

# Architecture-Centric Programming for Context-Aware Configuration

Vibha Sazawal and Jonathan Aldrich  
Department of Computer Science and Engineering  
University of Washington  
Box 352350  
Seattle, WA 98195-2350 USA  
{vibha, jonal}@cs.washington.edu

## ABSTRACT

As computing devices increase in number, it is essential that these devices configure and maintain themselves automatically. A device's *context* may be used to monitor the resources available in the local environment and connect to these resources dynamically.

Unfortunately, it is difficult to write pervasive computing applications that dynamically adapt to their context using current object-oriented languages. In this paper, we argue that first-class language support for software architecture can ease the development of context-aware ubiquitous computing systems. We illustrate our position with a context-sensitive display service, written in the language ArchJava, that uses location to choose among available display resources.

## 1. INTRODUCTION

Mark Weiser's vision of large quantities of computers throughout the environment is coming to fruition [5]. Computers of all shapes and sizes are appearing in our hands, on our walls, and in our living spaces. This abundance of devices is sure to overwhelm its intended users, unless each computer can handle its configuration and maintenance automatically. This challenge of self-configuration and self-maintenance is compounded by a frequent need for communication and connection among the many devices in order to offer *services* that benefit end-users.

One large contributor to a service's self-configuration is context. Context refers to the surroundings of a computing device upon which software components run. In a sea of computing devices, context allows a component to determine which other components can best aid it in completing its task. As the sea of devices and services moves and changes, context alerts components to new situations and enables re-configuration and reconnection.

Context is, of course, used for many purposes other than dynamic connection between components. For example, a service may collect and display contextual information as an interesting service in itself. Nonetheless, connection among software components is an important use of context and one that must succeed for computing devices to work together without requiring user intervention. In this paper, we restrict our interest to this specific use of context.

The use of contextual information for connection between components can be awkward. It is challenging to build services made of components that don't know with whom they connect until they have run-time contextual information. It is even more difficult to write services that can respond to changes in context by changing the services to which they are connected. In conventional object-oriented languages, context-dependant connection code is often mixed with application-specific code, making ubiquitous computing applications difficult to write, understand and evolve.

A new approach, known as *architecture-centric programming*, aims to augment traditional object-oriented techniques with additional language support for connection. Software architecture refers to a software system's components and the connections between those components<sup>1</sup>. Traditional object-oriented techniques provide the means to represent components well, but they lack adequate abstractions for component connection and configuration. In earlier work [3], we proposed that explicit implementation language support for software architecture has the potential to aid in the development of many dynamically adaptive systems. In this paper, we describe how an architecture-centric approach can specifically ease a special case of adaptive systems in which context-aware components make use of contextual information to connect to other components.

## 2. ARCHJAVA: A LANGUAGE FOR ARCHITECTURE-CENTRIC PROGRAMMING

The ArchJava language augments the Java programming language with additional syntax for architecture description [2, 1]. Components are expressed by defining a special kind of class identified by the `component` keyword. Connections are described by defining *ports* for each component, and then by connecting ports together with the `connect` keyword. Ports describe not only the set of methods that a component provides to the component on the side of the connection, but also the set of methods that the component requires the other component to implement.

---

<sup>1</sup>System-wide properties and/or rules that pertain to component structure or the nature of connections are also part of the software architecture. For example, a system's architecture may follow an architectural style that restricts the number of connections a component can have.

The original ArchJava language is limited to describing architectures within a single Java virtual machine, and all connections between components have synchronous method call semantics. However, a proposed extension to ArchJava described in an earlier paper [3] allows programmers to design *custom connector types* that can link components across machine boundaries using connector-specific protocols and semantics. In this paper, we argue that the extended language can ease the development of services that employ context information when creating connections between components. We describe the extended ArchJava language and its potential use for context-aware applications in an example below.

### 3. USING ARCHJAVA TO IMPLEMENT A CONTEXT-AWARE SERVICE

Consider a “Display Service” that runs on a user’s PDA, and displays a document on the nearest available display. The display service must dynamically discover its context: the location of the PDA and the set of display resources locally available. The service must also be able to move the presentation from one display to another as the user moves from location to location.

As a bootstrapping measure, this Display Service knows of a location-sensitive Well-Known Discovery Service that it uses to find available resources. The Display Service begins its work by asking the Well-Known Discovery Service for a Location Service that can track the PDA’s location as the user moves from place to place. It then connects to the Location Service and asks to be informed of the PDA’s location, and to be updated whenever the PDA moves. Upon receiving a location update, the Display Service asks the Well-Known Discovery Service for the nearest Display, passing the PDA’s updated location as the search criterion. The discovery service returns the name of the nearest available Display. Finally, the Display Service connects to the nearby Display and refreshes the document presentation there.

#### 3.1 Discussion of Problem Scenario

This scenario of the Display Service is a simple example of a system that uses context information to make connections. In this case, the context information being used is location. By sending location information along with requests to discover services, the Display Service ensures that the nearest services and displays are used whenever appropriate. In a more realistic system, decisions would likely be made using additional technical criteria, such as connection bandwidth, the capability of hardware upon which services run, and the qualities of the various displays available locally. Nonetheless, even in a more complex system, location would be one of the factors used to make connections. Thus, it is important to consider how contextual information like location can be used by components to easily connect with other components.

#### 3.2 Exploring Solution Possibilities with ArchJava

To illustrate the use of ArchJava with its proposed extension, let us examine the code that implements the context-aware configuration of the display service. We begin by describing how the architecture of the Display Service is defined in ArchJava. Source code for this example is located in the Appendix.

In ArchJava, an architecture is a component that is hierarchically composed from other components. In this case, we will define a component called `DisplayService` with two subcomponents: `Displayer` and `WellKnownDiscoveryClient`. The `Displayer` instance `displayer` encapsulates the functionality and state needed to perform the task of displaying presentations on local displays upon request. The `WellKnownDiscoveryClient` instance `WKDClient` encapsulates the functionality and state needed to reach the Well Known Discovery Service. The `connect` declaration listed on line 7 in the Appendix binds the `WKDisc` port of the `displayer` to the `WKDisc` port of `WKDClient`. Thus, when the `displayer` calls the required method `getName` in its `WKDisc` port, the well known discovery client’s provided method `getName` will be invoked and will return the appropriate `ServiceName`.

The `DisplayService` does not define any more concrete sub-components, because the other services and displays that `DisplayService` communicates with are externally created components that may be ephemeral. Connections that are dynamically created are specified in an ArchJava architecture via `connect` patterns. A `connect` pattern describes the types of components that will interact, but the actual component instances that communicate will be determined based on run-time information. The `DisplayService` architecture describes three `connect` patterns, linking the `Displayer` to a `LocationService`, a `DocumentService`, and a `Display`, respectively.

To support the need for custom, flexible connectors, a proposed change to ArchJava [3] involves the specification of a connection type. In our example, the connection type is called `RobustConnector`. The connection must implement a general `Connection` interface; this connection object is then used by the run-time system to communicate via the custom connector semantics. For example, the `RobustConnector` might implement asynchronous method call semantics, passing arguments across a network using a protocol such as SOAP. The `RobustConnector` constructor must be given a `ServiceName` that, like a URL, uniquely identifies the service to be connected to. The `connection constructor` defined within the `connect` pattern on line 11 returns a new `RobustConnector` created with a specific `ServiceName` and an argument that reifies the connecting port.

The application code of the `Displayer` begins in the function `setup` on line 40 of the Appendix. The `Displayer` begins by calling `getName` on its `WKDisc` port instance in order to look up the name of a location service. Using this name, the `Displayer` can connect to the location service (line 42) by invoking the connection constructor (required on line 28 and provided on line 11). Behind the scenes, the constructor of the `RobustConnector` instance sets up the connection in

a connector-specific manner, perhaps by opening a TCP/IP connection to the local discovery service. The connection constructor returns a new `Location` port instance, which the `Displayer` can then use to communicate with the location service. Thus, on line 43 the `Displayer` sends a message to the location service requesting updates on the PDA's current location.

The `Displayer` receives location updates via a call to its provided `updateLocation` method. Upon receiving an update, the `Displayer` requests the name of the nearest `Display` from the well-known discovery service and creates a new instance of the `Screen` port for communicating with the `Display`. Finally, the method invokes the screen refresh logic (not shown) that communicates with the `Display` through the `displayPort` in order to show the presentation on the new display.

In this example, the connection logic is encapsulated in the `DisplayService`, separated from the connector semantics (defined in `RobustConnector`) and from the application logic in the `Displayer`. Although the `RobustConnector` may be sending many messages behind the scenes for each inter-component method call, the `Displayer` code is separated from the communication code and can use the clean port interfaces to concentrate on the functional logic of displaying presentations. Anytime the `Displayer` instance receives a location update, it simply connects to the local display, creating a new port instance for each connection. Connections to other external services are equally straightforward.

#### 4. ADVANTAGES OF THE ARCHJAVA APPROACH

Although we do not yet have a compiler that allows us to run code utilizing custom connector types in ArchJava, we posit that some preliminary conclusions can be made. We discuss three criteria on which to evaluate our approach below: adaptivity, understandability, and correctness.

**Adaptivity.** The adaptivity of the system is enabled by its ability to use context information at run-time to inform connection decisions. This is implemented via the use of ports that can be created and destroyed at run-time. The `Displayer` can adapt to the locally-available set of displays and services simply by making new connections based on information from the discovery service. While ArchJava cannot ensure that the application logic of context-aware adaptive services is simple, it provides a connection abstraction that simplifies and encapsulates communication with external services.

**Understandability.** The understandability of the system is enhanced by the encapsulation of communication-related code into a connection object. When context is used to make decisions related to connection, connection code can become very complicated. Intermingling this code with task-related functionality results in code that is very hard to understand. Connection objects provide an appealing way to reduce confusion while continuing to employ powerful context-based connections.

Another reason why the ArchJava approach is understandable is the intuitive notion that most developers have about

architecture. The idea of a connector is very well understood by any developer that has had to connect two components together. It may take some adjustment to think of a connection as a first-class object, but at least the concept itself is familiar. Developers are also very comfortable with the idea of separating communication from computation (and its many benefits.) They already do this via use of message passing frameworks or remote procedure call libraries. The ArchJava approach allows them to also factor out much of the context-related code, which is used when making connection decisions but is not yet part of their communication libraries. In our example, the context-dependant code shown is largely independent of the application logic in `refreshDisplay` and other functions.

Finally, the ArchJava approach allows the developer to describe the architecture of their system, without committing to specific connections to specific component instances until run-time. This description allows readers of the code to quickly understand what types of components a `Displayer` instance communicates with without having to scan the entire `Displayer` source code.

**Correctness.** ArchJava verifies a property known as *communication integrity* [4, 1]. Communication integrity ensures that the `Displayer` instance only communicates with the services declared in the `DisplayService` architecture. (We assume that `Displayer` code does not directly use Java's networking library, a property that could also be checked in a straightforward way if necessary). Communication integrity guarantees that the architecture can be relied on as an accurate representation of communication in the system, increasing the understandability benefits.

ArchJava cannot ensure that context information is gathered correctly or used correctly. However, the framework set up by ArchJava may help developers reason about these concerns more easily.

#### 5. CONCLUSION

One common use of context is to determine what run-time connections should be established between components. We propose an approach for handling context-based connections with an architecture-centric language called ArchJava. While there is much more to context than simply connection, we argue that ArchJava can assist in the use of context and the development of adaptive systems that make heavy use of context. Benefits gained may include adaptability, understandability, and correctness. In future work, we plan to implement the ArchJava extensions discussed in this paper and apply ArchJava to adaptive systems that utilize context information. We will then be able to provide more definitive information about the benefits of our approach.

#### 6. ACKNOWLEDGMENTS

We would like to thank Anthony LaMarca, Stefan Sigurdson, Matt Lease, and other members of the Intel Research Lab in Seattle for their assistance in teaching us about adaptive, context-aware systems. This work was supported in part by NSF grants CCR-9970986 and CCR-0073379, and gifts from Sun Microsystems and IBM.

## 7. REFERENCES

- [1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning with archjava. In *Proceedings of ECOOP 2002*, June 2002.
- [2] J. Aldrich, C. Chambers, and D. Notkin. Archjava: Connecting software architecture to implementation. In *Proceedings of ICSE 2002*, May 2002.
- [3] J. Aldrich, V. Sazawal, C. Chambers, and D. Notkin. Architecture-centric programming for adaptive systems. In *Submission to the Workshop on Self-Healing Systems, 2002*, 2002.
- [4] D. Luckham and J. Vera. An event based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9), September 1995.
- [5] M. Weiser. The computer for the 21st century. *Scientific American*, September 1991.

## APPENDIX A. SOURCE CODE FOR DISPLAY SERVICE EXAMPLE

```
1 public component class DisplayService {
2
3     private final Displayer displayer = new Displayer();
4
5     private final WellKnownDiscoveryClient WKDClient = ...;
6
7     connect displayer.WKDisc, WKDClient.WKDisc;
8
9     connect pattern Displayer.Location, LocationService.Location
10        with RobustConnector {
11         connect(Port initiator, ServiceName name) {
12             return new RobustConnector(initiator, name);
13         }
14     };
15
16     connect pattern Displayer.Document, DocumentService.Document ...
17     connect pattern Displayer.Screen, Display.Screen ...
18     ...
19 }
20
21 public component class Displayer {
22
23     public port WKDisc {
24         requires ServiceName getName(String serviceType, LocationInfo locn);
25     }
26
27     public port interface Location {
28         requires connect(ServiceName name);
29
30         requires void requestUpdates(ServiceName name);
31         provides void updateLocation(ServiceName name, LocationInfo locn);
32     }
33
34     public port interface Document { ... }
35     public port interface DisplayInterface { ... }
36
37     private Location locationPort;
38     private Screen displayPort;
39
40     public void setup() {
41         ServiceName name = WKDisc.getName("Location", null);
42         locationPort = new Location(name);
43         locationPort.requestUpdates(getServiceName());
44         ...
45     }
46
47     public void updateLocation(ServiceName name, LocationInfo locn) {
48         ServiceName displayName = WKDisc.getName(
49             "Display", locn);
50         displayPort = new Screen(displayName);
51         refreshDisplay();
52     }
53     ...
54 }
```