Using Interaction Style to Match the Ubiquitous User Interface to the Device-to-Hand

Stephen W. Gilroy and Michael D. Harrison[13]

Dependability Interdisciplinary Research Collaboration,
Department of Computer Science, University of York, York YO10 5DD, UK.
steveg@cs.york.ac.uk

Abstract. Ubiquitous computing requires a multitude of devices to have access to the same services. Abstract specifications of user interfaces are designed to separate the definition of a user interface from that of the underlying service. This paper proposes the incorporation of interaction style into this type of specification. By selecting an appropriate interaction style, an interface can be better matched to the device being used. Specifications that are based upon three different styles have been developed, together with a prototype Style-Based Interaction System (SIS) that utilises these specifications to provide concrete user interfaces for a device. An example weather query service is described, including specifications of user interfaces for this service that use the three different styles as well as example concrete user interfaces that SIS can produce.

1   Introduction

The increasing availability of personalized and ubiquitous technologies leads to the possibility that whatever the device-to-hand is, it becomes the way to access services and systems. Therefore, interfaces to services must be designed for a variety of different types of device from desktop systems to handheld or otherwise portable devices. Different styles of interaction often suit different devices most effectively. While the appearance of ubiquitous devices has brought forth a proliferation of innovative interactive techniques, the broad categories and aspects of style as, for example, identified by Newman and Lamming [1] can still be applied. While a *key-modal* interface may be appropriate for a mobile telephone, with its limited screen and restricted keypad, a *direct manipulation* (DM) interface may be appropriate for a device based around touch / pen interactive techniques, such as current models of palmtop or tablet PCs. Typically in such situations a different low-level interface will have to be designed separately for each device. It is possible that several interaction styles may have to be supported for different users or parts of the system on the same device. As new technologies evolve to meet the demands of ubiquitous computing additional styles will emerge.

---

[13]Mailing address: Informatics Research Institute, University of Newcastle upon Tyne, NE1 7RU, UK. michael.harrison@ncl.ac.uk

Style-specific design considerations normally take the form of guidelines, heuristics or ad-hoc rationalizations by designers [2]. Designs to support many devices may be facilitated by incorporating interaction style explicitly into an implementation. In this paper we demonstrate that incorporating style-level descriptions into a model of a user interface can give more flexibility than forcing a single user interface model on a heterogeneous selection of devices. This paper is concerned with an approach in which interaction with a service is bound to the features of the platform through a mediating style description. The aim is to support an interface that is appropriate given the technological constraints or opportunities afforded by the platform. In section 2 the approach to the style-based interaction system is contrasted with other approaches to platform independent service provision. In section 3 the interaction style approach is described in more detail. In section 4 an implementation of a style-based system and the specifications that drive it are described. In section 5 an example of a weather system is used to illustrate the idea. In section 6 the approach is discussed again in relation to other similar approaches and in section 7 the paper draws conclusions.

2.   Modelling the Ubiquitous User Interface

Separating the user interface from application functionality [3] is a key theme in the delivery of interactive applications to multiple platforms. This is achieved by abstracting the interaction with a user interface from its presentation on a specific device. Model-based user interface development [4] provides useful tools to cleanly separate the parts of an application. However, its potential for easing cross-platform user interface development is less apparent when platforms differ in their support for styles.

The rise of ubiquitous computing and the proliferation of user appliances of widely differing capabilities and limitations have given new impetus to the need for cross-platform interface design. A provider of ubiquitous services typically wishes to target different users who may use devices of different capabilities, or a user or set of users who wish to migrate their use of services across several different devices.

Separation of application functionality and delivery via abstractly defined interfaces can be addressed in this broader context by the use of *service frameworks* [5] that organize and aggregate software functionality and data, and facilitate universal access to it. *Universal user interfaces* will provide interaction with services on a variety of devices, tailoring the interface to suit the device.

2.1   Service Frameworks

A service framework enables application functions to be delivered to devices whatever and wherever the devices are. The Web is an example of a framework for the delivery of many similar services through Hyper-Text Markup Language (HTML) files provided by web servers. Web services are delivered via Universal Resource Locators (URLs) that identify a particular service (usually requesting a single page of information). A user therefore makes the required service explicit by entering a URL

into the browser manually, through a bookmark, or via a hyperlink. Other frameworks, e.g., XWeb [6], use similar approaches to existing web services and provide better support for diverse interaction.

2.2 Universal Interface Specification

An application's behaviour can be defined independently of platform, through the use of services. However, a mechanism is required to map that behavior to the specific interface components of a device. Model-based approaches map abstractions of interaction objects onto platform-specific implementations. The interactive components of the interface, for example a text box for inputting text or a drop-down list for making a choice, are abstracted and encapsulated in terms of a relatively small set of "interactors" [7]. Other approaches utilize several levels of abstraction that may include low-level "widgets", as well as more abstract components such as "group" or "choice". The sets of widgets available on different platforms may not intersect in terms of detail but as long as the abstraction can be fulfilled by a widget that *is* available on a particular platform then a concrete interface can be rendered.

2.3 Problems of Abstract User Interface Models

Abstract interface models [6, 8-12] are problematic when abstraction is such that there is no convenient implementation of the low-level interaction objects on a particular platform. A model must be defined to either restrict the set of objects to ones that are common across all platforms, or provide a wider set of objects to cover the variation in platform. In the former case, the interface becomes the "lowest common denominator" of all target platform capabilities, and is unsuitable if a new platform has interaction objects that do not exist in the available set. In the latter case, abstract objects are a union of available platforms. This gives rise to the two-fold problem of an ever-expanding library, or "toolkit", of widgets and an overly complicated mapping scheme to select the correct widgets for a platform.

Presenting a user interface for a UIML [11,12] specification on a specific platform involves more than selecting an appropriate widget representation. An interface structure that is defined *canonically* may fit one platform but not another. It is then necessary to have different specifications for cross platform structure variations, or alternatively a generic structure specification, which may be overridden when mapping the parts of the interface to actual platform elements. This defeats some of the point of a single structure definition. UIML also assumes a one-to-one mapping of parts to toolkit implementations. If a part in one interface implementation is needed it is added to the canonical definition of parts, even if it is not mapped to a particular platform.

XWeb [6], on the other hand, provides a higher-level formal specification of semantic interaction than a simple widget mapping. However, it still suffers from the "structure" problems of UIML in that it uses "grouping" interactors that arrange other interactors in a hierarchical structure, incorporating a canonical XView. An XView defines which elements of a data tree are manipulated by each interactor. While

XWeb allows designers to reuse a view specification across clients with no extra effort, designs have to combine the interactors into views that are suitable for all platforms. The designer can therefore either design one set of views that maps to all client devices, or create a different set of views for different client types, losing the advantage of a single specification. Even if this is done, a new client with new interactor implementations might have usability problems with existing views, a problem encountered when speech widgets were implemented in an XWeb client [6].

3.   A Model of Interaction Style

A model that incorporates interaction style makes it possible to vary the structure or interface semantics applied across devices. User interface descriptions are defined on a per-style basis and a target device selects the description that best maps onto its capabilities. Hence, if a form-fill interaction style is most appropriate for the device in the context of a particular application then that style is bound to the application and mapped to the interactive components of the device. For another target device a dialogue style might be more appropriate and in this case, the same application software would be bound with this different style.

The number of styles supported in the model should be finite and small, to allow a designer to target the maximum number of devices with the minimum amount of effort. It should also be possible to add a completely new style by creating additional definitions for existing interfaces. Although a designer does not have to support all styles, compatibility will be lost if devices do not support the styles chosen.

Two distinguishing features of a style are the manner in which they guide the user to the desired task or function and how they gather required input from the user. There may be semantic relationships that are shared across styles but which manifest themselves in different ways.

The style-based interaction system described in section 4 incorporates support for three styles: form-fill, dialogue and menu. Although these three are considered "classic" styles that can be applied to desktop systems, they also apply equally to other kinds of device. The services provided may be targeted at both desktop and mobile devices. Form-fill would map onto a web-style interface on desktop type systems, dialogue for voice-based telephone systems and menu for mobile phones or embedded devices.

3.1   Form-Fill Style

Forms are two-dimensional rather than one-dimensional, so navigation is important. The organization of a form on the display of the device requires a logical structure so that it can be decomposed to suit different display capabilities [13].

Form elements have different interaction requirements. Simple elements just require text entry while complex elements involve groups of choices or data of a particular format and may be mandatory or optional. The relation between elements might mean that two elements are mutually exclusive, or that filling in an element

makes other elements or form sections mandatory. In addition, the elements that are filled in might affect what actions are available with the form data.

When the form is filled in, an action must be chosen to process the information. This is usually done by special commands, or buttons. An action might specify a certain set of form elements from which it processes information or the action invoked by a command might depend on the value of certain form elements. Validation of elements could occur before processing or feedback given if the processing finds invalid information.

A typical example of a form-fill style is the web-based form illustrated in figure 1(a).

Fig. 1(a). A Web-based Form Interface.

3.2 Dialogue Style

The key feature of this style is the structure of the dialogue with the user. As questions are posed, the user's answer determines the next question asked and that answer may be a piece of data that is gathered. A state-chart notation is useful in describing this interface. Each state is a mode of the interface, and the transitions between states are the available choices. On entering a state the appropriate prompt is displayed. Input and output in a question/answer interface is one-dimensional so, while it is limited in terms of interaction, it can be supported by devices without complex graphical capabilities and the conversational nature of interaction facilitates the use of speech. VoiceXML systems (figure 1(b)) are an example of a dialogue style of interface.
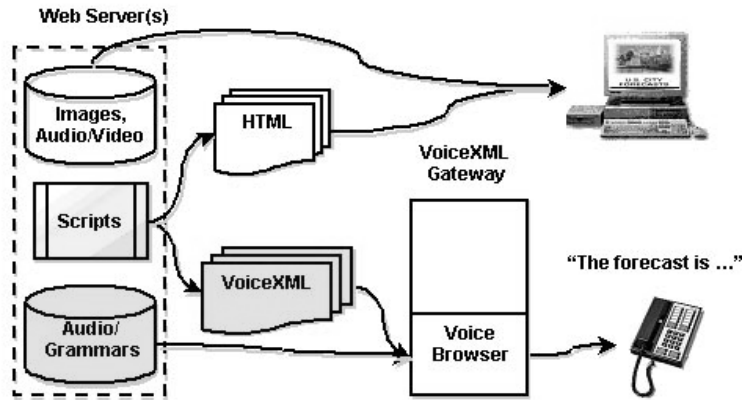
Fig. 1(b) A Voice XML Dialogue Interface.

3.3 Menu Style

The navigational structure of a menu style is governed by how best to partition the menu space to provide meaning to guide the user. Breadth is preferred over depth, as deep menus have the same orientation problems as dialogue structures. Devices that employ menu interfaces have a limited, customised input mechanism based around a small number of specialized buttons or keys. Input and navigation must be designed to facilitate easy mapping from an unknown layout of keys. Current generation mobile phones typically utilize a menu interface as shown in figure 1(c).
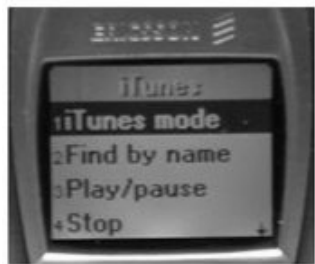

Fig. 1(c) A Mobile Phone Menu Interface.

4   Style-Base Interaction System (SIS) Framework

A prototype application framework supports interfaces using a variety of styles as outlined in section 3. The components of the framework are shown in figure 2. The framework consists of a runtime system that is configured by a set of eXtensible Mark-up Language (XML) specifications describing the service and style-based user interfaces of an application.
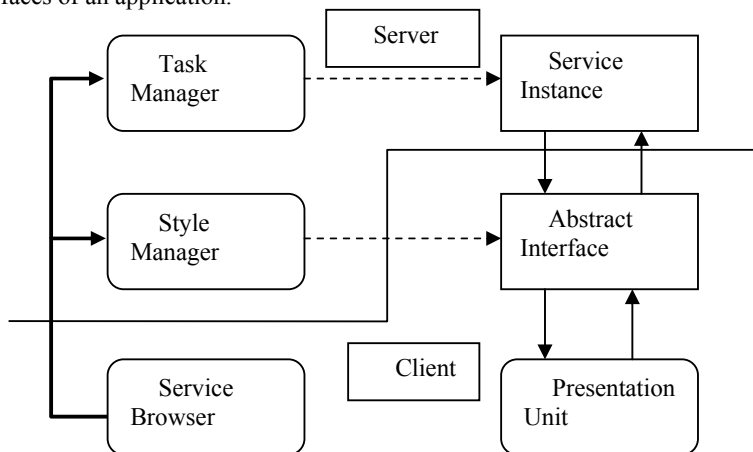
Fig. 2. SIS Framework.

SIS consists of both components that reside on a client appliance and those that can be managed on a remote server. Within a running ubiquitous application, this distinction is transparent. SIS is designed to switch easily between different style instantiations running on a single service instantiation. A user may thus migrate between different appliances without losing saved task-level information. It is feasible to swap a running style between different instances of the same service or two different services that both support the set of tasks required by the style definition.

The three components that deal with the initialization and management of an application are the *Service Browser* on the client, a *Style Manager* to look after styles and a *Task Manager* to look after the tasks required by services. Managers exist as separately running entities, possibly residing on remote servers, with their own resources and are configured using XML specifications of task and style. They use this configuration to generate the run-time components of the interface: *Service Instances* and an *Abstract Interface* for each style. Device specific *Presentation Units* provide concrete interface instantiations on each client. A weather service application is used to illustrate the approach.

4.1   Task Definition using Service Specifications

The XML specification of a service defines its tasks, required function and data storage. A task manager generates run-time instantiations of services called service instances from these specifications. A service instance provides the data storage for its component tasks and a list of all the tasks in the service. Task instantiations are shared between services that use them, and are maintained by the task manager. When a service instance needs a task, it calls the task using the manager that created it. Tasks are identified by a namespace scheme[14] to avoid clashes between tasks of the same name utilized by different services.

Functions. Service functions implement the tasks that are part of a service and "wrap" the logic implementation so that there is a consistent interface for use in SIS. SIS also allows external functions (utility functions) to manipulate data before it is used in a function call. An example service and utility function specification are shown in figure 3. The `class` and `method` attributes identify a function's Java implementation. The `<return>` and `<parameter>` elements identify the function's return type and required parameters respectively. Utility functions do not affect the state of the underlying application logic, but are assumed to perform some repeatable translation upon data. SIS therefore does not need to know the implementation of data types to be able to manipulate them.

```
<function class="WeatherService" method="getWeather"
name="GetWeather">
    <return type="weather">weatherData</return>
<parameter type="string">cityName</parameter>
</function>

<utility name="postalToCity" class="PostUtil"
method="postalToCity">
    <return type="alpha">cityName</return>
    <parameter type="string">postalCode</parameter>
</utility>
```

Fig. 3. Function Definitions: A Service Specification XML Fragment.

Tasks. A single task within a service represents the lowest level of interaction with an application that is understandable to the user. Tasks describe a flat pool of possible functions and define how they are invoked. Task parameters can be provided either by user input or by a stored value. In the case where a needed parameter is a stored value that is not initialized, that task can be defined as unavailable.

Each task can call on at most one service function to guarantee atomicity of tasks and avoid problems of sub-task ordering. The provision of utility functions is meant to encourage data representation issues to be separated from logic. Hence, logically

---

[14] A *namespace* is a unique identifier that labels a group of related items. Different groups can then use the same identifiers internally to label different items.

similar tasks may use the same underlying service function and use utility functions to manipulate the data they provide to that function.

An example task specification fragment is shown in figure 4. Note the definition of the mapping of input from the user (`<variable>` elements) to parameters of the service function (`<parameter>` elements). This mapping technique is described below.

```
<task name="Get City Weather" taskFunction="Get Weather">
    <variable type="simple">cityName</variable>
    <parameter type="alpha"
               source="task"
               store="lastCity">cityName</parameter>
</task>
```

Fig. 4. Task Definition: A Service Specification XML Fragment.

Mapping Tasks onto Functions. The data passed from tasks to their underlying function are defined in terms of input *variables* and function *parameters*. These are represented in task definitions by `<variable>` and `<parameter>` element tags. The types of parameters defined in the task exactly match the input parameters of the underlying service function. However, there need not be the same number of task parameters as variables. The manipulation of a variable to provide a parameter value is defined with the `<parameter>` element tag. It identifies the variable to be used, what mapping to perform and whether to store the generated parameter value for later use.

The default mapping, if no mapping is explicitly defined (as in figure 4), is no manipulation at all. Data is output as a parameter exactly as it is received as a variable.

```
<parameter type="alpha"
           source="task"
           mapping="utility"
           store="lastCity"
           name="postalToCity">
<parameter type="alpha"
           source="task">postalCode</parameter>
</parameter>
```

Fig. 5. Utility Mapping in a Task Parameter: A Service Specification XML Fragment.

A *utility mapping* (see figure 5) assigns a utility function to transform the data of a variable that defines a mapping from postcodes to city names. The `name` attribute identifies the utility function to use, and the nested `<parameter>` element tags describe the mapping for the utility function's parameters.

Extract mappings take an element of a record type and return one of the items within the record as specified in the parameter. (Figure 6 shows extraction of an ID value from an account record.)

```
<parameter type="alpha"
           source="task"
           mapping="extract">account
accID</parameter>
```

Fig. 6. Extract Mapping in a Task Parameter: A Service Specification XML Fragment.

Keeping Track of State. A task-based service keeps track of persistent state at a task level separately from any provision made by underlying logic. State therefore can be shared between tasks directly without the underlying logic. It is possible to support stateless implementations of the logic (such as with raw HyperText Transfer Protocol (HTTP) based systems). A task parameter can define a mapping from a state variable instead of a task variable. In figure 7, a state variable keeps track of the name of a city for which weather is requested and a task uses the name to give an update of that request.

```
<state> <variable type="string">lastCity</variable> </state>
...
<task name="Update Weather" taskFunction="Get Weather">
    <parameter type="alpha" source="store">lastCity</parameter>
</task>
```

Fig. 7. State Definition and Use in a Task Parameter: A Service Specification XML Fragment.

4.2   Interaction Style Specification

The key feature of the SIS approach is how tasks are implemented on different platforms. Each platform supports a set of presentation objects. Between the tasks and the presentation, each presentation style supports its own abstract user interface elements that gather input and display output to the user. These elements have their own distinctive way of navigating available tasks. No explicit layout or presentational information is contained in a style description; rather it is the semantic relationship between interface components that is described. It is the job of the presentation unit to resolve these relationships into an appropriate presentation.

Style instances are generated in the SIS client in order to facilitate fast user response. Therefore, events generated by presentation implementations are dealt with by style-specific, presentation-independent, objects that reside locally. The style manager generates each style instance from scratch locally on each client in order to customize a client's access to a common service.

Three styles are currently implemented but aim to provide a foundation for a potentially larger set.

4.2.1. Form-Fill Style

The style definition for a forms-based style involves: *field* elements for gathering user input, *actions* that can be invoked and a mapping from actions and fields to underlying tasks.

A field element is an abstract interactor that allows the user to enter a value to be used in a task, for example text entry, password entry, single choice, multiple choice, date entry, range entry and currency entry. Questions about whether a single choice entry would be represented by a drop-down list, radio buttons or some other selection method are deferred to platform implementation and depend on the actual data being selected and the layout constraints of the presentation. An example of a simple text field element and a single choice element are given in figure 8. The definition gives the type of the field element and the type of its value.

```
<field name="postalText" type="text"/>

-------------------------

<field name="accountChoice" type="choice" value="AccountType">
      <n-selection>1</n-selection>
      <selection-values source="utility">Get Accounts</selection-
values>
</field>
```

Fig. 8. Form-fill Style Specification: Example text field and single choice field element definitions.

Each style provides mechanisms for processing the data to produce an appropriate representation. Providers of services may specify functions that perform representational transformations. For example, in the form-fill style an output processor defines a set of items that can be extracted from a data type (see figure 9). Several output processors can be defined to work on the same types and used for different purposes.

```
<processor name="weatherOut" type="text">
      <input class="WeatherData">weatherData</input>
      <converter class="WeatherData">
         <item>
            <source>weatherData</source>
            <method>getWeatherText</method>
         </item>
      </converter>
</processor>
```

Fig. 9. Form-fill Style Specification: An example output definition.

A form is built out of fragments that map a set of fields to the inputs of a particular task. A fragment's task is only invoked if the requirements of the fields of that fragment are satisfied. A fragment also specifies an output processor that can extract information from the output of the task.

```
<form_fragment name="cityForm">
    <task>Get City Weather</task>
    <input req="mandatory">cityText</input>
    <output type="text">weatherOut</output>
</form_fragment>
```

Fig. 10. Form-fill Style Specification: An example form fragment definition.

This definition (figure 10) outlines a hierarchy of actions that may be invoked by a user and associates with each action a set of form fragments that are evaluated when that action is invoked. Typically an action would be invoked by the user pressing a submit button to indicate completion of the form ready for processing. An action is a semantic unit within the form. Trees of actions, together with form fragments allow a presentation to compose a form representation. The presentation decides whether fields are presented on several "pages" or on a single "page" and use different buttons to invoke different actions.

4.2.2. Dialogue Style

Dialogue style definitions are described by a set of grammars of input token combinations. Dialogue structures make use of these grammars to move between elements of the dialogue. A grammar used in a transition between states is called a *match set* and contains a list of match items that can be matched by a series of tokens in input. For example in figure 12 <matchitem> contains a main <token> whose contents must match the next input token and optionally a list of match items that can be matched after that token. Items are evaluated in list order. As soon as an item matches, no more items in a list are evaluated. An item only matches if its main token matches *and* one of its sub items matches. That a possibility is optional is supported by a special <lambda> match item that is matched if no other items in a list are matched.

```
<matchset name="CityMatch">
    <matchitem>
      <token>city</token>
      <matchitem>
      <token>name</token>
      </matchitem>
      <lambda/>
    </matchitem>
</matchset>
```

Fig. 11. Dialogue Style Specification: An example match set definition fragment.

The dialogue structure is a tree of states that has special task-invoking states as the leaf nodes in the tree (see figure 12). States are defined with <dialogue-state> element tags and contain possibly conditional prompts that are displayed if the dialogue stops at that state. A transition attribute identifies match sets or stored variables that a user's input must match. After a task is invoked, the dialog restarts at the root of the tree.

```
<dialogue-state>
      <prompt source="GetWeatherPrompt"/>
      <prompt source="GetUpdatePrompt">
       <condition task="Update Weather">
          <name>available</name>
          <value>true</value>
       </condition>
      </prompt>
      <dialogue-state transition="CityMatch">
          <prompt source="CityInput"/>
          <dialogue-state transition="$CITYVAR">
            <prompt source="CityWeather"/>
          </dialogue-state>
      </dialogue-state>
...
</dialogue-state>
```

Fig. 12. Dialogue Style Specification: An example dialogue tree definition fragment.

Task invocations are defined in special states that define the underlying task to be invoked, which dialogue variables to use, and the response to be generated with the output (figure 13).

```
<response name="weatherResponse" class="WeatherData">
    <output type="text">
      <method>getWeatherText</method>
    </output>
</response>
<task-state name="PostWeather">
    <task>Get Postal Weather</task>
    <parameter>$POSTVAR</parameter>
    <response>weatherResponse</response>
</task-state>
```

Fig. 13. Dialogue Style Specification: An example task state definition fragment.

Prompts can be either predefined questions or the response from a task invocation. Responses can also be shared between task instances. User variable input is transferred to the task states by use of a set of defined variables. The name of these variables can be used in place of a grammar match set in a transition between states.

4.2.3. Menu Style
A menu-based interface is specified by a tree of menu items (see figure 14). Each node representing an item has a label and an optional description of a task invocation. Only the leaves of the tree can have task invocations. Details of the task are wrapped into the menu item specification, with the name of the task and an output data extraction defined as usual, together with a list of inputs. Inputs can have a label to be displayed to the user when entering that input.

```
<menu-item>
    <label>Weather by PostCode</label>
    <task>Get Postal Weather</task>
    <input type="string">
        <name>postalCode</name>
        <label>Enter postal code</label>
    </input>
    <output class="WeatherData" method="getWeatherText"/>
</menu-item>
```

Fig. 14. Menu Style Specification: An example menu item definition.

This current version is limited to descriptions of simple menus, but as an aim of the specifications is to simplify interface definition for simple interfaces, the descriptions are also simple. It is envisioned that the specification will be extended to cope with more complicated menu semantics and user input.

4.3   Presentation

Presentation units run on the client device and prescribe a concrete user interface for style definitions. Each style will have a presentation unit tailored for it that runs on a particular device. A client presentation unit utilizes a reference to a remote service instance and the appropriate style instance. They give access to the internal object representations of tasks and the elements of styles. When a task is to be invoked, it passes the appropriate data to the service instance.

Current implemented presentation units use simple techniques to deal with physical layout and representational issues. An expansion of the presentation component in the future might include dealing with details of physical layout in an abstract way.

5   Creating Interfaces with Styles

An example weather service together with definitions of the three different styles of interfaces described above, and their rendering by presentation units is now described. The service provides a single function that returns a textual description of the weather for a given location supplied as a string.

5.1   The AnyWeather Service

The weather query service is described by a XML task specification for the service shown in figure 15. Three separate tasks perform the service:
1. Request the weather for a city by name (“Get City Weather”)
2. Request the weather for a city by postcode (“Get Postal Weather”)
3. Refresh the last weather request (“Update Weather”)

Requesting the weather for a city by name utilizes the underlying service function “Weather Service” directly, while a post-code based request requires the use of

an external utility function, "postalToCity", to convert postcodes to city names. The "Update Weather" task utilizes a state store object to keep track of the last city for which weather was requested.

```
<service location="http://www-
users.cs.york.ac.uk/~steveg/weather/">
<function class="WeatherService" method="getWeather" name="Get
Weather">
<return type="weather">weatherData</return>
<parameter type="string">cityName</parameter>
</function>
    <utility name="postalToCity" class="PostUtil"
method="postalToCity">
        <return type="alpha">cityName</return>
        <parameter type="string">postalCode</parameter>
    </utility>
    <state>
        <variable type="string">lastCity</variable>
    </state>
<task name="Get City Weather" taskFunction="Get Weather">
<variable type="simple">cityName</variable>
        <parameter type="alpha"
                   source="task"
                   store="lastCity">cityName</parameter>
</task>
<task name="Get Postal Weather" taskFunction="Get Weather">
<variable type="simple">postalCode</variable>
        <parameter type="alpha"
                   source="task"
                   mapping="utility"
                   store="lastCity"
                   name="postalToCity">
<parameter type="alpha" source="task">postalCode</parameter>
</parameter>
</task>
    <task name="Update Weather" taskFunction="Get Weather">
        <parameter type="alpha"
source="store">lastCity</parameter>
    </task>
</service>
```

Fig. 15. AnyWeather task specification.

5.2   Form-Fill Interface

The specification of the form-fill style for the AnyWeather service is shown in figure 16. Two fields are defined, one to enter city names ("cityText") and one to enter postcodes ("postalText"). A processor ("weatherOut") extracts the description of the weather from a WeatherData output object. Three form fragments, for each of the three tasks, use the defined processor for output and the two fields as inputs. The <sub-form> definitions match the form fragments to an action and a single display.

```
<style type="form"
       location="http://www.users.cs.york.ac.uk/~steveg/weather">
    <field name="cityText" type="text" />
    <field name="postalText" type="text" />
    <processor name="weatherOut" type="text">
        <input class="WeatherData">weatherData</input>
        <converter class="WeatherData">
            <item>
                <source>weatherData</source>
                <method>getWeatherText</method>
            </item>
        </converter>
    </processor>
    <form_fragment name="cityForm">
        <task>Get City Weather</task>
        <input requirement="mandatory">cityText</input>
        <output type="text">weatherOut</output>
    </form_fragment>
     …
    <form>
        <display type="text">weatherDisplay</display>
        <action-set>
            <action-set name="getWeather">
                <action name="getCity"/>
                <action name="getPostal"/>
            </action-set>
            <action name="updateWeather"/>
        </action-set>
        <sub-form>
            <fragment>cityForm</fragment>
            <action>getCity</action>
            <display>weatherDisplay</display
        </sub-form>
  …
    </form>
</style>
```

Fig. 16. AnyWeather form-fill style specification.

The form-fill presentation unit renders the form components on a single screen with two buttons representing the first sub-level of the action tree (see figure 17). The interface uses the requirements of the form fragments to evaluate which of the two user input tasks to invoke when the "Get Weather" button is pressed. The interface is told that "City Name" is mandatory for the "Get City Weather" task, but not required for the "Get Postal Weather" task, so if a city name is entered it can assume that the city task is required, and the button will invoke that task. In addition all non-required fields of that task will be disabled to help indicate which task has been chosen.
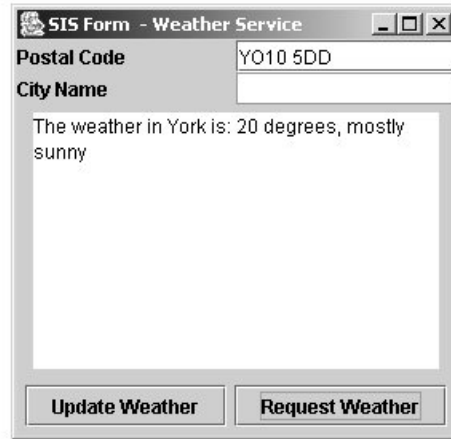
Fig. 17. Weather Service form-fill interface.

5.3   Dialogue Interface

The specification of the dialog style for the AnyWeather service is shown in figure 18. Prompts are defined for the initial dialog state and for requesting user input. A response extracts the weather description from a WeatherData object in much the same way as for the form-fill style. A task state for each of the available
tasks is assigned a response and an appropriate variable. Three match set grammars let a user enter a variety of phrases to select each of the tasks. For instance, a user can enter "postcode", "postal code" or just "postal" to access the `Get Postal Weather` task. A dialogue with three paths leads to the three tasks. The paths to the user input tasks have two states, one of which prompts the user to enter the appropriate input if it is not already in the token string. The update task doesn't require user input so only requires one state transition to reach it. The presentation unit for the dialogue renders the interface shown in figure 19.

```
<style type="dialogue">
<question name="GetWeatherPrompt">...</question>
<question name="GetUpdatePrompt">...</question>
<question name="CityInput">...</question>
<question name="PostInput">...</question>
<response name="weatherResponse" class="WeatherData">
     <output type="text"><method>getWeatherText</method></output>
</response>
<task-state name="PostWeather">
     <task>Get Postal Weather</task>
     <parameter>$POSTVAR</parameter>
     <response>weatherResponse</response>
</task-state>
...
<matchset name="PostMatch">
    <matchitem>
       <token>postcode</token>
    </matchitem>
    <matchitem>
      <token>postal</token>
     <matchitem>
        <token>code</token>
     </matchitem>
     <lambda/>
    </matchitem>
</matchset>
...
<dialogue-state>
   <prompt source="GetWeatherPrompt"/>
   <prompt source="GetUpdatePrompt">
    <condition task="Update Weather">
          <name>available</name>
           <value>true</value>
    </condition>
   </prompt>
    ...
   <dialogue-state transition="PostMatch">
       <prompt source="PostInput"/>
       <dialogue-state transition="$POSTVAR">
          <prompt source="PostWeather"/>
       </dialogue-state>
   </dialogue-state>
    ...
</dialogue-state>
</style>
```

Fig. 18. AnyWeather dialogue style specification.

Fig. 19. Weather Service dialogue interface.

```
<style type="menu"
       location="http://www-users.cs.york.ac.uk/~steveg/weather">
    <menu>
        <title>Weather Service Menu</title>
        <menu-item>
            <label>Weather by City</label>
            <task>Get City Weather</task>
            <input type="string">
                <name>cityName</name>
                <label>Enter a city name</label>
            </input>
            <output class="WeatherData" method="getWeatherText"/>
        </menu-item>
        <menu-item>
            <label>Weather by PostCode</label>
            <task>Get Postal Weather</task>
            <input type="string">
                <name>postalCode</name>
                <label>Enter postal code</label>
            </input>
            <output class="WeatherData" method="getWeatherText"/>
        </menu-item>
        <menu-item>
            <label>Update Weather</label>
            <task>Update Weather</task>
            <output class="WeatherData" method="getWeatherText"/>
        </menu-item>
    </menu>
</style>
```
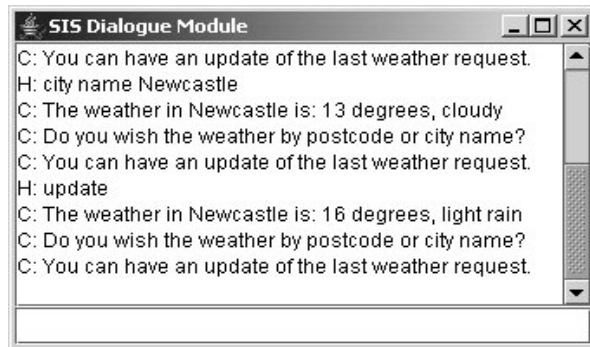
Fig. 20. AnyWeather menu style specification.

5.4   Menu Interface

The specification for the menu style of interface for AnyWeather is shown in figure 20. All three tasks are available from the main menu, one item per task. The two tasks requiring user input have inputs fields rendered as separate entry screens in a menu presentation implementation as shown in figure 21.
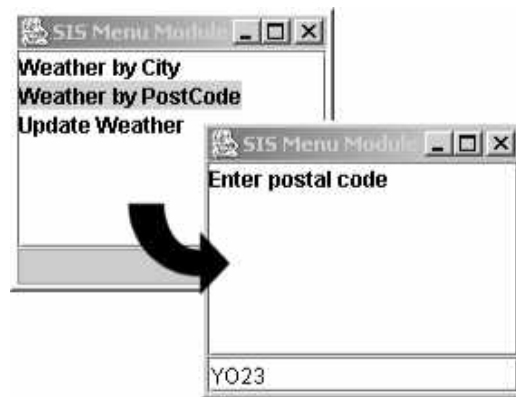
Fig. 21. Weather Service menu interface.

6   Discussion

The specifications in SIS separate the specification of the functionality of a ubiquitous application from the specification of its interface and provide a selection of different styles of interface so that an interface can more closely match the capabilities and limitations of a device. Both achievements are consistent with the original requirements of User Interface Management Systems (UIMS). Having a clean separation of function and interface has particular advantages when providing a selection of interface descriptions. It is clearly less important when providing a single "canonical" interface as in the case of XWeb and UIML (as discussed in section 2.3) or a UIMS vision based around a single type of device.

SIS achieves this separation by making the abstraction of functionality very simple. Any semantic relationships between the tasks must occur at the style level. In the AnyWeather service the relationship of tasks in the form-fill style (figure 16) is different from the dialogue style (figure 18), and this would be the case however systematically the layering was achieved.

Style specifications do not dictate how a presentation unit displays the information conveyed in the style. Presentation units on different devices display a style in different ways to fit that device even though the style definition is the same on each device. Applications can therefore use native applications on devices by having a presentation unit that renders interfaces in a way that is consistent with them. For instance a presentation unit could choose to display the AnyWeather form-fill actions as three separate buttons, rather than two, or indeed display the three sub-forms on different screens.

Although AnyWeather is designed to be simple to illustrate the basic ideas, more features can be added to each of the different styles. A further application of these features demonstrating SIS is based around an internet banking scenario. In this case more complex data types need to be supported, and this requires development of a

richer type system. List and record types can be implemented to help support more complex applications as well as user-defined custom types (similar to those in XWeb).

The relative size of dialogue style definitions might be said to be in conflict with the requirements for definitions for simple interfaces to be simple themselves. However, the benefit of having a clear, extensible specification means that the parsing engine of the system can be much simpler and allows for better integration with simple tools. In future, size might be alleviated without affecting the parsing engine by using transformations from more concise specifications into the current versions.

7   Conclusion

A model of interaction style has been devised that can be used to provide a range of possible interfaces to be presented on a device. Basing a single interface specification on simple (yet still abstract) concepts can work, but is limited if target devices are too diverse in their interactive capabilities. Conversely, tying the specification too closely to the capabilities of any one device leads to the situation of having a different specification for each device. Having a finite set of styles specifications can be complex enough to make fuller use of devices capabilities yet different and flexible enough to work on a wide range of devices. Interaction styles have potential to be viable for defining interfaces for ubiquitous interactive systems on many devices. Additional applications will provide the impetus for expanding the features of SIS, and demonstrate its potential and flexibility.

References

1. Newman, W., Lamming, M: Interactive System Design. Addison-Wesley (1995) 293—322
2. Shneiderman, B: Designing the User Interface, 3rd edition. Addison Wesley Longman (1998) 71-74
3. Edmonds, E.: The emergence of the separable user interface. In Edmonds, E., ed.: The Separable User Interface. Academic Press (1992) 5-18
4. Vanderdonckt, J.: Current trends in computer-aided design of user interfaces. In Vanderdonckt, J., ed.: Computer-Aided Design of User Interfaces Proc.of CADUI '96. Namur University Press (1996) xiii-xix
5. Abowd, G., Schilit, B.N.: Ubiquitous computing: The impact on future interaction paradigms and HCI research. In: CHI97 Extended Abstracts. (1997)
6. Olsen, D.R., Jefferies, S., Nielsen, S.T., Moyes, W., Fredrickson, P.: Cross-modal interaction using XWeb. UIST 2000. (2000) 191-200
7. Myers, B.A.: A new model for handling input. ACM Transactions on Information Systems (TOIS) **8** (1990) 289-320
8. Ponnekanti, S.R.,   et~al.: ICrafter: A service framework for ubiquitous computing environments. In: Proceedings of Ubicomp 2001. LNCS 2201 (2001) 56-75
9. Eisenstein, J., Vanderdonckt, J., Puerta, A.: Applying model-based techniques to the development of UIs for mobile computers. In: IUI01:2001 International Conference on Intelligent User Interfaces. (2001) 69—76

10. Muller, A., Forbrig, P., Cap, C.H.: Model-based user interface design using markup concepts. In: DSV-IS. Volume 2220 of Lecture Notes in Computer Science, Springer (2001) 16-27
11. Phanouriou, C.: UIML: A Device-Independent User Interface Markup Language. PhD thesis, Virginia Tech (2000)
12. Abrams, M., Phanouriou, C., Batongbacal, A., Williams, S.: UIML: an appliance-independent XML user interface language. In: Computer Networks. Volume 31. (1999) 1695-1708
13. Turau, V.: A framework for automatic generation of web-based data entry applications based 0on XML. In: ACM Symposium on Applied Computing (SAC 2002). (2002)

Discussion

[Gerrit van Deer Veer] You did not mention/elaborate interaction styles "direct manipulation" nor "command language". DM requires complex representation of n-dimensional interaction space and n-degrees of freedom user act to, command language seem completely upprite(?). Also, in envisioning scenarios of companies like Philips, NTT, Sun ("starfire") these styles are mixed.

> [Stephen Gilroy] We did not elaborate DM: it's very complex. We considered mixed styles (?). their analysis / Specification would be separate/unconnected.

[Ann Blanford] Walk-up-and-use isn't just device or just context – it's a tuple of device, context, user, task(s). i.e. There are combinations that work together and often that don't. Can these combinations make style selections simpler ?

> [Stephen Gilroy] Yes.

[Kevin Schneider] Within your categorization of interaction styles, are there different styles for each device ? For example, would there be a different interaction style for filling in a form on a PC versus filling in a form on a PDA.

> [Stephen Gilroy] No, it would be the same style. The device would handle the different presentations.