Revista Română de Interacțiune Om-Calculator 6 (3) 2013, 249-268

The Generic Interaction Protocol: Increasing portability of distributed physical user interfaces

Gervasio Varela[1], Alejandro Paz-Lopez[1], Jose A. Becerra[1], Richard J. Duro[1]

[1]Integrated Group for Engineering Research – University of A Coruña
C/ Mendizábal, S/N, 15403, Ferrol, Spain
E-mail: gervasio.varela@udc.es

Abstract. Natural user interfaces want to liberate the user from having to learn new concepts to interact with computers. They do that by taking advantage of our senses and our own knowledge about world in order to build the user interface. Physical user interfaces are a prominent example, where physical objects are enhanced to represent actions or information that the system must exchange with the user. One of the main drawbacks of physical user interfaces is often related with the difficulty to decouple system logic from the specific technology used to build the user interface, especially when multiple environments or scenarios should be supported. This paper presents an abstraction technology for user interaction devices that allows the building of physical user interfaces that are physically and logically decoupled from the system logic.

Keywords: usixml, distributed user interfaces, physical user interfaces, ambient intelligence, ubiquitous computing.

1. Introduction

Since the beginnings of user interaction research, natural user interfaces (UIs) have been one of the most relevant and studied topics. The idea was to liberate the user from having to learn new concepts in order to interact with computers, and take advantage of its own senses and knowledge about objects, physics and the world itself, to interact with computers.

Tangible UIs (Ullmer, 2000) (Patten, 2001), graspable UIs (Fitzmaurice, 1995) and physical UIs (Greenberg, 2001) (Ballagas, 2003) are prominent examples of natural interface research. The main idea shared among them is to rely on physical objects and their intrinsic and known properties, in order to build the user interaction primitives of a digital system. Tangible and graspable UIs, like tabletop UIs (Kaltenbrunner, 2005) (Jordá, 2007), usually allow the user to manipulate some physical objects of different

shapes or colours, and use the different manipulation actions to interact with the user. On the other hand, physical UIs rely much more on ad-hoc physical objects, or even on enhanced everyday objects, to provide physical representations of the interactive capabilities of the system.

The main idea behind physical UIs is to take advantage of physical objects to bridge the gap between the users and the state of a digital system. These objects are controlled by the digital system and provide a physical representation of actions or outputs that the system must present to the user. They range from everyday objects integrated in the environment, to specific setups like dedicated appliances or even cockpits.

A fairly typical and well-known example of a physical UI using everyday objects is a smart building or home, where some systems have been replaced with digitally controlled devices, like home automation devices. Regarding specific appliances, prominent examples could be the pinwheels[3] project, where a series of rolling pinwheels are used to show the network traffic of a server, or the famous marble answering machine[4] by Bishop, where physical balls are presented to the user to represent individual messages left in the answering machine, to hear a message the user selects a ball and introduces it in the machine.

One of the main advantages of using physical objects instead of virtual representations is that the knowledge of users about the physical objects is inherently richer than the knowledge about virtual ones:

- People already know the physics of things.
- Spatial positions and physical shapes matter and the user can infer new meanings or purposes for the same objects.
- If it is an everyday object, people usually have a previous knowledge about the purpose of the object and how to use it.

By introducing characteristic like these, an UI will be usually considered more natural and more integrated in the environment, which is an important characteristic for many kinds of systems. Ubiquitous Computing, Ambient Assisted Living or Ambient Intelligence (Augusto, 2011) (Dadlani, 2011) (Varela, 2013) systems have natural interaction and environment integration

[3] http://tangible.media.mit.edu/project/pinwheels/

[4] http://vimeo.com/19930744

as key objectives. But there are other systems that can benefit from such characteristics, like video games or simulators (Villar, 2007a) (Villar, 2007b).

Nevertheless, even if the benefits can be great, physical UIs introduce a large amount of complexity in the system from the point of view of the developers of a digital system, as it is usually quite difficult to decouple the system from the UI and its associated technologies.

More often than not, physical UIs are designed ad-hoc for a very specific use case, and they are implemented using custom hardware and connected to the digital system by specific interfaces. These systems are often coupled not only to the hardware devices, but also to the features of the scenario, like environment conditions and user characteristics. According to those specific features, the devices and technologies utilized, and even the shape and behaviour of the UI can change. This makes these systems difficult to deploy in different scenarios (Dadlani, 2011).

The variety of usage scenarios and environments is a key aspect of Ubiquitous Computing and Ambient Intelligence systems, because their goal is to be ubiquitous and provide their functionalities, in a natural way, regardless of where the users are. A UI to be perceived as natural should bear in mind the conditions, characteristics and preconceptions of the physical environment, so that it uses the most appropriate modalities and devices for each scenario.

Supporting different modalities and devices for user interaction makes the implementation of the system much more complex. There exist many toolkits for building smart physical objects (Ballagas, 2003) (Greenberg, 2001) (Villar, 2007), a wide variety of home automation technologies, and even for some scenarios, ad-hoc hardware built using Arduino or other custom solutions would be required. Developers are exposed to that variety of solutions and they must know how to use the different technologies involved, and how to exploit the diverse modalities.

The main objective of the work presented in this paper is to alleviate this complexity by decoupling the developers from the particularities of each hardware and interaction technology required for each scenario. This paper presents an UI development framework that, relying on model-driven engineering techniques and distributed hardware abstraction technologies, facilitates the development of UIs decoupled from the technologies and locations of the devices chosen to interact with the user. This framework has

two main components. A device abstraction technology, the Generic Interaction Protocol (GIP), that encapsulates the specific behaviour of sensor and appliance devices behind a generic interface of distributed interaction actions, and a model-driven UI management system allowing the description of the UI using high level models, whose abstract elements are connected at deploy time to a selection of distributed devices.

The solution proposed in this paper shares some similarities with the iStuff project (Ballagas, 2003). Both of them provide distributed access to smart physical objects, and both of them rely on proxy-like elements to encapsulate the specific behaviour of each device. Nevertheless there are two main differentiation factors. On the one hand, the GIP has been designed as an abstraction technology for user interaction devices, therefore any device can be accessed, from the developer point of view, using the same homogenous API, while iStuff has been designed as a technology to interconnect devices and not abstract them. On the other hand, the GIP follows a model-driven approach, thus allowing the utilization of model-driven engineering techniques to build physical UIs.

This paper is organized as follows. In section 2 a conceptual overview of the GIP is provided, while in section 3 an actual implementation using multi-agent systems is shown. Section 4 is dedicated to the exploration of an example system and how it can be implemented using the proposed solution. Finally, in section 5 some conclusions are extracted.

2. The General Interaction Protocol

The main objective of the solution presented in this paper is to decouple a digital system from the particularities and specifics of the multiple hardware technologies required to build a distributed physical UI. In this section we are going to present the conceptual framework and workflow of the solution, while in the next section we will present a concrete implementation of the framework.

The core element of the solution is the introduction of a new layer of abstraction for distributed hardware devices. This new layer encapsulates the specific behaviour of each device behind an interface of generic user interaction operations that can be accessed remotely. By relying only on the set of operations supported by this generic interface, physical UI developers

can design their UIs at an abstract level and implement them in a uniform way, independent of the underlying technologies. Only at deploy time, or even runtime, do they have to decide which concrete devices will be used in each scenario.

This new layer of abstraction is called the Generic Interaction Protocol (GIP) and it is conceptually conceived as a distributed communications protocol that defines a reduced set of interaction actions that create a generic remote interface to any kind of interaction device. By implementing this interface, any device or interaction resource can be accessed using the same set of concepts and operations, thus decoupling the application from the underlying interaction technologies and the location of the devices.

Error! Reference source not found. shows the conceptual architecture of the solution. As can be seen, the GIP is right between the system logic and the interaction resources (physical devices, graphical widgets, voice recognition software, etc.), decoupling them in two ways. On the one hand, as the GIP is a distributed protocol, it provides physical decoupling between the system logic and the UI components. On the other hand, it provides logical decoupling by isolating interaction resources behind a common generic interface.

The Generic Interaction Protocol has been designed to be integrated into a model-driven engineering solution for UI development. Following the abstraction level decomposition of the Cameleon (Balme, 2004) framework, it is designed to substitute the Concrete User Interface (CUI) level. The GIP provides a way to directly map Abstract User Interface (AUI) elements to the Final User Interface (FUI) elements at runtime, thus eliminating the need for a CUI model, because the actual hardware available is already known at the time the mapping is defined.

The common generic interface provided by the GIP is in charge of abstracting the behaviour of concrete interaction resources. Thus, it should be generic enough to support multiple kinds of modalities and interaction devices. We decided to inspire its design in a well-known UI abstraction technology, the UsiXML Abstract User Interface Model[5] (Limbourg, 2004)

Figure shows a simplified class diagram of the UsiXML Abstract User Interface model. UsiXML employs the concept of Abstract Interaction Unit

[5] http://www.usixml.eu

(AIU) as a representation of the typical widgets found in graphical user interface toolkits. Within the GIP this concept represents any kind of interaction element, such as physical devices like appliances or sensors, or even gestures or GUIs. Using the abstract UI model, a developer describes the application UI as a collection of interrelated AIUs representing the different high-level user interactions (data input/output, action triggering, and data selection) required by an application. The GIP interface is designed to match this set of generic interactions so that it is easier to establish a map between an AIU and an interaction resource that implements the GIP.

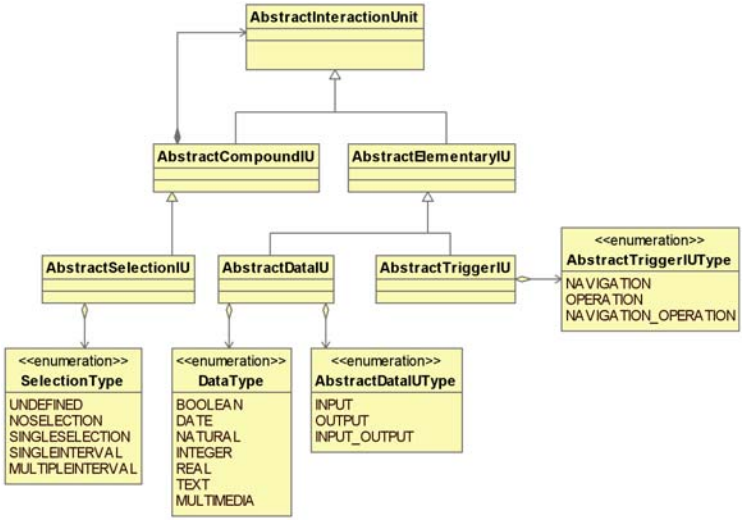

Figure 1. UsiXML Abstract User Interface simplified class diagram

The GIP is an event based distributed protocol following a hybrid model of publish/subscribe and one-to-one communications. On one hand, a series of publishers, the interaction resources, publish events notifying actions that the user has performed (input/selection of data or activation of an action) and a series of subscribers (usually the system UI controller) receive those events and react accordingly. On the other hand, when the system logic has to send some event to an interaction resource, it does it by using one-to-one communications, this way, the interaction resources are more decoupled from the system logic.

As can be seem in Figure , the UsiXML AUI model proposes three

different types of abstract interaction units (AIUs) to describe the UI interaction requirements at an abstract level:

- *Data unit*: can be either for input, output or both.
- *Selection unit*: for input of data from a predefined set.
- *Trigger unit*: for activating operations or actions.

The GIP interface, as can be seen in **Error! Reference source not found.**, is made up of five different events: input, output, selection, action and focus. The first four events are directly inspired by the UsiXML AUI model:

- *input*: an interaction resource informs its subscribers that the user has performed a data input action
- *output*: the system logic commands one or many interaction resources to output some data to the user
- *selection*: this event has two different meanings depending on the sender. If it is sent by an interaction resource, it means that the user has made a selection. Otherwise, it means that the system logic requires an interaction resource to show a selection to the user.
- *action*: an interaction resource informs its subscribers that the user has triggered an action
- *focus*: the system logic requires an interaction resource to gain focus over the user attention

It is not mandatory for every interaction resource to support all the GIP events. There can be many kinds of interaction resources with different levels of support for user actions. Some will support input and output, while others will support only input or will not be able to reclaim the focus of the user.

Every GIP event has an associated set of data properties that indicate the data an interaction resource is able to either output to the user, or gather from it as input. These data items are represented as a string and have an associated basic type (integer, double, byte or string), so that the system can know what kind of information an interaction resource is able to represent.

In order to allow some level of customization of the user interface, GIP events are enhanced with a set of properties called Interaction Hints (IH). They are a set of fixed properties that developers can use to provide indications to the interaction resources about an interaction action (for

example: priority, size or colour). The support for IHs is not mandatory, and each interaction resource can interpret them as it wants.

Even if the GIP design has been inspired by the UsiXML AUI model, it is generic enough to be used in combination with other UI abstraction languages or technologies, or even without previously specifying the UI at an abstract level. Nevertheless, in our implementation, which will be presented in the next sections, we are using the UsiXML AUI model to describe the UI of a system, and then, at deploy time, the Abstract Interaction Units are associated and connected to one or several interaction resources.

In order to use the GIP to build a UI for a system, developers and installers must follow a workflow with three different steps:

- *Design the UI at an abstract level*: The goal of this step is to establish the user interaction requirements of the system, what inputs, outputs, actions, etc.
- *Select or implement the required interaction resources*: For each input, output, selection, etc., specified in the AUI, one or more interaction resources are required. Those interaction resources could be either pre-existing ones that already implement the GIP protocol and interface, or they can be built specifically for one application. The key point here is that even if they are newly developed devices, they behaviour must be encapsulated behind the GIP interface, decoupling the system from this ad-hoc device and, furthermore, allowing both of them, system and device, to be used independently in the future.
- *Deploy the system and UI*: This step implies the installation of the system in each scenario and the connection of the AIUs to the interaction resources.

The details of each step are implementation dependent because, as we said before, different languages or technologies can be used to specify the UI at abstract level and the GIP can be implemented in different ways.

3. Dandelion: A framework for physical UIs in Ambient Intelligence and Ubiquitous Computing

In the previous sections we have provided a conceptual overview of the GIP. In the current section we are going to present an actual implementation of it. This implementation is called Dandelion and it has been designed to build distributed physical UIs for Ambient Intelligence (AmI) and Ubiquitous Computing (UC) systems. In fact, Dandelion is being implemented as the user interface development framework of the HI^3 AmI platform (Paz-Lopez, 2012). It follows a model-driven approach inspired by Cameleon-RT (Balme, 2004), and it is integrated within the HI^3 architecture, which is described in detail in ((Paz-Lopez, 2012)(Varela, 2012)).

Dandelion can be divided into two main blocks. On one hand, a model-driven UI management system allows developers to describe the UI using the UsiXML abstract UI model. On the other hand, a GIP implementation provides a distributed communication interface encapsulating devices behind a set of generic interaction actions, thus allowing a decoupled connection of the abstract UI definition to a set of distributed elements that perform the real interaction with the user.

As part of the HI^3 platform, Dandelion is implemented using a multi-agent paradigm. Figure shows a block diagram containing the different components implementing Dandelion. The elements with the round edges and continuous lines are agents or multi-agent systems, while the elements with doted lines are implemented as plain old java objects.

The model-driven UI management system is implemented by the abstract UI model and the UI Controller (UIC), which is in charge of managing the connection between the abstract UI definition and the devices. The abstraction of devices is conceptually provided by the GIP and physically realized by the Final Interaction Objects (FIOs). They implement the GIP interface in order to provide software abstractions of specific devices.

Following Figure , from top to bottom; the application logic and the UI description models are the only elements provided by the developer. It describes the UI using the UsiXML Abstract UI model, and then connects application data objects and actions to that UI description. The UIC

manages this connection by keeping a mapping between the data objects, the elements of the UI model and the real devices that will be used to interact with the user. Those real devices are represented in the system by the Final Interaction Objects (FIOs). As indicated before, they encapsulate the specific logic, characteristics and particularities of each device behind the GIP interface.

The physical connection between the UIC and the FIOs is decoupled by the GIP. The UIC monitors the application, and when it detects a change in the data objects associated to the UI model, it sends a GIP event to the FIOs mapped to the corresponding UI element. Conversely, the UIC acts as the subscriber in the GIP, so it is subscribed to GIP events coming from different FIOs. When the UIC receives a GIP event, it uses the mapping to know which application data object or action needs to be updated or fired. This process is transparent to the application and the developer.

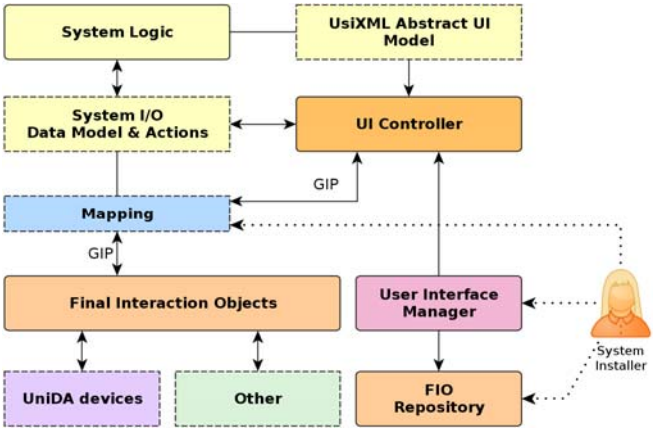

Figure 2. Block diagram of the Dandelion system implementing the GIP

In Dandelion, the GIP is implemented as a distributed agent communications protocol using the FIPA platform and taking advantage of the publish/subscribe communications features of the HI^3 platform. Because of that, the UIC and every FIO are agents implementing the GIP interface. They use the agent messaging support of the HI3 platform to publish GIP events, and receive and process them.

The mappings between elements described in the abstract UI model and the FIOs are stored by the UIC, but they are managed by the User Interface

Manager (UIM). While Dandelion is designed to support the autonomous selection of FIOs at runtime, in the current version this mapping is performed manually by the system installer; who at deployment time selects, out of the available FIOs, those that better suit the requirements of the application, the user and the environment.

In the next subsections the UIC, the FIOs, and how they implement the GIP, are explored in detail.

3.1 The User Interface Controller

The User Interface Controller is implemented as an agent, and its operation is based on the observer pattern. Apart from specifying the UI at an abstract level, the developer must provide a set of data and action objects that will be used by Dandelion as the I/O interface between the application logic and the user. These objects represent either actions of the system that can be triggered by the user from the UI, or data that must be output to the user or retrieved from the user as input.

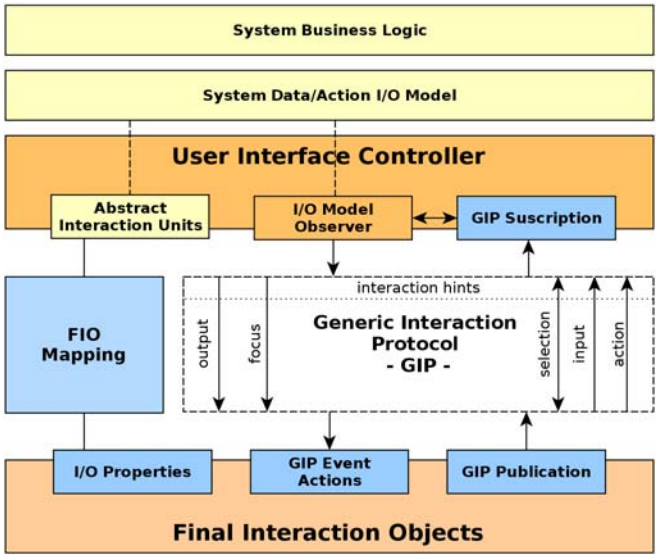

Figure 3. Detailed view of the Dandelion GIP implementation and the relationships between the UIC and the FIOs

Every data or action object must be associated to an Abstract Interaction

Unit (AIU). This association is established programmatically by the developers using the API of the UIC, which allows them to associate a data or action object to an AIU from the UI model, and a callback object that will be used by the UIC to notify the GIP events to the system logic. It is also at this time when the optional Interaction Hints can be established for each association between AIU and data/action object.

As shown in Figure , the UIC uses the observer pattern to monitor I/O objects, detect changes, and send the data to the FIOs, and vice versa. When a GIP event is received by the UIC (it is subscribed to the GIP events of the associated FIOs), it looks for its associated AIU, and uses the callback to notify the change to the system logic. The information about what AIU is associated to what FIO is obtained from a mapping that is stored by the UIC. This mapping is defined by the system installer at deployment time; who connects the data objects and actions to the different GIP properties supported by each FIO. It is stored by the UIM until the system is running, when the mapping is transferred to the UIC. If the user moves from one place to another, or if the installer wants to deploy the system in another scenario, it is only required to change that mapping, which can be done even at runtime without affecting the system logic.

The combination of the abstract UI model with the observer pattern to drive the connection between the system logic and the UI, allows a great level of decoupling between the system and the user interface. Nevertheless, this generic description of the UI and its behaviour has a great impact on the possibility of customizing the UI. This problem can be alleviated to some extend by using the GIP Interaction Hints.

3.2 The Final Interaction Objects

The Final Interaction Objects are the end elements in charge of physically interacting with the user. They are software abstractions of heterogeneous interaction resources that could be either hardware (appliances, sensors, etc.) or software (GUIs, voice recognition, etc.).

FIOs are implemented as agents realizing the GIP interface in the sensing/actuation layer of HI^3. Figure illustrates the relationship between FIOs, the GIP, and the UIC. They assume the role of GIP publishers, abstracting the behaviour of a device as a set of events that notify the

different actions the user is performing. Each FIO implementation is only required to support a subset of GIP events because there can be devices that only support input, output, etc. The supported GIP events and the data properties a FIO can input/out are specified in a FIO description, which can be consulted by the system installer using the FIO repository. See Figure .

The data properties associated to a FIO indicate the kind of data a FIO can obtain from the user as input or selection, or show to the user as output. They can be any basic type: integer, float, string or boolean. Each FIO can provide one or many properties, each of them associated to one kind of GIP event. For example, a FIO can specify that it is able to output a string and input an integer. This combination of data property type and GIP event is what a system installer associates to an AIU at deploy time. For example, an output AIU for a string type can be associated to an LCD display encapsulated by a FIO with an output event and a string data property.

For every different kind of device or interaction software used by a system, a FIO must exist that abstracts it using the GIP. Obviously, a key point is that developers should not need to develop their FIOs, or at least, not many of them. They should be provided by Dandelion itself or by the manufacturers of the interaction resources.

In order to alleviate the problem of developing FIOs for the wide number of different devices and technologies available, Dandelion makes use of the hardware abstraction layer of HI^3, UniDA, which provides a generic interface to remotely access and use any kind of hardware device. In UniDA every device is accessed using the same generic operations and concepts, and each type of device is reduced to a set of common operations. It is, consequently, possible to use similar devices from different manufacturers or technologies using the same exact API. This way, one FIO can support a wide number of physical devices.

4. Building a distributed physical UI with Dandelion

In this section we are going to show, step by step, how Dandelion can be used to implement a distributed physical UI. For this purpose we are going to explore the implementation of a ubiquitous music player application. This application follows the user from one place to another while playing her

favourite music. As the user moves, the application should continue to play the music, but also to provide a way to control the reproduction.

In order to achieve a truly ubiquitous music player, the UI must be well integrated in each environment and it should be perceived as natural. Therefore, in contrast to a classical system that would use a graphical UI, our example system will use a hybrid approach with a combination of user interaction modalities depending on the environment characteristics and the devices available in each scenario.

```
<AbstractUIModel
        xmlns:usi="http://UsiXML-XSD/AbstractUIModel"
        xmlns:xs="http://www.w3.org/2001/XMLSchema">

        <AbstractCompoundIU id="TrackInfo" role="grouping" importance="3"
                repetitionFactor="1">
                …
                <AbstractDataIU id="SongTitle"
                        role="" importance="5" repetitionFactor="1"
                        maxCardinality="1" minCardinality="0"
                        defaultValue="" dataType="TEXT" dataFormat=""
                        dataIUType="OUTPUT" isDynamic="true"/>
                ...
                <AbstractTriggerIU id="LikeOperation"
                        role="" importance="4" repetitionFactor="1"
                        triggerIUType="OPERATION"/>
                ...
        </AbstractCompoundIU>
        <AbstractCompoundIU id="PlayerControl" role="grouping" importance="5"
                repetitionFactor="1">
                ...
                <AbstractTriggerIU id="PreviousTrackOperation"
                        role="" importance="2" repetitionFactor="1"
                        triggerIUType="OPERATION"/>
                ...
                <AbstractSelectionIU id="VolumeSelection" role="" importance="5"
                        repetitionFactor="1" isContinuous="true"
                        selectionType="SINGLESELECTION"
                        step="1" start="0" end="10"/>
        </AbstractCompoundIU>
        ...
</AbstractUIModel>
```

Figure 4. An excerpt of the abstract UI description for the environmental audio player.

For illustration purposes we have selected three different use scenarios: a living room, a car and practicing sports outside. Each scenario has its own

particular characteristics and preconceptions that affect how the user expects to interact with the system. Consequently, while the system logic remains the same, the shape of the UI would change according to the use environment. For example, for the living room scenario, the user would expect to use a remote to control the reproduction, while for the car; she would expect to use the embedded buttons available on the steering wheel.

As specified in section 3, the design and development of a UI using Dandelion can be divided into three different steps.

The first step requires the analysis of the system in order to determine its interaction requirements and then the formalization of those requirements into a description of the UI at an abstract level using the UsiXML AUI model.

Our ubiquitous music player does not differ too much from a typical music player in terms of functionality. Therefore it must perform the usual operations like play or pause, next track, previous track or select the volume of the music. Figure shows an excerpt of the description of the AUI for the proposed example. We have used a XML syntax of the UsiXML AUI model to describe the different interaction elements required by the UI, such as, for example:

- Abstract Data Interaction Units of type output for the “Album Art” and “Song Title”.
- Abstract Trigger Interaction Units for the “Like” and “Next Track” operations.
- Abstract Selection Interaction Unit for the “Volume” selector.
- And Abstract Compound Interaction Units for building virtual containers of Interaction Units with related semantics.

Once the description document of the UI is ready, the next step is to define the data and action objects that will be used to interchange information between the business logic and the UI. We call them the interaction I/O interface. These objects will be observed by the UIC, so that when the business logic modifies them, the UIC will send GIP events to the appropriate FIOs, and vice versa. For example, for the “Song Title”, a String object will be used, while for the “Volume selection” it will be an array of Integer objects as the list for selection, and an Integer object for the selected one. Regarding the trigger interaction units, they must be associated to action objects. They represent only identifiers of business

logic actions. When the UIC receives an action GIP event, it notifies the business logic by issuing a callback method and passing the action object as an argument identifying the action that has been activated.

With the abstract user interface defined, and the interaction I/O interface objects associated to the different interaction units, the next step is to select or build the required Final Interaction Objects for each Abstract Interaction Unit. This step depends on the purpose of the system and the place or places where it will be deployed. If it is going to be deployed in many heterogeneous places, this step can be obviated by developers, and left in the hands of the installers, that will select the appropriate FIOs at deployment time. Nevertheless, developers can provide some guidelines about what modalities or devices to use for each Abstract Interaction Unit depending on the environment and user conditions.

The selection of the FIOs depends completely on the characteristics of the scenario as well as the devices available. Ideally, every device would implement the Dandelion GIP interface, so that they can be directly used as FIOs. Nevertheless, as this is not the case, in many scenarios developers or installers will need to implement themselves a FIO agent that abstracts a hardware device behind the GIP interface.

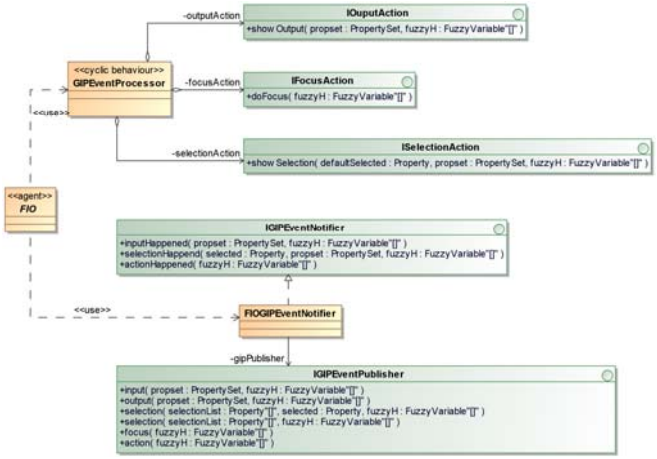

Figure 5. Dandelion Final Interaction Objects development framework

Dandelion makes the building of a FIO agent quite simple. Each FIO is implemented as an instance of the abstract FIO class, and uses the strategy

pattern to decouple the concrete logic required for processing each type of GIP event. Therefore, as shown in Figure 6, it is only required to implement three JAVA interfaces for processing incoming GIP events and implement the required agent behaviour logic to detect user input/selection, and publish the corresponding GIP events using the *FIOGIPEventNotifier* class.

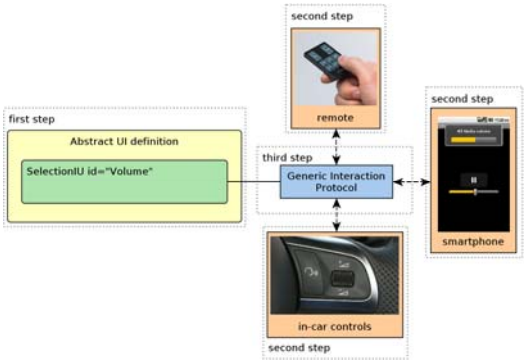

Figure 6. Each abstract element must be associated to one or many final interaction resources

Continuing with the example of the volume selector previously introduced. In order to adapt the system to the car, the home, and outdoor activities, we should provide at least three different implementations of the FIO used for volume selection, see Figure .

Let's explore in more detail the living room and outdoor scenarios. In our laboratory we have an experimental room where we have deployed a KNX home automation system. We are using it to simulate the living room scenario. We want to use two different devices, a KNX remote controller and a keypad with three pairs of buttons. One of the pairs will be used as a volume rocker to increase or decrease the volume. Regarding the outdoor scenario, we want to use the volume rocker of an Android smartphone to control the volume of the environmental music player.

In order to access the KNX devices and the Android smartphone buttons, we are going to use UniDA[6] (Varela, 2011), an open source hardware abstraction technology developed at our lab, which allows us to access many kinds of heterogeneous hardware devices using the same software interface

[6] http://www.github.com/UNIDA

In order to access the KNX devices and the Android smartphone buttons, we are going to use UniDA[7] (Varela, 2011), an open source hardware abstraction technology developed at our lab, which allows us to access many kinds of heterogeneous hardware devices using the same software interface.

The implementation of the two FIOs is quite similar, both of them generate selection GIP events, accept focus GIP events, use UniDA, and their physical shapes are also similar. The two of them use two buttons as a volume rocker to control volume selection. The two FIO agents use the UniDA API to subscribe themselves to the state of the two buttons, so that they get notified when a button is pressed. And they store a value for the volume; that will be updated accordingly when a notification about a button press is received. In case that the volume selection has changed, they will generate a selection GIP event indicating the selected volume and send it by calling the *selectionHappend()* method of the *FIOGIPEventNotifier*. The focus event is what makes them different. The remote has a buzzer and the smartphone uses the vibrator and the screen. When the remote controller FIO receives a focus event it will process it using an implementation of the *IFocusAction* interface that uses UniDA to activate the remote buzzer, while the smartphone FIO will use UniDA to activate the vibrator and make the screen blink temporarily. As can be seen, while the internal operation of each FIO is different depending on the interaction resource used, the external interface, the GIP, remains the same.

Finally, the last step to build the physical UI of our example, would be to deploy the application using the HI3 platform and associate the different AIUs of the UI to the FIO agents. This association can be specified programmatically, in a similar way as the association between AIUs and the I/O interface, or by using a configuration file for the User Interface Manager. This configuration file associates the identifiers of the AIUs with the agent identifiers of the FIOs. This step is carried out at deployment time, so it can change from one deployment to another.

With this example we have shown how a distributed physical UI can be built by using Dandelion, an implementation of the GIP using multi-agent technologies. It has been shown how the GIP is able to decouple the

[7] http://www.github.com/UNIDA

application logic from the specific characteristics of heterogeneous interaction resources by relying on proxy-like components, the FIOs, which abstract any device behind a generic interface populated by user interaction operations. Furthermore, it has been shown that the UI can be developed independently from the interaction resources used in each scenario, which are not required at all until deployment time.

5. Conclusions

This paper has presented a development framework for physical UIs in the context of Ambient Intelligence and Ubiquitous Computing. This framework allows the combination of model-driven engineering techniques with distributed device abstraction technologies to facilitate the development of physical UIs. It does so by providing an abstraction layer, the GIP, which decouples them from the various heterogeneous technologies and modalities used by the interaction resources.

The GIP interface reuses many concepts from the UsiXML AUI model, modifying and complementing them when necessary, in order to construct a remote generic interface capable of modelling the interactions available in the majority of interaction resources. The example presented in this paper has shown that the GIP interface achieves a great level of decoupling between the UI designers, system logic programmers, and the implementation of the specific user interaction resources.

Furthermore, a framework that implements the proposed abstraction layer, called Dandelion, has been presented. It uses multi-agent technologies to build a physical UI development framework based on the GIP concept.

The example demonstrating how to use Dandelion to build physical UIs has shown that the GIP provides a promising approach to reduce the coupling between systems and the end devices used to implement their physical UIs.

References

Ullmer, B.; Ishii, H., "Emerging frameworks for tangible user interfaces," IBM Systems Journal , vol.39, no.3.4, 915,931, 2000

James Patten, Hiroshi Ishii, Jim Hines, and Gian Pangaro. 2001. Sensetable: a wireless object tracking platform for tangible user interfaces. Proc. CHI '01. ACM, 253-260.

George W. Fitzmaurice, Hiroshi Ishii, and William A. S. Buxton. 1995. Bricks: laying the foundations for graspable user interfaces. In Proc. CHI '95, Irvin R. Katz, Robert Mack, Linn Marks, Mary Beth Rosson, and Jakob Nielsen (Eds.). ACM, 442-449.

Rafael Ballagas, Meredith Ringel, Maureen Stone, and Jan Borchers. 2003. iStuff: a physical user interface toolkit for ubiquitous computing environments. Proc. of the CHI '03. ACM, 537-544.

Saul Greenberg and Chester Fitchett. 2001. Phidgets: easy development of physical interfaces through physical widgets. In Proceedings of the 14th annual ACM symposium on User interface software and technology (UIST '01). ACM, 209-218.

Sergi Jordá, Günter Geiger, Marcos Alonso, and Martin Kaltenbrunner. 2007. The reacTable: exploring the synergy between live music performance and tabletop tangible interfaces. Proceedings TEI '0). ACM, New York, NY, USA, 139-146.

Kaltenbrunner, M., Bovermann, T., Bencina, R., & Costanza, E. (2005). TUIO: A protocol for table-top tangible user interfaces. In Proc. of the The 6th Workshop on Gesture in Human-Computer Interaction and Simulation.

Augusto, J. C., & McCullagh, P. Ambient Intelligence: Concepts and Applications. Int'l J. Computer Science and Information Systems, vol. 4, number 1, 1–28.

Dadlani, P, Peregrin Emparanza, J, & Markopoulos, P. Distributed User Interfaces in Ambient Intelligent Environments: A Tale of Three Studies. *Proc. 1st DUI*, University of Castilla-La Mancha (2011), 101-104.

Villar, N., Gilleade, K., RAMDUNYELLIS, D., & Gellersen, H. (2007). The VoodooIO Gaming Kit: A real-time adaptable gaming controller. Computers in Entertainment (CIE), 5(3), 7.

Villar, N., & Gellersen, H. (2007). A malleable control structure for softwired user interfaces. In Proceedings of the 1st international conference on Tangible and embedded interaction (pp. 49-56). ACM.

Balme, L., et al. Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces. *Proc. EUSAI 2004*, Springer-Verlang (2004), 291-302.

Limbourg, Q., Vanderdonckt, J. UsiXML: A User Interface Description Language Supporting Multiple Levels of Independence.Engineering Advanced Web Applications, Rinton Press, Paramus, 2004, 325-338.

Paz-Lopez, A., Varela, G., Becerra, J.A., Vazquez-Rodriguez, S., & Duro, R. J. Towards ubiquity in ambient intelligence: User-guided component mobility in the HI3 architecture. Science of Computer Programming, 11/2012, Elsevier (2012).

Varela, G., et al. Decoupled Distributed User Interfaces in the HI3 Ambient Intelligence Platform. *Proc. 6th UCAmI 2012*, Springer (2012), 161-164.

Varela, G, Paz-Lopez A, Becerra, J. A., Vazquez-Rodriguez, S., & Duro, R. J. UniDA: Uniform Device Access Framework for Human Interaction Environments. Sensors 11 (10), MDPI (2011), 9361–9392.