The universAAL Primer

Advanced

Introduction

The EU-funded project **universAAL** aims to produce an open platform that provides a standardized approach making it technically feasible and economically viable to develop AAL solutions. More information is available at http://universaal.org/.

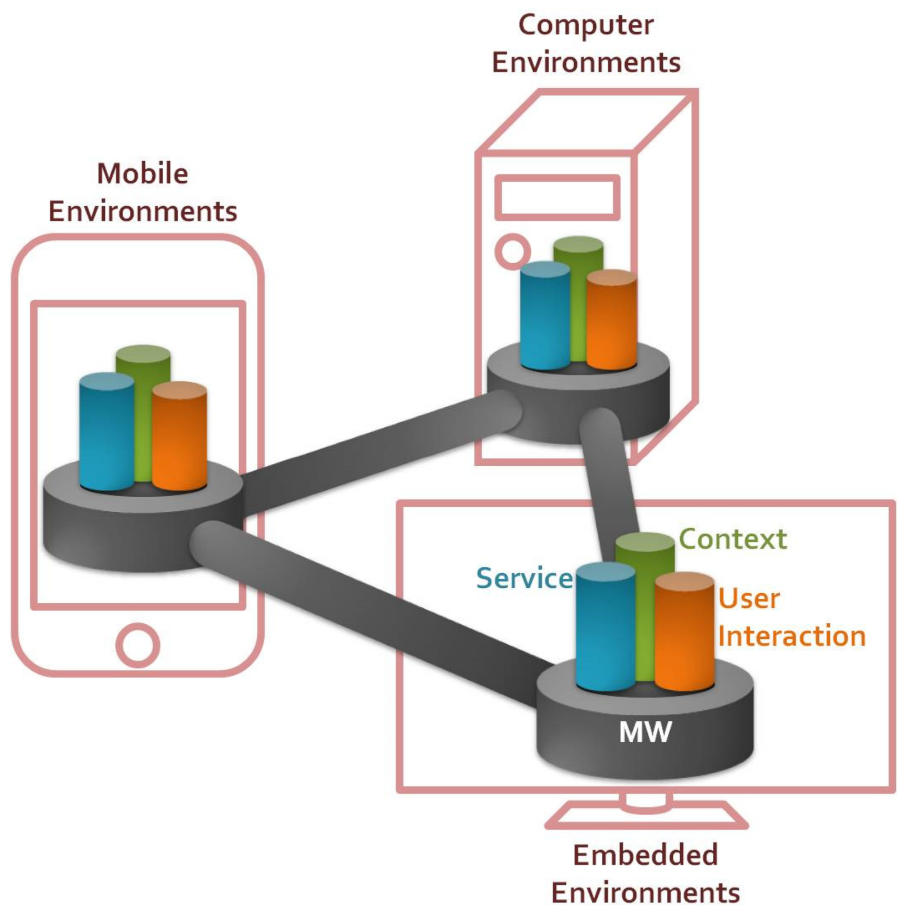
This **Advanced Primer Book** intends to give an overview of the universAAL Execution Platform and its API. The core concepts of **uAAL** are explained and these are accompanied by indicative diagrams and snippets of code showing API usage.

The targeted audience is code developers with some interest in the field of AAL, and knowledge about ICT and programming is strongly recommended, and required to understand the code snippets. It is also recommended that the reader has completed the **Basic Primer Book**. The code shown in this version of the **Advanced Primer Book** is in Java.

Take into account that the content of this **Primer** does neither replace nor take precedence under any circumstance over the official contents, code and API developed in project **universAAL**, reported in the public deliverables available in http://universaal.org/es/about/deliverables and documentation reachable at http://depot.universaal.org/.

The current version of this **Primer** is compatible with the release **2.0** of the **universAAL** Execution Platform. The software sources and binaries are available through http://forge.universaal.org/gf/.

Contents



The Middleware delivers buses across multiple instances in different environments

Middleware

The Middleware is the core part of uAAL platform and takes care that all uAAL nodes in a Space can cooperate one with each other. It establishes peer to peer communications between them so that they can share the different kinds of uAAL semantic communication: Context, Service and User Interaction, following the shared Ontological Model.

How middleware and platform are composed in layers

The **Container** is the part that lets the Middleware logic execute in different environments. There are different Containers so that the Middleware can run on devices with plain Java, computers or embedded systems running OSGi, or in Android smartphones, so far.

The Container determines how the components of uAAL and the applications that use it are coded and built. In OSGi they would be Bundles, in Android they would be APKs, and so on. Currently universAAL only officially supports these two.

The **Peering** part is responsible for interconnecting and communicating the instances of the Middleware regardless of where they are running, using technologies such as jSLP and jGroups.

More technologies can be added modularly. A hypothetic UPnP connector (like earlier versions) could discover other nodes with this connector, and use bridging options for using multiple technologies.

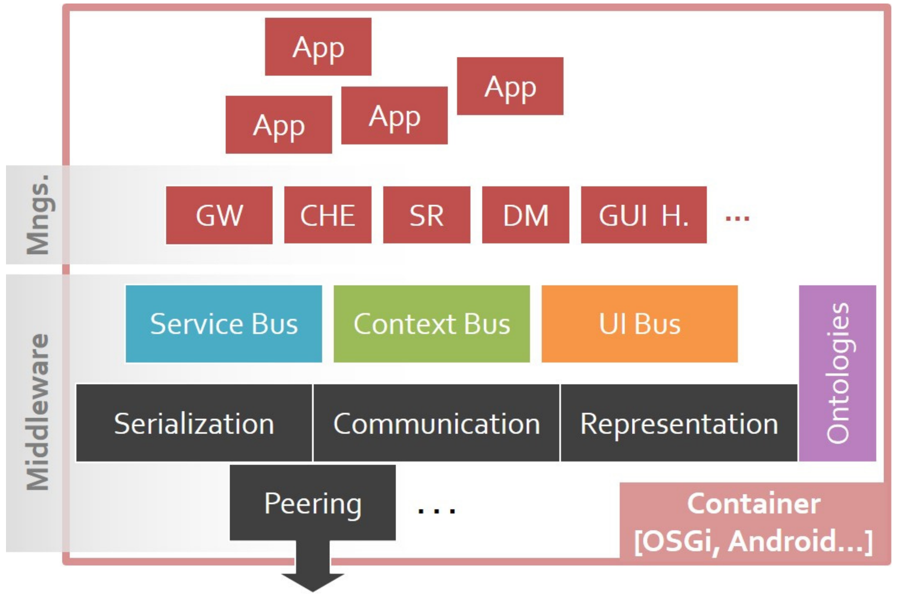
The **Communication** part is the one holding the ultimate logic of the Middleware that enables the flow of uAAL semantic information across peers, by defining specific-purposes Busses. These Busses is what applications connect to, and when they do so, they are in constant contact no matter the Device, Container or Peering technology they are running with. There is a bus for each type of communication (Context, Service and User Interaction), handling its own specific strategy, semantics, reasoning and match-making of participants.

Each Bus builds on top of the common components of the Middleware. The communication model is the "skeleton" for the buses. The representation model defines how ontologies are handled. The serialization component encrypts and parses messages across nodes.

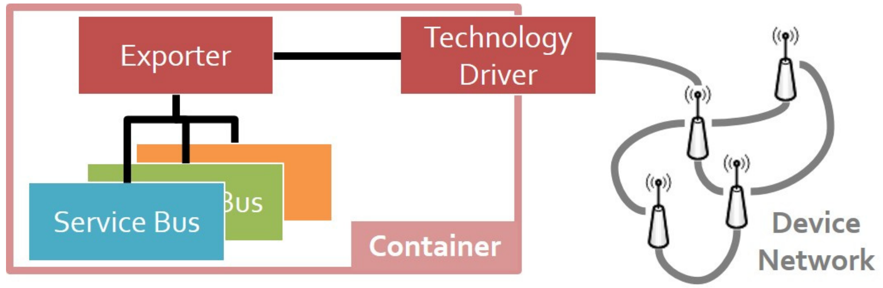
The **Managers** build upon the Middleware. They can be considered low level applications. Together with the Middleware they form the uAAL execution platform, and are required for its proper operation. Some are tied to certain Buses, while others are more widely used. They usually also provide functional APIs to the above final applications.

Finally a **uAAL Application** is any piece of software that can run on the Container and that makes use of the uAAL Buses or Managers, whether by consuming them or providing into them, in order to provide an AAL Service or a part of it.

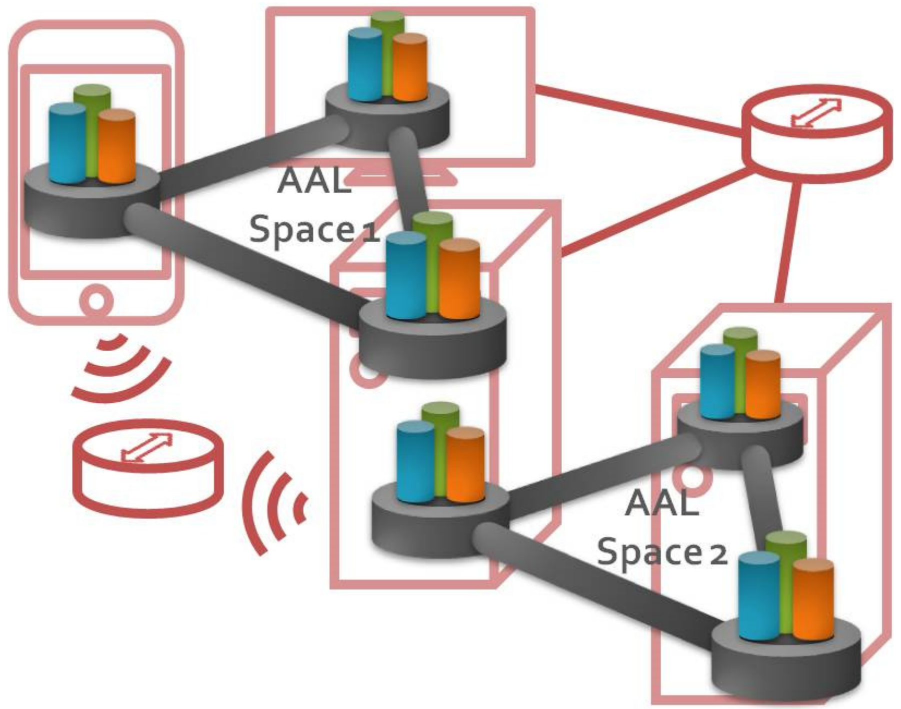
Regarding hardware sensor and actuator devices, these are connected through "exporters", which are just like an application exporting the devices interfaces and information into uAAL platform. There would be a different exporter per technology (KNX, ZigBee…).



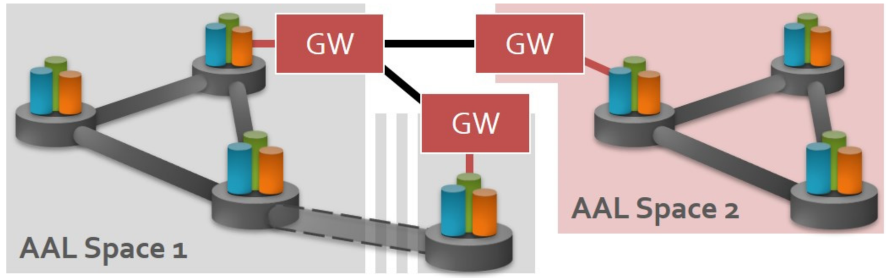
A **Layer-Oriented** scheme of uAAL middleware and managers forming the platform



Sensors and Actuators are plugged into the platform through exporter applications



uAAL Nodes only link to others in same space regardless of network or physical node



Gateways connect different spaces, or remote nodes into same space

What is an AAL space

The uAAL platform defines the concept of **AAL Space**. An AAL Space is a logical environment in which all uAAL Applications can communicate to each other and the platform through the same buses, seamlessly, regardless of the node or container they run within and the physical network beneath. The AAL Services provided by these applications are oriented to a specific human user (the Assisted Person) or set of users.

The official definition of AAL Space by uAAL is "a smart environment centred on its human users in which a set of embedded networked artefacts, both hardware and software, collectively realize the paradigm of Ambient Intelligence, mainly by providing for context-awareness and personalization, reactivity, and pro-activity".

The most typical Space is the Home of an Assisted Person, but other scenarios are possible in which an AAL Space can be set up in supermarkets, health care centres, hospitals, airports...

In every Space there is a single **Coordinator** node that takes care of creating it for the first time, keeps track of its status and ID, and acts as entry point for new nodes.

Communication across Spaces is not possible except through a special Manager called **AAL Space Gateway**, which takes care of message exchanges and authorization between different Spaces. However it also takes care of connecting remote nodes that belong to the original Space, accomplishing a similar function to that of a VPN, when networked connection through Peering is not possible.

Ontologies

Knowledge is shared in uAAL in the form of Ontologies. It is its information model. Ontologies are a way to represent real-life information so it can be understood by computers. You can think of Ontologies as a network of concepts linked by properties. One tricky thing is that while we usually think in tree-view, Ontologies are meshes.

Ontologies can be represented in some standard format such as serialized RDF, but in uAAL Ontologies have a code representation so they can be handled by the Middleware. There are tools to convert from one to another. Ontologies are usually grouped in domains representing a single field of knowledge, such as Devices, Health or Furniture.

How to build an ontology

In the Middleware representation Ontologies are composed by an Ontology class that defines the whole Ontology, a Factory for serialization, and a collection of classes representing the concepts.

Resources are how the concepts are represented. They are the nodes in the mesh. They are identified by a URI. They can inherit from other resources, and have properties that link to other resources or datatypes.

In the Middleware representation a Resource is like a class. In addition to being defined in the Ontology class, there is a class representing it that can be instantiated to be used from the code, and contains constants that

```
public class LightSource extends Device {
   // Resource URI start upper case, properties & instances lower
   public static final String MY_URI =
      LightingOntology.NAMESPACE + "LightSource";
   public static final String PROP_HAS_TYPE =
      LightingOntology.NAMESPACE + "hasType";
   public static final String PROP_SOURCE_BRIGHTNESS =
      LightingOntology.NAMESPACE + "srcBrightness";
   public LightSource() {super();}
   public LightSource(String uri) {super(uri);}
   // A concept resource class must comply with certain methods:
   public String getClassURI() {return MY_URI;}
   // This is for reducing serialization depending on property
   public int getPropSerializationType(String propURI) { ... }
   // This is to determine when an instance is properly built
   public boolean isWellFormed() { ... }
   // There can be as many helper methods as desired, or none
   public void setBrightness(int b) {
      setProperty(PROP_SOURCE_BRIGHTNESS, new Integer(b));
   }
   ...
}
```

Resource code that represents a light source concept

```
public class LightingFactory extends ResourceFactoryImpl {
   public Resource createInstance(String classURI, String
      instanceURI, int factoryIndex) {
      switch (factoryIndex) {
      case 0: return new LightSource(instanceURI);
      // ...new instances for each index used in Ontology class
}  }  }
```

Factory code for serialization purposes

```
public final class LightOntology extends Ontology {
   private static LightingFactory fact = new LightingFactory();
   public static final String NAMESPACE =
      "http://ontology.universaal.org/Lighting.owl#";

   public LightingOntology() {super(NAMESPACE);}

   public void create() {
      Resource r = getInfo();
      r.setResourceComment("Descriptive comment...");
      r.setResourceLabel("Lighting");
      addImport(DataRepOntology.NAMESPACE);
      // ... add imports for every other used ontology
      // Create Light Source concept resource
      OntClassInfoSetup oci =
         createNewOntClassInfo(LightSource.MY_URI, fact, 0);
      oci.setResourceComment("Descriptive comment...");
      oci.setResourceLabel("Light Source");
      oci.addSuperClass(Device.MY_URI);
      oci.addObjectProperty(LightSource.PROP_HAS_TYPE)
         .setFunctional(); // When cardinality 1:1
      oci.addDatatypeProperty(LightSource.PROP_SOURCE_BRIGHTNESS)
         .setFunctional(); // When cardinality 1:1
      oci.addRestriction(MergedRestriction
         .getAllValuesRestrictionWithCardinality(
         LightSource.PROP_HAS_TYPE, LightType.MY_URI, 1, 1));
      oci.addRestriction(MergedRestriction
         .getAllValuesRestrictionWithCardinality(
         LightSource.PROP_SOURCE_BRIGHTNESS, new IntRestriction(
         new Integer(0), true, new Integer(100), true), 1, 1));
      // ... create the rest of concepts
} }
```

Ontology code that defines all concepts and their properties

```
OntologyManagement.getInstance().register(moduleContext,
       new LightOntology());
```

Application code to register the ontology at startup

identify its URI and properties URI, and helper methods to handle it. All concept Resources that can be instantiated and handled from code extend from uAAL´s *ManagedIndividual* (or another super-class which will eventually extend *ManagedIndividual*), which extends from *Resource*. If the concept is not supposed to be instantiated it can be defined as abstract, despite this is not a feature of Ontologies.

Properties are links between the concepts. They are also identified by URIs and can also inherit from other properties. They can have restrictions upon them, like cardinality.

For each property there usually are helper methods in its Resource class to get, set and possibly add a value, although it is always possible to use the generic method *setProperty*. It always checks that the value set satisfies the restrictions. When cardinality is greater than 1, multiple values can be set as a List.

Datatypes are the native data formats, like Boolean, Integer and so on. They are always present by default and don´t have properties.

These types are available through *TypeMapper* to obtain their URI.

Enumerations are sets of instances of Resources, representing different specific values that a property can point to.

They cannot be instantiated and therefore are created as abstract Ontological Resources, with convenient methods to initialize them in the Ontology class. They have a class representing them too, but it´s different from those representing normal Resources.

Context Events

Context Events are the minimal unit of context information sharing and are built on the Ontological model of uAAL. The minimal context information that can be extracted from an Ontological model is a link of two concepts through a property, modelled as a triple with subject, predicate and object. This is known as a statement. That is the structure of a Context Event, along with metadata.

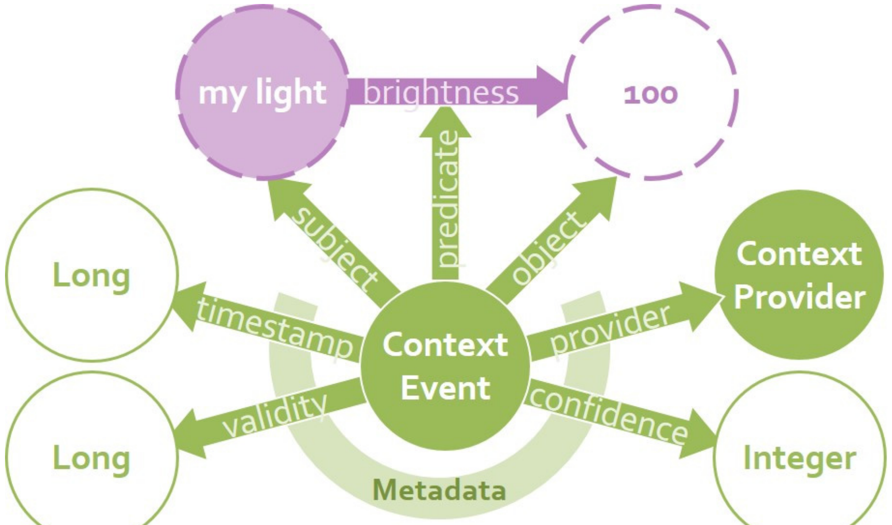
One way to understand Context Events is to think of them as simple natural language sentences. "User is in Kitchen", "Thermometer measures 25 ºC" or "Light is On" are typical examples of context information that can be shared in uAAL. This basic Subject-Predicate-Object triple, or SpO, is the core of the Context Event.

What the context event is composed of

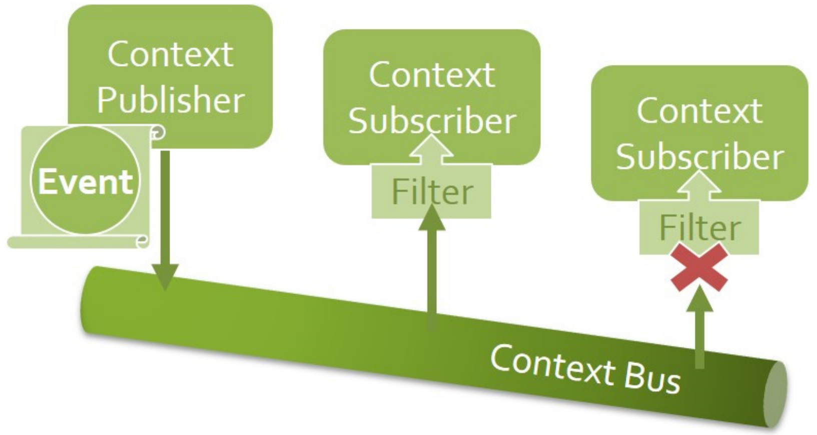
Subject is the concept the event is telling information about. It has to be an instance (an "individual") of a concept of the Ontological Model.

Predicate has to be one from all the properties that the Subject may have. It identifies the exact piece of information of interest about the Subject in this event, which value is the Object.

The **Object** is the value of the Subject´s property that works as Predicate. It is an instance (an "individual" or a "literal") of the type of concept that can be found at that Subject´s property in the Ontological Model.



A Context Event is composed of the mandatory SpO triple and some metadata



Publishers specify what events they broadcast and subscribers specify what they want

```
// Create an instance of the subject
LightSource light = new LightSource(MY_LIGHT_URI);
// Set the property to be used as predicate to a valid value
light.setBrightness(100);
// Create event with subject and predicate. Object is auto-set
ContextEvent ev = new ContextEvent(light,
   LightSource.PROP_SOURCE_BRIGHTNESS);
```

Application code to create a context event based on existing ontological instances

```
// Create the event directly
ContextEvent myevent=ContextEvent.constructSimpleEvent(
   MY_LIGHT_URI, LightSource.MY_URI,
   LightSource.PROP_SOURCE_BRIGHTNESS, new Integer(100));
```

Application code to create a context event directly

```
// Set some metadata:
//   Instantiate the Context Provider info as a Gauge
ContextProvider myprov = new ContextProvider(MY_PROVIDER_URI);
myprov.setType(ContextProviderType.gauge);
//   Set the provider in the event
myevent.setProvider(myprov):
//   Set the confidence to 75%
myevent.setConfidence(new Integer(75));
//   Set the expiration time to 1 day
myevent.setExpirationTime(new Long(86400000));
```

Application code to set metadata in a context event

The **Timestamp** of a Context Event is the metadata that marks the time at which an event was sent. It is a Long literal in UNIX format.

Context Provider is a piece of metadata that describes the application that generated the event. It is a concept contained in the default platform Ontological Model, with its own properties.

Confidence metadata indicates how reliable the Provider is about the information represented by the Context Event. It is an Integer literal representing a percentage.

Expiration Time metadata is a Long literal understood as milliseconds after which the information in the Event can be considered outdated.

Constructing a Context Event

The SpO triple is mandatory. Subject and Object must be instances of concepts validly connected by one of the Subject´s properties. More properties can be added to both Subject and Object instances but only the one designated as Predicate is guaranteed to be transmitted.

The Context Provider metadata is mandatory as well. If not manually set, it´s set automatically with the Publisher registration info. To do it manually, it has to be an instance of *ContextProvider*. To reduce Event size, set the Provider in the event manually without Provided Events (they are only needed in the Publisher constructor – see next section).

Timestamp metadata is also mandatory but is automatically added and cannot be set manually. Confidence and Expiration Time are optional.

Context Publishers

Context Publishers are applications that are capable of sending Context Events. They build these events with the Ontological model and broadcast them.

In fact the Publisher is the "part" of the application that sends context. An application may have many, but one is enough in most cases. Publishers define themselves and what kind of Events they intend to publish so that Context Bus can facilitate the matchmaking.

How to declare a publisher and send events

One way is to create an extension of *ContextPublisher*, but there is also a useful default implementation. Both need an instance of *ContextProvider* descriptor when constructed. It has two relevant properties to define.

Provider Type depends on what the whole application will do with the context information: Gauge if it only publishes context information it creates. Actuator if not only it publishes it but also allows to change it. Reasoner if the context published is inferred from other info.

Provided Events identify the type of events the publisher will send. They are described with Patterns, introduced in next section. If the application doesn´t know in advance what kind of events it will publish, it must use more loose Patterns, or even empty ones.

Once created, Publishers send Events by calling the *publish* method.

```
public class MyPublisher extends ContextPublisher {
   ...
   protected MyPublisher(ModuleContext context,
   ContextProvider providerInfo) {
      super(context, providerInfo);
         ...
   }
   ...
}
```

Context Publisher code showing only the typical constructor

```
// Instantiate the Context Provider info
ContextProvider myprov = new ContextProvider(MY_PROVIDER_URI);
// Set to type Gauge
myprov.setType(ContextProviderType.gauge);
// Set the provided events to "Unknown" with an empty Pattern
myprov.setProvidedEvents(new ContextEventPattern[] { new
   ContextEventPattern() });
// Create (and register) my Context Publisher. ModuleContext is
obtained elsewhere depending on the Middleware container
MyPublisher publisher = new MyPublisher(moduleContext, myprov);
...
// Publish an event
publisher.publish(myevent);
```

Application code to create and register a context publisher and publish a context event

```
// Create (and register) a Default Context Publisher. It simply
let´s you send events, which is enough in most cases
ContextPublisher publisher = new
   DefaultContextPublisher(moduleContext, myprov);
...
```

Application code to create a default context publisher provided by the platform

Context Subscribers

```
// This class will be a Context Subscriber
public class MySubscriber extends ContextSubscriber {
   // This is called when an event is received matching the
   subscription filter
   public void handleContextEvent(ContextEvent event) {
      //Do something with the event
   }
   ...
}
```

Context Subscriber code with the method to receive context events

```
// This defines two patterns. If any of them are met by an event,
it will be passed to handleContextEvent(event)
ContextEventPattern[] cep = new ContextEventPattern[2];
// This first pattern is for events about Lights from Gauge
Providers. Notice how ContextEvent is the root for Restrictions
cep[0] = new ContextEventPattern();
cep[0].addRestriction(MergedRestriction.getAllValuesRestriction(
   ContextEvent.PROP_RDF_SUBJECT, LightSource.MY_URI));
cep[0].addRestriction(MergedRestriction.getFixedValueRestriction(
   ContextProvider.PROP_CONTEXT_PROVIDER_TYPE,
   ContextProviderType.reasoner).appendTo(
   MergedRestriction.getAllValuesRestriction(
   ContextEvent.PROP_CONTEXT_PROVIDER,ContextProvider.MY_URI),new
   String[] { ContextEvent.PROP_CONTEXT_PROVIDER,
   ContextProvider.PROP_CONTEXT_PROVIDER_TYPE }));
// The second pattern is for events about any brightness change
cep[1].addRestriction(MergedRestriction.getAllValuesRestriction(
   ContextEvent.PROP_RDF_PREDICATE,
   LightSource.PROP_SOURCE_BRIGHTNESS));
// Create (and register) the Context Subscriber
MySubscriber subscriber = new MySubscriber(moduleContext, cep);
```

Application code to create filter patterns and create and register a subscriber

Context Subscribers are any application interested in consuming Context Events. They define a filter to restrict which types of Events they are exactly interested in.

Again, it is possible to have more than one Subscriber in an application.

How to register to receive events

It´s necessary to extend *ContextSubscriber*, there is no default implementation. When instantiated, it must be given the filter for events, as an array of Context Event Patterns. There are also methods in *ContextSubscriber* to dynamically change subscriptions afterwards.

Patterns are collections of restrictions over the fields of an Event, as if defining ontological restrictions over a Context Event concept. If one is received that complies with the restrictions, it´s passed to the Subscriber.

Restrictions in a Pattern work in an AND fashion, they all must be met to pass the event. But Patterns in the subscription array work in an OR fashion. An event will be received if it matches any of those Patterns.

Whenever an Event passes the filter used in the subscription, the *handleContextEvent* method of the Subscriber that defined it is called with that Event. The Event is guaranteed to contain the mandatory fields commented before: SpO, Timestamp and Provider. Other metadata may not be present unless required in the filtering Pattern.

Service Ontology

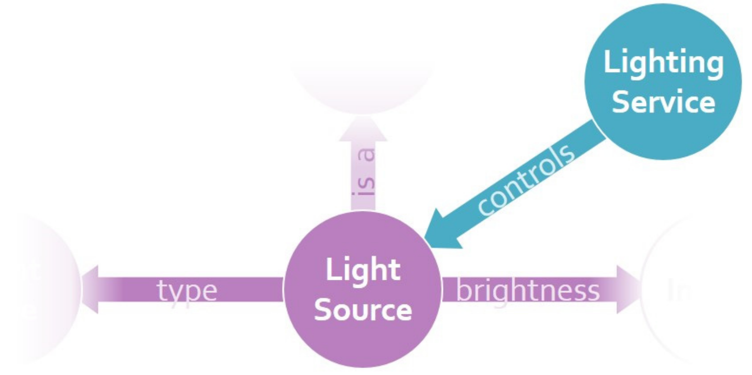
Service Ontology is the shared model between service requester and provider. It works as an anchor to the Ontological model. It also allows restrictions over the original model to make services more specific.

Service Ontologies are usually included in the related ontology domain module, but more can also be created in other modules or in the applications themselves. The module containing the Service Ontology must be available to both requester and provider.

Creating service ontologies

A Service Ontology is in fact just a single concept that connects to the Ontology Model of the domain it is going to handle, by one or more properties. It inherits from the concept Service included in the Service Bus. It should be linked to the most “root-like” concept(s) of domain ontologies, thus allowing providers to handle all possible combinations of the domain concepts. The property (or properties) linking to the domain is usually named MANAGES, CONTROLS or HANDLES.

Ontological restrictions can be defined over the Service Ontologies and therefore over all properties reachable through it. These can be used to narrow the possible services that can be provided with this Service Ontology. But in general these restrictions are defined by instances of the Service Ontology created by Service Callees (see next section), while generic Service Ontologies should be exactly that: generic.



Service Ontologies can be seen as an anchor to the ontological model

```
// Lighting concept is a service controlling the lights. The
property CONTROLS links it to LightSource concept
oci = createNewOntClassInfo(Lighting.MY_URI, factory, 4);
oci.addSuperClass(Service.MY_URI);
oci.addObjectProperty(Lighting.PROP_CONTROLS);
oci.addRestriction(MergedRestriction.getAllValuesRestriction(
   Lighting.PROP_CONTROLS, LightSource.MY_URI));
```

Ontology code to create a service ontology for lighting services

```
// This class represents the Lighting service concept
public class Lighting extends Service {
   public static final String MY_URI =
      LightingOntology.NAMESPACE + "Lighting";
   public static final String PROP_CONTROLS =
      LightingOntology.NAMESPACE + "controls";
   ...
}
```

Ontology concept code that represents the service ontology

Service Callees and Profiles

```
public class ProvidedService extends Lighting {
   // Profiles and restrictions will be placed here
   static final ServiceProfile[] profs = new ServiceProfile[2];
   private static Hashtable myRestrictions = new Hashtable();
   static {
      // This extends Lighting without using an Ontology class
      OntologyManagement.getInstance().register( new
      SimpleOntology(MY_URI, Lighting.MY_URI, new
      ResourceFactoryImpl() { public Resource
      createInstance(String classURI, String instURI, int index){
         return new ProvidedService(instURI); }
      }));
      // Custom restriction to say it only controls light bulbs
      addRestriction(MergedRestriction.getFixedValueRestriction(
         LightSource.PROP_HAS_TYPE, ElectricLight.lightBulb),
         new String[] { Lighting.PROP_CONTROLS,
         LightSource.PROP_HAS_TYPE }, myRestrictions);
      // Then define here the Service Profiles:
      // This profile is for getting all controlled lights
      ProvidedService prof1 = new ProvidedService(PROF1_GETALL);
      prof1.addOutput(OUTPUT_LAMPS, LightSource.MY_URI, 0, 0,
         new String[] { Lighting.PROP_CONTROLS });
      profs[0] = prof1.myProfile;
      // This one is for turning on a lamp (set brightness 100%)
      ProvidedService prof2 = new ProvidedService(PROF2_TURNON);
      prof2.addFilteringInput(INPUT_LAMP, LightSource.MY_URI, 1,
         1, new String[] { Lighting.PROP_CONTROLS });
      prof2.myProfile.addChangeEffect(
         new String[]{Lighting.PROP_CONTROLS,
         LightSource.PROP_SOURCE_BRIGHTNESS }, new Integer(100));
      profs[1] = prof2.myProfile;
      ...
   }
}
```

Provided Service code extending service ontology and defining restrictions and profiles

Service Callees are those applications that provide services of certain Service Ontology. They do so by registering Service Profiles.

Service Profiles are the equivalent to methods. They represent the operation to perform. Starting at the Service Ontology they describe arguments as a Path to a concept on which an Effect is expected.

An application can actually have more than one Callee. Each Callee can answer to multiple Profiles. The Profiles are registered when the Callee is constructed. It is also possible to add more or remove later.

A Callee can use its own instance of a Service Ontology to define its own restrictions to narrow its provided services. This is done by creating an extension of the Service Ontology, usually named Provided Service. The Service Profiles are usually defined in this Provided Service class too.

What is necessary to create a service callee

A **Provided Service** class is the most usual way to define the provided services. On one hand, it can define the desired restrictions because it is an extension of the Service Ontology to be provided. On the other hand, it´s also used to define the Service Profiles that will be handled by the Callee, each identified by a URI. If no restrictions are needed then it could also be possible not to have this Provided Service class and use the Service Ontology directly instead, declaring the Profiles elsewhere.

Then there is the **Service Callee** itself. It is an extension of *ServiceCallee* from the Service Bus. When created, it must be passed the Service Profiles it will answer to. Whenever a call is received that matches one of the profiles, its method *handleCall* will be invoked, along with the call. The Callee must then identify which Service Profile was called by looking into the call´s requested URI. Then it can work with the call, extract the arguments it knows it has, do something with it and return a response (success, error, timeout…), along with outputs, if any.

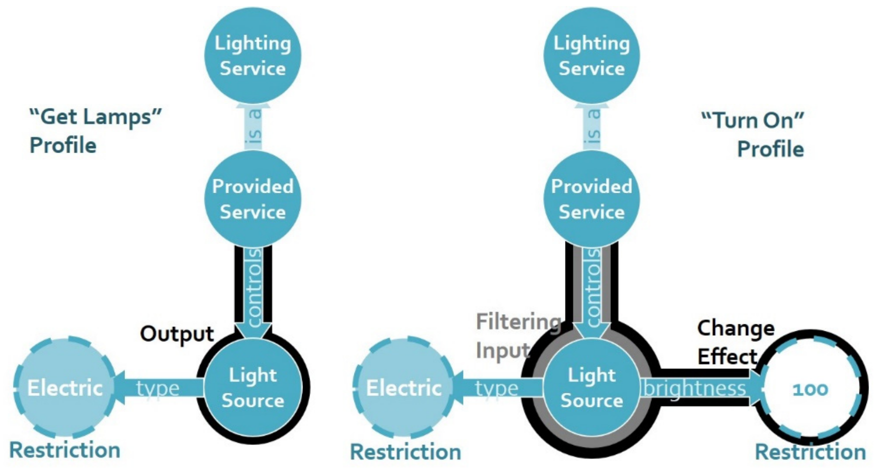
Declaring services with the service profiles

A Profile is created by first defining which Service Ontology is being used as starting point. This is done by creating instances of the Provided Service. Then the arguments are added, as many as need. The meaning of the operation is actually described by the combination of arguments and their restrictions.

Each argument is identified by a URI that can be later used by the Callee to handle arriving calls to the Profile. Each argument is described by a Path and an Effect.

The **Path** leads to a concept in the Ontological Model, starting from the Service Ontology. It is represented as an array of the successive properties to follow from the Service Ontology to the desired concept.

The **Effect** determines what is expected to happen with the concept at the end of the path. Possible effects are Add, Change, Remove, Filter and Output. These are set depending on the method of *Service* used to define the argument.



Service Profiles differ in the arguments defined over the service ontology model

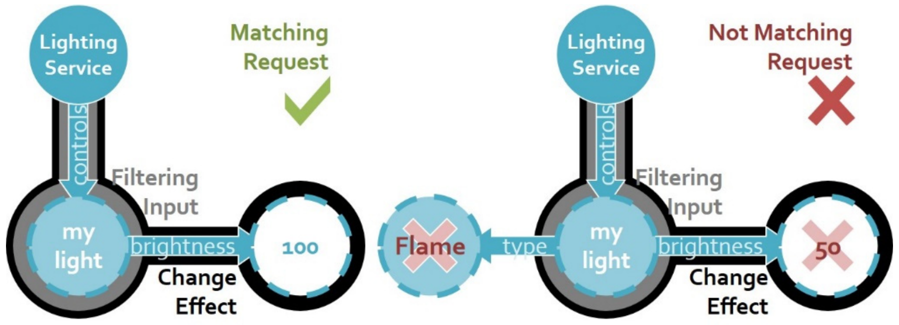
```
public class MyCallee extends ServiceCallee {
   // Called if a request matches a profile (find out which)
   public ServiceResponse handleCall(ServiceCall call) {
      String operation = call.getProcessURI();
      if(operation.startsWith(
         ProvidedService. PROF2_TURNON)){
         // Profile arguments can be taken from the call
         Object input = call.getInputValue(
            ProvidedService.INPUT_LAMP);
         // Make the light turn on here... then return success
         return new ServiceResponse(CallStatus.succeeded);
}  }  }
```

Service Callee code with the method to receive calls

```
// Create (and register) the Service Callee with the profiles
MyCallee callee=new MyCallee(moduleContext,ProvidedService.profs)
```

Application code to create and register a callee

Service Callers and Requests



Service Requests that match or don´t the "turn on" profile

```
public class MyCaller extends ServiceCaller {
   // Handles async. responses, when sendRequest(call) is used
   public void handleResponse(String requestID, ServiceResponse
      response) {
      // Handle the response
}  }
```

Service Caller code with the method to handle asynchronous responses

```
// Create the caller, which doesn´t need extra info
MyCaller caller = new MyCaller(moduleContext);
// Call "get all lamps". Using Lighting instead of Provided
Service and not specifying light type still matches the profile.
ServiceRequest req = new ServiceRequest(new Lighting(), null);
req.addRequiredOutput(REQUIRED_OUTPUT_LAMPS,
   new String[] { Lighting.PROP_CONTROLS });
// Synchronous call. Asynchronous would be sendRequest(call)
ServiceResponse rsp=caller.call(req);
// Deal with response. There are other methods to get outputs.
if (rsp.getCallStatus() == CallStatus.succeeded) {
   List lampList = rsp.getOutput(REQUIRED_OUTPUT_LAMPS, true);
}
```

Application code to create a request, create a caller, send the request and get outputs

Service Callers are the applications that request the execution of a service. This is achieved by issuing Service Requests. The Requests are matched to registered Profiles and if they are ontologically equivalent, the Callee(s) that registered them will be called and will give an answer.

Service Requests are the counterpart of Service Profiles, built the same way but declare what the Caller wants to execute.

One Caller is enough for most applications, but there can be more. There is also a default implementation. Nothing is needed at construction.

Calling services with service requests

The default implementation is enough to send requests and getting the response. But if this must be handled asynchronously, *ServiceCaller* must be extended, and responses handled in *handleResponse* method. In any case, requests are sent with *call* method of the Caller.

What is sent in the call is a Service Request, which is built the same way as the Service Profile it intends to call: First create it with the root Service Ontology, and then add arguments with paths and effects. But because services are semantic it doesn´t have to be identical to the Profile: just call what is needed. All services which Profiles match the Request will respond, which means that a response may have answers from many services. This is relevant if outputs are requested.

User Interaction Forms

Forms are the ontological representation of the typical user interaction components, like textual inputs, multiple selections, buttons, and so on. Forms are created by UI Callers and sent to UI Handlers to be rendered, filled by user, and sent back to UI Callers to be processed.

Just like the other Ontological models, Forms can be extended, but the most usual interfacing options can be achieved with the default Forms.

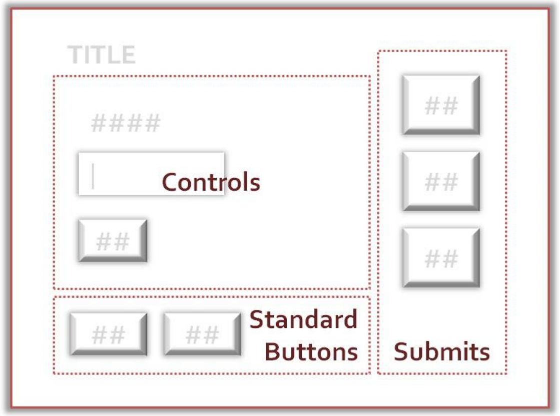
What the forms are built of

There are three groups in a Form. **Controls** is where common UI Elements are put, including trigger buttons. **Submits** is for common buttons that finish a Dialog and/or do something. And **Standard Buttons** is for system buttons, not usually changed by applications.

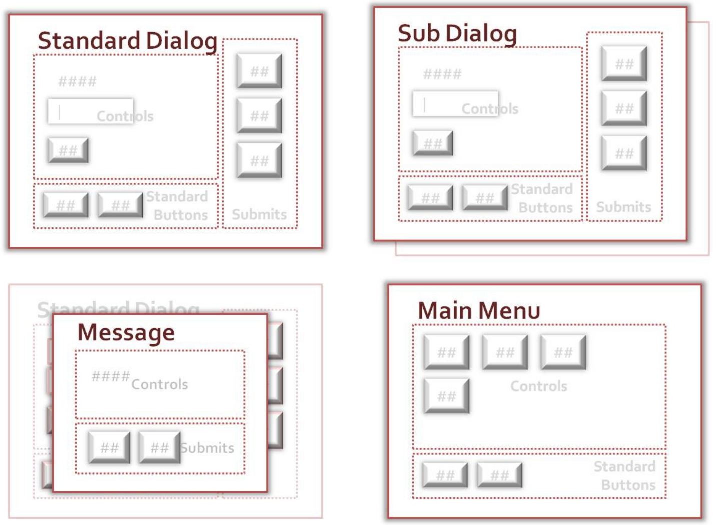
A Dialog is an interaction with a Form. There are 4 different types. **Main Menu** is for the main screen only, and is not used by applications. **Standard Dialog** is the normal Dialog. **Subdialog** is a Dialog triggered by a previous one, which comes back after the Subdialog finishes. **Message** is a popup that may appear regardless of current dialog, on top of it.

How to compose a user interaction form

The first thing is instantiating a Form depending on the type of Dialog that is intended, with the right method of *Form* class. An instance of an



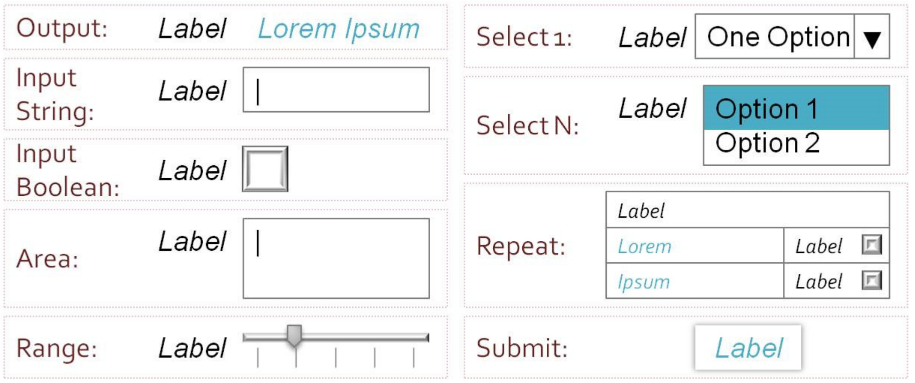
Typical arrangement of the groups of a form in a graphical interface



Typical graphical representations of the different types of dialogs

```
// Create a normal Dialog, and don´t use Resource (it´s empty)
Form f = Form.newDialog("Title of the Dialog", new Resource());
// Show a simple output saying what it does
SimpleOutput out = new SimpleOutput(f.getIOControls(), null,
   null, "Select a Lamp");
// A choice selector to select one of 2 lamps
Select1 sel = new Select1(f.getIOControls(), new Label("Lamps",
   null), PATH_LAMP, null, new Integer(1));
sel.addChoiceItem(new ChoiceItem("Lamp1", null, new Integer(1)));
sel.addChoiceItem(new ChoiceItem("Lamp2", null, new Integer(2)));
// Buttons to turn on and off
new Submit(f.getSubmits(), new Label("On", null), ID_SUBMIT_ON);
new Submit(f.getSubmits(), new Label("Off", null),ID_SUBMIT_OFF);
```

Application code to create a form



Typical graphical representations of the default UI elements

Ontological Resource can be used to later fill the fields of the Form with initial values, but it is not mandatory, and not necessary in most cases.

UI Elements are added as they are created, to the group passed in their constructor. Most elements can optionally have a Label, an initial value and a Property Path. The Path is necessary for Input Elements: it is used to retrieve the value introduced by the user. It can also be used to link the Element value with the Resource used to build the Form, if any. If no Resource is used (or an empty one is) the Path can be arbitrary.

Outputs are for displaying information to the user, without requesting any information. Property Paths are not mandatory for these. Multimedia outputs will be limited by the handler rendering the information.

Inputs are Elements where a value is expected to be set by the user. There are Elements for free textual input or pre-set choices.

Submits are for issuing actions. In graphical interfaces these are buttons, so Submits are often called buttons. They have an ID used to know which Submit was issued ("pressed") by the user.

Labels are identifiers of UI Elements and must not be mistaken for Outputs. They are only for the user to identify the different Elements.

Groups and **Repeats** are not Elements themselves but containers that can be used to place elements inside (Remember the Controls, Submits and Standard Buttons in the Form? Those are Groups). The Repeat is used for creating tables or lists of similar elements.

User Interaction Callers

User Interaction Callers are the applications that want to have some kind of direct interaction with the user. They build a Form that represents exactly what they want to show to the user and what they want in return.

There can be many UI Callers in an application, but one is usually enough. It is the only thing needed for both sending and receiving on UI Bus.

Sending requests and handing responses with a caller

UI Requests ship the Forms to the UI Bus. To address the proper Handler, they are passed the target user, language, required privacy disclosure and priority level. Requests are sent with *sendUIRequest* method of the Caller.

Once the request is displayed and filled by the Handler, the **UI Response** returns to UI Caller through *handleUIResponse*. The Caller can analyse which Submit triggered the response, and can extract the filled inputs from within with the right Property Path. If no more Forms are sent from there, the system will automatically go back to the Main Menu.

Requests can be sent at any time but to allow a user to manually "launch" an application dialog this must be registered in Main Menu. This is done though Service bus. A special Service Profile must be registered and there must be a Service Callee that launches the application main dialog when the Profile is requested. The profile is called by the Dialog Manager Main Menu configuration files, adding a special line for each button.

```
public class MyUICaller extends UICaller {
   // This is called when the response to a UI request arrives
   public void handleUIResponse(UIResponse resp) {
      // Check which submit “button” was “pressed”
      if (resp.getSubmissionID().equals(ID_SUBMIT_ON)) {
         // This is how user input is taken
         lampNumber=(Integer)resp.getUserInput(PATH_LAMP);
         // Turn on the lamp identified by lampNumber
      }
      ...
      // If no more Forms are sent here, system returns to menu
}
```

UI Caller code with the method to handle responses

```
// Create the caller, which does not need extra registration info
MyUICaller uicaller = new MyUICaller(moduleContext);
// Create a UI request for a given user and send the Form f
UIRequest req = new UIRequest(new User(USER_URI), f,
   LevelRating.middle, Locale.ENGLISH, PrivacyLevel.insensible);
// Send the UI request
uicaller.sendUIRequest(out);
```

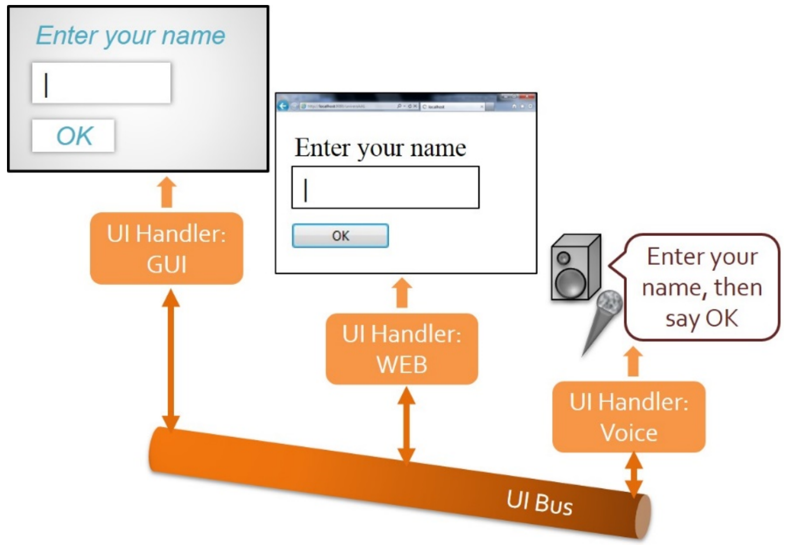
Application code to create a caller, create a ui request with a form and send the request

```
// Profile that starts the app UI. Service Callee must handle it
and send UI request with app dialog when PROF_STARTUI is called
ServiceProfile prof =
   InitialServiceDialog.createInitialDialogProfile(
      "http://my.ont.org/MyApp.owl#MyMenu ",
      "http://mycompany.com ", "My App Menu", PROF_STARTUI);
```

Provided Service code including a new profile to start app main dialog

```
/My App|http://mycompany.com|http://my.ont.org/MyApp.owl#MyMenu
```

Main Menu txt file line that will display a button that will call the above profile



UI Handlers render the same form differently

```
public class MyUIHandler extends UIHandler {
   // Called when a request is sent to this Handler. It must
   render the Form it carries and put the user input inside
   public void handleUICall(UIRequest uiRequest) { ... }

   // If some preferences of the user change, the Handler may
   want to adapt to them, like screen size
   public void adaptationParametersChanged(String dialogID,
           String changedProp, Object newVal) { ... }

   // It may happen that a dialog is interrupted for some reason
   public Resource cutDialog(String dialogID) { ... }

   ...
}
```

UI Handler code with the required methods only

User Interaction Handlers

User Interaction Handlers are special types of applications in charge of translating the Forms sent by UI Callers to a physical rendering that a user can interact with, such as a GUI, a sound output or Web page. Then interpret the user responses to fill in the information requested by UI Callers into the Form and send it back. There can be several UI Handlers in different locations, with different modalities, and the UI Callers are oblivious to them, thus achieving multi-modal and multi-location interaction.

UI Handlers are special applications that are considered Managers in the platform. Developers of everyday applications would never have to deal with UI Handler code, since the UI Bus makes these applications completely agnostic of these issues, unless a developer is interested in developing specifically a new UI Handler.

How user interaction handlers work

Despite coding UI Handlers is out of scope in this document, it may be interesting to know how they work. Every UI Request sent by UI Callers is added automatically some metadata about the addressed user. This is used by the UI Bus to find the most suitable Handler, because Handlers are registered with a set of adaptation parameters. The Handler with the most appropriate parameters is selected to render the Request, and then will have to send back the user input.

Managers

A uAAL Application is the software part of an AAL Service, and is understood as a piece of software that communicates with others by making use of the uAAL Execution Platform. A Manager is an Application that is part of the platform itself and is necessary for its proper operation, or provides relevant basic services or events for other applications.

Some Mangers are mandatory for the platform to work. Some have to be present only in one Node but not the rest. Others simply provide convenient services for upper Applications or other Managers.

List of managers provided by uAAL

Context History Entrepôt: It stores all Context Events sent through Context Bus and maintains the status of the context information updated according to them. It provides services for consulting the history of events and obtaining the current context information status. The platform can run without it but it is crucial for other Managers, so it can be considered mandatory. There must only be one instance of it.

Profiling & Space Servers and Editor: These servers provide convenient services in the Service Bus for applications to obtain context (profile and space) information without having to deal with CHE. The editors allow a human user to edit this information. They are not mandatory but strongly recommended.

Situation Reasoner: Listens to Context Bus and analyses basic Context Events. Based on a set of rules and conditions it composes new higher level events based on those, and they are sent through Context Bus. It provides services to add or edit those rules. It is an optional Manager.

Dialog Manager: This Manger is mandatory for the UI Bus to work properly. It takes care of maintaining the uAAL Main Menu interface and initiating the application interfaces listed there. It also handles UI dialog stack and manages adaptation properties, which are vital for enabling the right UI Handler. There must only be one of these.

UI Handlers: Refer to the "User Interaction Handlers" section for details. Currently uAAL provides a few "official" handlers, like Swing-based GUI Handler for PC mouse + keyboard interaction and Web Handler for Web Browser interaction. Other experimental Handlers are in store.

Resource Server: Mandatory for most UI Handlers, applications can store multimedia resources here that can be later referenced by the Handlers.

Remote Import/Export: These managers act as proxies that can represent external web services as universAAL services inside the space, or publish universAAL services of the space to the outside as web services.

Other Managers: There are other optional Managers providing useful features such as Security Authorization, Document Encoder or the Device Exporters and AAL Space Gateway described in the first section.

The universAAL Primer - Advanced