Technical Standard

Service-Oriented Architecture Ontology

Technical Standard

Service-Oriented Architecture Ontology

ISBN: TBA

Document Number: TBA

Published by The Open Group, Month 2010.

Comments relating to the material contained in this document may be submitted to:

The Open Group
Thames Tower
37-45 Station Road
Reading
Berkshire, RG1 1LX
United Kingdom

or by electronic mail to:

ogspecs@opengroup.org

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 a draft Technical Standard for Service-Oriented Architecture Ontology. It has been developed by the Service-Oriented Infrastructure project of The Open Group’s SOA

Working Group. It is a draft that is made available for comment, and is not a formal Open Group publication. An Open Group Publication may eventually result, following comment, revision, and review. That publication will not necessarily reflect the contents of this draft in any way.

Trademarks

Referenced Documents

The following documents are referenced in this Technical Standard:

[BPMN] Business Process Modeling Notation, Version 1.1, Object Management Group; available from www.omg.org

[OASIS RM] OASIS Reference Model for Service-Oriented Architecture, Version 1.0, Organization for the Advancement of Structured Information Standards; available from www.oasis-open.org

[OWL] OWL Web Ontology Language Reference, W3C Recommendation, 10 February 2004, World-Wide Web Consortium; available from www.w3.org/TR/owl-ref

[SoaML] Service-Oriented Architecture Modeling Language, Object Management Group; available from www.omg.org

[TOGAF] The Open Group Architecture Framework, The Open Group; available from www.opengroup.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 Service-Oriented Architecture (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
   - — 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 concepts and how they are used in the enterprise and its environment
- Architects, as metadata for architectural artifacts
- Architecture methodologists, as a component of SOA meta models
- System and software designers for guidance in terminology and structure

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 about a composite system in terms of services, processes, events, policies etc.

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 core 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 sub-languages: OWL-Lite, OWL-DL, and OWL-Full. This ontology uses OWL-DL, the sub-language that provides the greatest expressiveness possible while retaining computational completeness and decidability.

This document is structured as follows:

- This chapter provides an introduction to the whole document.
- Chapter 2….
- Appendix A contains the formal OWL definitions of the ontology, collected together.
- Appendix B describes the relation of this ontology to other work.

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 examples 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 examples provided in this Technical Standard are describing one way in which the ontology could be applied in practical situations. Different applications of the ontology to the same situations would nevertheless be possible. The precise instantiation of the ontology in particular practical situations is a matter for users of the ontology; as long as the concepts and constraints defined by the ontology are correctly applied, the instantiation is valid.

1.4 Conformance

There are two kinds of applications that can potentially conform to this ontology. One is other OWL based ontologies (typically extensions of the SOA ontology). The other is a non-OWL application such as a meta model or a piece of software.

A conforming OWL application (derived OWL based ontology):

- Must conform to the OWL standard

- Must include (in the OWL sense) the whole of the ontology contained in Appendix A of this Technical Standard
- Can add other OWL constructs, including class and property definitions
- Can import other ontologies in addition to the SOA ontology

A conforming non-OWL application:

- Must include a defined and consistent transform to the ontology contained in Appendix A of this Technical Standard
- May intersect only a subset of the SOA ontology constructs
- Can add other constructs, including class and property definitions
- Can leverage other ontologies in addition to the SOA ontology

1.5 Terminology

The words and phrases MUST, REQUIRED, SHALL, MUST NOT, SHALL, NOT, SHOULD, RECOMMENDED, SHOULD NOT, NOT RECOMMENDED and MAY are used in this Technical Standard with the meanings defined in IETF RFC 2119.

1.6 Typographical 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 Technical Standard describes a formal OWL ontology. For purposes of exposition, it includes UML diagrams that illustrate the classes and properties of the ontology.

Note that these diagrams make some simplifications. The OWL definitions contained in this specification constitute the authoritative definition of the ontology.

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 System and Element

2.1 Introduction

System and *Element* are two of the core concepts of this ontology. Both are concepts that are well-understood by practitioners, including the notion that systems have elements and that systems can be hierarchically combined (systems of systems). What differs from domain to domain is the specific nature of systems and elements, for instance an electrical system has very different kinds of elements than an SOA system.

In the ontology only elements and systems within the SOA domain are considered. Some SOA sub-domains use the term *component* rather than the term *element*. This is not contradictory, as any *component* of an SOA system is also an *element* of that (composite) system.

This chapter describes the following classes of the ontology:

- **Element**
- **System**

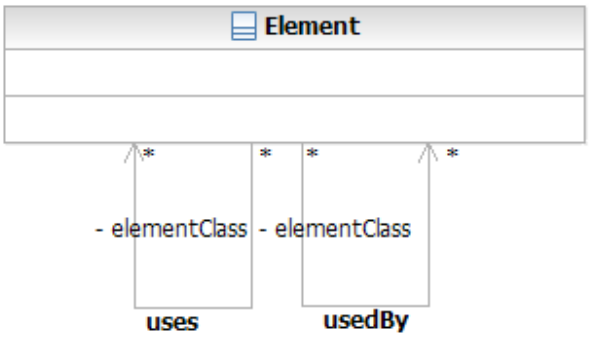
In addition if defines the following properties:

- **uses** and **usedBy**
- **represents** and **representedBy**

2.2 The *Element* Class

```
<owl:Class rdf:about="#Element">
</owl:Class>
```

An *element* is an opaque entity that is atomic at a given level of abstraction. The *element* has a clearly defined boundary. By its nature, to an external observer any possible internal structure (inside the boundary) of the *element* is unknown. The concept of *element* is captured by the **Element** class, which is illustrated in the figure below.



In the context of the SOA ontology we consider mainly elements that belong to the SOA domain. Furthermore we adopt a mainly functional view point, meaning that we consider mainly functional elements.

Elements are not exclusive to the four named subclasses (System, Actor, Task and Service) described later in this ontology. Other examples of elements are things like software components or technology components (such as Enterprise Service Bus's etc.)

2.2.1 The *uses* and *usedBy* Properties

```
<owl:ObjectProperty rdf:about="#uses">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="usedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="uses"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Elements may use other elements in various ways. In general the notion of some element using another element is well understood by practitioners for all of models, executables and physical objects. What differs from domain to domain is the way in which such use is perceived. The **uses** property, and its inverse **usedBy**, capture the abstract notion of an element using another.

The **uses** property captures not just transient relations. Instantiations of the property can include "uses at this instant", "has used", and "may in future use".

2.2.2 Organizational example

Using an organizational example, typical instances of **Element** are organizational units and people. Whether to perceive a given part of an organization as an organizational unit or as the set of people within that organizational unit is an important choice of abstraction level:

- Inside the boundary of the organizational unit we want to express the fact that an organizational unit uses the people that are members of it. Note that the same person can in fact be a member of (be used by) multiple organizational units.

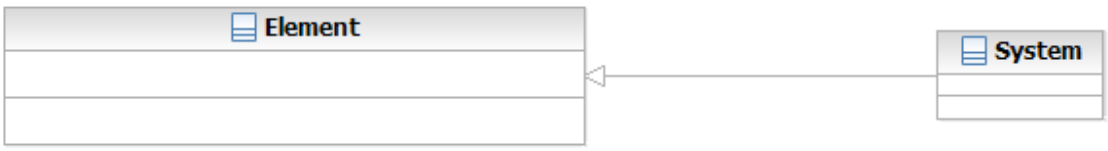
- Outside the boundary The internal structure of an organizational unit must remain opaque to an external observer, as the enterprise wants to be able to change the people within the organizational unit within having to change the definition of the organizational unit itself.

This simple example expresses that some elements have an internal structure. In fact from an internal perspective they are an organized collection of other simpler things (captured by the *System* class defined below).

2.3 The *System* Class

```
<owl:Class rdf:ID="System">
  <owl:disjointWith>
    <owl:Class rdf:ID="Task"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Service"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Element"/>
  </rdfs:subClassOf>
</owl:Class>
```

A *system* is an organized collection of other, simpler things. Specifically things in a system collection are instances of **Element**, each such instance being used by the system. The concept of *system* is captured by the **System** class, which is illustrated in the figure below.



In the context of the SOA ontology we consider only SOA based systems. Furthermore we adopt a mainly functional view point of those systems. Note that a fully described instance of **System** should have by its nature (as a collection) a *uses* relationship to at least one instance of **Element**.

Since System is a subclass of **Element**, all systems have a boundary and are opaque to an external observer (black box view). This excludes from the **System** class structures that have no boundary, such as The Universe. From an SOA perspective this is not really a loss since all interesting SOA systems do have the characteristic of being possible to perceive from an outside (consumer) perspective. Furthermore having **System** as a subclass of **Element** allows us to

naturally express the notion of systems of systems – the lower level systems are simply elements used by the higher level system.

At the same time as supporting an external view point (black box view, see above) all systems must also support an internal view point (white box view) expressing how they are an organized collection. As an example for the notion of a service this would typically correspond to a service specification view versus a service realization view[1].

It is important to realize that even though systems using elements express an important aspect of the **uses** property it is not necessary to "invent" a system just to express that some element uses another. In fact even for systems we may need to be able to express that they can use elements outside their own boundary - though this in many cases will preferably be expressed not at the system level, but rather by an element of the system using that external **Element** instance.

System is defined as disjoint with the *Service* and *Task* classes. Instances of these classes are considered not to be collections of other things. **System** is specifically not defined as disjoint with the *Actor* class since an organization is many cases is in fact just a particular kind of system[2].

2.3.1 Examples

2.3.1.1 Organizational example

Continuing the organizational example from above, we can now express that an organizational unit as an instance of system has the people in it as members (and instances of element).

2.3.1.2 Service composition example

Using a service composition example, services A and B and instances of **Element** and the composition of A and B is an instance of **System** (that uses A and B).

See also below for a formal definition of the concepts of service and service composition (and a repeat of the example in that more precise concext).

2.3.1.3 Car wash example

Consider a car wash business. The company as a whole is an organizational unit and can be instantiated in the ontology in the following way:

- *CarWashBusiness* is an instance of **System**
- *Joe* (the owner) is an instance of **Element** and used by (owner of) *CarWashBusiness*
- *Mary* (the secretary) is an instance of **Element** and used by (employee of) *CarWashBusiness*

[1] Similar to the way that SOAML defines services as having both a black box/specification part and a white box/realization part.
[2] We choose not to define a special intersection class to represent this fact.

- *John* (the pre-wash guy) is an instance of **Element** and used by (employee of) *CarWashBusiness*

- *Jack* (the washing manager and operator) is an instance of **Element** and used by (employee of) *CarWashBusiness*

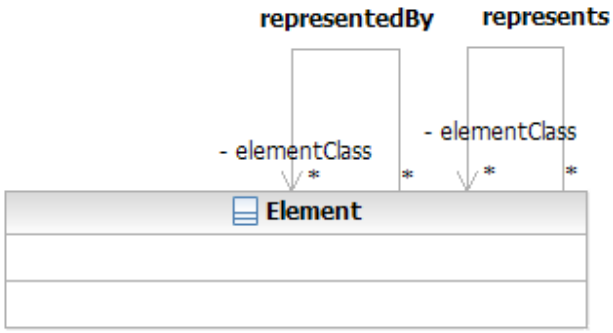
2.4 The *represents* and *representedBy* Properties

```
<owl:ObjectProperty rdf:about="#represents">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Element"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="representedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="represents"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

An SOA environment is intrinsically hierarchically composite[3], in other words the elements of SOA systems can be repeatedly composed to ever higher levels of abstraction. One aspect of has already been addressed by the **uses** and **usedBy** properties in that we can use these to express the notion of systems of systems. This is still a very concrete relationship though, and does not express the concept of architectural abstraction. We find the need for architectural abstraction in various places such as a role representing the people playing that role, an organizational unit representing the people within it[4], an architectural building block representing an underlying construct[5] and an Enterprise Service Bus representing the services that are accessible through it[6]. The concept of such an explicitly changing view point, or level of abstraction, is captured by the *represents* and *representedBy* properties illustrated in the figure below.

[3] See also Section 3.2 for a definition of the *Composition* class.
[4] Subtly different from that same organizational unit using the people within it, as the represents relationship indicates the organizational unit as a substitute interaction point.
[5] For instance important to enterprise architects wanting to explicitly distinguish between constructs and building blocks.
[6] For instance relevant when explicitly modeling operational interaction and dependencies.



It is important to understand the exact nature of the distinction between using an element E1 and using another element (E2) that represents E1. If E1 changes then anyone using E1 directly would experience a change but someone using E2 would not experience any change.

When applying the architectural abstraction via the **represents** property there are three different architectural choices that can be made:

- An element represents another element in a very literal way, simply by hiding the existence of that element and any changes to it. There will be a one-to-one relationship between the instance of **Element** and the (different) instance of **Element** that it represents. A simple real world example is the notion of a broker acting as an intermediary between a seller (that does not wish to be known) and a buyer.

- An element represents a particular aspect of another element. There will be a many-to-one relationship between many instances of **Element** (each of which represents a different aspect), and one (different) instance of **Element**. A simply real world example is the notion that the same person can play (be represented by) many different roles.

- An element is an abstraction that can represent many other elements. There will be a one-to-many relationship between one instance of **Element** (as an abstraction) and many other instances of **Element**. A simple real world example is the notion of an architectural blueprint representing an abstraction of many different buildings being built according to that blueprint.

Note that is most cases an instance of **Element** will represent only one kind of thing[7].

2.4.1 Examples

2.4.1.1 Organizational example

Expanding further on the organizational example, assume that a company desires to form a new organizational unit O1. There are two ways of doing this:

[7] Specifically an instance of **Element** will typically represent instances of at most one of the classes **System**, **Service**, **Actor** and **Task** (with the exception of the case where the same thing is both an instance of **System** and an instance of **Actor**). See later sections for the definitions of **Service**, **Actor** and **Task**.

- Define the new organization directly as a collection of people P1, P2, P3 and P4. This means that the new organization is perceived to be a leaf in the organizational hierarchy, and that any exchange of personnel means that its definition needs to change

- Define the new organization as a higher level organizational construct, joining together two existing organizations O3 and O4. Coincidentally O3 and O4 between them may have the same four people P1, P2, P3 and P4, but the new organization really doesn't know, and any member of O3 or O4 can be changed without needing to change the definition of the new organization. Furthermore, any member of O3 is intrinsically **not** working in the same organization as the members of O4 (in fact need not even be aware of them) – contrary to the first option where P1, P2, P3 and P4 are all colleagues in the same new organization.

In this way the abstraction aspect of the **represents** property induces an important difference in the semantics of the collection defining the new organization. Any instantiation of the ontology can and should use the **represents** and **representedBy** properties to crisply define the implied semantics and lines of visibility/change.

2.4.1.2 Car wash example

Joe chooses to organize his business into two organizational units, one for the administration and one for the actual washing of cars. This can be instantiated in the ontology in the following way:

- *CarWashBusiness* is an instance of **System**

- *AdministrativeSystem* is an instance of **System**

- *Administration* is an instance of **Element** that represents *AdministrativeSystem* (the opaque organizational unit aspect, aka ignoring anything else about *AdministrativeSystem*)

- *CarwashBusiness* uses (has organizational unit) *Administration*

- *CarWashSystem* is an instance of **System**

- *CarWash* is an instance of **Element** that represents *CarWashSystem* (the opaque organizational unit aspect, aka ignoring anything else about *CarWashSystem*)

- *CarWash* is a member of *CarWashBusiness*

- *Joe* (the owner) is an instance of **Element** and now used by *AdministrationSystem*

- *Mary* (the secretary) is an instance of **Element** and now used by *AdministrationSystem*

- *John* (the pre-wash guy) is an instance of **Element** and now used by *CarWashSystem*

- *Jack* (the wash manager and operator) is an instance of **Element** and now used by *CarWashSystem*

3 Actor and Task

3.1 Introduction

People, organizations, and the things they do are important aspects of SOA systems. *Actor* and *Task* capture this as another set of core concepts of the ontology. Both are concepts that are generic and have relevance outside the domain of SOA. For the purposes of this SOA ontology we have chosen to give them specific scope in that tasks are intrinsically atomic[8] and actors are restricted to people and organizations[9].

This chapter describes the following classes of the ontology:

- **Actor**
- **Task**

In addition if defines the following properties:

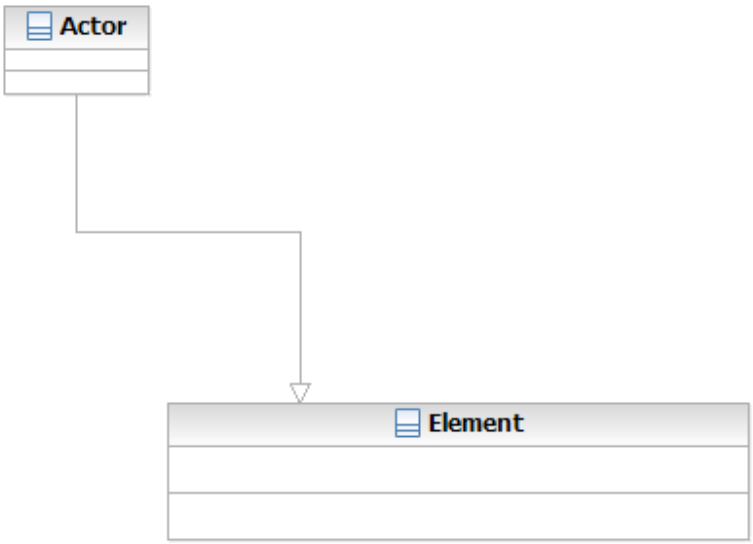
- **does** and **doneBy**

3.2 The *Actor* class

```
<owl:Class rdf:about="#Actor">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Element"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:ID="Task"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Service"/>
  </owl:disjointWith>
</owl:Class>
```

An *actor* is a person or an organization[10]. The concept of *actor* is captured by the **Actor** class, which is illustrated in the figure below.

[8] Corresponding to for instance the BPMN 2.0 definition of Task.
[9] Excluding for instance software.
[10] In principle this classification is not exhaustive as for example an animal might also be an actor. For the purposes of this ontology only the person or organizations kinds of actors are described.



Actor is defined as disjoint with the *Service*, and *Task* classes. Instances of these classes are considered not to be people or organizations. **Actor** is specifically not defined as disjoint with **System** since an organization in many cases is in fact just a particular kind of system[11].

3.2.1 Examples

*3.2.1.1 The **uses** and **usedBy** properties applied to **Actor***

In one direction an actor can itself use things such as services, systems and other actors. In the other direction an actor can for instance be used by another actor or by a system (as an element within that system such as an actor in a process)

*3.2.1.2 The **represents** and **representedBy** properties applied to **Actor***

As mentioned in the introduction to this section, actors are intrinsically part of SOA systems. Yet in many cases as an element of an SOA system we talk about not the specific person or organization, rather an abstract representation of them that participates in processes, provides services etc. In other words we talk about elements representing actors.

As examples a broker may represent a seller that wishes to remain anonymous, a role may represent (the role aspect of) multiple actors and an organizational unit may represent the many people that are part of it.

Note that we have chosen not to define a “role class”, as we believe that using **Element** with the **represents** property is a more general approach which does not limit the ability to also define role based systems. For all practical purposes there is simply a “role subclass” of **Element**, a subclass that we have chosen not to define explicitly.

[11] We choose not to define a special intersection class to represent this fact.

3.2.1.3 *Organizational example*

Continuing the organizational example from above, we can now express that P1, P2, P3 and P4 as instances of **Element** in fact are (people) instances of **Actor**. We can also express (if we so choose) that all of O1, O2, O3 and O4 are (organization) actors from an action perspective at the same time that they are systems from a collection/composition perspective.

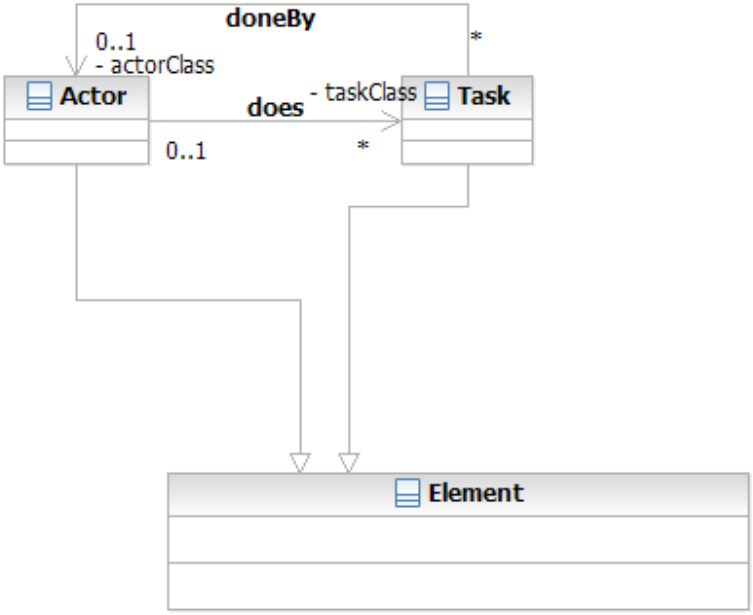
3.2.1.4 *Car wash example*

See section 8.1 for the complete organizational aspect of the car wash example.

3.3 The *Task* class

```
<owl:Class rdf:about="#Task">
  <owl:disjointWith>
    <owl:Class rdf:ID="System"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Actor"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Service"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Element"/>
  </rdfs:subClassOf>
</owl:Class>
```

A *task* is an atomic action which accomplishes a defined result. Tasks are done by people or organizations, specifically by instances of **Actor**. The concept of *task* is captured by the **Task** class, which is illustrated in the figure below.



Task is defined as disjoint with the *System*, *Service*, and *Actor* classes. Instances of these classes are considered not to be atomic actions.

3.3.1 The *does* and *doneBy* Properties

```
<owl:ObjectProperty rdf:about="#doneBy">
  <rdfs:domain rdf:resource="#Task"/>
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="does">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#doneBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>

<owl:Class rdf:ID="Task">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="doneBy"/>
```

```
          </owl:onProperty>
          <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >0</owl:minCardinality>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:maxCardinality>
          <owl:onProperty>
            <owl:ObjectProperty rdf:about="#doneBy"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>
```

Tasks are naturally thought of as being done by people or organizations. If we think of tasks as being the actual things done, then the natural cardinality is that each instance of **Task** is done by at most one instance of **Actor**[12]. On the other hand the same instance of **Actor** can (over time) easily do more than one instance of **Task**. The **does** property, and its inverse **doneBy**, capture the relation between an actor and the tasks it does.

3.3.2 Examples

*3.3.2.1 The **uses** and **usedBy** properties applied to **Task***

In one direction the most common case of a task using another element is where an automated task[13] uses a service as its realization. In the other direction a task can for instance be used by a system (as an element within that system such as a task in a process)

*3.3.2.2 The **represents** and **representedBy** properties applied to **Task***

As mentioned in the introduction to this section, tasks are intrinsically part of SOA systems. Yet in many cases as an element of an SOA system we talk about not the actual thing being done, rather an abstract representation of it that is used as an element in systems, processes etc. In other words we talk about elements representing tasks.

As a simple example an abstract activity in a process model (associated with a role) may represent a concrete task (done by a person fulfilling that role). Note that due to the atomic

[12] Due to the atomic nature of instances of **Task** we rule out the case where such an instance is done jointly by multiple instances of **Actor**. The cardinality can be zero if someone chooses not the instantiate all possible actors.
[13] In an orchestrated process, see Section 5 for the definition of *process* and *orchestration*

nature of a task it does not make sense to talk about many elements representing different aspects of it.

3.3.2.3 Organizational example

Continuing the organizational example from above, we can now express which tasks that are done by P1, P2, P3 and P4 as (people) instances of **Actor**, and how those tasks can be elements in bigger systems that describe things such as organizational processes. Section 5 will deal formally with the concept of *composition*, including properly defining the concept of a *process* as one particular kind of *composition*.

3.3.2.4 Car wash example

As an important part of the car wash system, John and Jack perform certain manual tasks required for washing a car properly:

- *Jack* and *John* are instances of **Actor**
- *WashWindows* is an instance of **Task** and is done by *John*
- *PushWashButton* is an instance of **Task** and is done by *Jack*

4 Service, ServiceContract and ServiceInterface

4.1 Introduction

Service is another core concept of this ontology. It is a concept that is well-understood in practice, but is not easy to define. The ontology is based on the following definition of *service*:

"A service is a logical representation of a repeatable activity that has a specified outcome. It is self-contained and is a 'black box' to its consumers."

The word activity is here used in the general English language sense of the word, not in the process specific sense of that same word (i.e. activities are not necessarily process activities). The ontology purposefully omits "business" as an intrinsic part of the definition of *service*. The reason for this is that the notion of business is relative to a person's viewpoint – as an example, one person's notion of IT is another person's notion of business (the business of IT). *Service* as defined by the ontology is agnostic to whether the concept is applied to the classical notion of a business domain or the classical notion of an IT domain.

Other current SOA specific definitions of the term service include:

- *"A mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description"* (from the [OASIS SOA] Reference Model)
- *"A capability offered by one entity or entities to others using well-defined 'terms and conditions' and interfaces"* (from the [OMG SoaML] Specification)

Within the normal degree of precision of the English language, these definitions are not contradictory; they are stressing different aspects of the same concept. All three definitions are SOA specific though, and represent a particular interpretation of the generic English language term service.

This chapter describes the following classes of the ontology:

- **Service**
- **ServiceContract**
- **ServiceInterface**
- **InformationType**

In addition if defines the following properties:

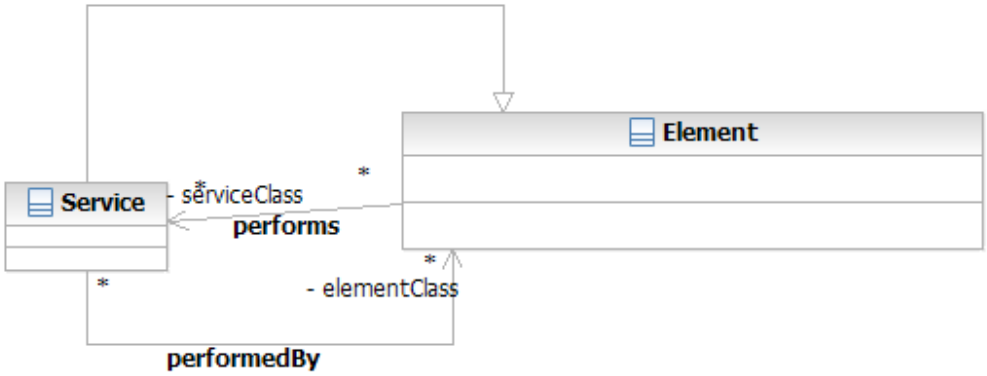
- **performs** and **performedBy**
- **hasContract** and **isContractFor**

- **involvesParty** and **isPartyTo**
- **specifies** and **isSpecifiedBy**
- **hasInterface** and **isInterfaceOf**
- **hasInput** and **isInputAt**
- **hasOutput** and **isOutputAt**

4.2 The *Service* Class

```
<owl:Class rdf:about="#Service">
  <owl:disjointWith>
    <owl:Class rdf:ID="System"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Task"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Actor"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Element"/>
  </rdfs:subClassOf>
</owl:Class>
```

A *service* is a logical representation of a repeatable activity that has a specified outcome. It is self-contained and is a ‘black box’ to its consumers. The concept of *service* is captured by the **Service** class, which is illustrated in the figure below.



In the context of the SOA ontology we consider only SOA based services. Furthermore we adopt a mainly functional view point of those services.

Service is defined as disjoint with the *System*, *Task* and *Actor* classes. Instances of these classes are considered not to be services themselves, even though they may provide capabilities that can be offered as services.

4.2.1 The *performs* and *performedBy* Properties

```
<owl:ObjectProperty rdf:ID="performs">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Service"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="performedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="performs"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

As a service itself is only a logical representation, any service is *performed* by something. The something that *performs* a service must be opaque to anyone interacting with it, an opaqueness which is the exact nature of the **Element** class. This concept is captured by the *performs* and *performedBy* properties as illustrated in figure X above. This also captures the fact that services can be performed by elements of other types than systems[14].

Note that the same instance of **Service** can be performed by many different instances of Element. As long as the service performed is the same an external observer cannot tell the

[14] This includes elements such as software components, actors and tasks

difference[15]. Conversely any instance of **Element** may perform more than one service or none at all.

Terminology used in an SOA environment often include the notions of service providers and service consumers. There are two challenges with this terminology:

- It does not distinguish between the contractual obligation aspect of consume/provide and the interaction aspect of consume/provide. A contractual obligation does not necessarily translate to an interaction dependency, if for no other reason than because the realization of the contractual obligation may have been sourced to a third party.

- Consuming or providing a service is a statement that only makes sense in context – either a contractual context or an interaction context. These terms are consequently not well suited for making statements about elements and services in isolation.

The above are the reasons why the ontology has chosen not to adopt consume and provide as core concepts, rather instead allows consume or provide terms used with contractual obligations and/or interaction rules described by *service contracts*[16].

4.2.2 Examples

*4.2.2.1 The **uses** and **usedBy** properties applied to **Service***

In one direction it does not really make sense to talk about a service that uses another element. While the thing that performs the service might very well include the use of other elements (and certainly will in the case of *Service Composition*), the service itself (as a purely logical representation) does not use other elements.

In the other direction we find the most common of all interactions in an SOA environment, the notion that something uses a service by interacting with it. Note that from an operational perspective this interaction actually reaches somewhat beyond the service itself by involving the following typical steps:

- Picking the service to interact with[17]

- Picking an element that performs that service[18]

- Interacting with the chosen element (that performs the chosen) service[19]

*4.2.2.2 The **represents** and **representedBy** properties applied to **Service***

Concepts such as service mediations, service proxies, Enterprise Service Bus's etc. are natural to those practitioners that describe and implement the operational aspects of SOA systems. From an ontology perspective all of these can be captured by some other element representing the service – a level of indirection that is critical when we do not want to bind operationally to a particular

[15] For contractual obligations, SLA's etc. see the definition of the *ServiceContract* class in Section XYZ
[16] See the definition of the *ServiceContract* class later in this chapter.
[17] This statement is agnostic to whether this is done dynamically at runtime or statically at design and/or construct time
[18] In a typical SOA environment this is most often done "inside" an Enterprise Service Bus
[19] Often also facilitated by an Enterprise Service Bus

service endpoint, rather want to preserve loose coupling and the ability to switch embodiments as needed. Note that by leveraging the *represents* and *representedBy* properties in this fashion we additionally encapsulate the relatively complex operational interaction pattern that was described in the section above (picking the service, picking an element that performs the service and interacting with that chosen element).

While a service being represented by something else is quite natural it is harder to imagine what the service itself might represent. To some degree we have already captured the fact that a service represents any embodiment of it, only we have chosen the use the *performs* and *performedBy* properties to described this rather than the generic *represents* and *representedBy* properties. As a consequence we do not expect practical applications of the ontology to have services represent other elements.

4.2.2.3 Car wash example

Joe offers two different services to his customers, a basic wash and a gold wash. This can be instantiated in the ontology in the following way (subset to the part relevant for these two services):

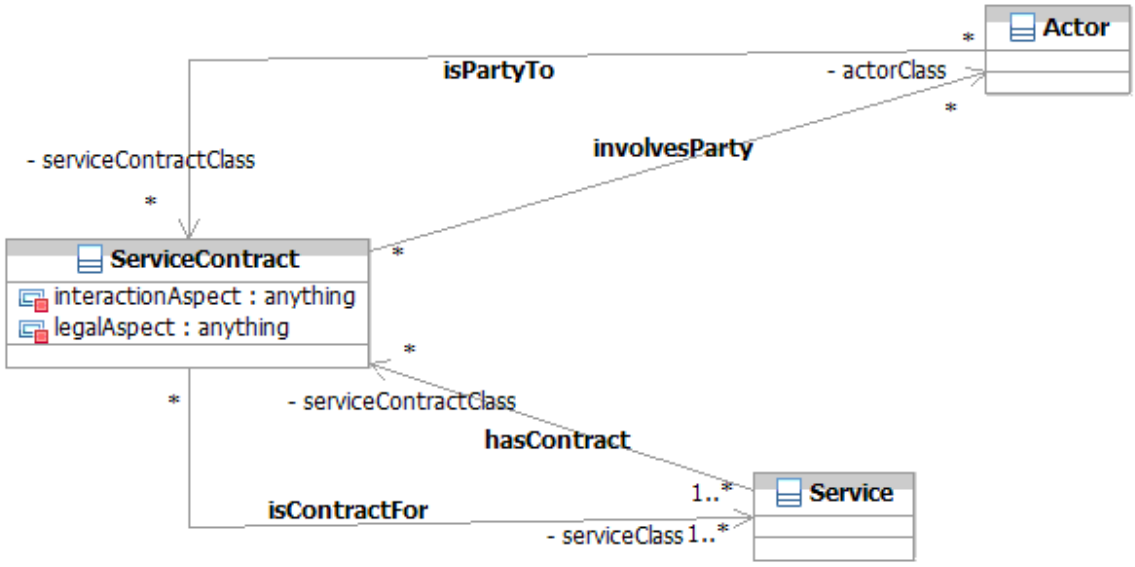
- *GoldWash* is an instance of **Service**
- *BasicWash* is an instance of **Service**
- *CarWash* performs both *BasicWash* and *GoldWash*
- *WashManager* represents both BasicWash and *GoldWash* (i.e. is the interaction point where customers can order services as well as pay for them)

Note the purposeful use of *WashManager* representing both services. This is due to Joe deciding that in his car wash customers are not to interact with the washing machinery directly, rather must instead interact with whomever (actor) is fulfilling the role of wash manager.

4.3 The *ServiceContract* Class

```
<owl:Class rdf:about="#ServiceContract">
</owl:Class>
```

In many cases specific agreements are needed in order to define how to use a service. This can either be because of a desire to regulate such use or can simply be because the service will not function properly unless interaction with it is done in a certain sequence. A *service contract* defines the terms, conditions and interaction rules that interacting participants must agree to (directly or indirectly). A *service contract* is binding on all participants in the interaction, including the service itself and the element that provides it for the particular interaction in question. The concept of *service contract* is captured by the **ServiceContract** class, which is illustrated in the figure below.



4.3.1 The *interactionAspect* and *legalAspect* Datatype Properties

```
  <owl:DatatypeProperty
rdf:about="#interactionAspect">
    <rdfs:domain rdf:resource="#ServiceContract"/>
  </owl:DatatypeProperty>

  <owl:DatatypeProperty rdf:about="#legalAspect">
    <rdfs:domain rdf:resource="#ServiceContract"/>
  </owl:DatatypeProperty>

  <owl:Class rdf:about="#ServiceContract">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="legalAspect"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
```

```
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="legalAspect"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="interactionAspect"/>
        </owl:onProperty>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:about="#interactionAspect"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
```

```
  </owl:Class>
```

Service contracts explicitly regulate both the interaction aspects (see the hasContract and *isContractFor* properties) and the legal agreement aspects (see the *involvedParty* and *isPartyTo* properties) of using a service. The two types of aspects are formally captured by defining the **interactionAspect** and **legalAspect** datatype properties on the **ServiceContract** class. Note that the second of these attributes, the legal agreement aspects, includes concepts such as Service Level Agreements.

If desired it is possible as an architectural convention to split the interaction and legal aspects into two different service contracts. Such choices will be up to any application using this ontology.

4.3.2 The *hasContract* and *isContractFor* Properties

```
  <owl:ObjectProperty rdf:about="#isContractFor">
    <rdfs:domain rdf:resource="#ServiceContract"/>
    <rdfs:range rdf:resource="#Service"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="hasContract">
    <owl:inverseOf>
      <owl:ObjectProperty
rdf:about="#isContractFor"/>
    </owl:inverseOf>
  </owl:ObjectProperty>

  <owl:Class rdf:about="#ServiceContract">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="isContractFor"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

The **hasContract** property, and its inverse **isContractFor**, capture the abstract notion of a service having a service contract. Anyone wanting the use a service must obey the interaction

aspects (as defined in the **interactionAspect** datatype property) of any service contract applying to that interaction. In that fashion the interaction aspects of a service contract are context independent; they capture the defined or intrinsic ways in which a service may be used.

By definition any service contract must be a contract for at least one service. It is possible that the same service contract can be a contract for more than one service; for instance in cases where a group of services share the same interaction pattern or where a service contract (legally[20]) regulates the providing and consuming multiple services.

4.3.3 The *involvesParty* and *isPartyTo* Properties

```
    <owl:ObjectProperty rdf:about="#isPartyTo">
      <rdfs:range
rdf:resource="#ServiceContract"/>
      <rdfs:domain rdf:resource="#Actor"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:ID="involvesParty">
      <owl:inverseOf>
        <owl:ObjectProperty rdf:ID="isPartyTo"/>
      </owl:inverseOf>
    </owl:ObjectProperty>
```

In addition to the rules and regulations that intrinsically apply to any interaction with a service (the interaction aspect of service contracts captured in the **interactionAspect** datatype property) there may be additional legal agreements that apply to certain actors and their use of services. The **involvesParty** property, and its inverse **isPartyTo**, capture the abstract notion of a service contract specifying legal obligations between actors in the context of using the one or more services that the service contract is a contract for.

While the **involvesParty** and **isPartyTo** properties define the relationships to actors involved in the service contract, the actual legal obligations on each of these actors is defined in the **legalAspect** datatype property on the service contract. This includes the ability to define who is the provider and who is the consumer from a legal obligation perspective.

There is a many to many relationship between service contracts and actors. A given actor may be party to none, one or many service contracts. Similarly a given service contract may involve none, one or multiple actors (none in the case where that particular service contract only specifies the **interactionAspect** datatype property. Note that important we allow for sourcing contracts where there is a legal agreement between actor A and actor B (both of which are party to a service contract), yet actor B has sourced the performing of the service to actor C (aka actor C performs the service in question, not actor B).

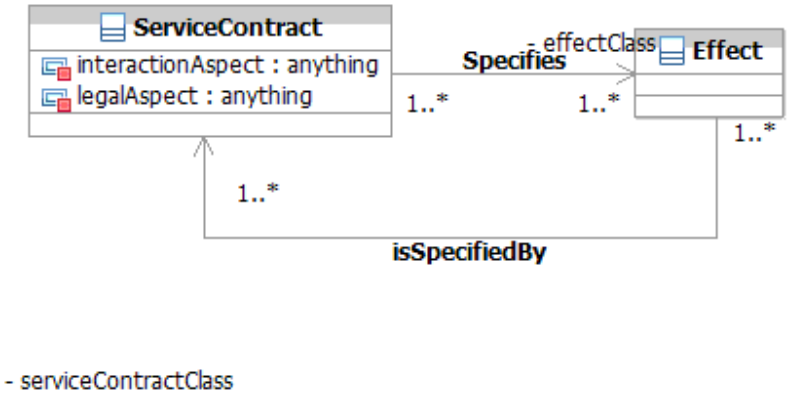
[20] See the *involvesParty* and *isPartyTo* properties.

The **involvesParty** property together with the **legalAspect** datatype property on **ServiceContract** capture not just transient obligations. They include the ability to express "is obliged to at this instant", "was obliged to", and "may in future be obliged to".

4.3.4 The *Effect* Class

```
<owl:Class rdf:about="#Effect">
  <owl:disjointWith>
    <owl:Class rdf:ID="ServiceInterface"/>
  </owl:disjointWith>
</owl:Class>
```

A service has *effects*. These comprise the outcome of the service, and are how it delivers value to its consumers. The concept of *effect* is captured by the **Effect** class, which is illustrated in the figure below.



Note that the **Effect** class purely represents how results or value is delivered to someone interacting with a service. Any possible internal side effects are explicitly not covered by the **Effect** class.

Effect is defined as disjoint with the *ServiceInterface* class[21]. Interacting with a service through its service interface can have an outcome or provide a value (an instance of **Effect**) but the service interface itself does not constitute that outcome or value.

4.3.5 The *specifies* and *isSpecifiedBy* Properties

```
<owl:ObjectProperty rdf:about="#specifies">
  <rdfs:domain rdf:resource="#ServiceContract"/>
```

[21] The *ServiceInterface* class is defined later in the document.

```
      <rdfs:range rdf:resource="#Effect"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:about="#isSpecifiedBy">
      <owl:inverseOf>
        <owl:ObjectProperty rdf:about="#specifies"/>
      </owl:inverseOf>
    </owl:ObjectProperty>

    <owl:Class rdf:ID="Effect">
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:minCardinality>
          <owl:onProperty>
            <owl:ObjectProperty
rdf:ID="isSpecifiedBy"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>

    <owl:Class rdf:about="#ServiceContract">
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:onProperty>
            <owl:ObjectProperty rdf:ID="specifies"/>
          </owl:onProperty>
          <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:minCardinality>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>
```

While a service intrinsically has an effect every time someone interacts with it, in order to trust the effect to be something in particular, the effect needs to be specified as part of a service contract. The **specifies** property, and its inverse **isSpecifiedBy**, capture the abstract notion of a service contract specifying a particular effect as part of the agreement for using a service. Note

that the specified effect can apply to both the **interactionAspect** datatype property (simply specifying what will happen when interacting with the service according to the service contract) and the **legalAspect** datatype property (specifying a contractually promised effect).

Anyone wanting a guaranteed effect of the interaction with a given service must ensure that the desired effect is specified in a service contract applying to that interaction. By definition any service contract must specify at least one effect. In the other direction an effect must be an effect of at least one service contract; this represents that fact that we have chosen only to formalize those effects that are specified by service contracts (and not all intrinsic effects of all services).

4.3.6 Examples

4.3.6.1 Service Level Agreements

A service level agreement on a service has been agreed upon by organizations A and B[22]. This can be represented in the ontology as follows:

- *A* and *B* are instances of **Actor**
- *Service* is an instance of **Service**
- *ServiceContract* is an instance of **ServiceContract**
- *ServiceContract* isContractFor *Service*
- *ServiceContract* involvesParty *A*
- *ServiceContract* involvesParty *B*
- The **legalAspect** datatype property on *ServiceContract* describes the Service Level Agreement

4.3.6.2 Service sourcing

Organizations A and B have agreed on B providing certain services for A, yet B wants to source the actual delivery of those services to third party C. This can be represented in the ontology as follows:

- *A*, *B* and *C* are instances of **Actor**
- *Service* is an instance of **Service**
- *C* provides *Service*
- *ServiceContract* is an instance of **ServiceContract**
- *ServiceContract* isContractFor *Service*

[22] It is important to realize that a Service Level Agreement always has a context of the parties that have agreed to it, involving at a minimum one legal “consumer” and one legal “provider”

- *ServiceContract* involvesParty *A*
- *ServiceContract* involvesParty *B*
- The **legalAspect** datatype property on *ServiceContract* describes the legal obligation of B to provide *Service* for A

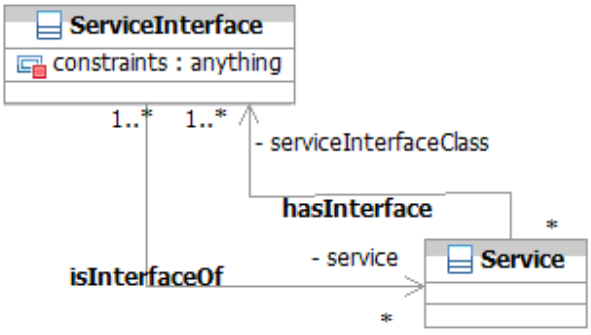
4.3.6.3 *Car wash example*

See section 8.2 for the complete **Service** and **ServiceContract** aspects of the car wash example.

4.4 The *ServiceInterface* Class

```
<owl:Class rdf:about="#ServiceInterface">
  <owl:disjointWith>
    <owl:Class rdf:ID="Service"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="ServiceContract"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Effect"/>
  </owl:disjointWith>
</owl:Class>
```

An important characteristic of services is that they have simple, well-defined interfaces. This makes it easy to interact with them, and enables other elements to use them in a structured manner. A *service interface* defines the way in which other elements can interact and exchange information with a service. This concept is captured by the **ServiceInterface**, class which is illustrated in the figure below.



The concept of an interface is in general well-understood by practitioners, including the notion that interfaces define the parameters for information going in and out of them when invoked. What differs from domain to domain is the specific nature of how an interface is invoked and how information is passed back and forth. Service interface are typically, but not necessarily, message based (to support loose coupling). Furthermore service interfaces are always defined independently from any service implementing them (to support loose coupling and service mediation).

From a design perspective interfaces may have more granular operations or may be composed of other interface. We have chosen to stay at the concept level and not include such design aspects in the ontology.

ServiceInterface is defined as disjoint with the **Service**, **ServiceContract** and **Effect** classes. Instances of these classes are considered not to define (by themselves) the way in which other elements can interact and exchange information with a service. Note that that there is a natural synergy between **ServiceInterface** and the **interactionAspect** datatype property on **ServiceContract** as the latter defines any multi-interaction and/or sequencing constraints on how to use a service through interaction with its service interfaces.

4.4.1 The *Constraints* Datatype Property

```
  <owl:DatatypeProperty rdf:about="#constraints">
    <rdfs:domain
rdf:resource="#ServiceInterface"/>
  </owl:DatatypeProperty>

  <owl:Class rdf:about="#ServiceInterface">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="constraints"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:about="#constraints"/>
```

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

The **Constraints** datatype property on **ServiceInterface** captures that notion that there can be constraints on the allowed interaction such as only certain value ranges allowed on given parameters. Depending on the nature of the service and the service interface in question these constraints may be defined either formally or informally[23].

4.4.2 The *hasInterface* and *isInterfaceOf* Properties

```
    <owl:ObjectProperty rdf:about="#hasInterface">
      <rdfs:domain rdf:resource="#Service"/>
      <rdfs:range rdf:resource="#ServiceInterface"/>
    </owl:ObjectProperty>

    <owl:ObjectProperty rdf:ID="isInterfaceOf">
      <owl:inverseOf>
        <owl:ObjectProperty
rdf:about="#hasInterface"/>
      </owl:inverseOf>
    </owl:ObjectProperty>

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

[23] The information case being relevant at a minimum for certain types of real world services.

```
</owl:Class>
```

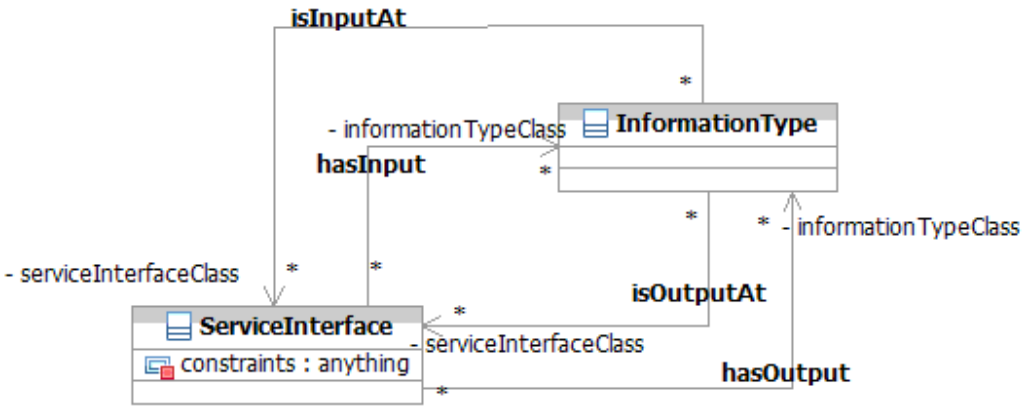
The **hasInterface** property, and its inverse **isInterfaceOf**, capture the abstract notion of a service having a particular service interface.

In one direction any service must have at least one service interface; anything else would be contrary to the definition of a service as a representation of a repeatable activity that has a specified outcome and is a 'black box' to its consumers. In the other direction there can be service interfaces that are not yet interfaces of any defined services. Also the same service interface can be an interface of multiple services. The latter does not mean that these services or the same, nor even that they have the same effect, it only means that it is possible to interact with all these services in the manner defined by the service interface in question.

4.4.3 The *InformationType* Class

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

A service interface can enable another element to give information to or receive information from a service (when it uses that service); specifically the types of information given or received. The concept of *information type* is captured by the **InformationType** class, which is illustrated in the figure below.



In any concrete interaction through a service interface the information types on that interface are instantiated by information items, yet for the service interface itself it is the types that are important. Note that the **constraints** datatype property on **ServiceInterface**, if necessary, can be used to express constraints on allowed values for certain information types.

4.4.4 The *hasInput* and *isInputAt* Properties

```
  <owl:ObjectProperty rdf:ID="hasInput">
    <rdfs:domain
rdf:resource="#ServiceInterface"/>
    <rdfs:range rdf:resource="#InformationType"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isInputAt">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="hasInput"/>
    </owl:inverseOf>
  </owl:ObjectProperty>
```

The **hasInput** property, and its inverse **isInputAt**, capture the abstract notion of a particular type of information being given when interacting with a service through a service interface.

Note that there is a many to many relationship between service interfaces and input information types. A given information type may be input at many service interfaces or none at all. Similarly a given service interface may have many information types as input or none at all. It is important to realize that some services may have only inputs (triggering an asynchronous action without a defined response) and other services may have only outputs (elements performing these services execute independently yet may provide output that is used by other elements).

4.4.5 The *hasOutput* and *isOutputAt* Properties

```
  <owl:ObjectProperty rdf:ID="hasOutput">
    <rdfs:domain
rdf:resource="#ServiceInterface"/>
    <rdfs:range rdf:resource="#InformationType"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isOutputAt">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="hasOutput"/>
    </owl:inverseOf>
  </owl:ObjectProperty>
```

The **hasOutput** property, and its inverse **isOutputAt**, capture the abstract notion of a particular type of information being received when interacting with a service through a service interface.

Note that there is a many to many relationship between service interfaces and output information types. A given information type may be output at many service interfaces or none at all. Similarly a given service interface may have many information types as output or none at all. It

is important to realize that some services may have only inputs (triggering an asynchronous action without a defined response) and other services may have only outputs (elements performing these services execute independently yet may provide output that is used by other elements).

4.4.6 Examples

4.4.6.1 Interaction sequencing

A service contract on a service expresses that the services interfaces on that services must be used in a certain order:

- *Service* is an instance of **Service**
- *ServiceContract* is an instance of **ServiceContract**
- *ServiceContract* isContractFor *Service*
- *X* is an instance of *ServiceInterface*
- *X* isInterfaceOf *Service*
- *Y* is an instance of *ServiceInterface*
- *Y* isInterfaceOf *Service*
- The **interactionAspect** datatype property on *ServiceContract* describes that *X* must be used before *Y* may be used.

4.4.6.2 Car wash example

See section 8.2 for the complete **ServiceInterface** aspect of the car wash example.

5 Composition and its subclasses

5.1 Introduction

The notion of *Composition* is a core concept of SOA. Services can be composed of other services. Processes are composed of actors, tasks and possibly services. Experienced SOA practitioners intuitively apply composition as an integral part of architecting, designing and realizing SOA systems, in fact any well structured SOA environment is intrinsically composite in the way services and processes support business capabilities. What differs from practitioner to practitioner is the exact nature of the composition, the composition pattern being applied.

This chapter describes the following classes of the ontology:

- **Composition** (as a subclass of **System**)
- **ServiceComposition** (as a subclass of **Composition**)
- **Process** (as a subclass of **Composition**)

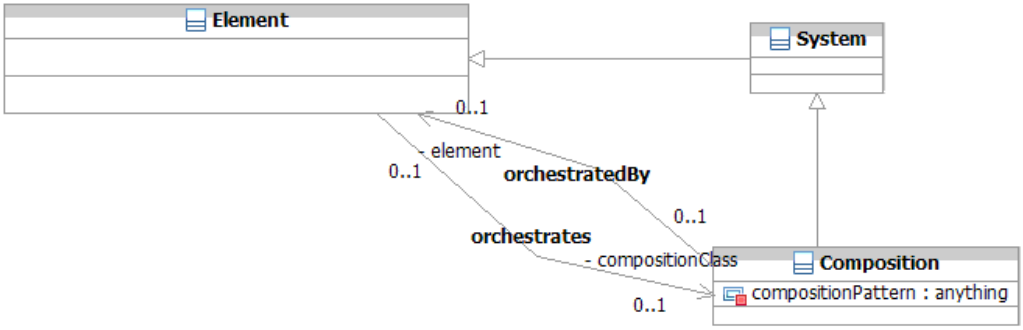
In addition if defines the following datatype property:

- **compositionPattern**

5.2 The *Composition* Class

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

A *composition* is the result of assembling a collection of things for a particular purpose. The concept of *composition* is captured by the **Composition** class, which is illustrated in the figure below.



Being intrinsically (also) an organized collection of other, simpler things, the **Composition** class is a subclass of the **System** class. While a composition is always also a system, a system is not necessarily a composition in that it is not necessarily a result of anything – note here the difference between a system producing a result and the system itself being a result. A perhaps more tangible difference between a system and a composition is that the latter must have associated with it a specific composition pattern that renders the composition (as a whole) as the result when that composition pattern is applied to the elements used in the composition. One implication of this is that there is not a single member of a composition that represents (as an element) that composition as a whole, in other words the composition itself is not one of the things being assembled. On the other hand *composition* is in fact a recursive concept (as are all subclasses of **System**) – being a system, a composition is also an element which means that it can be used by a higher level composition.

In the context of the SOA ontology we consider only SOA based compositions. Furthermore we adopt a mainly functional composition view point. Note that a fully described instance of **Composition** must have by its nature a *uses* relationship to at least one instance of **Element**[24]. Again (as for **System**) it is important to realize that a composition can use elements outside its own boundary.

Since **Composition** is a subclass of **Element**, all compositions have a boundary and are opaque to an external observer (black box view). The composition pattern in turn is the internal view point (white box view) of a composition. As an example for the notion of a service composition this would correspond to the difference between seeing the service composition as an element providing a (higher level) service or seeing the service composition as a composite structure of (lower level) services.

5.2.1 The *compositionPattern* Datatype Property

```
  <owl:DatatypeProperty
rdf:about="#compositionPattern">
      <rdfs:domain rdf:resource="#Composition"/>
    </owl:DatatypeProperty>
```

[24] It need not necessarily have more than one as the composition pattern applied may be for instance simply a transformation.

```
  <owl:Class rdf:about="#Composition">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="compositionPattern"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="compositionPattern"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

As discussed above any composition must have associated with it a specific composition pattern, that pattern describing the way in which a collection of elements is assembled to a result. The concept of a composition pattern is captured by the **compositionPattern** datatype property. Note that even though certain kinds of composition patterns are of special interest within SOA (see below), the **compositionPattern** data type property may take any value as long as that value describes how to assemble the elements used by the composition that it is associated with.

5.2.1.1 The Orchestration composition pattern

One kind of composition pattern that has special interest within SOA is an *Orchestration*. In an orchestration (a composition whose composition pattern is an orchestration), there is one particular element used by the composition that oversees and directs the other elements. Note that the element that directs an orchestration by definition is different than the orchestration (**Composition** instance) itself.

Think of an orchestrated executable workflow as an example of an orchestration. The workflow construct itself is one of the elements being used in the composition, yet is different from the composition itself – the composition itself is the result of applying (executing) the workflow on the processes, actors, services etc. that are orchestrated by the workflow construct.

A non IT example is the foreman of a road repair crew. If the foreman chooses to exert direct control over the tasks done by his crew than the resulting composition becomes an orchestration (with the foreman as the director and provider of the composition pattern). Note that under other circumstances, with a different team composition model, a road repair crew can also act as a collaboration or a choreography[25].

As the last example clearly shows, using an orchestration composition pattern is not a guarantee that "nothing can go wrong". That would in fact depend on the orchestration director's ability to handle exceptions.

5.2.1.2 The Choreography composition pattern

Another kind of composition pattern that has special interest within SOA is a *Choreography*. In a choreography (a composition whose composition pattern is a choreography) the elements used by the composition interact in a non-directed fashion, yet with each autonomous member knowing and following a pre-defined pattern of behavior for the entire composition.

Think of a process model as an example of a choreography. The process model does not direct the elements within it, yet does provide a predefined pattern of behavior that each such element is expected to conform to when "executing".

5.2.1.3 The Collaboration composition pattern

A third kind of composition pattern that has special interest within SOA is a *Collaboration*. In a collaboration (a composition whose composition pattern is a collaboration) the elements used by the composition interact in a non-directed fashion, each according to their own plans and purposes without a pre-defined pattern of behavior. Each element simply knows what it has to do and does it independently, initiating interaction with the other members of the composition as applicable on its own initiative. This means that there is no overall pre-defined "flow" of the collaboration, though there may be a run time "observed flow of interactions".

A good example of a collaboration is a work meeting. There is no script for how the meeting will unfold and only after the meeting has concluded can we describe the sequence of interactions that actually occurred.

5.2.2 The *orchestrates* and *orchestratedBy* Properties

```
    <owl:ObjectProperty
rdf:about="#orchestratedBy">
      <rdfs:domain rdf:resource="#Composition"/>
      <rdfs:range rdf:resource="#Element"/>
    </owl:ObjectProperty>
```

[25] See below for definitions of collaboration and choreography.

```
      <owl:ObjectProperty rdf:about="#orchestrates">
        <owl:inverseOf>
          <owl:ObjectProperty
  rdf:ID="orchestratedBy"/>
        </owl:inverseOf>
      </owl:ObjectProperty>

    <owl:Class rdf:about="#Composition">
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:maxCardinality>
          <owl:onProperty>
            <owl:ObjectProperty
rdf:ID="orchestratedBy"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >0</owl:minCardinality>
          <owl:onProperty>
            <owl:ObjectProperty
rdf:ID="orchestratedBy"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>

    <owl:Class rdf:about="#Element">
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >0</owl:minCardinality>
```

```
          <owl:onProperty>
            <owl:ObjectProperty
rdf:ID="orchestrates"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
      <rdfs:subClassOf>
        <owl:Restriction>
          <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
          >1</owl:maxCardinality>
          <owl:onProperty>
            <owl:ObjectProperty
rdf:about="#orchestrates"/>
          </owl:onProperty>
        </owl:Restriction>
      </rdfs:subClassOf>
    </owl:Class>
```

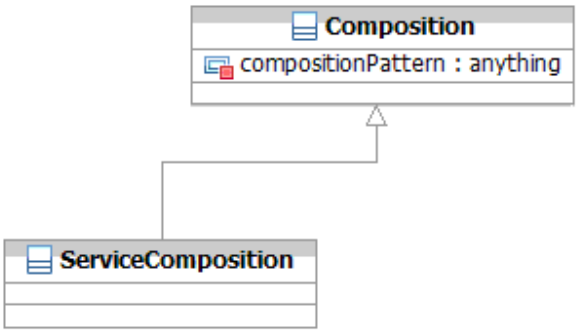
As defined above an orchestration has one particular element that oversees and directs the other elements used by the composition. This type of relationship is important enough that we have chosen to capture the abstract notion in the **orchestrates** property and its inverse **orchestratedBy**.

In one direction a composition has at most one element that orchestrates it, and the cardinality can only be 1 if in fact the composition pattern of that composition is an orchestration. In the other direction an element can orchestrate at most one composition which then must have an orchestration as its composition pattern.

5.3 The *ServiceComposition* Class

```
  <owl:Class rdf:ID="ServiceComposition">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Composition"/>
    </rdfs:subClassOf>
  </owl:Class>
```

A key SOA concept is the notion of *service composition*, the result of assembling a collection of services in order to perform a new higher level service. The concept of *service composition* is captured by the **ServiceComposition** class, which is illustrated in the figure below.



As a *service composition* is the result of assembling a collection of services, **ServiceComposition** is naturally a subclass of **Composition**.

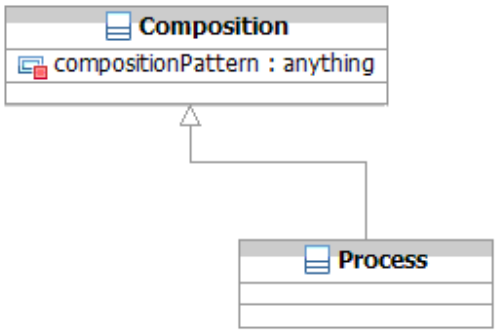
A service composition may, and typically will, add logic (or even "code") via the composition pattern. Note that a service composition is **not** the new higher level service itself[26], rather performs (as an element) that higher level service.

5.4 The *Process* Class

```
<owl:Class rdf:ID="Process">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Composition"/>
  </rdfs:subClassOf>
</owl:Class>
```

Another key SOA concept is the notion of *process*. A *process* is a composition whose elements are composed into a sequence or flow of activities and interactions with the objective of carrying out certain work.[27] The concept of *process* is captured by the **Process** class, which is illustrated in the figure below.

[26] Due to the **System** and **Service** classes being disjoint
[27] This definition is consistent with for instance the BPMN 2.0 definition of what a process is.



Elements in process compositions can be things like Actors, Tasks, Services, other processes etc. A process always adds logic via the composition pattern, the result is more than the parts. According to their collaboration pattern, processes can be:

- Orchestrated: When a process is orchestrated in a Business Process Management System then the resulting IT artifact is in fact an orchestration, i.e. has an orchestration collaboration pattern. This type of process is often called a “Process Orchestration”.
- Choreographed: E.g. a process model representing a defined pattern of behavior. This type of process is often called a “Process Choreography”.
- Collaborative: No (pre)defined pattern of behavior (model), the process represents observed (executed) behavior

5.5 *Service Composition* and *Process* examples

5.5.1 Simple service composition example

Using a service composition example, services A and B are instances of **Service** and the composition of A and B is an instance of **ServiceComposition** (that uses A and B):

- *A* and *B* are instances of **Service**
- *X* is an instance of **ServiceComposition**
- *X* uses both *A* and *B* (composes them according to its service composition pattern)

Note that there are various ways in which the service composition pattern can compose A and B, all of which are relevant in one situation or another. For example interfaces of X may or may not include some subset of the interfaces of A and B. Furthermore the interfaces of A and B may or may not be also be (directly) invocable without going through X – that is a matter of the service contracts and/or access policies apply to the A and B. Finally X may also use other elements that are not services at all (examples are composition code, adaptors etc.).

5.5.2 Process example

Using a process example, tasks T1 and T2 are instances of **Task**, roles R1 and R2 are instances of **Element** and the composition of T1, T2, R1 and R2 is an instance of **Process** (that uses T1, T2, R1 and R2):

- *T1* and *T2* are instances of **Task**
- *R1* and *R2* are instances of **Element**
- *Y* is an instance of **Process**
- *Y* uses all of *T1, T2, R1* and *R2* (composes them according to its process composition pattern)

5.5.3 Process and service composition example

Elaborating on the process example above, if T1 is done using service S then:

- *S* is an instance of **Service**
- *T1* uses *S*

Note that depending on the particular design approach chosen (and the resulting composition pattern), Y may or may not use S directly. This depends on whether Y carries the binding between T1 and S or whether that binding is encapsulated in T1.

5.5.4 Car wash example

See section 8.3 for the **Process** aspect of the car wash example.

6 Policy

6.1 Introduction

Policies, the actors defining them and the things that they apply to are important aspects of any system, certainly also SOA systems with their many different interacting elements. Policies can apply to any element in a system. The concept of *Policy* is captured by the **Policy** class and its relationships to the **Actor** and **Thing** classes.

This chapter describes the following classes of the ontology:

- **Policy**

In addition it defines the following properties:

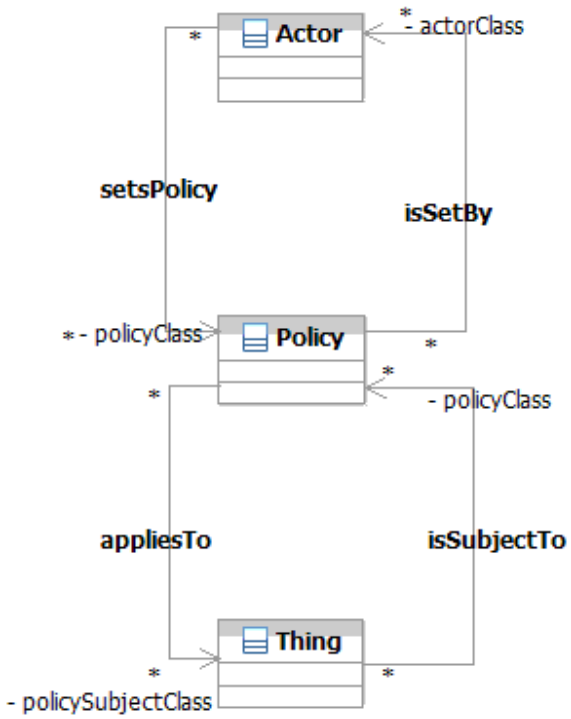
- **appliesTo** and **isSubjectTo**
- **setsPolicy** and **isSetBy**

6.2 The *Policy* Class

```
<owl:Class rdf:about="#Policy">
  <owl:disjointWith>
    <owl:Class rdf:ID="InformationType"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="ServiceInterface"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Element"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Effect"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Event"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="ServiceContract"/>
```

```
    </owl:disjointWith>
  </owl:Class>
```

A *policy* is a statement of direction that an actor may intend to follow or may intend that another actor should follow. Knowing the policies that apply to something makes it easier and more transparent to interact with that something. The concept of *policy* captured by the **Policy** class, which is illustrated in the figure below.



Policy as a concept is generic and has relevance outside the domain of SOA. For the purposes of this SOA ontology it has not been necessary or relevant to restrict the generic nature of the **Policy** class itself. The relationships between **Policy** and **Actor** are of course bound by the SOA specific restrictions that have been applied on the definition of **Actor**.

From a design perspective policies may have more granular parts or may be expressed and made operational through specific rules. We have chosen to stay at the concept level and not include such design aspects in the ontology.

Policy is distinct from all other concepts in this ontology hence the **Policy** class is defined as disjoint with all other defined classes. In particular **Policy** is disjoint with **ServiceContract**.

While policies may apply to service contracts[28], or conversely be referred to by service contracts as part of the terms, conditions and interaction rules that interacting participants must agree to, service contracts are themselves not policies as they do not describe an intended course of action.

6.2.1 The *appliesTo for* and *isSubjectTo* Properties

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

<owl:ObjectProperty rdf:ID="isSubjectTo">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="appliesTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Policies can apply to other things than elements; in fact policies can apply to anything at all, including other policies[29]. The **appliesTo** property, and its inverse **isSubjectTo**, capture the abstract notion that a policy can apply to any instance of **Thing**. Note specifically that **Element** is a subclass of **Thing**, hence policies by inference can apply to any instance of **Element**.

In one direction a policy can apply to zero[30], one or more instances of **Thing**. Note that having a policy apply to multiple things does not mean that these things are the same, only that they are (partly) regulated by the same intent. In the other direction an instance of **Thing** may be subject to zero, one or more policies. Note that where multiple policies apply to the same instance of **Thing** this is often because the multiple policies are from multiple different policy domains (such as security and governance).

The SOA ontology does not attempt to enumerate different policy domains; such policy focused details are deemed more appropriate for a policy ontology. It is worth pointing out that a particular policy ontology may also restrict (if desired) the kinds of things that policies can apply to.

6.2.2 The *setsPolicy* and *isSetBy* Properties

```
<owl:ObjectProperty rdf:about="#setsPolicy">
  <rdfs:domain rdf:resource="#Actor"/>
  <rdfs:range rdf:resource="#Policy"/>
</owl:ObjectProperty>

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

[28] Such as security policies on who may change a given service contract.
[29] For instance a security policy might specify which actors that have the authority to change some other policy.
[30] In the case where a policy has been formulated but not yet explicitly applied to anything.

```
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="setsPolicy"/>
    </owl:inverseOf>
  </owl:ObjectProperty>
```

The **setsPolicy** property, and its inverse **isSetBy**, capture the abstract notion that a policy can be set by one or more actors.

In one direction a policy can be set by zero[31], one or more actors. Note specifically that some policies are set by multiple actors in conjunction, meaning that all these actors need to discuss and agree on the policy before it can take effect[32]. In the other direction an actor may potentially set (or be part of setting) multiple policies.

The SOA ontology purposefully separates the setting of the policy itself and the application of the policy to one or more instances of **Thing**. In some cases these two acts may be inseparable bound together, yet in other cases they are definitely not[33].

Also 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. For example, government food and hygiene regulations (a policy that is law) cover restaurant services independently of anything desired or defined by the restaurant owner.

6.2.3 Examples

6.2.3.1 Car wash example

See section 8.4 for the **Policy** aspect of the car wash example.

[31] In the case where actors setting the policy by choice are not defined or captured.
[32] A real world example would be two parents in conjunction setting policies for acceptable child behavior.
[33] One such example is an overall compliance policy that is formulated at the corporate level yet applied by the compliance officer in each line of business.

7 Event

7.1 Introduction

Events and the elements that generate or respond to them are important aspects of any event emitting system. SOA systems are in fact often event emitting, hence *event* is defined as a concept in the SOA ontology.

This chapter describes the following classes of the ontology:

- **Event**

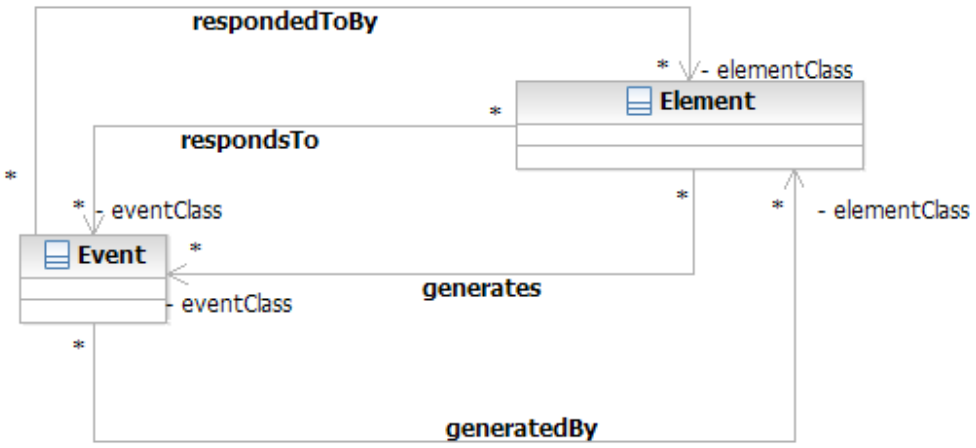
In addition it defines the following properties:

- **generates** and **generatedBy**
- **respondsTo** and **respondedToBy**

7.2 The *Event* Class

```
<owl:Class rdf:about="#Event">
</owl:Class>
```

An *event* is something that happens, to which an element may choose to respond. Events can be responded to be any element. Similarly events may be generated (emitted) by any element. Knowing the events generated or responded to by an element makes it easier and more transparent to interact with that element. Note that some events may occur whether generated or responded to by an element or not. The concept of *event* captured by the **Event** class, which is illustrated in the figure below.



Event as a concept is generic and has relevance outside the domain of SOA. For the purposes of this SOA ontology it has not been necessary or relevant to restrict the generic nature of the **Event** class itself.

From a design perspective events may have more granular parts or may be expressed and made operational through specific syntax or semantics. We have chosen to stay at the concept level and not include such design aspects in the ontology.

7.2.1 The *generates* and *generatedBy* Properties

```
<owl:ObjectProperty rdf:ID="generates">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="generatedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="generates"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Events can, but need not necessarily, be generated by elements. The **generates** property, and its inverse **generatedBy**, capture the abstract notion that an element generates an event.

Note that the same event may be generated by many different elements. Similarly the same element may generate many different events.

7.2.2 The *respondsTo* and *respondedToBy* Properties

```
<owl:ObjectProperty rdf:ID="respondsTo">
  <rdfs:domain rdf:resource="#Element"/>
  <rdfs:range rdf:resource="#Event"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="respondedToBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="respondsTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>
```

Events can, but need not necessarily, be responded to by elements. The **respondsTo** property, and its inverse **respondedToBy**, capture the abstract notion that an element responds to an event.

Note that the same event may be responded to by many different elements. Similarly the same element may respond to many different events.

8 Complete car wash example

This chapter contains the complete car wash example that has been used in parts throughout the definitional chapters of the ontology.

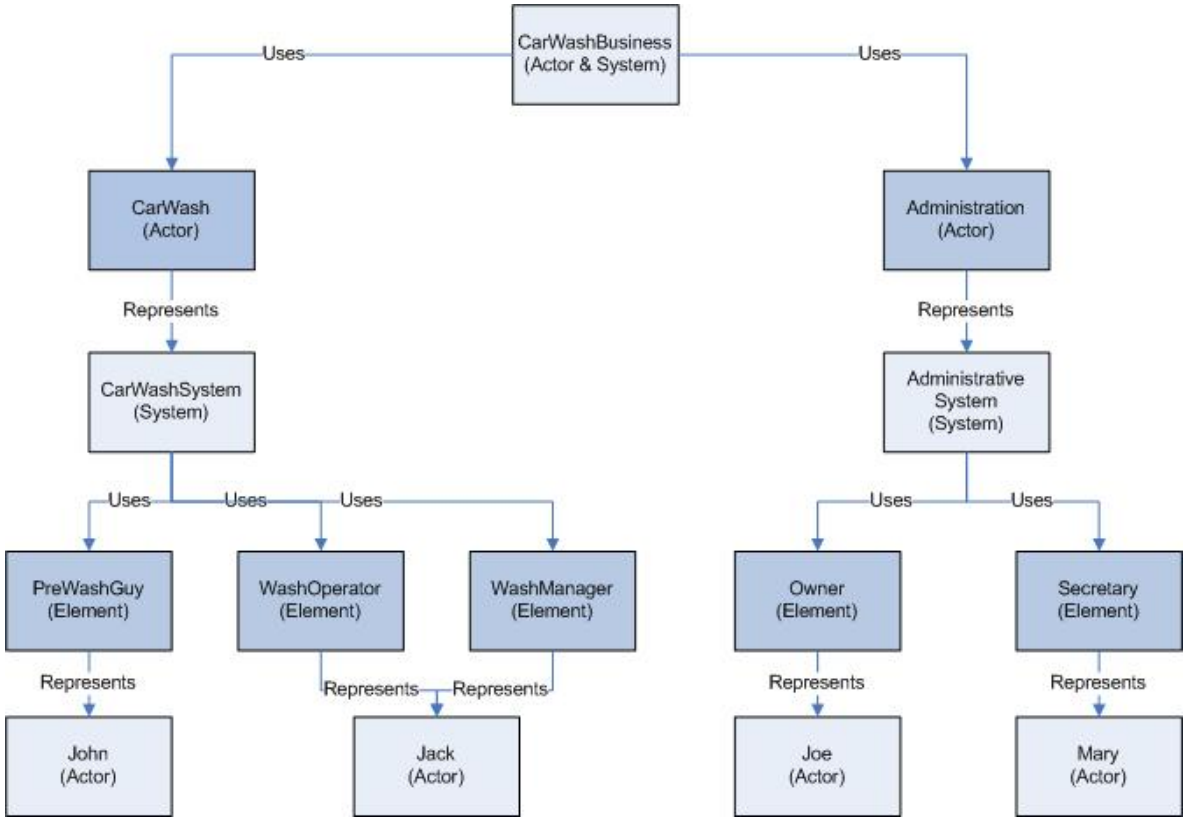
8.1 The organizational aspect

Joe the owner chooses to organize his business into two organizational units, *Administration* and *CarWash*:

- *CarWashBusiness* is an instance of both **Actor** and **System**
- *Administration* is an instance of **Actor** (organizational unit)
- *CarWash* is an instance of **Actor** (organizational unit)
- *CarWashBusiness* uses (has organizational units) *Administration* and *CarWash*
- *AdministrativeSystem* is an instance of **System**
- *Administration* represents *AdministrativeSystem*
- *CarWashSystem* is an instance of **System**
- *CarWash* represents *CarWashSystem*

And using well defined roles within each organization:

- *Owner* (role) is an instance of **Element** and is used by *AdministrativeSystem*
- *Joe* is an instance of **Actor** and is represented by (has role) *Owner*
- *Secretary* (role) is an instance of **Element** and is used by *AdministrativeSystem*
- *Mary* is an instance of **Actor** and is represented by (has role) *Secretary*
- *PreWashGuy* (role) is an instance of **Element** and is used by *CarWashSystem*
- *John* is an instance of **Actor** and is represented by (has role) *PreWashGuy*
- *WashManager* (role) is an instance of **Element** and is used by *CarWashSystem*
- *WashOperator* (role) is an instance of **Element** and is used by *CarWashSystem*
- *Jack* is an instance of **Actor** and is represented by (has roles) both *WashManager* and *WashOperator*



8.2 The washing services

Joe offers two different services to his customers, a basic wash and a gold wash:

- *GoldWash* is an instance of **Service**
- *BasicWash* is an instance of **Service**
- *CarWash* performs both *BasicWash* and *GoldWash*
- *WashManager* represents both BasicWash and *GoldWash* (i.e. is the interaction point where customers can order services as well as pay for them)

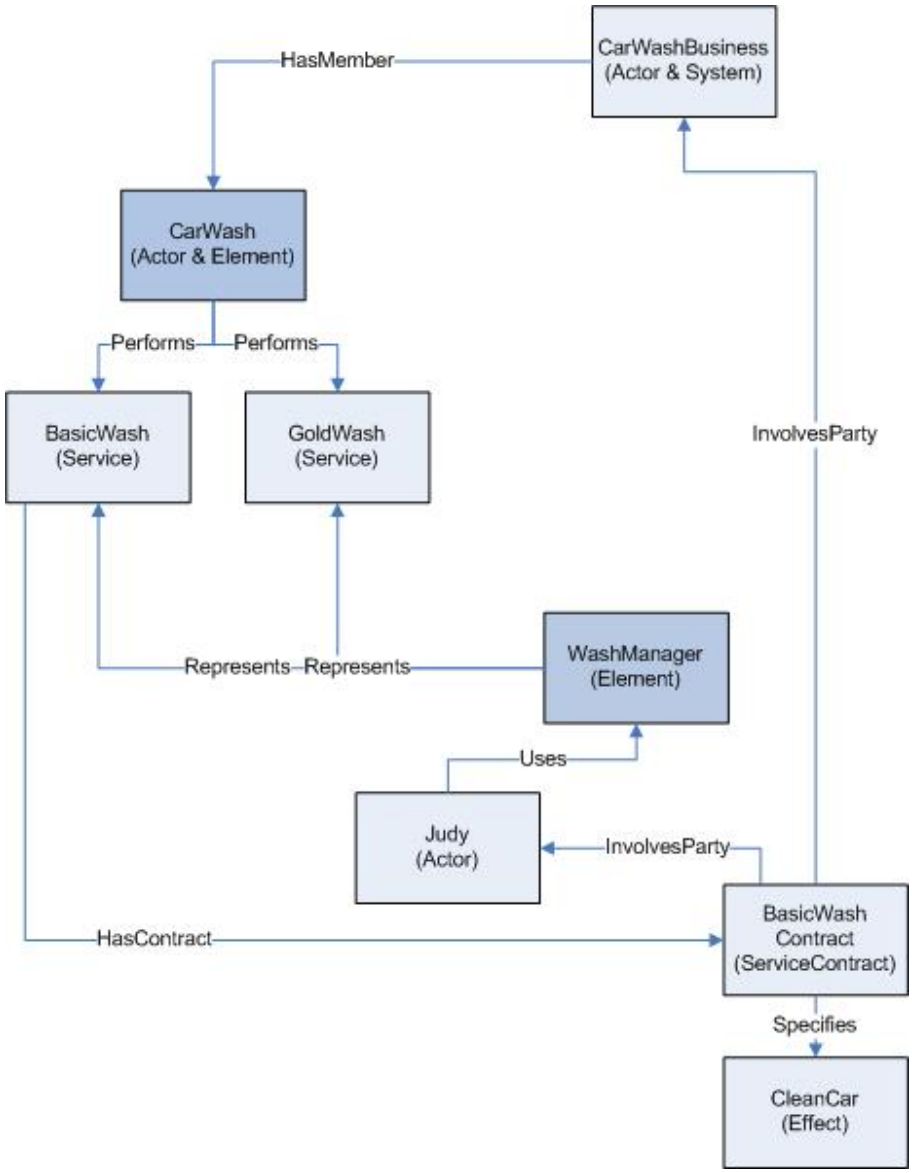
In return for payment, Joe's *BasicWash* service cleans the car of customer Judy:

- *Judy* is an instance of **Actor** (the customer)
- *BasicWashContract* is an instance of **ServiceContract**
- *BasicWash* has contract *BasicWashContract*
- *CleanCar* is an instance of **Effect**
- *BasicWashContract* specifies *CleanCar* as its effect

- *BasicWashContract* involves parties *CarWashBusiness* and *Judy* and specifies that *Judy* (as the legal consumer) pays *CarWashBusiness* (as the legal provider) $10 for the one consumption of *BasicWash* with the effect of (one) *CleanCar*. Note that *BasicWash* is actually performed by *CarWash* and not by the legal provider *CarWashBusiness* – in this particular example *CarWash* happens to be a member of *CarWashBusiness* but such need not always be the case, *CarWash* could have been some 3rd party provider.

- *Judy* uses *WashManager* (in order to invoke the *BasicWash* service)

Note that in this example Judy does not interact with the (abstract) *BasicWash* service directly, rather she interacts with the *WashManager* that represents the service. This is due to Joe deciding that in his car wash customers are not to interact with the washing machinery directly.



8.2.1 The interfaces to the washing services

The way to interact with the car wash services is simple for the customer; he or she simply gives money to the wash manager and asks to have to the car washed using one of the two available wash services. Due to the fact that Joe has decided to interpose the wash manager between the customer and the washing machine, the customer actually never interacts with the wash services themselves. We could have chosen to formally define a proxy service provided by the wash manager but have omitted that level of formally in this real world example.

The wash manager in turn does interact with the wash services through their interfaces defined as follows:

- *WashingMachineInterface* is an instance of **ServiceInterface**
- *TypeOfWash* is an instance of **InformationType**
- *WashingMachineInterface* has input *TypeOfWash*
- *BasicWash* has interface *WashingMachineInterface*
- *GoldWash* has interface *WashingMachineInterface*

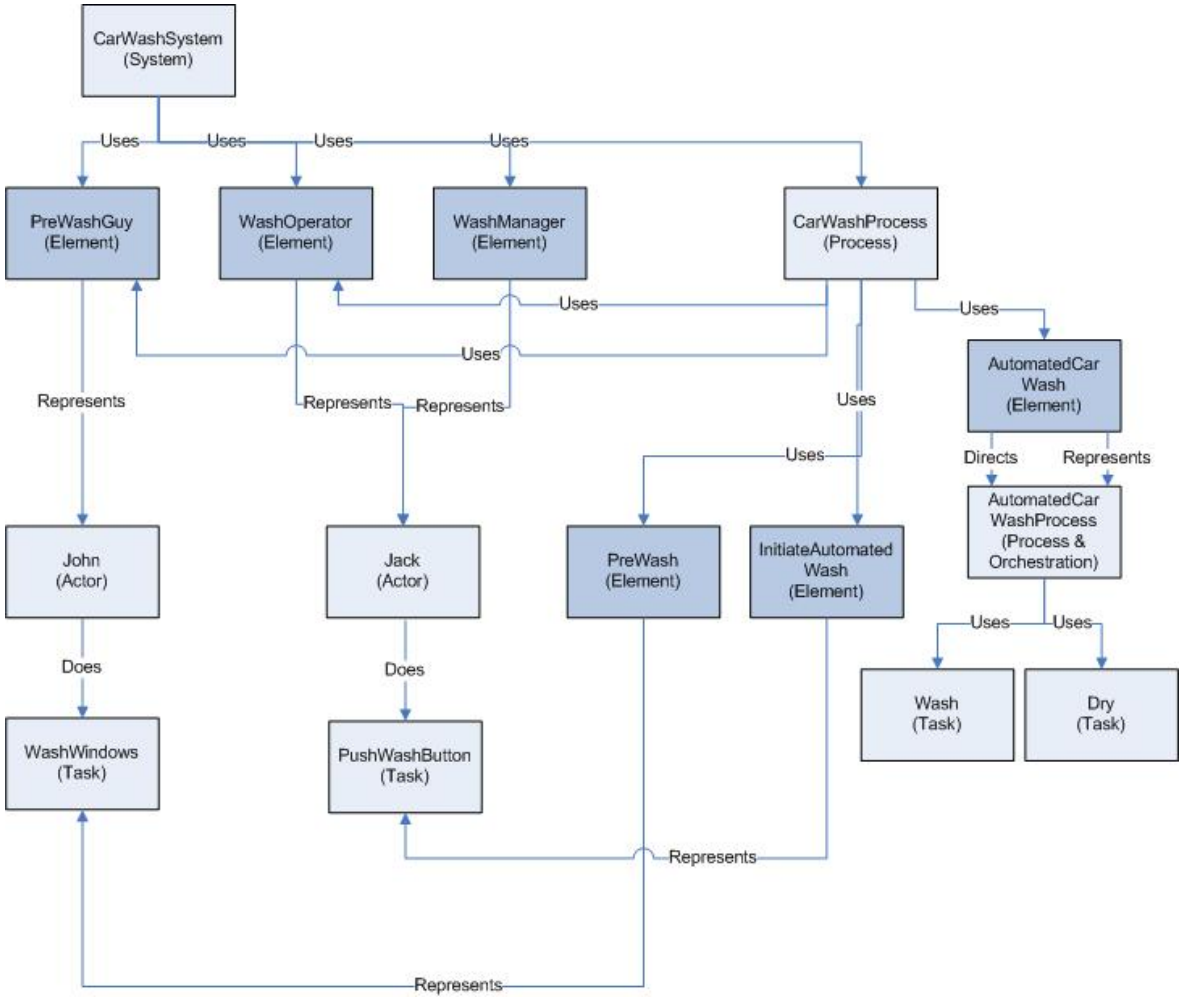
Note how both washing service in fact have the same service interface. Even though Joe has chosen to offer basic wash and gold wash as two different services, both are in effect done by the same washing machine (one simply has to choose the type of wash when initializing the washing machine).

8.3 The washing processes

An important part of the car wash system is the car washing process itself:

- *AutomatedCarWashProcess* is an instance of both **Process** and **Orchestration**
- *Wash* is an instance of **Task** and is used by *AutomatedCarWashProcess*
- *Dry* is an instance of **Task** and is used by *AutomatedCarWashProcess*
- *AutomatedCarWash* is an instance of **Element** (the automated washing machine) and represents *AutomatedCarWashProcess* (encapsulates the process) as well as directs *AutomatedCarWashProcess*
- *CarWashProcess* is an instance of **Process** and is used by (part of) *CarWashSystem* (no need to create an explicit opaque building block)
- *AutomatedCarWash* is used by *CarWashProcess* (automated activity in the process)
- *WashWindows* is an instance of **Task** and is done by *John*
- *PreWash* is an instance of **Element**, represents *WashWindows* and is used by *CarWashProcess* (logical activity in the process)

- *PrewashGuy* is a member of *CarWashProcess* (role in the process)
- *PushWashButton* is an instance of **Task** and is done by *Jack*
- *InitiateAutomatedWash* is an instance of **Element**, represents *PushWashButton* and is used by *CarWashProcess* (logical activity in the process)
- *WashOperator* is a member of *CarWashProcess* (role in the process)



8.4 The washing policies

Joe sets a payment up-front policy for the washing services:

- *PaymentUpFront* is an instance of both **Policy**
- *PaymentUpFront* is set by *Joe*
- *PaymentUpFront* applies to both *GoldWash* and *BasicWash*

Note how the *PaymentUpFront* policy enhances the service contract *BasicWashContract*. While *BasicWashContract* only specifies that *Judy* has to pay $10 for one consumption of the *BasicWash* service, the *PaymentUpFront* policy makes it specific that payment has to happen up front. One of the advantages of separating policy from service contract is that the payment policy can be changed independently of the service contract. For instance at some later point in time Joe may decide that recurring customer need not pay up-front, and can institute this change in policy without changing anything else related to *CarWashBusiness*.

9 Complete internet purchase example

NEEDS TO BE WRITTEN

A The OWL Definition of the Ontology

The ontology is available online at:

NEED LOCATION FOR PUBLISHED VERSION OF OWL DEFINITION

and is reproduced below.

```
<?xml version="1.0"?>

<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-
syntax-ns#"
    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#"

xmlns="http://www.semanticweb.org/ontologies/2010/
01/core-soa.owl#"

xml:base="http://www.semanticweb.org/ontologies/20
10/01/core-soa.owl"
>


  <!-- ontology -->


  <owl:Ontology rdf:about=""/>


  <!-- classes -->


  <owl:Class rdf:ID="Event">
    <owl:disjointWith>
```

```
      <owl:Class rdf:ID="Policy"/>
    </owl:disjointWith>
  </owl:Class>


  <owl:Class rdf:ID="InformationType">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
  </owl:Class>


  <owl:Class rdf:ID="ServiceCompostion">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Composition"/>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:Class rdf:ID="Effect">
    <owl:disjointWith>
      <owl:Class rdf:about="#Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="ServiceInterface"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="isSpecifiedBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
```

```
  <owl:Class rdf:ID="Task">
    <owl:disjointWith>
      <owl:Class rdf:ID="Policy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="System"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Actor"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Service"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Element"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="doneBy"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >0</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:about="#doneBy"/>
        </owl:onProperty>
```

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


  <owl:Class rdf:about="#System">
    <owl:disjointWith>
      <owl:Class rdf:ID="Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Service"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Element"/>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:Class rdf:about="#Service">
    <owl:disjointWith>
      <owl:Class rdf:ID="System"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Task"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Actor"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="ServiceInterface"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Element"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
```

```
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="hasInterface"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:Class rdf:about="#Policy">
    <owl:disjointWith>
      <owl:Class rdf:ID="InformationType"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="ServiceInterface"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Element"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Effect"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Event"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="ServiceContract"/>
    </owl:disjointWith>
  </owl:Class>


  <owl:Class rdf:about="#Actor">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Element"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:ID="Task"/>
    </owl:disjointWith>
```

```
    <owl:disjointWith>
      <owl:Class rdf:ID="Service"/>
    </owl:disjointWith>
  </owl:Class>


  <owl:Class rdf:about="#Composition">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="System"/>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="compositionPattern"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="compositionPattern"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
```

```
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="orchestratedBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >0</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="orchestratedBy"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:Class rdf:about="#ServiceInterface">
    <owl:disjointWith>
      <owl:Class rdf:ID="Service"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="ServiceContract"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Effect"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Policy"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
```

```
          <owl:DatatypeProperty
rdf:ID="constraints"/>
        </owl:onProperty>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:about="#constraints"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>


  <owl:Class rdf:about="#Element">
    <owl:disjointWith>
      <owl:Class rdf:ID="Policy"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >0</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="orchestrates"/>
        </owl:onProperty>
      </owl:Restriction>
```

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


  <owl:Class rdf:about="#ServiceContract">
    <owl:disjointWith>
      <owl:Class rdf:ID="ServiceInterface"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Policy"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="legalAspect"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
```

```
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="legalAspect"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:ID="interactionAspect"/>
        </owl:onProperty>
        <owl:maxCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:maxCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty
rdf:about="#interactionAspect"/>
        </owl:onProperty>
        <owl:minCardinality
rdf:datatype="http://www.w3.org/2001/XMLSchema#int
"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty
rdf:ID="isContractFor"/>
```

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


  <owl:Class rdf:about="#Process">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="Composition"/>
    </rdfs:subClassOf>
  </owl:Class>


  <!-- object properties -->


  <owl:ObjectProperty rdf:about="#isPartyTo">
    <rdfs:range rdf:resource="#ServiceContract"/>
    <rdfs:domain rdf:resource="#Actor"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="involvesParty">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="isPartyTo"/>
```

```
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:about="#orchestratedBy">
    <rdfs:domain rdf:resource="#Composition"/>
    <rdfs:range rdf:resource="#Element"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:about="#orchestrates">
    <owl:inverseOf>
      <owl:ObjectProperty
rdf:ID="orchestratedBy"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:about="#isContractFor">
    <rdfs:domain rdf:resource="#ServiceContract"/>
    <rdfs:range rdf:resource="#Service"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="hasContract">
    <owl:inverseOf>
      <owl:ObjectProperty
rdf:about="#isContractFor"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:about="#setsPolicy">
    <rdfs:domain rdf:resource="#Actor"/>
    <rdfs:range rdf:resource="#Policy"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isSetBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="setsPolicy"/>
    </owl:inverseOf>
```

```
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="generates">
    <rdfs:domain rdf:resource="#Element"/>
    <rdfs:range rdf:resource="#Event"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="generatedBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="generates"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:about="#represents">
    <rdfs:domain rdf:resource="#Element"/>
    <rdfs:range rdf:resource="#Element"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="representedBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="represents"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="hasInput">
    <rdfs:domain
rdf:resource="#ServiceInterface"/>
    <rdfs:range rdf:resource="#InformationType"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isInputAt">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="hasInput"/>
    </owl:inverseOf>
  </owl:ObjectProperty>
```

```
<owl:ObjectProperty rdf:about="#doneBy">
  <rdfs:domain rdf:resource="#Task"/>
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="does">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#doneBy"/>
  </owl:inverseOf>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:about="#specifies">
  <rdfs:domain rdf:resource="#ServiceContract"/>
  <rdfs:range rdf:resource="#Effect"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:about="#isSpecifiedBy">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:about="#specifies"/>
  </owl:inverseOf>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:ID="appliesTo">
  <rdfs:domain rdf:resource="#Policy"/>
</owl:ObjectProperty>

<owl:ObjectProperty rdf:ID="isSubjectTo">
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="appliesTo"/>
  </owl:inverseOf>
</owl:ObjectProperty>


<owl:ObjectProperty rdf:about="#hasInterface">
  <rdfs:domain rdf:resource="#Service"/>
  <rdfs:range rdf:resource="#ServiceInterface"/>
```

```
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isInterfaceOf">
    <owl:inverseOf>
      <owl:ObjectProperty
rdf:about="#hasInterface"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="respondsTo">
    <rdfs:domain rdf:resource="#Element"/>
    <rdfs:range rdf:resource="#Event"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="respondedToBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="respondsTo"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="performs">
    <rdfs:domain rdf:resource="#Element"/>
    <rdfs:range rdf:resource="#Service"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="performedBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="performs"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:about="#uses">
    <rdfs:domain rdf:resource="#Element"/>
    <rdfs:range rdf:resource="#Element"/>
  </owl:ObjectProperty>
```

```
  <owl:ObjectProperty rdf:ID="usedBy">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="uses"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <owl:ObjectProperty rdf:ID="hasOutput">
    <rdfs:domain
rdf:resource="#ServiceInterface"/>
    <rdfs:range rdf:resource="#InformationType"/>
  </owl:ObjectProperty>

  <owl:ObjectProperty rdf:ID="isOutputAt">
    <owl:inverseOf>
      <owl:ObjectProperty rdf:ID="hasOutput"/>
    </owl:inverseOf>
  </owl:ObjectProperty>


  <!-- datatype properties -->


  <owl:DatatypeProperty rdf:about="#legalAspect">
    <rdfs:domain rdf:resource="#ServiceContract"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty rdf:about="#constraints">
    <rdfs:domain
rdf:resource="#ServiceInterface"/>
  </owl:DatatypeProperty>


  <owl:DatatypeProperty
rdf:about="#compositionPattern">
    <rdfs:domain rdf:resource="#Composition"/>
  </owl:DatatypeProperty>
```

```
  <owl:DatatypeProperty
rdf:about="#interactionAspect">
    <rdfs:domain rdf:resource="#ServiceContract"/>
  </owl:DatatypeProperty>


</rdf:RDF>
```

B Relationship to other SOA standards

.A joint paper that has been written that positions the SOA Ontology with other architectural standards. The “Navigating the SOA Open Standards Landscape Around Architecture” joint white paper from OASIS, OMG, and The Open Group was written to help the SOA community at large navigate the myriad of overlapping technical products produced by these organizations with specific emphasis on the “A” in SOA; i.e., Architecture.

This joint white paper explains and positions standards for SOA reference models, ontologies, reference architectures, maturity models, modeling languages, and standards work on SOA governance. It outlines where the works are similar, highlights the strengths of each body of work, and touches on how the work can be used together in complementary ways.. It is also meant as a guide to users of these specifications for selecting the technical products most appropriate for their needs, consistent with where they are today and where they plan to head on their SOA journeys.

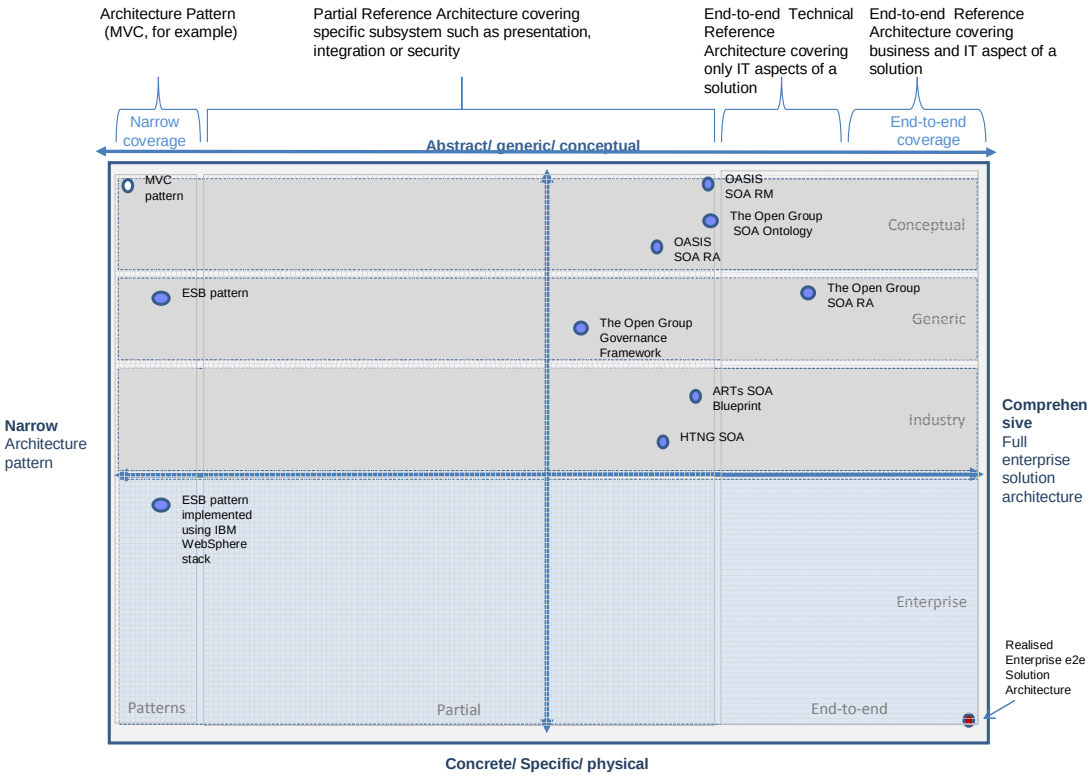
While the understanding of SOA and SOA Governance concepts provided by these works is similar, the evolving standards are written from different perspectives. Each specification supports a similar range of opportunity, but has provided different depths of detail for the perspectives on which they focus. Therefore, although the definitions and expressions may differ somewhat, there is agreement on the fundamental concepts of SOA and SOA Governance.

The following is a summary of the positioning and guidance on the specifications:

- The OASIS Reference Model for SOA (SOA RM) is the most abstract of the specifications positioned. It is used for understanding of core SOA concepts
- The Open Group SOA Ontology extends, refines, and formalizes some of the core concepts of the the SOA RM. It is used for understanding of core SOA concepts and facilitates a model-driven approach to SOA development.
- The OASIS Reference Architecture for SOA Foundation is an abstract, foundation reference architecture addressing the ecosystem viewpoint for building and interacting within the SOA paradigm. It is used for understanding different elements of SOA, the completeness of SOA architectures and implementations, and considerations for cross ownership boundaries where there is no single authoritative entity for SOA and SOA governance. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/soa-ra-pr-01.pdf
- The Open Group SOA Reference Architecture is a layered architecture from consumer and provider perspective with cross cutting concerns describing these architectural building blocks and principles that support the realizations of SOA. It is used for understanding the different elements of SOA, deployment of SOA in

enterprise, basis for an industry or organizational reference architecture, implication of architectural decisions, and positioning of vendor products in SOA context. http://www.opengroup.org/projects/soa-ref-arch/uploads/40/19713/soa-ra-public-050609.pdf

- The Open Group SOA Governance Framework is a governance domain reference model and method. It is for understanding SOA governance in organizations. The OASIS Reference Architecture for SOA Foundation contains an abstract discussion of governance principles as applied to SOA with particular application to governance across boundaries. http://www.opengroup.org/projects/soa-governance/uploads/40/19263/SOA_Governance_Architecture_v2.4.pdf
- The Open Group SOA Integration Maturity Model (OSIMM) is a means to assess an organization's maturity within a broad SOA spectrum and define a roadmap for incremental adoption. It is used for understanding the level of SOA maturity in an organization. http://www.opengroup.org/projects/osimm/uploads/40/19756/OSIMM_v2.1_6-04-09_Review.doc
- The Object Management Group SoaML Specification supports services modeling UML extensions. It can be seen as an instantiation of a subset of the Open Group RA used for representing SOA artifacts in UML. http://www.omg.org/cgi-bin/doc?ad/08-11-01



Fortunately, there is a great deal of agreement on the foundational core concepts across the many independent specifications and standards for SOA. This could be best explained by broad and common experience of users of SOA and its maturity in the marketplace. It also provides assurance that investing in SOA-based business and IT transformation initiatives that incorporate and use these specifications and standards helps to mitigate risks that might compromise a successful SOA solution.

It is anticipated that future work on SOA standards may consider the positioning in this paper to reduce inconsistencies, overlaps, and gaps between related standards and to ensure that they continue to evolve in as consistent and complete a manner as possible.

Glossary

BPMN	Business Process Modeling Notation
ESB	Enterprise Service Bus
OWL	Web Ontology Language
HTTP	Hypertext Transfer Protocol
SOA	Service-Oriented Architecture
SoaML	Service-Oriented Architecture Modeling Language

Index (NOT COMPLETE - NEEDS UPDATE)