

Architecture-Centric Programming for Adaptive Systems

Jonathan Aldrich Vibha Sazawal Craig Chambers David Notkin

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350 USA
+1 206 616-1846

{jonal, vibha, chambers, notkin}@cs.washington.edu

Abstract

Ubiquitous computing services are a fast-growing and challenging class of self-healing systems that must adapt to constant failures and environmental changes. Writing robust ubiquitous computing code is difficult in current programming systems. The architecture, interfaces, and logic of the program are often obscured by infrastructure details, making the development and evolution of these systems difficult and error-prone.

We are exploring whether implementation language support for software architecture can aid in the development and evolution of ubiquitous computing systems. One such approach, embodied in the ArchJava language, allows programmers to express the software architecture of an application within Java source code. In this paper, we propose an extension to ArchJava allowing programmers to define custom connectors. Custom connectors are useful in many different contexts; we show how they can be used to implement part of the PlantCare ubiquitous computing application in ArchJava.

1. Introduction

The ubiquitous computing vision is that embedded computers will be present throughout our living and working environment, working seamlessly together to facilitate user tasks. Ubiquitous computing services are challenging to develop, however, because they must adapt to frequent changes and failures in their environment. These services operate in environments where devices are constantly being moved, upgraded, disconnected, and turned off. A service must rely on many other services to do its tasks, yet the exact identities of these services will inevitably change over time. Ubiquitous computing systems must *self-heal* in response to these changes. In addition to the challenge of self-healing, ubiquitous services must also be flexible enough to communicate with heterogeneous types of devices using many different communication protocols.

Due to these complications, it is unsurprising that even simple ubiquitous computing systems are difficult to write. The most common solution is to base services on a distributed systems library or middleware infrastructure. This approach, however, may allow application code and self-healing logic to be lost in the complexities of communication code. As a result, it is difficult to

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WOSS'02, November 18-19, 2002, Charleston, SC, USA.
Copyright 2002 ACM 1-58113-609-9/02/0011...\$5.00.

see what a service depends on and how it communicates with other services, which can result in errors both during initial development and as the system evolves.

Ubiquitous Architectures. We are exploring whether an architectural approach to writing ubiquitous computing systems can help with these problems. Our intended goal is not to define new ways that systems can heal themselves. Instead, we hope to make it easier for system implementers to reason about the self-healing properties of their code. An architectural approach has the potential benefit of showing the communication and dependencies in a system directly, facilitating program understanding. In addition, we hope to make ubiquitous systems easier to evolve by separating architectural and communication code from the application-specific logic and self-healing code.

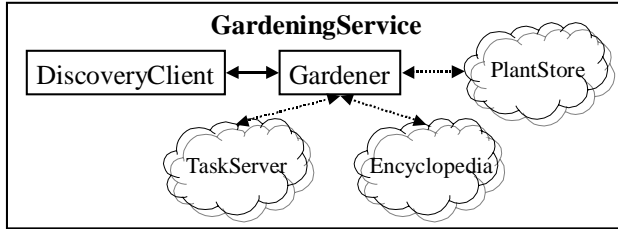
Our Approach. The ArchJava language integrates a specification of software architecture into Java implementation code [ACN02]. In this paper, we show how ArchJava can be extended to support custom connector types, similar to the custom connectors in UniCon [SDK+95] and C2 [MDT98]. Custom connectors can then be written to support ubiquitous services.

Language-based approaches have well-known drawbacks, including barriers to adoption and limiting implementers to the chosen language. However, language-based approaches have potential advantages as well. Our system is the first to extend an existing implementation language with architecture constructs, documenting the architecture along with the code and thus making it more directly accessible. Furthermore, developers are not limited to the connectors and connection protocols supported by a library or system, but can easily write their own as needed.

2. The ArchJava Language

In this section, we illustrate our approach by applying it to the PlantCare ubiquitous computing application. PlantCare is a project at Intel Research Seattle that uses a collection of sensors and a robot to care for houseplants autonomously in a home or office environment [LBK+02]. This application illustrates many of the challenges of ubiquitous computing systems: it must be able to configure itself and react robustly to failures and changes in its environment.

The Gardening Service. Figure 1 shows the architecture of the gardening service, one of several services in the PlantCare system. The gardening service consists of a central gardener component that uses three external services as well as a client for a well-known discovery service. The gardener periodically executes a cycle of code that cares for plants as follows. First, the gardener requests from the `PlantStore` a list of all the plants in the system and the sensor readings from each plant. For each plant, it queries the `Encyclopedia` to determine how that plant should



```

public component class Gardener
    extends StateMachineNode {
    public port discovery {
        requires ServiceID find(String serviceType);
    }
    public port interface PlantInfo {
        requires void statusQuery();
        provides void statusReply(PlantStatus data);
        requires connect(ServiceID id);
    }
    private PlantInfo plantInfoPort;

    public startStateCycle() {
        ServiceID ID = discovery.find("Plant Store");
        plantInfoPort = new PlantInfo(ID);
        plantInfoPort.statusQuery();
        ...
    }
    // remaining Gardener implementation not shown
}

public component class GardeningService {
    private final Gardener gardener
        = new Gardener(getServiceID());
    private final DiscoveryClient client = ...;

    connect client.discovery, gardener.discovery;
    connect pattern Gardener.PlantInfo,
        PlantStore.InfoClient with RainConnector {
        connect(ServiceID id) {
            return new RainConnector(id, ...);
        }
    }
    // other architectural connections not shown
}

```

Figure 1. The architecture of the gardening service

be cared for. After comparing the recommended and actual plant humidity levels, it adds or removes watering tasks from the TaskServer so that each plant remains in good health.

We have chosen to include the interfaces of relevant external services as part of the gardening service architecture, because then we can use the connectors in the architecture to reason about the protocols used to communicate with these services. A more conventional architectural depiction would represent these protocols as connectors in an enclosing architecture. However, in ubiquitous computing systems, there is no way to statically specify the entire enclosing architecture, because the services available in a system may change frequently as devices move and connections fail. Instead, the gardening service architecture includes a partial view of the surrounding architecture, including the external components with which the gardener communicates.

Components and Ports. Below the visual architectural diagram in Figure 1 is the ArchJava code describing the architecture. In ArchJava, a component like Gardener is an instance of a special kind of class identified by the **component** keyword. Components communicate with their environment through *ports*. Each port encapsulates a list of **provides** methods implemented

by the component, and **requires** methods that the component invokes on other components through the port. For example, the gardener’s discovery port contains a required method that is used to find the ServiceID address for a particular type of service. The first line in the startStateCycle method invokes this find method on the discovery port. The DiscoveryClient that is connected to the port will receive the method call, and will return the address of an available PlantStore service.

A *port interface* describes the interface of a port that will be instantiated one or more times on demand. The Gardener component uses port interfaces to connect to external services, because the failure or replacement of a service could force the Gardener to reconnect to a different implementation of that service. The PlantInfo port interface contains a required method statusQuery that allows the gardener to request the current sensor data for all the plants in the system, and a provided method statusReply that the PlantStore calls to return the requested data.

In order to connect to an external service through this port, the startStateCycle method creates a connection with the syntax `new PlantInfo(...)`, passing in a ServiceID that identifies the location of the PlantStore service. This expression invokes the required *connection constructor* declared in port interface PlantInfo using the **connect** keyword. This constructor, declared in the **connect pattern** in the GardeningService architecture, will return a concrete port of type PlantInfo. The next statement in startStateCycle then makes a call through the newly instantiated PlantInfo port, requesting sensor data on all the plants in the plant store.

Architectures. The architecture of the gardening service is defined in the GardeningService component class. The gardening service is composed from two concrete subcomponents: a Gardener and a DiscoveryClient. Ordinary Java code creates these subcomponents when the GardeningService is instantiated and stores them in **final** fields. The **connect** declaration in GardeningService links the specified subcomponent ports together.

While the discovery client is a concrete component that is fixed for the lifetime of the GardeningService, the other services that the Gardener communicates with are externally created components that may be ephemeral. A *connect pattern* describes the types of components that interact, but the actual component instances that communicate will be determined based on run-time information—in this case, the ServiceID returned by the discovery service.

Custom Connectors. ArchJava provides a facility to define connector types with custom semantics. When declaring the connect pattern between the gardener and the plant store, the gardening service defines a connection constructor (defined with the **connect** keyword). This constructor accepts a ServiceID and uses it to create and return a new RainConnector that connects to the given service. Here, RainConnector is a connection library class that implements the Rain protocol used in the PlantCare system. When methods are invoked through connections of type RainConnector, the custom connector code will package the method name and arguments as an XML message, send them over a TCP/IP connection, and call the

appropriate provided method on the other side. Since Rain messages are asynchronous and do not return a response, RainConnector also defines a custom typechecker that verifies that methods in the connected ports have a `void` return type.

3. Discussion

In order to explore our approach, we applied the ArchJava language to the source code for the Gardener component described above (about 300 lines of Java code in total). Initial feedback on our design from the PlantCare researchers has been positive. While we have not yet fully implemented the custom connector extension in our compiler, we have formed some preliminary conclusions based on a comparison of the two versions of the source code.

Self-Healing. Ubiquitous computing services must be robust to communication failures and to failures in other services. The gardening service uses a simple self-healing strategy: if it does not receive a response to a query within a timeout interval, it begins the state cycle again, re-establishing connections to the components it depends on. ArchJava supports this self-healing strategy by providing customizable connectors that can be created and configured at run time.

Program Understanding. The ArchJava version of the gardening service code has a number of characteristics that make it easier to understand the service's implementation. In the Java version, the information about which messages are sent and received is spread throughout the source code. Figure 1 shows how the ArchJava architecture documents the sent and received messages explicitly as required and provided methods in the ports of Gardener, making it easier to understand the interactions between the gardener and other services.

Figure 1 also shows how the ArchJava source code documents the architecture of the service, showing which other services the gardener depends on. This information is completely missing from the original gardener source code; it would have to be deduced from the types of messages exchanged. Another benefit is that the connector specification explicitly documents that the Rain communication protocol is used between components; this would be especially valuable if the gardener used different protocols to communicate with different external services.

Figure 2 shows the ArchJava version of the code that responds to a `PlantInfoReply` message from the encyclopedia. Here, ArchJava's abstraction mechanisms for inter-component communication make the application logic of the gardener clearer. In the original Java code, a single `handleMessageIn` method responds to all incoming messages. The `PlantInfoReply` message is one case in a long list of messages; the code stores the plant care information in an internal data structure and then calls a separate `sendTasksRequest` function to send out the next batch of messages. In the ArchJava version, this response code is more cleanly encapsulated in a single method, which responds to the original message and then sends the next set of messages through the task port. The process of sending a message is also simpler and cleaner in ArchJava. The programmer simply calls a method in the `taskPort`, rather than constructing a custom message and sending it using the Rain library.

Correctness. The ArchJava language performs a number of checks that help to ensure the correctness of the `GardenerService` implementation. For example, the

```
void infoReply(PlantInfoReply data) {
    careMap.put(data.name, data);
    state = AWAITING_TASKS;
    try {
        taskPort.taskQuery("Water Plants");
    } catch (Exception ex) {
        // an error occurred, restart the cycle
        ex.printStackTrace();
        resetState();
    }
}
```

Figure 2. Gardener message response code in ArchJava

RainConnector typechecker verifies interface compatibility between the ports of Gardener and the connected ports of the external services at compile time. In the original Java code, this problem would show up as a run time error when a component does not recognize a message that was sent to it.

Software Evolution. Because of ArchJava's explicit abstractions for ports and connectors, some evolutionary steps are easier to perform. For example, if a service needs to interact with a device that cannot generate XML messages, we can replace RainConnector with a new connector type that can communicate with the more restricted device. Also, we can reuse an existing service in a new environment by simply inserting adaptor components that retrofit the old service to the message protocol expected by the new environment. In both cases, ArchJava's explicit descriptions of component interfaces and connections make architectural evolution easier.

An important criterion to consider in the evolvability of a system is the degree to which the system's modularization hides information within a single module. One benefit of the ArchJava version of the gardening service is that the gardener's functionality is encapsulated in Gardener while the communication protocol used is encapsulated in GardeningService. The ports of Gardener serve as the interfaces used to hide this information. Thus, in the ArchJava code, the gardening functionality can be changed independently of the communication protocol, facilitating evolution of this service.

Acknowledgements

We would like to thank the anonymous reviewers, as well as Anthony LaMarca, Stefan Sigurdsson, Matt Lease, and other members of the PlantCare group at Intel Research Seattle for their help, comments, and suggestions.

References

- [ACN02] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting Software Architecture to Implementation. In ICSE '02, Orlando, Florida, May 2002.
- [LBK+02] A. LaMarca, W. Brunette, D. Koizumi, M. Lease, S. B. Sigurdsson, K. Sikorski, D. Fox, and G. Borriello. PlantCare: An Investigation in Practical Ubiquitous Systems. In UbiComp '02.
- [MDT98] N. Medvidovic, E. M. Dashofy, and R. N. Taylor. Employing Off-the-Shelf Connector Technologies in C2-Style Architectures. In CSS'98, Irvine, CA, October 1998.
- [SDK+95] M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. In IEEE Trans. Software Engineering, 21(4), April 1995.