aQtiveSpace

how it works!

2nd October 1999

Alan Dix

aQtive limited

Birmingham Research Park
Vincent Drive
Birmingham, B15 2SQ, UK
www.aqtive.com

* onCue, aQtiveSpace and aQtive are trademarks of aQtive limited
onCue and aQtiveSpace are the subject of pending patents

aQtiveSpace – how it works!

Alan Dix
aQtive limited
alan@aqtive.com

Abstract

This document describes the computational model and primitives of aQtiveSpace, a component-based architecture designed specifically for context-rich applications. aQtive's onCue product, a form of context-sensitive intelligent toolbar, uses aQtiveSpace as its computational infrastructure. The current implementation of aQtiveSpace is built using Java, but the principles are language independent. Components in aQtiveSpace, called Qbits, have a number of nodes, which another Qbit can set, get or call as well as a variety of listen-type mechanisms. The particular combination of primitives makes it possible to configure Qbits statically or dynamically. Also, the primitives make it easy to capture status-phenomena, which are particularly important for context-aware applications.

Contents

overview 2
fundamental parts of aQtiveSpace...3
Qbits...3
nodes..3
plug and play...4
example – currency converter..5
demand-driven currency converter..................................5
data-driven currency converter.......................................5
contrast..6
advanced features..7
conversations...7
asynchronous interactions...7
aQtiveSpace in Java 8

background and overview

aQtiveSpace is a software framework for producing context-sensitive applications from small components, which we call **Qbits**. It is particularly suited for systems that dynamically reconfigure themselves as new Qbits are added. aQtiveSpace is used as the underlying framework for various aQtive products, in particular onCue.

onCue was launched in the UK on 6th July 1999. By the end of the year more than 1/2 million users in the UK will have onCue on CD and more have downloaded onCue through the Internet. onCue is a unique product. It has aspects of an active toolbar, an intelligent portal and a software agent. It watches everything that is copied to the clipboard, uses 'appropriate intelligence' to suggest suitable Internet services and desktop applications, and automates the use of the data in chosen services. In addition, onCue is extensible – new services can be added to extend its functionality and tune it for particular market areas.

aQtiveSpace is ideal for implementing such a system and a parallel document (onCue – how it works), describes the way in which onCue is constructed on top of aQtiveSpace. This document looks at the aQtiveSpace framework in its own right.

aQtiveSpace has been developed from a strong theoretical standpoint, in particular, it is heavily influenced by status-event analysis. The discrete nature of computation means that at an implementation level everything reduces to events and the system's response to them. This is reflected in the majority of specification notations and implementation platforms. However, many aspects of the physical world are of a different kind, status phenomena, which always have a value that can be sampled. In context-aware applications, such as onCue many of the contextual elements are better viewed as status rather than prematurely decomposed into events. The primitives in aQtiveSpace, although by their nature discrete, are designed to enable an effective and natural encoding of status phenomena. This makes it easier to build context-aware applications, such as onCue, on top of aQtiveSpace. Also the direct mapping between intended user interface behaviour and underlying architecture means that the resulting systems behave correctly and consistently.

Another theoretical underpinning of aQtiveSpace is the theory and practice of agent and component technology. To some extent aQtiveSpace (although developed independently) has similarities with Java Beans. Components (Qbits) have named attributes (nodes) that you can set, get or listen for changes on. aQtiveSpace adds to these basic primitives however, first the thing that is registered with a 'listenable' node is not a special callback function or object, but a settable node. Also, there is a parallel of 'listen' called 'give', which takes a 'gettable' node and asks it for values when required. This symmetry makes it far easier to externally plug together Qbits both statically and dynamically.

fundamental parts of aQtiveSpace

Qbits

The components in aQtiveSpace are called Qbits. In quantum mechanics, qubit refers to a property that is effectively half a bit. A single qubit carries no information in itself, but, when combined with suitable other qubits, does yield useful information. In a similar fashion the Qbits in aQtiveSpace are usually impotent individually, but when combined yield substantial power to the user.

Each Qbit in aQtiveSpace has a series of named **Nodes**. The nodes act somewhat like the named attributes and methods of an object, but with some differences and additional semantics. The nodes are like plugs and sockets by which the Qbit can be connected to its environment and to each other.

nodes

Each node performs one or more of 6 kinds of interaction:

Table 1. Node interactions

set	–	a value can be given to the node (e.g. setting an attribute)
get	–	a value can be requested from the node (e.g. getting the value of an attribute)
call	–	the node can be called as in a normal object method call
listen	–	the node can give a value to a 'settable' node
give	–	the node can request a value form a 'givable' node
supply	–	the node can invoke a 'callable' node

[note – the names of these interactions may change as the latter three have been found confusing!]

These interactions can be classified in two ways:

- by data flow
- by initiative (control flow)

Data Flow – In the case of set and give, data flows into the node. In the case of get and listen data flows out from the node. In the case of call and supply the flow is bidirectional.

Initiative – In the case of set, get and call, the control comes from the outside (external initiative), another Qbit (or arbitrary Java code) has invoked the relevant set, get or call method on the node. In the case of listen, give and supply, the control comes form within (internal initiative) as the node invokes the appropriate intercation when it is ready.

The internal initiative interactions correspond to 'callbacks' found in many systems. They each have a means (in the reference implementation, listen, give and supply methods) of establishing a connection to one or more other nodes and they then invoke those nodes when ready.

The interactions can be matched in pairs as each internal initiative interaction has a corresponding external interaction. For example, a listenable node is given a settable node in its listen method. It invokes the set method on the node everytime it is ready to donate a value.

Table 2. Interaction characteristics

interaction	data flow	initiative	pair
set	in	external	
get	out	external	
call	bidirectional	external	
listen	out	internal	set
give	in	internal	get
supply	bidirectional	internal	call

The input and output data of each node (where relevant) are also typed (e.g. number, text, image).

plug and play

The nodes of one Qbit can be connected to another where they are compatable (i.e. they can function as complementary pairs and have compatable types).

We can represent the node interactions s a 'Lego' block where control flow runs from left to right, input is represented by a hole (wanting to be filled) and output asa peg:

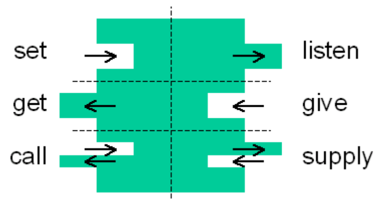

Figure 1. Node as a Lego block

In this representation, two nodes can be connected if they have a corresponding hole/peg combination (and to take the analogy further, if we regard tpyes as the shape of peg, then the shapes must also correspond).

The method by which nodes are described means that Qbits can be dynamically connected together by other Qbits or program code. This is different from, for example, object oriented programming languages, when objects tend to 'know about' a lot of other objects.

Of course, Qbits may have internal structure, including other Qbits, but their external behaviour is very like a Lego brick that can be freely connected to others.

Although, connections can be established statically when an application is configured, the plug-and-play nature of Qbits mean it is particularly easy to connect them and disconnect them form one another while a program is running.

The right hand side interactions, listen, give and supply, are established by a method/function call on the node which establishes the link, later when the

example – currency converter

Let's look at two variants of a pounds-to-dollars currency converter to see how the above interactions work together.

They each make use of the same 'exchange rate' Qbit that monitors online sources to obtain the current pounds-to-dollars exchange rate. It has one node 'rateNow' which is both getable and listenable. The current exchange rate can be obtained by 'get'-ting the node value or by registering a settable node with the listen part of the node. In the latter case the listening node will be 'set' if the rate changes.

demand-driven currency converter

The first 'currency converter' Qbit has two nodes. The 'convert' is a callable node, given an amount in pounds it returns the corresponding amount in dollars. The 'rate' is a giveable node and is where it looks to get the current exchange rate.

As part of the configuration of the system, the 'rateNow' node of the 'exchange rate' Qbit is registered with the 'rate' node of the 'currency converter' Qbit. Now when and external call is made to the 'convert' node, the currency converter Qbit asks its rate node to obtain the value, it performs a 'get' on the rateNow node which obtains the required value. The currency converter is then able to complete the conversion and return the resuklt of the 'convert' call.

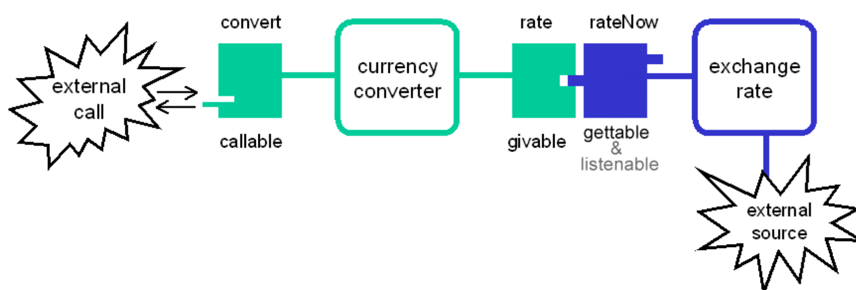

Figure 2. Demand driven currency exchange rate Qbit

This is a demand'driven currency converter as it 'demands' the rate when it needs it.

data-driven currency converter

Now imagine a slight variant of the currency converter. This one has a 'convert' node just like the first, but instead of a givable 'rate' node, instead it has a settable 'rate' node. This can be linked to the same exchange rate Qbit as before, but using a slightly different method.

This time it is the 'rate' node that is registered as a listener to the 'rateNow' node. When the exchange rate node notices a change in the current rate it checks to see if there are any registered listeners, and if so does a 'set' on each with the new rate. The currency convereter is then responsible for keeping an internal copy of the value (probably associated with the 'rate' node. When it is next asked to perform a conversion, it simply uses this saved value of the rate knowing that it is up-to-date.

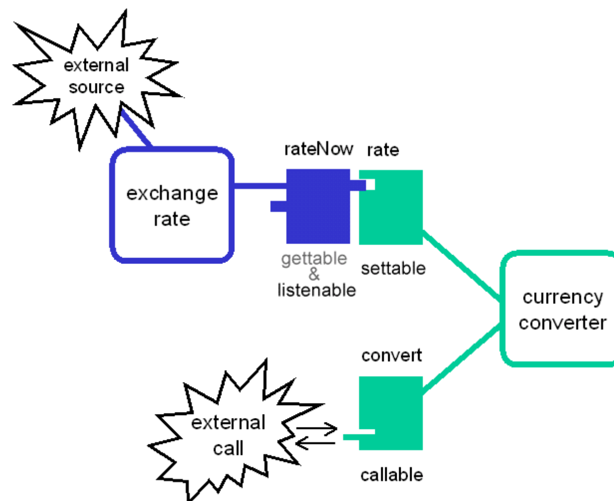

Figure 3. Data driven currency exchange rate Qbit

This is a data-driven currency converter as it is given the rate when the data becomes available.

contrast

Both of these perform the same function, they perform a conversion that is correct with respect to the current status of the exchange rate. The two examples are different ways of obtaining up-to-date status information. They show that:

1. the framework is equally good at managing demand-driven and data-driven actions
2. the same components (e.g. exchange rate Qbit) can be used in either style.
3. the Qbits can be fitted together dynamically without 'knowing about' one another

advanced features

conversations

As well as the data that is passed in an interaction, a 'context' object can also be passed. The initiator of the interaction supplies the context object. In the case of set, get and call this is passed to the Node when the interaction is invoked. In the case of listen, give and supply, a context is passed to the Node when the relationship is established. For example, node A is 'listenable' and node B is 'settable'. Some external code (possibly part of B's Qbit, but possibly completely external) creates a context object C then establishes a listen relationship with a method such as:

```
A.listen(C,B)
```

Later A has data (D) ready and invokes B's set interaction with code like:

```
B.set(C,D)
```

It passes the context on, so that B can, for example, match the 'set' with the corresponding 'listen'.

These contexts can be used to establish 'conversations' where several independent interactions can be regarded as part of a more protracted pattern. In particular, 'call; can be regarded as a simple conversation ('set' followed by corresponding 'get'), which has been included as a primitive for convenience.

asynchronous interactions

All the node interactions may have synchronous and asynchronous versions. For example, one can invoke the 'set' method and wait for the method to return indicating that the set was successful (synchronous). Alternatively, one can use a variant that establishes the request to set the node, but allows the 'setting' Qbit/code to continue to execute (asynchronous). This asynchronous version of 'set' can be thought of as a sort of 'fire and forget' mode. For 'call' and 'get', where a return value is required, a 'callback' can be registered for the value when it is ready (this may be a 'settable' node, or a simpler callback object depending on the implementation).

Asynchronous interactions are particularly useful in networked environments where there may be considerable delays if Qbits are located on different parts of the network. The current reference implementation allows either form of interaction and automatically converts between synchronous and asynchronous versions.

aQtiveSpace in Java

The current aQtiveSpace implementation is coded in Java. Although this is particularly well suited for certain aspects of the implementation, it is not a necessary part of aQtiveSpace and it would be possible to have aQtiveSpace implementations built over other platforms.

The implementation consists of many Java classes and interfaces, but the critical two are the Qbit interface and the Node interface.

The **`Qbit`** interface defines four methods:

`String getName()`
get the name of the Qbit (optional)

`Node getNode(String name)`
get the named node

`Node[] getNodes(NodeTypeSpec spec)`
find all nodes of a particular type

`Node getNode(String name, NodeTypeSpec spec)`
get the node and verify its type

The 'getNodes' method is important as it makes it possible to discover the interaction possibilities of a Qbit without knowing the names of its nodes beforehand (a reflection mechanism).

The **`Node`** interface is more complicated, as it includes methods corresponding to all the interaction kinds:

First of all there are several methods to get the name, parent Qbit and type of the node:

`String getName()`
Get the name of this node..
`Qbit getQbit()`
Return a reference to the Qbit that this node is part of.
`NodeType getType()`
Get the type (input type, output type and interactions) of this node.
`boolean is(Interaction inter)`
Says whether the node can perform the required interaction. The parameter `inter` can be either a specific interaction kind such as `GIVE`, or `SET`, or can be a combination interaction kind such as: '`SET.and(GET)`'.

Then there are methods for set get and call, with a synchronous and asynchronous version of each.

`void set(Context, SyncSetOptions, Data)`
A synchronous version of set. This method will wait until the set has happened or an exception is returned.
`void set(Context, SetOptions, Data, SetManager setter)`
An asynchronous version of set. Any exceptions are returned to the `setter` object at some later point in time. This function returns immediately. The setter object may be null if a result is not expected or needed.
`Data get(Context, SyncGetOptions)`
A synchronous version of get. This method will block until the data is available.

`void get(Context, GetOptions, GetManager getter)`
An asynchronous version of get. The results of the get are returned to the `getter` object at some later point in time. This function returns immediately.

`Data call(Context, SyncCallOptions, Data)`
A synchronous version of call. This method will block until the parameter data has been sent and a result is returned.

`void call(Context, CallOptions, Data, CallManager)`
An asynchronous version of call. The results of the call are returned to the caller object at some later point in time. This function returns immediately.

`void listen(Context, ListenOptions, Node listener, ListenManager)`
This method sets up a listening dependency between nodes. When an event occurs, a set message will be sent/set method will be called on the node `listener`. This method is essentially an event registration.

`void give(Context, GiveOptions, Node giver, GiveManager)`
This method sets up a giving (supply) dependency between nodes. When a value is needed by this node it will get it from the `giver`. This method is to get() what listen() is to set().

`void supply(Context, SupplyOptions, Node supplier, SupplyManager)`
This method sets up a supply (need) dependency between nodes. When a function is required by this node it will call the `supplier`. This method is to call() what give() is to get().

Finally there are methods to cancel a part-finished interaction (asynchronous) and to deregister listeners etc.

`void cancel(Context, CancelOptions)`
Method that allows a client of the node to quit the current interaction.

`void unlisten(Context, UnlistenOptions)`
This method removes a listening dependency between nodes. This method is essentially an event deregistration.

`void ungive(Context, UngiveOptions)`
This method removes a give (supply) dependency between nodes. This method is to get() what unlisten() is to set().

`void unsupply(Context, UnsupplyOptions)`
This method removes a supply (need) dependency between nodes. This method is to call() what ungive() is to get().