**, an important subclass of **Technology Actor**, and **Enterprise**, an important subclass of **Organization Actor**. These classes are illustrated in Figure 21.

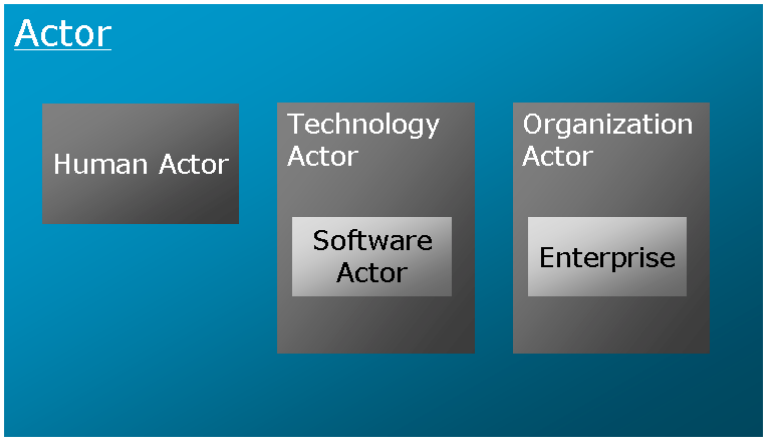

Figure 21: Kinds of Actor

The human, technology and organization kinds of actor are mutually exclusive. An actor can not be both a human and a piece of technology, both a human and an organization, or both an organization and a piece of technology. But the classification is not exhaustive. There could be other kinds of actor, for example animals, for which the ontology does not have classes.

Section 2.3 also stated that the term "actor" can be used to describe a real individual or a role – an abstraction of an individual. This distinction is explored in more depth in section 4.3.

4.6.2 The *Human Actor* Class

```
<owl:Class rdf:ID="HumanActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
</owl:Class>
```

A *human actor* is an actor that is a human being. The **Human Actor** class is defined as a subclass of the **Actor** class.

Joe is a *human actor*, for example.

4.6.3 The *Technology Actor* Class

```
<owl:Class rdf:ID="TechnologyActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#HumanActor"/>
</owl:Class>
```

A *technology actor* is an actor that is a machine or other piece of technology. It could be hardware, software, or both. The **Technology Actor** class is defined as a subclass of the **Actor** class.

An actor can not be both a human being and a piece of technology. The **Technology Actor** and **Human Actor** classes are therefore disjoint.

Joe's car-wash machine is a *technology actor*.

A technology actor can be composed of other technology actors. (By contrast, a human actor would not normally be regarded as being composed of other actors, of any kind.) For example, a car-wash machine might be composed of brushes, motors, etc, each of which could be regarded as being an actor in its own right.

4.6.4 The *Software Actor* Class

```
<owl:Class rdf:ID="SoftwareActor">
   <rdfs:subClassOf rdf:resource="#TechnologyActor"/>
</owl:Class>
```

A software actor is an executing software program. This is a particular kind of technology actor. The **Software Actor** class is defined, as a subclass of **Technology Actor**, to capture this concept.

Most car-wash machines are controlled by software. A running car-wash machine control program would be a *Software Actor*.

4.6.5 The *Organization Actor* Class

```
<owl:Class rdf:ID="OrganizationActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
   <rdfs:subClassOf rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#HumanActor"/>
   <owl:disjointWith rdf:resource="#TechnologyActor"/>
</owl:Class>
```

An *organization actor* is a system whose components can be people, technology items, and other things, and that is regarded as a single actor. This concept is captured by the **Organization Actor** class, which is defined as a subclass of **Actor**, and also of **System**.

An organization is not actually a person or a piece of technology. The **Organization Actor**, **Technology Actor**, and **Human Actor** classes are disjoint.

The city authority that issues Joe's street trading permit is an *organization actor*. It is a complex organization that includes computer systems, clerks, elected representatives, committees, and a mayor. But it can be regarded as a single actor. (For some legal purposes, it even counts as a kind of person, although the ontology does not include this idea.)

4.6.6 The *Enterprise* Class

```
<owl:Class rdf:ID="Enterprise">
   <rdfs:subClassOf rdf:resource="#OrganizationActor"/>
</owl:Class>
```

The bodies that have service-oriented architectures are typically enterprises. An *enterprise* is a definable collection of human and asset resources that provide products or services for use or consumption by outside entities. Enterprises include commercial, industrial and government organizations. The **Enterprise** class is defined, as a subclass of **Organization Actor**, to capture this concept.

The city authority that issues Joe's permit is an *enterprise* – it performs local government services for its citizens.

4.7 Software Services

4.7.1 Overview

Software services are the most important solution building blocks in a service-oriented software architecture. They are represented in the ontology by instances of the **Software Service** class.

4.7.2 The *Software Service* Class

```
<owl:Class rdf:ID="SoftwareService">
   <rdfs:subClassOf rdf:resource="#Service"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParticipant" />
      <owl:someValuesFrom rdf:resource="#SoftwareActor" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParticipant" />
      <owl:allValuesFrom rdf:resource="#SoftwareActor" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

A *software service* is a service that is performed by software programs. This concept is captured by the **Software Service** class, which is defined as the class of services in which

- At least one software actor takes part, and
- All the actors that take part are software actors.

Many software programs perform services. In software architectures that are not classed as service-oriented, the fact that a program performs a service is not stressed, the provider of the service is often not identified, and the contract between the provider and the consumers of the service is usually implicit. In a service-oriented architecture, software services and their providers are clearly identified, and the services and the contracts for providing them are often formally described.

In addition, a service-oriented software architecture often incorporates some or all of the concepts that are introduced in the following subsections. These concepts relate to activities and services in general, and their application to software services is an important facet of SOA.

4.8 Service Orchestration and Choreography

4.8.1 Overview

Loosely-coupled services can easily be combined in different ways to do different things. Such a combination constitutes a composition of the constituent services. The activity produced by this composition is often itself a service.

In the context of a service-oriented architecture, solutions often consist of compositions of services: that is, of orchestrations and choreographies.

This section describes those two styles of service composition. Both styles define subclasses of the **Composition** class, as illustrated in Figure 22.

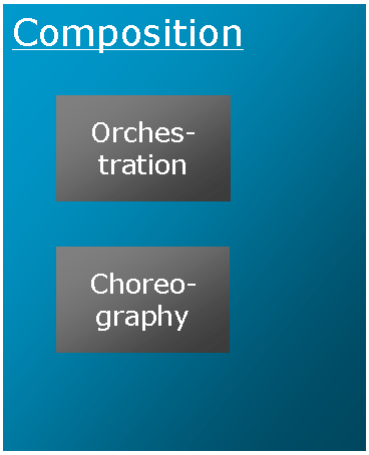

Figure 22: Orchestration and Choreography

Both of these subclasses consist of compositions of activities that produce other activities. The difference between these two styles is that:

- In an *orchestration*, there is one particular component activity of the composition that oversees and directs the other component activities; and
- In a *choreography*, the component activities of the composition are autonomous but have a defined pattern of behavior with respect to each other.

The **Orchestration** and **Choreography** subclasses of the **Composition** class, and the **has direction activity** property that is associated with the **Orchestration** class, are defined in the following subsections.

4.8.2 The *Orchestration* Class

```
< owl:Class rdf:ID="Orchestration">
  <rdfs:subClassOf rdf:resource="#Composition"/>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#hasComponent" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#produces" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasDirectionActivity" />
      <owl:cardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
         >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class >
```

An *orchestration* is a composition of activities that includes a *direction activity* that oversees and directs the others. This concept is captured by the **Orchestration** class, which is defined as a subclass of the **Composition** class, with a restriction on the **has component** property that ensures that all the components of an orchestration are activities, a restriction on the produces property that ensures that the thing produced by the composition is an activity, and a restriction on the **has direction activity** property that ensures that each orchestration includes a direction activity. (The **has direction activity** property is defined in Section 4.8.3.)

An orchestration has one and only one direction activity. The cardinality constraint on **has direction activity** ensures that this is the case.

Muhammad's car-valet service, described in Section 2.6.2, is produced by an *orchestration*. Muhammad performs its direction activity, and his friends Joe, Masha and Juan perform the other component services of the composition: car washing, vacuuming, and shampooing.

Note that this list of component services does not include the messaging service performed by Lin. A common design technique is to identify a set of components as being "infrastructure". These infrastructure components can be assumed to be used by solutions, without being included in them. Lin's messaging service is here assumed to be part of the infrastructure, and is not included in the orchestration solution. What constitutes "infrastructure" is usually an architectural decision. The relevant concepts are described in Section 5.2.6.

In a service-oriented software solution, the component services of an orchestration are software services performed by software programs. For example, a service-oriented car-wash software solution might be composed of the following software services: customer welcome; car-wash

program selection; payment; car-wash execution; and a control service that schedules the other four.

4.8.3 The *has direction activity* and *is direction activity of* Properties

```
<owl:ObjectProperty rdf:ID="hasDirectionActivity">
  <rdfs:subPropertyOf rdf:resource="#hasComponent"/>
  <rdfs:domain rdf:resource="#Orchestration"/>
  <rdfs:range rdf:resource="#Activity"/>
  <owl:inverseOf rdf:resource="#isDirectionActivityOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isDirectionActivityOf">
  <rdfs:subPropertyOf rdf:resource="#isComponentOf"/>
  <rdfs:domain rdf:resource="#Activity"/>
  <rdfs:range rdf:resource="#Orchestration"/>
  <rdfs:domain rdf:resource="#Orchestration"/>
  <rdfs:range rdf:resource="#Activity"/>
  <owl:inverseOf rdf:resource="#hasDirectionActivity"/>
</owl:ObjectProperty>
```

The **has direction activity** property, and its inverse **is direction activity of**, capture the relation between an orchestration and its component activity that oversees and directs its other component activities.

The **has direction activity** property is defined as a subproperty of the **has component** property, and its inverse **is direction activity of** is defined as a subproperty of **is component of**, because the direction activity of an orchestration is one of its component activities.

Muhammad's car-valet service orchestration *is directed by* his direction activity. The service-oriented car-wash software orchestration *is directed by* its control service.

4.8.4 The *Choreography* Class

```
< owl:Class rdf:ID="Choreography">
  <rdfs:subClassOf rdf:resource="#Composition"/>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#hasComponent" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#produces" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasDirectionActivity" />
      <owl:cardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
         >0</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class >
```

A *choreography* is a set of autonomous activities that have a defined pattern of behavior with respect to each other. This concept is captured by the **Choreography** class, which is defined as a subclass of the **Composition** class, with a restriction on the **has component** property that ensures that all the components of a choreography are activities, a restriction on the **produces** property that ensures that the thing produced by the composition is an activity, and a restriction on the **has direction activity** property that ensures that an orchestration does not have a direction activity. (This last restriction implies that a choreography can not also be an orchestration.)

There is no single activity that directs the other activities in a choreography. A choreography distributes the control and relies on the ability of its component activities to understand and respond to events.

Muhammad's car-valet business expands. He buys premises and engages staff, training them as washers, vacuumers, and shampooers. He no longer directs operations. Now, when a car comes in to be valeted, the first available washer, the first available vacuumer, and the first available shampooer go to work on it. His car-valet service is now produced by a choreography of services, rather than an orchestration.

It would be possible to design a car-wash solution without a control program. The customer welcome service might generate a customer welcomed event; the car-wash program selection service could respond to this and generate a program selected event; the payment service could respond to this and generate a payment made event; and the car-wash execution service might respond to this event and execute the car wash. This solution would be a *choreography*.

4.9 Messaging

4.9.1 Overview

Loosely-coupled services can interact conveniently by exchanging messages, rather than by invoking each other directly. This approach has a number of advantages, and is often used in service-oriented architecture.

Web services often exchange messages using the Simple Object Access Protocol and the Hypertext Transfer Protocol . This is a prime example of the messaging approach.

Another example is the use of an Enterprise Service Bus (ESB) to carry messages between services. This enables centralized monitoring, control and transformation of the information passing between the services, and can be a basis for solutions that satisfy requirements for semantic interoperability, management, and security. In this case, the ESB is a technology actor. It performs a messaging service, whose consumers are the actors taking part in the services that exchange the messages.

The concept of interface is described in Section 3.4. A messaging service can be a way by which interfaces are implemented within a system. The relevant classes and properties are illustrated in Figure 23.

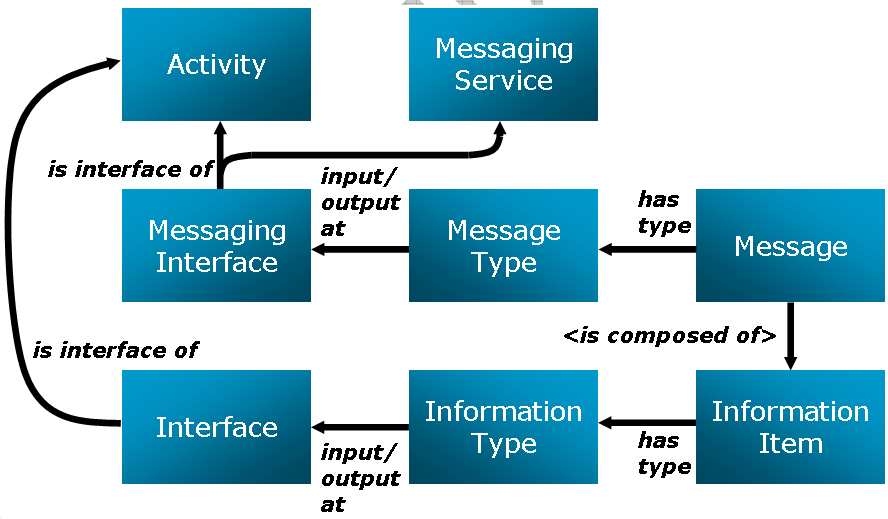

Figure 23: Messaging Interface

An activity can have an interface, and particular information types can be input at and output at that interface. Messages composed of those information items are information items that have **message type** information types. They are input at and output at a messaging interface, which is also an interface of the activity, distinct from the first interface. The messaging interface is an interface of the activity, and is also an interface of a messaging service, which conveys the messages.

4.9.2 Example

In Muhammad's car valet service, described in Section 2.6.2, Lin performs a *messaging service*. She takes messages from Muhammad to Joe, Masha, and Juan. Joe, Masha, Juan, and Muhammad are the consumers of this service.

For messaging between software services, there are a number of Enterprise Service Bus (ESB) products that perform messaging services and can be bought "off-the-shelf".

4.9.3 The *Message* Class

```
<owl:Class rdf:ID="Message">
  <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>
```

A *message* is an information item that is sent by an actor to one or more other actors (the *recipients* of the message). In addition to information to be processed by the recipients (the *message body*), a message may include other information, for example to identify the recipients or the sender, or to describe characteristics of the message such as its priority.

This concept is captured by the **Message** class, which is defined as a subclass of the **Information Item** class.

In the case of Muhammad's car-valet service, the *messages* exchanged are verbal and informal. For example, Muhammad asks Lin to tell Joe that a customer has brought in a car that needs washing. In the case of software services, the software actors that take part in the services exchange *messages* that have formal definitions.

4.9.4 The *Message Type* Class

```
<owl:Class rdf:ID="MessageType">
  <rdfs:subClassOf rdf:resource="#InformationType"/>
</owl:Class>

<owl:Class rdf:about="#Message">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasType"/>
         <owl:someValuesFrom rdf:resource="#MessageType" />
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

A *message type* identifies a particular class of messages. This concept is captured by the **Message Type** class, which is a subclass of the **Information Type** class.

The restriction on the **Message** class ensures that each message has an information type that is an instance of the **Message Type** class.

An application of the ontology might define only a single instance of the **Message Type** class (for example, **message**), or it might define multiple instances (for example, **business message**, **personal message**, and **infrastructure-generated message**).

4.9.5 The *Messaging Service* Class

```
<owl:Class rdf:ID="MessagingService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</owl:Class>
```

A *messaging service* is a service that carries messages between actors. This concept is captured by the **Messaging Service** subclass of the **Service** class.

Lin provides a *messaging service* that is consumed by Muhammad and his friends when Muhammad directs his car-valet service.

4.9.6 The *Messaging Interface* Class

```
<owl:Class rdf:ID="MessagingInterface">
  <rdfs:subClassOf rdf:resource="#Interface"/>
  <rdfs:subClassOf>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasInputInformation"/>
          <owl:someValuesFrom rdf:resource="#MessageType" />
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasOutputInformation"/>
          <owl:someValuesFrom rdf:resource="#MessageType" />
        </owl:Restriction>
      </owl:unionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#MessagingService">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasInterface"/>
         <owl:someValuesFrom rdf:resource="#MessagingInterface" />
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

A *messaging interface* is an interface by which messages are passed to or received from a messaging service. This concept is captured by the **Messaging Interface** class, which is defined to be a subclass of the class of interfaces at which messages are input or output.

The restriction on the **Messaging Service** class ensures that every messaging service has at least one messaging interface.

The *messaging interface* to Lin's messaging service is informal; Muhammad simply asks him to take the messages to Joe and his other friends. Where software actors exchange messages using a messaging service performed by a service bus, the *messaging interfaces* are formally defined in the relevant software programming languages.

4.10 Discovery

4.10.1 Overview

In the world of business, services are advertised by their providers in many ways, and it is by this means that consumers learn of the existence of services and are able to make choices between competing services. The concept has been taken into the technical paradigm of service orientation, leading to the definition of technical standards for service descriptions and service discovery.

This section describes the ontology constructs corresponding to the business concepts that are related to service advertisement, and that apply also to service discovery in SOA. These are the classes **Registry**, **Registry Entry**, **Visibility, and Registry Service**, and the properties **is contained in**, **is registered in**, **is in scope of**, and **has visibility**. They are illustrated in Figure 24 (except for the Registry Service class, which is described in Section 4.10.9).

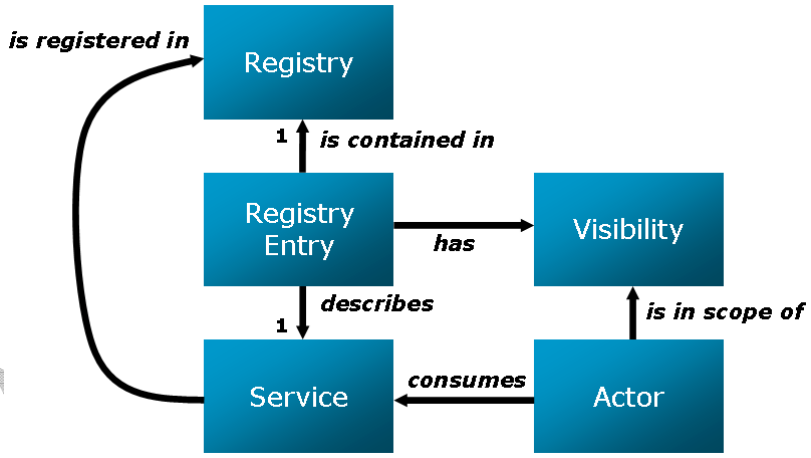

Figure 24: Registries and Visibility

Comment [CJH1]: Figure replaced to correct a typo: is contained in arrow wrong way up.

Services are described by registry entries, which are contained in registries. Registries are regarded as information items that are composed of these entries. A registry entry that describes a service has a particular visibility. This determines which potential consumers can find out about the service through the registry. These actors are in the scope of that visibility.

4.10.2 The *Registry* Class

```
<owl:Class rdf:ID="Registry">
   <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>
```

A *registry* is an organized set of descriptions of services that are visible to potential consumers.

A registry is a collection of information. **Registry** is therefore defined as a subclass of **Information Item**.

Service registry is a technical concept of SOA. A service registry is a set of descriptions of services in a machine-readable form, for example, written in the Web Services Description Language (WSDL). But the concept of a registry for businesses and their activities, including services, is a general business concept.

A "Yellow Pages" telephone directory is an example of such a *registry*. Joe's friend Muhammad's car valet service has an entry in his local "Yellow Pages". There are many other common examples of registries, including business directories, lists of members published by trade associations, and so on.

There are a number of commercially-available products that are used to manage service *registries* for software services.

4.10.3 The *Registry Entry* Class

```
<owl:Class rdf:ID="RegistryEntry">
   <rdfs:subClassOf rdf:resource="#Description"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#isContainedIn"/>
         <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
           >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

A *registry entry* is a description of a service that is contained in a registry. This concept is captured by the **Registry Entry** class. This class is defined as a subclass of **Description**, with a restriction on the **is contained in** property that ensures that each registry entry is contained in exactly one registry

It is quite common for a policy for a service to require that its registry entries should take a particular form, or have a particular visibility. A registry entry can thus be affected by rules that are the conditions of a policy.

The Yellow Pages description of Muhammad's car-valet service is an example of a *registry entry*. This is an informal textual description. The descriptions of software services in a software service registry are examples also. These are generally formal descriptions; for example, they may be expressed in the Web Services Description Language [WSDL].

4.10.4 The *contains* and *is contained in* Properties

```
<owl:ObjectProperty rdf:ID="contains">
  <rdfs:domain rdf:resource="#Registry"/>
  <rdfs:range rdf:resource="#RegistryEntry"/>
  <owl:inverseOf rdf:resource="#isContainedIn"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isContainedIn">
  <rdfs:domain rdf:resource="#RegistryEntry"/>
  <rdfs:range rdf:resource="#Registry"/>
  <owl:inverseOf rdf:resource="#contains"/>
</owl:ObjectProperty>
```

The **contains** property, and its inverse **is contained in**, capture the relation between a registry and the descriptions that are included in it.

Registries are regarded as information items that are composed of registry entries. A registry entry is contained in a registry if and only if it is a component of the composition of information items that produces the registry. (This fact is not captured by the OWL definitions.)

Muhammad's local "Yellow Pages" *contains* the description of his car-valet service. Software service registries *contain* formal descriptions of software services.

4.10.5 The *registers* and *is registered in* Properties

```
<owl:ObjectProperty rdf:ID="registers">
   <rdfs:domain rdf:resource="#Registry"/>
   <rdfs:range rdf:resource="#Service"/>
   <owl:inverseOf rdf:resource="#isRegisteredIn"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRegisteredIn">
   <rdfs:domain rdf:resource="#Service"/>
   <rdfs:range rdf:resource="#Registry"/>
   <owl:inverseOf rdf:resource="#register"/>
</owl:ObjectProperty>
```

The **registers** property, and its inverse **is registered in**, capture the relation between a registry and the services whose descriptions it contains.

Muhammad's car-valet service *is registered in* the local "Yellow Pages". Software services are *registered in* software service registries.

4.10.6 The *Visibility* Class

```
<owl:Class rdf:ID="Visibility">
   <rdfs:subClassOf rdf:resource="#InformationType"/>
</owl:Class>
```

An instance of the **Visibility** class defines a set of actors that can see (or read) an information item. One such instance might be **public visibility**, defining the set of all actors. Another might be **top secret** defining, in the context of a particular organization, a particular, small set of individuals.

The visibility of an information item is information about that item. It is meta-information. **Visibility** is therefore defined as a subclass of **Information Type**.

The ontology does not define any specific instances of the class **Visibility**. **Public** and **top secret** are examples of instances of this class that users of the ontology might define. How they define such instances is a matter for them, the ontology imposes no conditions on such definitions.

Anyone can walk down the street and see Joe's placard. But only local telephone service subscribers, who receive copies of the Yellow Pages, are likely to see the entry for Muhammad's car valet service. It would be reasonable to define **public** and **local subscribers** instances of the **Visibility** class, as appropriate visibilities for these two service descriptions.

In the case of software services, a solution often involves the definition of specific instances of the **Visibility** class. **Public**, **subscribers**, **managers**, and **system administrators** are typical examples.

4.10.7 The *is in scope of* and *has in scope* Properties

```
<owl:ObjectProperty rdf:ID="isInScopeOf">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Visibility"/>
   <owl:inverseOf rdf:resource="#hasInScope"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInScope">
   <rdfs:domain rdf:resource="#Visibility"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isInScopeOf"/>
</owl:ObjectProperty>
```

The **is in scope of** property, and its inverse **has in scope**, capture the relation between a member of the set of actors defined by a visibility and that visibility.

With typical definitions, all actors would be *in scope of* **public** visibility, and those actors that are subscribers to the local telephone service would be *in scope of* **local subscribers** visibility.

If **public**, **subscribers**, **managers**, and **system administrators** are visibilities defined for a particular software services solution, the corresponding classes of actors would typically be all system users, customers that have paid to use certain services, management staff of the enterprise providing the services, and system administrator staff of that enterprise.

4.10.8 The *has visibility* and *is visibility of* Properties

```
<owl:ObjectProperty rdf:ID="hasVisibility">
   <rdfs:subPropertyOf rdf:resource="#hasType"/>
   <rdfs:domain rdf:resource="#RegistryEntry"/>
   <rdfs:range rdf:resource="#Visibility"/>
   <owl:inverseOf rdf:resource="#isVisibilityOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isVisibilityOf">
   <rdfs:subPropertyOf rdf:resource="#isTypeOf"/>
   <rdfs:domain rdf:resource="#Visibility"/>
   <rdfs:range rdf:resource="#RegistryEntry"/>
   <owl:inverseOf rdf:resource="#hasVisibility"/>
</owl:ObjectProperty>
```

The **has visibility** property and its inverse **is visibility of** capture the relation between a registry entry and a visibility defining a set of actors that can see it.

Has visibility is a sub-property of **has type**, and **is visibility of** is a sub-property of **is type of**.

With the visibility definitions suggested in section 4.10.6, Muhammad's Yellow Pages entry *has visibility* **local subscribers**.

A company that charges for the provision of software services often arranges matters so that the descriptions of those services *have visibility* **subscribers**. Descriptions of system configuration services often *have visibility* **system administrator**.

Note that visibility of description is distinct from accessibility of service. It may be possible for an actor to see a description of a service but not to consume that service, and it may be possible for an actor to consume a service without being in then scope of visibility of a description of it. The reasons for giving a service description a particular visibility are often linked to those for restricting access to the service, but the concepts should not be confused.

4.10.9 The *Registry Service* Class

```
<owl:Class rdf:ID="RegistryService">
   <rdfs:subClassOf rdf:resource="#Service"/>
</owl:Class>
```

A *registry service* is a service that maintains a registry, storing descriptions of services and making them available to potential consumers. This concept is captured by the **Registry Service** class.

The term "registry" is often used in common speech to mean an actor that performs a registry service. In this ontology, the term is not used in this sense; it is used only to mean the collection of information maintained by a registry service.

Commercial "registry" products are technology actors that perform *registry services*.

4.11 Virtualization

4.11.1 Overview

A service has a well-defined interface, and its function and performance are assured by contract, but how it is implemented is transparent to the consumer. This means that the service provider can employ different resources on different occasions when the service is used. This can enable the service to be provided consistently when resources are not consistently available, or when there is variable demand for the service.

This technique is commonly employed in business operations. For example, "24/7" support capability can be provided by call centers in different time zones, each of which operates for 8 hours of the 24. The caller is switched to whichever call center is working at the time of the call. In a further application of the principle of virtualization, the call can be taken by any of the support personnel in that center, and will be routed to the first one to become free.

The technique is also used in service-oriented software solutions. When used at the infrastructure level, it can provide a form of grid computing.

The **Virtual Actor** and **Virtualized Service** classes that are described in the following subsections capture two of the essential concepts of virtualization as used in service-oriented software solutions.

4.11.2 The *Virtual Actor* Class

```
<owl:Class rdf:ID="VirtualActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
   <rdfs:subClassOf rdf:resource="#System"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasComponent"/>
         <owl:allValuesFrom rdf:resource="#Actor"/>
      </owl:Restriction>
   </rdfs:subClassOf>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasComponent"/>
         <owl:allValuesFrom rdf:resource="#Realization"/>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

A *virtual actor* is an abstract actor that can be realized by multiple real actors. This concept is captured by the **Virtual Actor** class, which is a subclass of the **Actor** class and also of the **Abstraction** class.

The realization of a virtual actor is a system of real actors that all perform the same function. This is captured by the first restriction on the **has component** property.

Each of the individual real actors is also a realization of the virtual actor. The second restriction captures the idea that they are realizations, but the idea that they are realizations of the original virtual actor can not readily be captured in OWL.

Juan tells Muhammad that he is taking a part-time job with a carpet-cleaning company, and will only be available to do shampooing in Muhammad's car-valet service on Tuesdays and Thursdays. Muhammad has another friend, Samir, who agrees to take Juan's place on the other days of the week. Muhammad then has a "shampooer" *virtual actor*. The combination of Samir and Juan is a realization of this actor. Samir and Juan are also realizations of it individually.

In Joe's expanding automatic car-wash operation, Joe decides to install multiple automatic car-wash machines on the most popular sites. The customer pays for the car-wash, then uses the first machine to become free. There is a car-wash machine *virtual actor*, which is realized by a set of actual car-wash machines, each of which is individually a realization of the virtual car-wash machine.

4.11.3 The *Virtualized Service* Class

```
<owl:Class rdf:ID="VirtualizedService">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Service" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParticipant" />
      <owl:someValuesFrom rdf:resource="#VirtualActor" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

A *virtualized service* is a service in which a virtual actor takes part. This concept is captured by the **Virtualized Service** class, which is defined as the intersection of the **Service** class with the class of activities that have virtual actors as participants.

The ontology does not distinguish between a fully virtualized service, in which all the participants are virtual, and a partially virtualized service, in which only some of them are.

Muhammad's car-valet service, and Joe's car-wash service, as described in Section 4.11.1, are both (partially) *virtualized services*.

5 Architecture and Governance

5.1 Introduction

This chapter builds on the general concepts relating to design that were described in the first part of Chapter 4 to introduce concepts relating to architecture and governance, and finally to describe the concept of a service-oriented architecture, which may use the design pattern concepts described in the last part of Chapter 4.

Although many of the concepts related to architecture and governance are more general in nature, this chapter is intended to address enterprise architectures that include significant information technology components, and particularly to address architectures based on the use of software services.

This chapter first describes concepts related to architecture and architecture development. It then addresses the idea of solutions instantiating an architecture. Governance is a crucial aspect of the use of service-oriented architectures, and the ideas of governance of activities related to architecture development and system implementation and operation are described. Finally, the chapter describes the concept of service-oriented architecture, and how it relates to the other concepts of the ontology.

5.2 Architecture

5.2.1 Overview

An *architecture* is the fundamental organization of a system embodied in its components, their relationships to each other and the environment, and the principles guiding its design and evolution. (See [IEEE 1471].)

The clause "and the principles guiding its design and evolution" is an important part of this definition, and this is particularly the case for SOA. A system typically evolves through the creation of new solutions – as abstractions which are then realized by implementation. Where the abstract solutions follow the architecture's fundamental organization and are created in accordance with its principles, they are *instantiations* of the architecture.

Systems – even very complex ones – can develop by chance. (Darwin's theory of natural selection provides the ultimate illustration of this.) But the commercial and technical systems of enterprises today are usually formed through deliberate choices. Architects are employed to develop architectures for these systems, which are instantiated by solutions, which are then implemented. When developing an architecture for a system, an architect thinks in terms of architecture building blocks, which are abstractions of the components of the system.

The evolution of systems by chance is time-consuming, error-prone, and expensive. The use of architecture saves time, avoids mistakes, and cuts costs.

This section describes the concepts of architecture, architecture building block, and architecture development activity. It defines the **Architecture**, **Architecture Building Block** and **Architecture Development Activity** classes, and the **is architecture of** and **has infrastructure** properties. The concepts associated with instantiation are described in Section 5.3.

5.2.2 Example

The design of Joe's first car-wash system might possibly be described as architecture, of a very trivial kind, applying as it does only to one particular system with no plans for evolution. But Joe's business prospers and, as his operations expand, he increasingly sees the need for real architecture, including the establishment of principles that will help his business to evolve and grow.

This applies first of all in the business domain. Should he borrow to buy more sites? Or should he start a franchise operation? Should he set up to provide car-wash implants at gas stations and supermarkets? Or should he start selling gas and groceries at his own locations? As he takes these decisions, he establishes principles that are the foundation of his business architecture.

He needs to establish principles in the technical domain too. Buying whichever model of car-wash machine happens to be cheapest for each new site may save money in the short term, but Joe realizes that it will leave him with a collection of equipment that will be a nightmare to operate and maintain. He lays down standards for what kinds of machines he will purchase. This is the beginning of his technical architecture.

When he has just one or two stand-alone sites, he is not very concerned about information technology. As he acquires further sites, and partners with grocery stores who ask him to integrate with their point-of sale systems, he finds himself thinking about IT more and more. But it is when he decides to buy a central computer system and network all his car-wash locations to it that he really comes to understand the need for an enterprise IT architecture. He hires a Chief Architect.

The new Chief Architect, Kimi, talks to Joe to find out what he wants. Joe has a grand vision. He wants the central system first of all to collect accounting information, prevent fraud, provide usage statistics, and handle some on-line diagnostics. Then, it must interface to his grocery store partners, and other future partners as yet unspecified, and to his suppliers, and be able to be rapidly reconfigured to support new marketing initiatives. Finally, Joe wants to start a loyalty program, and to provide a web service through which its members can book their washes in advance, so that they don't have to wait in line on Sunday mornings.

Kimi nods her head, and recommends SOA.

5.2.3 The *Architecture* Class

```
<owl:Class rdf:ID="Architecture">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#Composition"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
   <owl:disjointWith rdf:resource="#Policy"/>
   <owl:disjointWith rdf:resource="#Abstraction"/>
   <owl:disjointWith rdf:resource="#Realization"/>
</owl:Class>
```

The concept of architecture is captured by the **Architecture** class.

An architecture can have component architecture building blocks. An architecture can therefore be a system. The **Architecture** class is not disjoint with the **System** class.

The concept of architecture is distinct from the concepts other than system that have been introduced so far, and the **Architecture** class is defined to be disjoint with the classes corresponding to these concepts.

In particular, although architecture is an abstract concept, an architecture is not an abstraction of anything. A different term, *is architecture of*, is used to describe the relation between an architecture and a system that has that architecture. The **Architecture** class is disjoint with the **Abstraction** class.

Also, an architecture can be composed of other architectures, but that composition of architectures is not itself an architecture. The **Architecture** class is disjoint with the **Composition** class.

The concepts of architecture and architecture building block, and their relation to the concepts of system and abstraction, are illustrated in Figure 25.

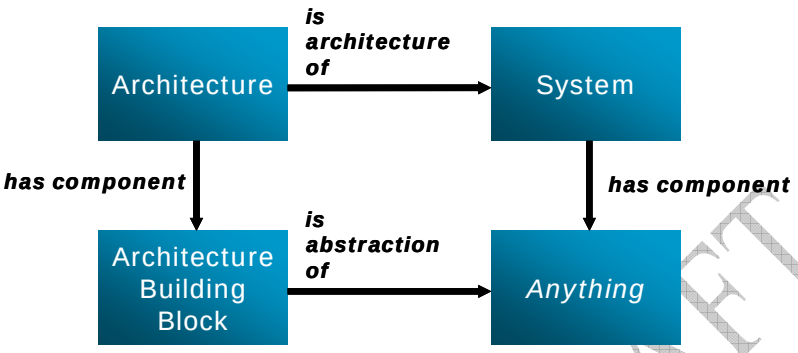

Figure 25: Architecture

Kimi forms an architecture team. Starting from the idea of a service-oriented architecture, they debate a number of questions. “What are the business operations?” “What is the business information that the system must deal with?” “What kinds of software service are needed to support the business operations?” “What kinds of contract and policy should apply to them?” “How should the services be described, and where should the descriptions be kept?” “How should the services interface to each other and exchange information?” “How should services be composed of other services?” “What infrastructure – registry, service bus, etc. – is needed to support the services?” “What operating systems and hardware should it run on?” As they answer these questions, they are defining a service-oriented architecture for Joe’s car-wash operations.

5.2.4 The *has architecture* and *is architecture of* Properties

```
<owl:ObjectProperty rdf:ID="hasArchitecture">
   <rdfs:domain rdf:resource="#System"/>
   <rdfs:range rdf:resource="#Architecture"/>
   <owl:inverseOf rdf:resource="#isArchitectureOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isArchitectureOf">
   <rdfs:domain rdf:resource="#Architecture"/>
   <rdfs:range rdf:resource="#System"/>
   <owl:inverseOf rdf:resource="#hasArchitecture"/>
</owl:ObjectProperty>
```

The **has architecture** property, and its inverse **is architecture of**, capture the relation between systems and their architectures.

Note that “has architecture” has a broad meaning. It includes “has had architecture in past” and “may have architecture in future”, as well as “has architecture at this precise moment.”

The architecture created by Joe’s team *is an architecture of* the system comprising his car-wash operations. It embodies the organization of the components of that system when his new central computer has just become operational. It does not embody the components of that system at the time when it is being created, but it is nevertheless referred to as *an architecture of* the system at that time.

5.2.5 The *Architecture Building Block* Class

```
<owl:Class rdf:ID="ArchitectureBuildingBlock">
   <rdfs:subClassOf rdf:resource="#Abstraction"/>
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty rdf:resource="#isComponentOf" />
       <owl:someValuesFrom rdf:resource="#Architecture" />
     </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>
```

An *architecture building block* is an abstraction that is a component of an architecture. This concept is captured by the **Architecture Building Block** class, which is defined as a subclass of the **Abstraction** class, with a restriction on the **is component of** property that ensures that every architecture building block is a component of an architecture.

Almost anything can be an architecture building block, including activities, actors, effects, changes, information items, compositions, events, interfaces, contracts, and policies. The **Architecture Building Block** class is not defined as disjoint with any of the other classes of this ontology.

In particular, an architecture building block can have component architecture building blocks. The **Architecture Building Block** class is therefore not defined as disjoint with the **System** class.

Joe's architecture team identifies a number of *architecture building blocks*. At the top level are building blocks such as **car-wash machine**, **point-of-sale system**, and **central computer system**. These have lower-level component building blocks, such as **car-wash services**, **payment services**, and **services bus**, and these may be in turn have component building blocks at even lower levels.

There is no clear definition of exactly what should constitute an architecture building block. What constitutes a building block in any particular architecture is determined by the judgment of the architects concerned. It would be possible to define **service** as a building block, and not define separate building blocks such as **car-wash services** and **payment services** for services of different kinds. Equally, it would be possible to define individual services such as **accept money payment** as building blocks.

As explained in Section 1.3, the ontology defines the relations between terms, but does not prescribe exactly how they should be applied, and different applications of the ontology to the same situation are perfectly possible. The ability of architects to use the term "architecture building block" in different ways is a good example of this.

5.2.6 The *has infrastructure* and *is infrastructure of* Properties

```
<owl:ObjectProperty rdf:ID="hasInfrastructure">
   <rdfs:subPropertyOf rdf:resource="#hasComponent"/>
   <rdfs:domain rdf:resource="#Architecture"/>
   <rdfs:range rdf:resource="#ArchitectureBuildingBlock"/>
   <owl:inverseOf rdf:resource="#isInfrastructureOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isInfrastructureOf">
   <rdfs:subPropertyOf rdf:resource="#isComponentOf"/>
   <rdfs:domain rdf:resource="#ArchitectureBuildingBlock"/>
   <rdfs:range rdf:resource="#Architecture"/>
   <owl:inverseOf rdf:resource="#hasInfrastructure"/>
</owl:ObjectProperty>
```

In Section 4.8.2 it was noted that a common design technique is to identify a set of components as being "infrastructure". These *infrastructure components* can be assumed to be used by solutions, without being explicitly included in them. An architecture can identify common infrastructure components for the solutions that instantiate it. These are represented in the architecture by architecture building blocks. The instantiations of these architecture building blocks are solution building blocks that are the design components.

The relation between an architecture and its architecture building blocks that represent common infrastructure components is captured by the **has infrastructure** property and its inverse **is infrastructure of**. These building blocks are components of the architecture. The **has infrastructure** property is therefore a subproperty of **has component**, and **is infrastructure of** is a subproperty of **is component of**.

Almost anything can be an architecture building block, and any of these things can be *infrastructure of* the architecture. Activities that form part of system operation, such as messaging activities, and the actors that perform them, such as ESBs, are commonly identified as *infrastructure of* an architecture. Infrastructure can also include technology actors that take part in development activities: programming language compilers, for example. A service-oriented architecture that has service orchestration as a principle might include a "BPEL engine" building block in its infrastructure. (This would be a technology actor taking part in operational or development activities, depending on whether it was an interpreter or compiler.)

Joe's car-wash system uses an enterprise service bus. His car-wash architecture includes a **service bus** architecture building block, which *is infrastructure of* his architecture.

5.2.7 The *Architecture Development Activity* Class

```
<owl:Class rdf:ID="ArchitectureDevelopmentActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasEffect" />
      <owl:someValuesFrom>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#isChangeTo" />
          <owl:someValuesFrom rdf:resource="#Architecture" />
        </owl:Restriction>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

The **Architecture Development Activity** class captures the concept of an activity that develops an architecture. It is defined as the intersection of the **Activity** class with the class of things that have effects that are changes to instances of the **Architecture** class. This is illustrated in Figure 26.

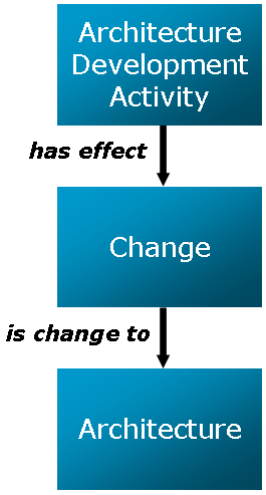

Figure 26: Architecture Development

Note that an instance of the **Architecture Development Activity** class may change more than one architecture, will often make more than one change to the architectures that it changes, and will generally also have other effects, including changes to architecture building blocks that are components of those architectures.

Joe's business continues to grow. His operations expand, nation-wide and internationally. To support this, he decides to have regional computing centers to perform most of the functions of his single central system, with that system now performing only a few functions, such as

corporate accounting. This requires a substantial change to the architecture, including the introduction of a new *regional system* building block, and changes to the way that the building blocks are made up (although the basic architectural principles remain unaltered).

The development of Joe's car-wash system architecture is an *architecture development activity*, in which the members of the architecture team take part. Note, again, that this Technical Standard does not prescribe exactly how the terms of the ontology are applied. It is possible to regard the initial creation and the subsequent change to include regional centers as separate activities. Equally, it is possible to regard them as part of a single, ongoing, architecture development activity. (And, since an activity can be composed of other activities, the single ongoing development and the separate developments of which it is composed can all be regarded as activities.)

5.3 Instantiation

5.3.1 Overview

An *instantiation* of an architecture is a solution that changes a system that has the architecture in accordance with the architecture's fundamental organization and principles. The relevant classes and properties are illustrated in Figure 27.

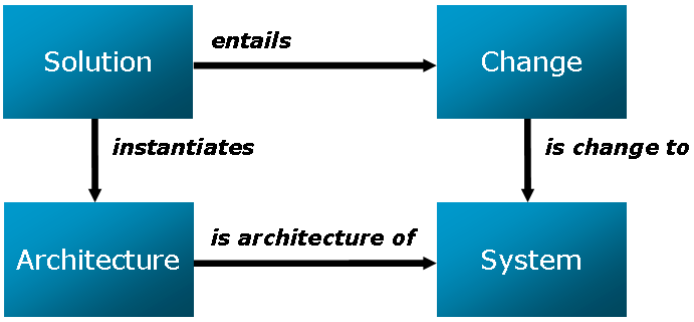

Figure 27: Architecture Instantiation

An architecture is typically a system of architecture building blocks. A solution may relate to one or more particular architecture building blocks. A solution that instantiates an architecture and relates to a particular architecture building block is also said to instantiate that building block. The corresponding classes and properties are illustrated in Figure 28.

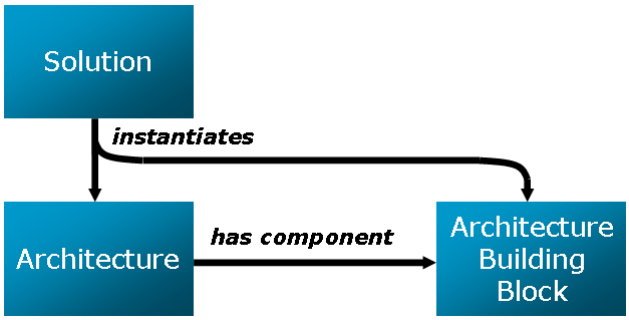

Figure 28: Building Block Instantiation

5.3.2 Example

When Joe's first (central system) architecture has been defined, and the system is in place, one of his grocery store partners wants to give a free car wash to each customer spending over $100 on groceries, and Joe agrees to support this. His car-wash software includes a **payment** orchestration that has a number of component services, including **payment method selection**, **cash payment**, and **credit card payment**. All that is needed is to change the **payment method selection** service, add a new **partner promotion payment** service, and re-orchestrate. This is the basis of a new solution that satisfies the requirement and entails a change to the system in accordance with the architecture's fundamental organization and principles. It does not entail a change to the architecture. It is an *instantiation* of the architecture.

The architecture includes a **payment** architecture building block. The old and new payment orchestrations are both solutions that *instantiate* it. (They are not only solutions in their own right – payment solutions – they are also solution building blocks of an overall promotions solution that includes changes to other parts of the system, such as accounts processing.)

The situation is different when Joe decides to introduce regional centers to satisfy his requirement to expand the business nation-wide and internationally. This is a solution that entails changes to the architecture as well as to the system. Joe commissions an architecture development activity, which makes the changes to the architecture. The regional-center solution is then further developed, to the point where it can be implemented. This solution, too, is an *instantiation* of the car-wash system architecture, as it is in accordance with the changed architecture's fundamental organization and principles. This seems rather confusing.

To avoid confusion, Kimi introduces formal version control, and labels the central system architecture as "Car-Wash Architecture Version 1", and the regional-centre architecture as "Car-Wash Architecture Version 2". He can now refer to three architectures: "Car-Wash Architecture", "Car-Wash Architecture Version 1", and "Car-Wash Architecture Version 2". This is a valid way of applying the ontology that is useful in many situations. The regional center solution is an *instantiation* of Car-Wash Architecture and Car-Wash Architecture Version 2, but not of Car-Wash Architecture Version 1.

5.3.3 The *instantiates* and *is instantiated by* Properties

```
<owl:ObjectProperty rdf:ID="instantiates">
  <rdfs:domain rdf:resource="#Solution"/>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Architecture"/>
        <owl:Class rdf:about="#ArchitectureBuildingBlock"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
  <owl:inverseOf rdf:resource="#isInstantiatedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isInstantiatedBy">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Architecture"/>
        <owl:Class rdf:about="#ArchitectureBuildingBlock"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Solution"/>
  <owl:inverseOf rdf:resource="#instantiates"/>
</owl:ObjectProperty>
```

The **instantiates** property, and its inverse **is instantiation of**, capture the relation between a solution that instantiates an architecture or architecture building block and that architecture or architecture building block.

All the solutions described in Section 5.3.2 *are instantiations of* the Car-Wash Architecture. The central system solution and the grocery promotion solutions *are instantiations of* Car-Wash Architecture Version 1. The regional centre solution *is an instantiation of* Car-Wash Architecture Version 2.

The grocery promotions payments solution and the previous payments solution are both *instantiations of* the *payment* architecture building block, as described in Section 5.3.2.

5.4 Governance

5.4.1 Overview

The term “governance” is originally from political theory, where it refers to a system by which a political unit is controlled, and to the exercise of that control. The term is now also used in relation to enterprises, where it applies to all aspects of enterprise operation, including architecture development and implementation. Good governance is widely recognized as being crucial for successful deployment of SOA.

In the context of an enterprise, *governance* refers to the control of the conduct of an activity of the enterprise. This activity may be the entire operation of the enterprise ("Enterprise Governance") or a particular activity in which the enterprise is engaged. The "political unit" here is the set of people that take part in the activity, and the control that is exercised over them is restricted to their participation in it.

A system that controls the conduct of an activity is a *governance regime*. Its components include rules for how the activity is conducted, and governance activities that develop and interpret the rules and ensure that they are followed. The corresponding classes and properties are illustrated in Figure 29.

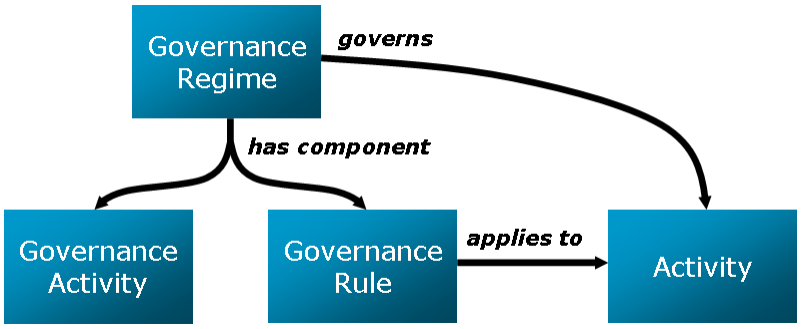

Figure 29: Governance

This section describes the **Governance Regime**, **Governance Rule**, and **Governance Activity** classes, and the **governs** property.

5.4.2 Example

As Joe's business activities expand, he finds an increasing need for *governance* within his car-wash enterprise. Even in the early stages, he has to make it clear who can purchase supplies, and on what terms. As his team grows, he must make it clear who can take which decisions. As he obtains more and more information technology, he needs to control activities that include procurement, development and operation of IT: he must set up a regime of IT governance.

When Joe's architecture team starts work on Version 1 of the Car-Wash architecture, it agrees governance procedures for the architecture development itself at an early stage. A later stage of the architecture development includes the definition of governance procedures for the implementation of solutions that are instantiations of the architecture.

As the business grows, it becomes increasingly reliant on the effectiveness of these procedures. At a typical moment when Joe is expanding internationally, he has proceeding at the same time:

- An Architecture Development project for Version 3 of the architecture, to incorporate country centers in addition to regional centers within each country;

- A project to document the architecture of a rival car-wash operation that Joe has just taken over, and to recommend a solution for integrating it with Joe's operation;
- A dozen solution development and implementation projects to meet a large number of varied requirements, some of which may lead to architecture development work to produce minor version updates (the architecture is currently at Version 2.3);
- A large number of implementation projects to upgrade to the current version some sites that still assume earlier versions of the architecture (many sites are at Version 2.1 or 2.2, and some are still at Version 1);
- A problem management process in which a team is working its way through a list of bug reports, to which new reports are constantly added (there are currently 152 unresolved bugs listed).

This makes the architecture development and implementation a very complex process indeed. In general, a project affects multiple systems, and a system is affected by several projects. At any given time, each person or team is usually working on multiple projects. And a service must often be designed to meet the needs of multiple projects. Without good governance, the situation would quickly degenerate into chaos.

5.4.3 The *Governance Regime* Class

```
<owl:Class rdf:ID="GovernanceRegime">
   <rdfs:subClassOf rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
   <owl:disjointWith rdf:resource="#Policy"/>
   <owl:disjointWith rdf:resource="#Abstraction"/>
   <owl:disjointWith rdf:resource="#Realization"/>
   <owl:disjointWith rdf:resource="#Architecture"/>
</owl:Class>
```

A *governance regime* is a system that controls the conduct of an activity. This concept is captured by the **Governance Regime** class.

A governance regime has components that are rules and activities. The **Governance Regime** class is a subclass of the **System** class. A governance regime can be a solution (satisfying requirements for governance) and can be an abstraction or a realization. The Governance Regime class is not disjoint with the **Solution**, **Abstraction** and **Realization** classes, but is disjoint with the other top-level classes defined in this ontology.

Control of the conduct of an activity can include constraining the actions performed by the actors that take part in it, and limiting the effects of those actions and of the activity.

Kimi ensures that there is a *governance regime* that controls all of the activities concerned with architecture development, solution design, solution implementation, and system operation. This regime includes rules about labelling the things produced by these activities, keeping track of versions, providing traceability between solutions and requirements, and performing quality checks. It also includes activities that check that the rules are followed, such as authorization procedures for activities, sign-off procedures for deliverables, and quality review procedures. And it includes an Architecture Board that monitors these activities, interprets the rules and, if necessary, changes them. It is a complex system, but necessary to the successful conduct of Joe's business.

5.4.4 The *governs* and *is governed by* Properties

```
<owl:ObjectProperty rdf:ID="governs">
   <rdfs:domain rdf:resource="#GovernanceRegime"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#isGovernedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isGovernedBy">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#GovernanceRegime"/>
   <owl:inverseOf rdf:resource="#governs"/>
</owl:ObjectProperty>
```

The **governs** property, and its inverse **is governed by**, capture the relationship between a governance regime that controls how an activity is carried out and that activity.

The governance regime instituted by Kimi *governs* the architecture development, solution design, solution implementation, and system operation activities of Joe's car-wash enterprise.

5.4.5 The *Governance Rule* Class

```
<owl:Class rdf:ID="GovernanceRule">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Rule" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isComponentOf" />
      <owl:someValuesFrom rdf:resource="#GovernanceRegime" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

A *governance rule* is a rule that is part of a governance regime. This concept is captured by the **Governance Rule** class, which is defined as the class of rules that are components of governance regimes.

Joe's governance regime includes *governance rules* such as "No architecture development activity takes place until it is approved by the Architecture Board", and "Installation of a system is signed off by the site manager".

5.4.6 The *Governance Activity* Class

```
<owl:Class rdf:ID="GovernanceActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isComponentOf" />
      <owl:someValuesFrom rdf:resource="#GovernanceRegime" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>
```

A *governance activity* is an activity that is part of a governance regime. This concept is captured by the **Governance Activity** class, which is defined as the class of activities that are components of governance regimes.

The authorization, sign-off and quality review procedures of Joe's governance regime, and the activities of his Architecture Board, are all *governance activities*.

5.5 Service-Oriented Architecture

5.5.1 Overview

Service-Oriented Architecture (SOA) is an architectural style that supports service orientation.

Service orientation is a way of thinking in terms of services and service-based development and the outcomes of services.

An *architectural style* is the combination of distinctive features in which architecture is performed or expressed.

The SOA architectural style has the following distinctive features:

- It is based on the design of the services – which mirror real-world business activities – comprising the enterprise (or inter-enterprise) business processes.
- Service representation utilizes business descriptions to provide context (i.e., business process, goal, rule, policy, service interface, and service component) and implements services using service orchestration.
- It places unique requirements on the infrastructure – it is recommended that implementations use open standards to realize interoperability and location transparency.
- Implementations are environment-specific – they are constrained or enabled by context and must be described within that context.
- It requires strong governance of service representation and implementation.
- It requires a "Litmus Test", which determines a "good service".

This section describes the **Service Oriented Architecture** class. The instances of this class are architectures in the service-oriented style.

The SOA architectural style is based on the design of services that mirror real-world business activities. The concept of service was introduced in Section 2.2.3. This is a concept that comes from the world of business and is used also in the world of IT. The **Service** class can have instances that are real-world business activities and can also have instances that are software services that mirror those activities. The concept of an activity being a business activity for an enterprise (or other actor) is described in Section 3.3.

Service representation utilizes business descriptions to provide context. The concept of description is described in Section 2.5.5 and the concept of a description that is a registry entry for a service is described in Section 4.10.3. A formal description could use the terms of this ontology, such as **business activity** (Section 3.3.3), **Rule** (Section 3.5.3), **Policy** (Section 3.5.9), **Interface** (Section 3.4.3), and **component** (Section 2.6.5).

Service-based solutions can be implemented using orchestration (Section 4.8.2), and also using choreography (Section 4.8.4), messaging (Section 4.9), discovery (Section 4.10), and virtualization (Section 4.11).

SOA places unique requirements on the infrastructure. The concept of an architecture's infrastructure is described in Section 5.2.6. The building blocks that are the infrastructure of a service-oriented architecture are likely to include messaging services (Section 4.9.5), registry services (Section 4.10.9), and other activities that are fundamental to the development and operation of the architecture's systems. It is also likely to include technology actors (Section 4.6.3) that take part in development and operation activities.

Implementations (Section 4.5) are constrained or enabled by context. They are realizations (Section 4.3.3) of solutions (Section 4.2.4) that satisfy requirements (Section 4.2.2) that are defined by the context.

SOA requires strong governance of service representation and implementation. Governance is provided by governance rules (Section 5.4.5) and governance activities that enforce those rules (Section 5.4.6) within the context of a governance regime (Section 5.4.3).

SOA requires a "Litmus Test", which determines a "good service". This ontology does not help architects and designers to determine what makes a "good service". This is a crucial aspect of service-oriented architecture, but it is beyond the scope of usefulness of a formal ontology.

This section introduces just one formal definition, that of the **Service Oriented Architecture** class, which captures the concept of a service-oriented architecture.

5.5.2 The *Service Oriented Architecture* Class

```
<owl:Class rdf:ID="ServiceOrientedArchitecture">
   <rdfs:subClassOf rdf:resource="#Architecture"/>
</owl:Class>
```

A *service-oriented architecture* is an architecture that is based on the principle of service orientation. This concept is captured by the **Service Oriented Architecture** class, which is defined as a subclass of the **Architecture** class.

Joe's car-wash architecture is a *service-oriented architecture*.

An *architecture* is the fundamental organization of a system embodied in its components, their relationships to each other and the environment, and the principles guiding its design and evolution. (See [IEEE 1471].)

The principles guiding the design and evolution of Joe's car-wash architecture include some basic SOA principles:

- It is based on the design of services – which mirror real-world business activities – comprising the enterprise (or inter-enterprise) business processes;
- The services are loosely-coupled with simple interfaces that are formally described;
- Service composition is used to define solutions to meet requirements;
- Service virtualization is used to optimize use of resources.

(In this example, dynamic discovery is not a principle of the architecture. The architecture team has not identified it as being useful in the context of Joe's business. SOA is not a "one size fits all" approach: different sets of SOA principles are appropriate for different enterprises.)

The components of Joe's car-wash architecture are architecture building blocks, which include:

- Building blocks of his business architecture, such as **car-wash services**, **loyalty program**, **partner relations**, and **accounts**;
- Information system architecture building blocks, such as **car-wash operation**, which has components like **car-wash control**, **car-wash booking**, and **payment services**;
- Technology architecture building blocks such as **car-wash machine**, and **services bus**.

The architecture description shows how these building blocks relate to each other and to the environment.

A number of building blocks, such as the **car-wash machine** and **services bus** technology architecture building blocks, are identified as infrastructure of the architecture.

Solutions that meet Joe's business requirements are designed and implemented as instantiations of this architecture.

A The OWL Definition of the Ontology

The ontology is available online at **http://www.opengroup.org/projects/soa-ontology/uploads/40/16940/draft200.owl** and is reproduced below.

```
<?xml version="1.0"?>
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns="http://www.opengroup.org/soa/v01.owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xml:base="http://www.opengroup.org/projects/soa-
ontology/uploads/40/16940/draft200.owl">
  <owl:Ontology rdf:about=""/>

<owl:Class rdf:ID="Service">
   <rdfs:subClassOf rdf:resource="#Activity"/>
</owl:Class>

<owl:Class rdf:ID="Actor"/>

<owl:ObjectProperty rdf:ID="provides">
   <rdfs:domain rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isProvidedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isProvidedBy">
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#provides"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Service">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#isProvidedBy"/>
         <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="consumes">
   <rdfs:domain rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isConsumedBy"/>
</owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:ID="isConsumedBy">
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#consumes"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Effect">
   <owl:disjointWith rdf:resource="#Service"/>
   <owl:disjointWith rdf:resource="#Actor"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="isEffectOf">
   <rdfs:domain rdf:resource="#Effect"/>
   <owl:inverseOf rdf:resource="#hasEffect"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasEffect">
   <rdfs:range rdf:resource="#Effect"/>
   <owl:inverseOf rdf:resource="#isEffectOf"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Service">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasEffect"/>
         <owl:minCardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:minCardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Change">
   <owl:disjointWith rdf:resource="#Service"/>
   <owl:disjointWith rdf:resource="#Actor"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="isChangeTo">
   <rdfs:domain rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isChangedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isChangedBy">
   <rdfs:range rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isChangeTo"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Change">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#isChangeTo"/>
         <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
      </owl:Restriction>
```

```
   </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="InformationItem">
   <owl:disjointWith rdf:resource="#Service"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
</owl:Class>

<owl:Class rdf:ID="InformationType">
   <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasType">
   <owl:inverseOf rdf:resource="#isTypeOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isTypeOf">
   <owl:inverseOf rdf:resource="#hasType"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#InformationItem">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasType"/>
         <owl:allValuesFrom rdf:resource="#InformationType"/>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Description">
   <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="describes">
   <rdfs:domain rdf:resource="#Description"/>
   <owl:inverseOf rdf:resource="#isDescribedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isDescribedBy">
   <rdfs:range rdf:resource="#Description"/>
   <owl:inverseOf rdf:resource="#describes"/>
</owl:ObjectProperty>

<owl:Class rdf:about="#Description">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#describes"/>
         <owl:cardinality
            rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
            >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
```

```
</owl:Class>

<owl:Class rdf:ID="System">
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasComponent">
   <owl:inverseOf rdf:resource="#isComponentOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isComponentOf">
   <owl:inverseOf rdf:resource="#hasComponent"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Composition">
   <rdfs:subClassOf rdf:resource="#System"/>
</owl:Class>

<owl:FunctionalProperty rdf:ID="produces">
   <owl:inverseOf rdf:resource="#isProducedBy"/>
</owl:FunctionalProperty>

<owl:ObjectProperty rdf:ID="isProducedBy">
   <owl:inverseOf rdf:resource="#produces"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Activity">
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#Actor"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="takesPartIn">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#hasParticipant"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasParticipant">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#takesPartIn"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Event">
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Activity"/>
</owl:Class>
```

```
<owl:ObjectProperty rdf:ID="respondsTo">
   <rdfs:domain>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Activity"/>
            <owl:Class rdf:about="#Actor"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:domain>
   <rdfs:range rdf:resource="#Event"/>
   <owl:inverseOf rdf:resource="#isRespondedToBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRespondedToBy">
   <rdfs:domain rdf:resource="#Event"/>
   <rdfs:range>
      <owl:Class>
         <owl:unionOf rdf:parseType="Collection">
            <owl:Class rdf:about="#Activity"/>
            <owl:Class rdf:about="#Actor"/>
         </owl:unionOf>
      </owl:Class>
   </rdfs:range>
   <owl:inverseOf rdf:resource="#respondsTo"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isTheOccurrenceOf">
   <rdfs:domain rdf:resource="#Change"/>
   <rdfs:range rdf:resource="#Event"/>
   <owl:inverseOf rdf:resource="#occursAs"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="occursAs">
   <rdfs:domain rdf:resource="#Event"/>
   <rdfs:range rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isTheOccurrenceOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isBusinessActivityOf">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#hasBusinessActivity"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasBusinessActivity">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#isBusinessActivityOf"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Interface">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
```

```
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="isInterfaceOf">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#hasInterface"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInterface">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#isInterfaceOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isEventAt">
   <rdfs:domain rdf:resource="#Event"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#includesEvent"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="includesEvent">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#Event"/>
   <owl:inverseOf rdf:resource="#isEventAt"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isInputAt">
   <rdfs:domain rdf:resource="#InformationType"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#hasInputInformation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInputInformation">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#InformationType"/>
   <owl:inverseOf rdf:resource="#isInputAt"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isOutputAt">
   <rdfs:domain rdf:resource="#InformationType"/>
   <rdfs:range rdf:resource="#Interface"/>
   <owl:inverseOf rdf:resource="#hasOutputInformation"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasOutputInformation">
   <rdfs:domain rdf:resource="#Interface"/>
   <rdfs:range rdf:resource="#InformationType"/>
   <owl:inverseOf rdf:resource="#isOutputAt"/>
</owl:ObjectProperty>
```

```
<owl:Class rdf:ID="Rule">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="appliesTo">
   <rdfs:domain rdf:resource="#Rule"/>
   <owl:inverseOf rdf:resource="#isAffectedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isAffectedBy">
   <rdfs:range rdf:resource="#Rule"/>
   <owl:inverseOf rdf:resource="#appliesTo"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasCondition">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Contract"/>
        <owl:Class rdf:about="#Policy"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Rule"/>
  <owl:inverseOf rdf:resource="#isConditionOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isConditionOf">
  <rdfs:domain rdf:resource="#Rule"/>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Contract"/>
        <owl:Class rdf:about="#Policy"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
  <owl:inverseOf rdf:resource="#hasCondition"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Contract">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
```

```
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="isContractFor">
  <rdfs:domain rdf:resource="#Contract"/>
  <owl:inverseOf rdf:resource="#isSubjectOfContract"/>
</owl:ObjectProperty>

<owl:FunctionalProperty rdf:about="#isContractFor" />

<owl:ObjectProperty rdf:ID="isSubjectOfContract">
   <rdfs:range rdf:resource="#Contract"/>
   <owl:inverseOf rdf:resource="#isContractFor"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isPartyTo">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Contract"/>
   <owl:inverseOf rdf:resource="#hasParty"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasParty">
   <rdfs:domain rdf:resource="#Contract"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isPartyTo"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Policy">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasPolicy">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Policy"/>
   <owl:inverseOf rdf:resource="#isPolicyOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isPolicyOf">
  <rdfs:domain rdf:resource="#Policy"/>
  <rdfs:range rdf:resource="#Actor"/>
  <owl:inverseOf rdf:resource="#hasPolicy"/>
</owl:ObjectProperty>
```

```
<owl:FunctionalProperty rdf:about="#isPolicyOf" />

<owl:ObjectProperty rdf:ID="isPolicyFor">
   <rdfs:domain rdf:resource="#Policy"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#isSubjectOfPolicy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isSubjectOfPolicy">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#Policy"/>
   <owl:inverseOf rdf:resource="#isPolicyFor"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Requirement">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
   <owl:disjointWith rdf:resource="#Policy"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="requires">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Requirement"/>
   <owl:inverseOf rdf:resource="#isRequiredBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRequiredBy">
   <rdfs:domain rdf:resource="#Requirement"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#requires"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Solution">
</owl:Class>

<owl:ObjectProperty rdf:ID="satisfies">
   <rdfs:domain rdf:resource="#Solution"/>
   <rdfs:range rdf:resource="#Requirement"/>
   <owl:inverseOf rdf:resource="#isSatisfiedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isSatisfiedBy">
   <rdfs:domain rdf:resource="#Requirement"/>
   <rdfs:range rdf:resource="#Solution"/>
   <owl:inverseOf rdf:resource="#satisfies"/>
```

```
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="entails">
   <rdfs:domain rdf:resource="#Solution"/>
   <rdfs:range rdf:resource="#Change"/>
   <owl:inverseOf rdf:resource="#isEntailedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isEntailedBy">
   <rdfs:domain rdf:resource="#Change"/>
   <rdfs:range rdf:resource="#Solution"/>
   <owl:inverseOf rdf:resource="#entails"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="SolutionBuildingBlock">
  <rdfs:subClassOf rdf:resource="#Abstraction" />
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isComponentOf" />
      <owl:someValuesFrom rdf:resource="#Solution" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Abstraction">
   <owl:disjointWith rdf:resource="#Requirement"/>
</owl:Class>

<owl:Class rdf:ID="Realization">
   <owl:disjointWith rdf:resource="#Abstraction"/>
   <owl:disjointWith rdf:resource="#Requirement"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="isAbstractionOf">
   <rdfs:domain rdf:resource="#Abstraction"/>
   <rdfs:range rdf:resource="#Realization"/>
   <owl:inverseOf rdf:resource="#isRealizationOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRealizationOf">
   <rdfs:domain rdf:resource="#Realization"/>
   <rdfs:range rdf:resource="#Abstraction"/>
   <owl:inverseOf rdf:resource="#isAbstractionOf"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Design">
   <rdfs:subClassOf rdf:resource="#Abstraction"/>
   <rdfs:subClassOf rdf:resource="#Solution"/>
</owl:Class>

<owl:Class rdf:ID="DesignActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
```

```
      <owl:onProperty rdf:resource="#hasEffect" />
      <owl:someValuesFrom>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#isChangeTo" />
          <owl:someValuesFrom rdf:resource="#Design" />
        </owl:Restriction>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="Implementation">
  <owl:equivalentClass>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isRealizationOf" />
      <owl:someValuesFrom rdf:resource="#Design" />
    </owl:Restriction>
  </owl:equivalentClass>
</owl:Class>

<owl:Class rdf:ID="ImplementationActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasEffect" />
      <owl:someValuesFrom>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#isChangeTo" />
          <owl:someValuesFrom rdf:resource="#Implementation" />
        </owl:Restriction>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="HumanActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
</owl:Class>

<owl:Class rdf:ID="TechnologyActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#HumanActor"/>
</owl:Class>

<owl:Class rdf:ID="SoftwareActor">
   <rdfs:subClassOf rdf:resource="#TechnologyActor"/>
</owl:Class>

<owl:Class rdf:ID="OrganizationActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
   <rdfs:subClassOf rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#HumanActor"/>
   <owl:disjointWith rdf:resource="#TechnologyActor"/>
</owl:Class>
```

```
<owl:Class rdf:ID="Enterprise">
   <rdfs:subClassOf rdf:resource="#OrganizationActor"/>
</owl:Class>

<owl:Class rdf:ID="SoftwareService">
   <rdfs:subClassOf rdf:resource="#Service"/>
<rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParticipant" />
      <owl:someValuesFrom rdf:resource="#SoftwareActor" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParticipant" />
      <owl:allValuesFrom rdf:resource="#SoftwareActor" />
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Orchestration">
  <rdfs:subClassOf rdf:resource="#Composition"/>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#hasComponent" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#produces" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasDirectionActivity" />
      <owl:cardinality
          rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasDirectionActivity">
  <rdfs:subPropertyOf rdf:resource="#hasComponent"/>
  <rdfs:domain rdf:resource="#Orchestration"/>
  <rdfs:range rdf:resource="#Activity"/>
  <owl:inverseOf rdf:resource="#isDirectionActivityOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isDirectionActivityOf">
  <rdfs:subPropertyOf rdf:resource="#isComponentOf"/>
```

```
  <rdfs:domain rdf:resource="#Activity"/>
  <rdfs:range rdf:resource="#Orchestration"/>
  <rdfs:domain rdf:resource="#Orchestration"/>
  <rdfs:range rdf:resource="#Activity"/>
  <owl:inverseOf rdf:resource="#hasDirectionActivity"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Choreography">
  <rdfs:subClassOf rdf:resource="#Composition"/>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#hasComponent" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
  <owl:Restriction>
      <owl:onProperty rdf:resource="#produces" />
      <owl:allValuesFrom rdf:resource="#Activity" />
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasDirectionActivity" />
      <owl:cardinality
         rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
         >0</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Message">
  <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>

<owl:Class rdf:ID="MessageType">
  <rdfs:subClassOf rdf:resource="#InformationType"/>
</owl:Class>

<owl:Class rdf:about="#Message">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasType"/>
         <owl:someValuesFrom rdf:resource="#MessageType" />
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="MessagingService">
  <rdfs:subClassOf rdf:resource="#Service"/>
</owl:Class>

<owl:Class rdf:ID="MessagingInterface">
  <rdfs:subClassOf rdf:resource="#Interface"/>
```

```
  <rdfs:subClassOf>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasInputInformation"/>
          <owl:someValuesFrom rdf:resource="#MessageType" />
        </owl:Restriction>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#hasOutputInformation"/>
          <owl:someValuesFrom rdf:resource="#MessageType" />
        </owl:Restriction>
      </owl:unionOf>
    </owl:Class>
  </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:about="#MessagingService">
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasInterface"/>
         <owl:someValuesFrom rdf:resource="#MessagingInterface" />
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="Registry">
   <rdfs:subClassOf rdf:resource="#InformationItem"/>
</owl:Class>

<owl:Class rdf:ID="RegistryEntry">
   <rdfs:subClassOf rdf:resource="#Description"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#isContainedIn"/>
         <owl:cardinality
           rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
           >1</owl:cardinality>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="contains">
  <rdfs:domain rdf:resource="#Registry"/>
  <rdfs:range rdf:resource="#RegistryEntry"/>
  <owl:inverseOf rdf:resource="#isContainedIn"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isContainedIn">
  <rdfs:domain rdf:resource="#RegistryEntry"/>
  <rdfs:range rdf:resource="#Registry"/>
  <owl:inverseOf rdf:resource="#contains"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="registers">
```

```
   <rdfs:domain rdf:resource="#Registry"/>
   <rdfs:range rdf:resource="#Service"/>
   <owl:inverseOf rdf:resource="#isRegisteredIn"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isRegisteredIn">
   <rdfs:domain rdf:resource="#Service"/>
   <rdfs:range rdf:resource="#Registry"/>
   <owl:inverseOf rdf:resource="#register"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="Visibility">
   <rdfs:subClassOf rdf:resource="#InformationType"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="isInScopeOf">
   <rdfs:domain rdf:resource="#Actor"/>
   <rdfs:range rdf:resource="#Visibility"/>
   <owl:inverseOf rdf:resource="#hasInScope"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasInScope">
   <rdfs:domain rdf:resource="#Visibility"/>
   <rdfs:range rdf:resource="#Actor"/>
   <owl:inverseOf rdf:resource="#isInScopeOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="hasVisibility">
   <rdfs:subPropertyOf rdf:resource="#hasType"/>
   <rdfs:domain rdf:resource="#RegistryEntry"/>
   <rdfs:range rdf:resource="#Visibility"/>
   <owl:inverseOf rdf:resource="#isVisibilityOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isVisibilityOf">
   <rdfs:subPropertyOf rdf:resource="#isTypeOf"/>
   <rdfs:domain rdf:resource="#Visibility"/>
   <rdfs:range rdf:resource="#RegistryEntry"/>
   <owl:inverseOf rdf:resource="#hasVisibility"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="RegistryService">
   <rdfs:subClassOf rdf:resource="#Service"/>
</owl:Class>

<owl:Class rdf:ID="VirtualActor">
   <rdfs:subClassOf rdf:resource="#Actor"/>
   <rdfs:subClassOf rdf:resource="#System"/>
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasComponent"/>
         <owl:allValuesFrom rdf:resource="#Actor"/>
      </owl:Restriction>
   </rdfs:subClassOf>
```

```
   <rdfs:subClassOf>
      <owl:Restriction>
         <owl:onProperty rdf:resource="#hasComponent"/>
         <owl:allValuesFrom rdf:resource="#Realization"/>
      </owl:Restriction>
   </rdfs:subClassOf>
</owl:Class>

<owl:Class rdf:ID="VirtualizedService">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Service" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParticipant" />
      <owl:someValuesFrom rdf:resource="#VirtualActor" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="Architecture">
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#Composition"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
   <owl:disjointWith rdf:resource="#Policy"/>
   <owl:disjointWith rdf:resource="#Abstraction"/>
   <owl:disjointWith rdf:resource="#Realization"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="hasArchitecture">
   <rdfs:domain rdf:resource="#System"/>
   <rdfs:range rdf:resource="#Architecture"/>
   <owl:inverseOf rdf:resource="#isArchitectureOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isArchitectureOf">
   <rdfs:domain rdf:resource="#Architecture"/>
   <rdfs:range rdf:resource="#System"/>
   <owl:inverseOf rdf:resource="#hasArchitecture"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="ArchitectureBuildingBlock">
   <rdfs:subClassOf rdf:resource="#Abstraction"/>
   <rdfs:subClassOf>
     <owl:Restriction>
       <owl:onProperty rdf:resource="#isComponentOf" />
       <owl:someValuesFrom rdf:resource="#Architecture" />
     </owl:Restriction>
   </rdfs:subClassOf>
```

```
</owl:Class>


<owl:ObjectProperty rdf:ID="hasInfrastructure">
   <rdfs:subPropertyOf rdf:resource="#hasComponent"/>
   <rdfs:domain rdf:resource="#Architecture"/>
   <rdfs:range rdf:resource="#ArchitectureBuildingBlock"/>
   <owl:inverseOf rdf:resource="#isInfrastructureOf"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isInfrastructureOf">
   <rdfs:subPropertyOf rdf:resource="#isComponentOf"/>
   <rdfs:domain rdf:resource="#ArchitectureBuildingBlock"/>
   <rdfs:range rdf:resource="#Architecture"/>
   <owl:inverseOf rdf:resource="#hasInfrastructure"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="ArchitectureDevelopmentActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasEffect" />
      <owl:someValuesFrom>
        <owl:Restriction>
          <owl:onProperty rdf:resource="#isChangeTo" />
          <owl:someValuesFrom rdf:resource="#Architecture" />
        </owl:Restriction>
      </owl:someValuesFrom>
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:ObjectProperty rdf:ID="instantiates">
  <rdfs:domain rdf:resource="#Solution"/>
  <rdfs:range>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Architecture"/>
        <owl:Class rdf:about="#ArchitectureBuildingBlock"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:range>
  <owl:inverseOf rdf:resource="#isInstantiatedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isInstantiatedBy">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Architecture"/>
        <owl:Class rdf:about="#ArchitectureBuildingBlock"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
```

```
  <rdfs:range rdf:resource="#Solution"/>
  <owl:inverseOf rdf:resource="#instantiates"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="GovernanceRegime">
   <rdfs:subClassOf rdf:resource="#System"/>
   <owl:disjointWith rdf:resource="#Activity"/>
   <owl:disjointWith rdf:resource="#Actor"/>
   <owl:disjointWith rdf:resource="#Effect"/>
   <owl:disjointWith rdf:resource="#Change"/>
   <owl:disjointWith rdf:resource="#InformationItem"/>
   <owl:disjointWith rdf:resource="#Event"/>
   <owl:disjointWith rdf:resource="#Interface"/>
   <owl:disjointWith rdf:resource="#Rule"/>
   <owl:disjointWith rdf:resource="#Contract"/>
   <owl:disjointWith rdf:resource="#Policy"/>
   <owl:disjointWith rdf:resource="#Abstraction"/>
   <owl:disjointWith rdf:resource="#Realization"/>
   <owl:disjointWith rdf:resource="#Architecture"/>
</owl:Class>

<owl:ObjectProperty rdf:ID="governs">
   <rdfs:domain rdf:resource="#GovernanceRegime"/>
   <rdfs:range rdf:resource="#Activity"/>
   <owl:inverseOf rdf:resource="#isGovernedBy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isGovernedBy">
   <rdfs:domain rdf:resource="#Activity"/>
   <rdfs:range rdf:resource="#GovernanceRegime"/>
   <owl:inverseOf rdf:resource="#governs"/>
</owl:ObjectProperty>

<owl:Class rdf:ID="GovernanceRule">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Rule" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isComponentOf" />
      <owl:someValuesFrom rdf:resource="#GovernanceRegime" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="GovernanceActivity">
  <owl:intersectionOf rdf:parseType="Collection">
    <owl:Class rdf:about="#Activity" />
    <owl:Restriction>
      <owl:onProperty rdf:resource="#isComponentOf" />
      <owl:someValuesFrom rdf:resource="#GovernanceRegime" />
    </owl:Restriction>
  </owl:intersectionOf>
</owl:Class>

<owl:Class rdf:ID="ServiceOrientedArchitecture">
```

```
   <rdfs:subClassOf rdf:resource="#Architecture"/>
</owl:Class>

</rdf:RDF>
```