NORTHWESTERN
UNIVERSITY

Electrical Engineering and Computer Science Department

Technical Report
NWU-EECS-09-13
May 15, 2009

Pennyworth: A platform for building context-aware applications for everyday use

Chris Karr & Darren Gergle

Abstract

In this work we describe Pennyworth, a software system that provides both a reusable context inference engine and an infrastructure that supports context-aware features across a variety of heterogeneous applications.

Keywords: human-computer interaction, context-aware computing, context toolkit, context-sensing, sensor-based interactions

Authors' note: This technical report describes Pennyworth, a context-aware platform for building everyday end-user applications. The design and development of the system described within has continued and the system has evolved in terms of platform support, underlying architecture, and application interfaces. To obtain the most recent developments and iterations of Pennyworth, please visit

http://www.pennyworthproject.org/

Introduction

> *Our hero opens the door and steps inside the room. Before he can close the door, the room has turned on the lamps and drawn the blinds to showcase a spectacular sunrise. As he finds a comfortable position in the seat behind his workstation in the center of the room, his computing terminal brings up a list of e-mails that arrived overnight.*
>
> *The clock-like device to the right of his screen activates and displays a small list of urgent activities. He notices that he has not yet picked up the dry cleaning for the weekend's festivities. He taps the item on the screen and the phone in his pocket chirps to confirm that it will remind him to retrieve the articles on his way home that evening.*
>
> *Back on the main screen, he scans the messages received overnight. He is pleased that there are no outstanding emergencies and he can resume work on the project begun the day before. As he sits up in his chair and opens the project files, the room draws the blinds behind him to block the intense glare of the young day. The overhead speakers begin to quietly play a classic Sinatra tune.*
>
> *Our hero goes to work...*

This small fictional blurb reads like a passage from a Phillip K. Dick or Stephen Baxter science fiction story. It invites the reader to imagine the protagonist in a futuristic setting that features advanced technology such as a room that opens the window long enough to display the sunrise and a clock and phone cooperating to remind him to pick up his dry cleaning.

The tragedy of this anecdote is that it reads like science fiction. Outside of research labs and the homes of technology billionaires, most of us never experience such an environment. Our devices remain ignorant of each other and fail to coordinate in any meaningful manner to help us accomplish our daily objectives. We find ourselves manually selecting our favorite music and resuming work on a project that often involves hunting for a variety of electronic resources strewn across a variety of folders and online hosts.

Nothing described in the anecdote should be considered science fiction. The technology to adapt a room for its inhabitant can be found in commodity home automation products. A modern desktop computer system includes sufficient technology to activate itself and restore the desktop to a prior state before the user even touches a keyboard. A device called the Chumby can be the clock-like device that communicates a task reminder to the mobile phone. Automating a media program to play activity-appropriate music is a simple exercise in scripting.

So, if all of the technological building blocks are presently available to make our anecdote a reality, why do we spend our days interacting with primitive systems that demand we adapt to them instead of the reverse?

The answer is simple: two major components are missing from our systems. The first missing component is *a reusable context-inference engine*. This is the technology that allows a system to infer your situation based upon information that it can collect from the surrounding environment.

The second missing component is *infrastructure* that supports context-aware features across a variety of heterogeneous applications.

In the following chapters, we describe *Pennyworth*, a software system that provides both the context-inference engine and necessary infrastructure that enables scenarios such as the one described above. In the next chapter, we discuss the prior efforts that inspired the creation of this system as well the limitations of those efforts that made creating Pennyworth necessary. Drawing upon that discussion, in the third chapter, we describe the architecture of the system by highlighting the successful elements it adopts from previous work as well as design decisions made to avoid some of the pitfalls that plagued earlier efforts. The fourth chapter describes how the Pennyworth system can be deployed in practice by illustrating how context-awareness improves existing applications and highlighting new opportunities for novel work to address the challenges imposed by context-awareness.

Before we discuss how to create context-aware software using the Pennyworth system, it is useful to understand why such a system is absent and not already ubiquitous on our existing systems.

Whither the context-aware applications?

In the introduction, we argue that adaptive context-aware systems are possible today and the main limitation to their deployment is the lack of reusable general-purpose context-sensing software and the necessary infrastructure required to coordinate across a variety of software applications and hardware devices.

We argue that context-aware systems are both useful and desirable because they address a fundamental flaw inherent in the construction of *all* software and devices: the idea of a *typical user*[1].

Since most software users do not create their own programs, they rely upon developers to build, test, and support the software that they use. Developers do not typically create unique "one-off" applications for individual users, but instead rely upon some form of aggregate user to guide their system's design and implementation. This reliance guarantees that unless an individual perfectly fits the mold of the developer's aggregate user, he will have to adapt in order to use the application in the most effective manner.

The ubiquity of application preferences and user-defined options underscore this problem. The choices exposed by these interfaces illustrate areas where the designer believes that their conception of a typical user does not overlap with every other person in the product's target user base. Thus in the broadest case, applications that are sufficiently context-aware to infer the local user's identity (relying upon mechanisms such as user accounts) can use an *identity-configuration mapping* to overcome the limitation that designers cannot create customized software for each individual user. The user still must configure the software to match his preferences, but once that configuration is set, a context-aware technology can free its users from restoring their preferences after another finishes using it.

This may seem like an insignificant point that is irrelevant in the age of multiuser systems where the system manages the identity-configuration mapping in the background. This does describe the world that the majority of users currently inhabit. However, experience demonstrates that identity is not always sufficient and that adding other elements to the identity-configuration mapping is often necessary.

For example, e-mail applications with built-in notification systems post alerts when new messages arrive. Since users have different preferences about how they should be notified, the application designer includes a configuration panel. The options exposed by these panels allow the user to select whether to use a visual notification (such as a bouncing icon or an unread message count) or sound effects.

---

[1] Cooper (2004) argues against using a typical user when designing products, instead opting for a collection of fictional users called *personas*. While this is a useful tool for making design decisions, personas exhibit the same problems discussed above as a single monolithic typical user.

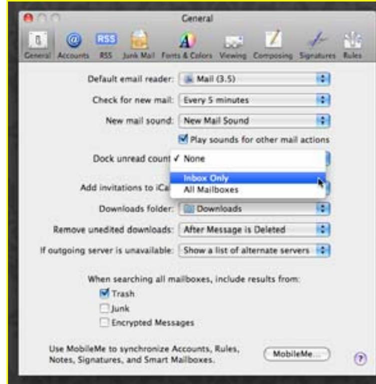

Figure 1: Apple's Mail.app program allows users select a variety of audible and visual notification options.

If the identity-configuration mapping were sufficient, then these configuration options would always be sufficient for users to manage their preferences. However, as anyone who has ever given an important slideshow presentation will attest, mail notification preferences are not only a function of the user's identity, but also a function of other factors such as the user's current activity (e.g. "giving a presentation" vs. "reading e-mail") and location (e.g. "home office" vs. "lecture hall").

Unless software designers wish to force their users to manually reconfigure their systems each time a situation like a slideshow presentation occurs, *software and devices must move beyond the identity-configuration mapping and begin using contextual information to provide their users the best configuration for the current situation*. We argue below that robust software that distinguishes between users' situation, combined with a pervasive infrastructure for sensing and disseminating context provides the solution to this problem.

Previous Efforts

In the next chapter, we describe Pennyworth, a context inference engine designed to provide application developers the necessary tools to begin creating context-aware software. However, Pennyworth is only a recent addition to a long line of context-inference engines created over the past decade and a half. Its primary innovations are that it makes the context-sensing mechanisms *visible* and *malleable* by end users and it provides application developers a useful set of tools for constructing context-aware applications without requiring machine learning and sensing expertise. This development is the result of a natural progression of systems where the idea of context morphed from an isolated application-specific notion to a system-wide property to a socio-technical construct that extended from the circuits of the machine into something negotiable between software and its user.

Schlit and Theimer (1994) first introduced the term *context-aware computing* when they described their location-based application that monitored users' and objects' location in physical space. This early instantiation of a sensor-based context-aware system was limited in that the context was wholly contained within a single application and not made available to other programs.

Dey, Abowd, and Wood (1998) overcame this limitation by creating the first reusable context-aware components in the CityDesk framework. CityDesk allowed applications to share contextual information in order to expose relevant services as the user worked. For example, CityDesk would make visible the option to open a web browser when the user highlighted a URL in another application. Drawing upon their experience using CityDesk, Dey and Abowd created the Context Toolkit (Salber, Dey & Abowd 1999) as a reusable framework for creating general-purpose context-aware applications.

The Context Toolkit adopted the idea of reusable components from object-oriented desktop systems as a model for creating context-aware software. In their approach, applications subscribe to context widgets, which hide the complexity of the sensing mechanism from the application programmer. A developer writing a location-based application would subscribe to the location widget. An application that monitored meeting attendance would subscribe to presence widget. The Context Toolkit supported higher-level concepts (such as activity) by allowing developers to write aggregator widgets where developers could write code that interpreted readings from a collection of other widgets to generate new context information. A location widget combined with a social proximity widget could determine if the user was attending a meeting or chatting with coworkers in the hall.

One drawback to the Context Toolkit is that it assumed that the mappings from sensor readings to application outcomes were unambiguous and that developers would readily recognize and preemptively encode appropriate behaviors that map incoming sensor data to application outcomes. Fogarty recognized that this assumption was idealistic and used Munguia Tapia's insight that statistical models augmented with sensors could be used to predict higher-level concepts (Munguia Tapia, Intille & Larson, 2004). Fogarty extended this approach to create statistical machine learners that classified sensor readings to predict contextual attributes of a user, such as interruptibility (Fogarty, Hudson & Lai 2004) and activity (Fogarty, Au & Hudson 2006).

Using machine learners as context-interpreters, Fogarty created the Subtle toolkit that added the learners to the Context Toolkit's basic architecture (Fogarty & Hudson 2007). Like its predecessor, Subtle included support for sensors and a means of disseminating context information to other applications. Furthermore, Subtle made writing context-aware applications more approachable by applying machine learners to predict higher-level concepts using models trained by the user. By transforming sensor readings into higher-level concepts, Subtle allowed developers to create applications without worrying about interpreting individual sensor readings. Instead, developers created applications that monitored changes in the higher-level concepts (such as activity or interruptibility) and acted upon that information. By using machine learners as the interpreters between the sensors and the applications, Subtle eliminated the need for any direct relationships between the two architectural components.

Unfortunately, Subtle's architectural innovations have been overshadowed by its lack of availability, large resource and runtime requirements, and non-existent user interface. While the system aptly demonstrated how to decouple sensing and inference, it's lack of focus of practical

considerations (availability, usability & performance) render it unsuitable for widespread deployment. Pennyworth picks up where Subtle left off by

Mainstream Efforts

Despite the active work in context sensing within the research community, mainstream developers have been slow to adopt or implement context-aware functionality other than simple location sensing. The current state of the art of mainstream software more closely resembles Schilit and Theimer's fifteen year-old efforts than the more recent research work. Microsoft's Sensor and Location Platform for Windows 7[2] is a notable exception and implements a similar architecture to the Context Toolkit, but it suffers from the same limitations in that it requires *application* developers to directly process low-level sensor information into useful higher-level concepts. MarcoPolo (Symonds 2006) is another attempt at enabling mainstream context-aware software, but it requires that users define exhaustive sets of rules to interpret their context instead of leveraging the computer's ability to make inferences.

Consequently, there is a significant gap between researchers inventing context-inference techniques and developers writing software for everyday users. Researchers produce powerful systems that demonstrate context-inference is possible, but fail to follow through and make their systems usable in wide deployments. Software developers create practical systems, but these are crippled by simple and crude approaches used to infer context.

The raison d'être of the Pennyworth system is to bring these two parties together by providing a robust and usable system that enables users and developers to begin using research-grade context inference technologies in the activities of everyday life.

---

[2] http://www.microsoft.com/whdc/device/sensors/default.mspx

Bridging the gap: Pennyworth

Fundamentally, the major problems preventing mainstream developers from applying the insights generated in the research community all have straightforward solutions. The problem of mainstream developers not adopting research approaches in their own work can be overcome by embedding that knowledge in a reusable software framework. The lack of sufficient developer support can be addressed by a toolkit creator actively choosing to make a commitment to document the system, assist users and developers, and promote the toolkit in order to build a sustainable community around it. Prompting developers to think beyond location-based services is best accomplished by making available a variety of novel applications that stimulate developers to imagine context-aware technologies of their own.

In addition to being a tool for mainstream developers, a context-awareness framework deployed in a variety of applications "in the wild" provides an ideal proving ground for developing and testing theories about whether context meaningfully improves interaction between people and machines, the consequences of specific configurations, and what life is like in a context-aware information ecology. None of these questions will be answered definitively until research-grade technology is available to mainstream users.

In this chapter, we describe Pennyworth[3], a robust context-inference engine designed for production use by mainstream developers. Pennyworth features a theoretically informed architecture and design that embodies the best current understandings as well as addressing shortcomings in the previous work.

This chapter contains three major sections. The first section provides an overview of the Pennyworth system from the perspective of the end-user. The next section describes Pennyworth from the perspective of an application developer who wishes to include context-aware functionality in their product. The final section describes Pennyworth from a theoretical perspective addresses issues and themes found in the HCI literature about context-aware systems.

Pennyworth overview

Pennyworth is a user-space application that uses its sensing capabilities to continuously observe the local environment surrounding the user's desktop computer. Pennyworth learns how to interpret this information by relying upon the user to train it to recognize the user's *activity*, *location*, and *social context*.

[3] The following sections describe the Pennyworth application as implemented on the Mac OS X platform. While some differences will exist between the flagship Mac version and ports to other platforms, the approaches and principles described here will remain consistent across operating systems and hardware platforms.

Pennyworth runs as a background application, but it exposes a modest user interface that allows the user to

1. Monitor the inbound sensor readings
2. Track the application's context inferences.
3. Correct the system as needed.
4. Review the system's generated context models.
5. Override the system with rules when necessary.
6. Create and install scripts that enable new sensors and application control.
7. Configure which sensors are active and which are disabled.

The most significant difference between Pennyworth and previous systems like the Context Toolkit (Salber, Dey & Abowd 1999) and Subtle (Fogarty & Hudson 2007) is that Pennyworth allows end-users to take a more active role in managing the system. The implementation of Pennyworth takes great pains to be as accessible and transparent to the end-user as possible. The rationale for this fundamental design principle is described in the theoretical section later in this chapter.

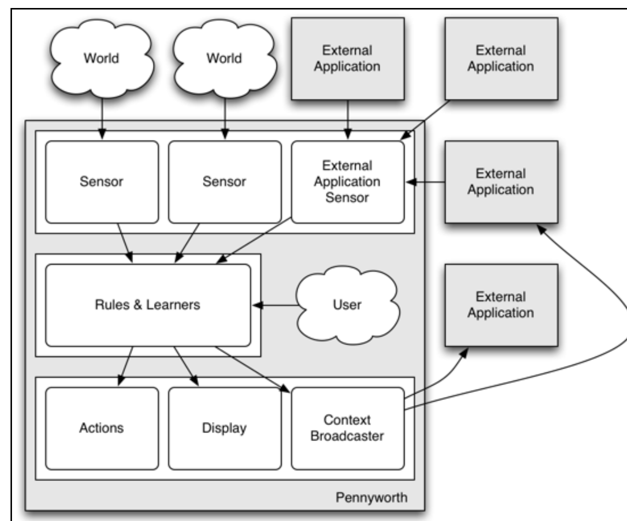

Figure 2: Pennyworth Architecture

Pennyworth is not intended run in isolation; rather, it communicates the user's context to a variety of other applications so that those applications can react in their own specific ways to changes in the user's situation. Throughout this chapter, we will use the example of a context-aware time tracking tool to illustrate how Pennyworth provides context information to other programs. In this scenario, the user trains Pennyworth to recognize their context and the time tracking tool simply logs changes in the user's context for later reflection analysis. (This model is similar to the approach that traditional financial management software employs to allow users to track their spending habits.)

To begin using the system, the user must first train the system to distinguish between several contexts. When a user first launches Pennyworth, the application makes itself visible by adding a small bell icon in the user's menu bar. When the user clicks the icon, a menu appears that allows

the user view and control the system in a variety of ways. Using the small footprint of a system menu bar item as the primary interface to the context inference features allows the user to keep the system running as long as they are signed into the system.

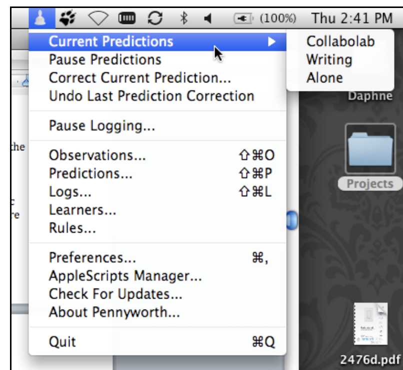

Figure 3: Pennyworth's features are accessible from the user's menu bar

In Figure 3, the currently predicted context is visible from the drop-down menu. In this example, the system is predicting that the user is alone and writing in a specific laboratory as the current context.

If any of these predictions are incorrect, the user may invoke a correction interface by selecting "Correct Current Prediction" from the menu.[4] This action summons a small training panel where the user corrects the system's predictions. The user may select a context label that has been previously used or he may elect to input an entirely new label. The system remembers new labels and suggests them in subsequent training interactions. Using this mechanism, users train the system using the labels that make the most sense for them – the system does not rely upon an arbitrary set of labels preselected by the system's designers. Furthermore, correcting the current context is a context change, and Pennyworth notifies our hypothetical user's time tracking software automatically, so no further interaction is necessary.

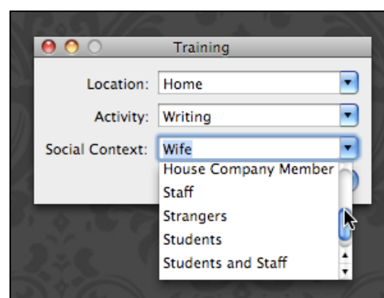

Figure 4: Pennyworth's training interface

If the user wishes to monitor the system's context predictions, he can do so using the menu (as shown in Figure 3), or the system can display a small panel that updates in real-time with the system's current context predictions.

[4] To expedite training and minimize interruptions to the user's workflow, the correction interface may be invoked using a system hot key (Command-Control-C) as well.

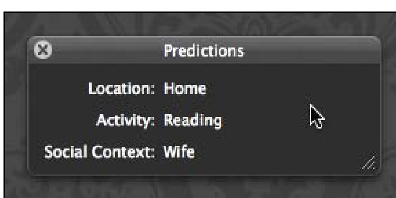

Figure 5: Current contexts predicted by the system. New predictions can also be reported using the Growl notification system.

This display is small enough that it may be placed in the corner of the desktop for peripheral monitoring while other activities are underway. Using this panel is helpful when training a fresh system, as the user can instantly recognize and correct errors in the prediction. As time and training elapses, the user may choose to hide this display when they feel that the trained model has become sufficiently accurate for their purposes.

Predictions are generated in real-time using input from a variety of sensors. The user can display the output of the sensors from the menu bar icon. This display reveals all of the active sensors on the system and the value that each is currently reporting. This information used by the machine learners to predict context and is always visible from this interface.

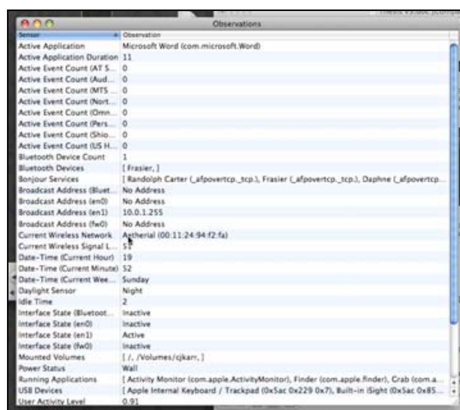

Figure 6: Active sensor readings

The information collected by the sensors is used by C4.5 decision tree learners (Quinlan 1993) to infer the user's current context[5]. The model generated by these learners is also visible to the user.

---

[5] In this thesis, the term *sensor* may be used interchangeably with the machine learning term *feature*. Strictly speaking, Pennyworth's sensors can and do interpret raw inputs to extract higher-level concepts. For example, the *idle time* sensor calculates the elapsed period since the user last interacted with the computer using an input device (e.g. mouse, keyboard, etc.). However, it also uses that information to keep a running average that describes the user's overall amount of engagement as a fraction between 0 and 1. The idle time measurement is subject to less interpretation than the activity level measurement, but both results are treated equally as *features* within the machine learner. The program uses the term *sensor*, as it is more congruent with end-users' mental models than the term *feature*.

The decision tree is presented using a nested-set diagram[6] and the user may view any of the generated context models used to infer their situation. In our hypothetical user's case, the current active application may be tightly bound to the user's activity (e.g. Active Application = Word □ Activity = Writing; Active Application = World of Warcraft □ Activity = Playing Video Game). The decision tree can recognize these relationships and build a model accordingly.

Making the decision tree visible allows the user to take a more active role in training and shaping the learners as well as providing a means of harmonizing the user's mental model of the system with the system image and generated context models (Lederer, et al. 2004). If user experiences difficulty training the system, he may review the model to try and identify troublesome sensors generating spurious readings (e.g. Current Hour >= 10 □ Activity = Writing; Current Hour < 10 □ Activity = Web Browsing). If a culprit can be identified, the user may disable it in the system's preferences. No prior research system grants users this level of independent control, which is extremely helpful in configuring sensors and learners that are most effective for in a given setting.

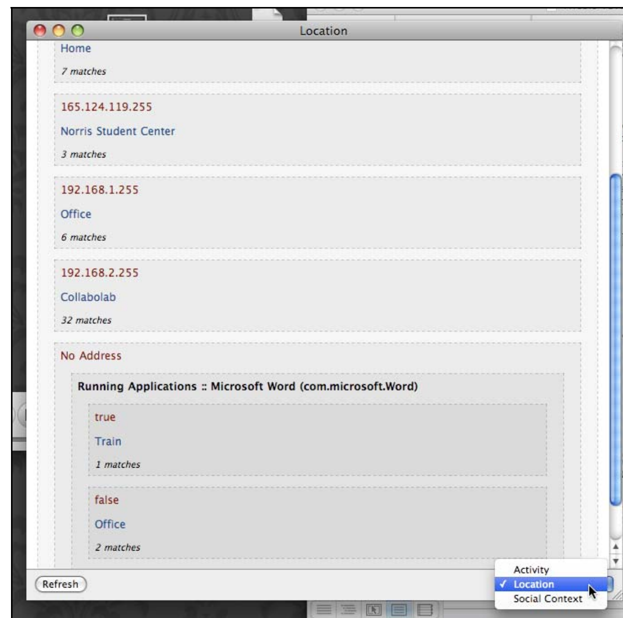

Figure 7: A context model that predicts location as a function of the local wireless network. In this case, the predicted location changes when the user's IP address changes. When no address is available, the system uses the active application to choose the predicted location.

While disabling problem sensors is an effective tool in training the system, there are troublesome contexts that a model will never be able to learn using Pennyworth's real-time training interface.[7]

---

[6] Nested set diagrams are useful for making the context models visible and understandable, but user feedback suggests that they are not sufficiently clear to the novice user. Future iterations of Pennyworth will explore using other techniques (such as flowcharts) to better represent this information.

[7] An additional interface that preserved sensor readings and asked the user to provide *post hoc* labels may be able to predict some of these troublesome concepts, but would likely produce more erroneous labels as users were forced to self-report their context at the recorded time. Such an

For example, an “Away” context is one where the user is away from the system. Since training the learners requires an active interaction between the user and the system, learning “Away” is impossible, as the user must be present to provide that label. To address this kind of problem, Pennyworth implements a rules system (not unlike that found in MarcoPolo) that wraps the context model. Using the rules system, the user can define “Away” as a state that satisfies a particular set of sensor conditions. The user may define an entire collection of rules that the system uses to infer context before resorting to the machine learners.

No previous context-inference systems combine both machine learners and a rule system to create a single context model. This combination allows the system to take advantage of both approaches: the machine provides superhuman attention to detail and the ability to detect novel patterns, while the users distill their own knowledge into rules. Rules allow users to specify significant relationships between sensors and contexts that may be difficult to impossible for the machine learners to discover on their own, while the machine learners free users from the need to exhaustively specify every rule that relates sensor readings to context.

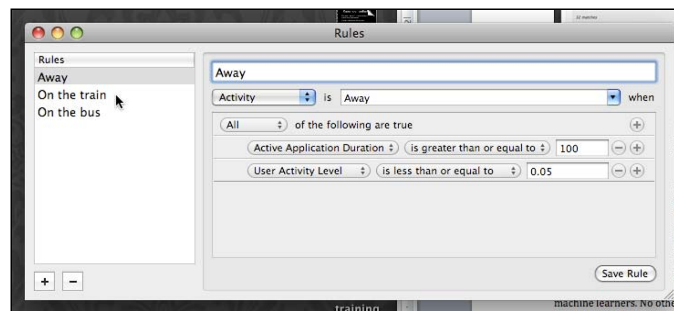

Figure 8: Rules are compound Boolean expressions that evaluate sensor readings to infer context.

The user has a tremendous amount of access and control over Pennyworth’s context inference features. However, errors will occur in training and corrective action may be required. In the simple case where a user incorrectly inputs their context, the system provides a simple mechanism where the user can undo the last correction (as seen in Figure 2). However, deeper corrections may be required and the user may “edit” the learners by renaming, merging, or deleting labels provided in the system.

Figure 9: Labels may be corrected within the preferences.

---

interface is a future option if both the real-time machine learner and user-defined rules are insufficient to predict users’ context, but does not appear to be necessary at the time of writing.

If the user experiences problems training the system to recognize a specific context, he can selectively "reboot" a learner by deleting the problem label and beginning fresh with a new set of enabled sensors and fresh training instances. Renaming labels allows the user to update descriptions of their context in addition to providing a mechanism for merging labels. For example, if the user began training the system to recognize the difference between "Writing a proposal" and "Writing a letter", and the distinction between the two ceases to be useful, he may rename both to "Writing" without retraining the system to recognize the new label.

While manipulating labels allows the user to fix minor problems, there may be instances when the learner must be completely reset and the training begun anew. For example, our hypothetical user may experience a career change, and the model trained to distinguish between activities in the old job may not work as well as a fresh model trained with information from the new job. The system provides a convenient "reset" button for these cases.

Anecdotally, Pennyworth can generate useful context models with as few as fifty or sixty training examples. Since the system generates new models within seconds of receiving training labels[8], the user can abandon previously learned models, retain the system, and resume using it productively in a period as short as a couple of hours.

Enabling context-aware applications

The discussion thus far covered how users interact with the Pennyworth system to provide it meaningful labels and to train the system to infer their context from those labels and continuous sensor readings. However, the utility of a system that only infers context is extremely limited. Pennyworth provides a variety of mechanisms to translate its inferences into meaningful action.

Previous systems approached this problem by taking a "library" view of the world. That is, designers of these systems assumed that other application developers would link their program to their context inference engines to provide context-awareness features *within* the target application. Pennyworth functions more as a system service that target applications *consult* for context information. By taking a service-oriented approach, new applications can use the context models that were trained for other applications without requiring the user to train every new application that they add to their system.

Despite its advantages, even the service-oriented approach will be inadequate for all but the most trivial circumstances. The service-oriented approach still assumes that the third party developer is aware of Pennyworth and programs accordingly. Unfortunately, until context-inference engines achieve a critical mass of adoption that attracts the attention of application developers, the engines will continue to be ignored and no compatible software will be released.

[8] The C4.5 decision tree algorithm is sufficiently efficient and can generate context models in a few seconds or less with as many as sixty training examples. The current implementation of the algorithm runs in approximately O(n log(n)) time. As the number of training examples increase, model generation time increases as well. However, with as many as 100 training examples, Pennyworth generates 3 context models in less than a minute.

Pennyworth avoids this chicken-and-egg problem by including the ability to take action on its own using the host system's native scripting architecture. Rather than rely upon developers to make compatible applications, Pennyworth empowers users to independently endow other applications with context-awareness. From the system's menu bar item, users can invoke the script manager interface to download and install new action scripts or write their own.

In the case of our hypothetical user, he may have a preferred time tracking tool that he wishes to make context-aware. If the existing time tracker exposes a generic scripting interface, he can craft a script that bridges Pennyworth and the preexisting software.

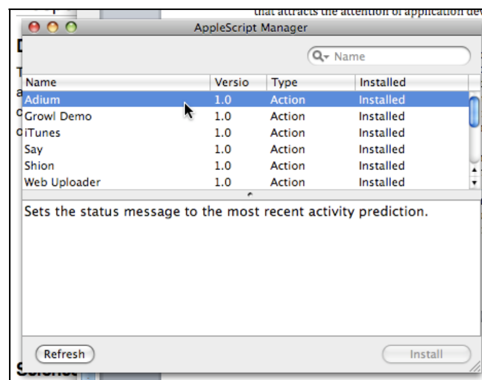

Figure 10: The script manager allows users to use context to automate their own applications.

Pennyworth executes each of these scripts each time the user's context changes. Scripts are designed to be small and easily understood by end users. For example, this script sets the current activity in the user's time tracking tool:

```
on prediction(type, prediction)
  -- type is the kind of context: "Activity", "Location", "Social
  --                                "Context"
  -- prediction is the value: "Writing", "Home", "Alone"
  if (type is equal to "Activity") then
    tell application "Finder"
      repeat with p in (processes whose visible is true and
                        name is not "Finder")
        try
          if (name of p is equal to "Generic Time Tracker") then
            tell application " Generic Time Tracker "
              set the current activity to prediction
            end tell
          end if
        end try
      end repeat
    end tell
  end if
end prediction
```

Similar scripts can be used to change the playing track of a media player, alter file-sharing settings using location as a guide, or change notification preferences when playing a game.

This mechanism works well on host systems that support a common application-scripting interface. On the Mac OS X platform, there is an existing expectation of application developers that they will implement scripting interfaces in their applications. This norm has the side effect of making hundreds of applications context-ready *without any extra involvement from the original developers*. Users decide how they wish for their applications to behave and they write or adapt scripts to implement those behaviors.

Ideally, the ability for users to make their own applications context-aware will spur adoption of the Pennyworth system and encourage developers to consider how their applications can productively apply the user's context. Developers take the first step to realizing this by simply making their applications scriptable.

However, as mentioned above, Pennyworth is a service that provides context inferences to other applications. Pennyworth implements this service as using the Observer design pattern (Gamma, et al. 1994). Applications subscribe to the Pennyworth service and the system broadcasts context updates asynchronously as they become available.

On the Mac, this mechanism is implemented using the system's notification center. Applications subscribe to context updates using the following method:

```
[[NSDistributedNotificationCenter defaultCenter]
                                        addObserver:self
                                           selector:@selector(update:)
                                               name:PREDICTION_FETCHED
                                                   object:nil];
```

When new context information is available, the "update:" method will be called:

```
- (void) update:(NSNotification *) theNote
{
    NSDictionary * userInfo = [theNote userInfo];
    NSString * key = [userInfo valueForKey:KEY];
    NSString * prediction = [userInfo valueForKey:PREDICTION];

    // Do something useful with the context prediction
}
```

In this block of code, developers implement the desired context-specific behavior. Using this mechanism, our hypothetical time tracker developer can make the bridging script described above obsolete by crafting his application to listen for context change notifications from Pennyworth.

If developers wish to have their own applications serve as sensors for Pennyworth, they can submit new information using Pennyworth's own scripting interface or broadcast updates in their own code. The following example shows how our hypothetical time tracker could contribute the current project selected as a sensor value:

```
NSMutableDictionary * note = [NSMutableDictionary dictionary];
[note setValue:@"Current Project" forKey:COCOA_SENSOR];
[note setValue:project.name forKey:COCOA_OBSERVATION_VALUE];
```

```
[note setValue:[NSNumber numberWithInteger:15]
        forKey:COCOA_OBSERVATION_DURATION];
 [[NSDistributedNotificationCenter defaultCenter]
    postNotificationName:COCOA_OBSERVATION object:@"Time Tracker"
                userInfo:note];
```

Pennyworth receives these broadcasts and integrates the readings into the learner framework as if the information had originated from an internal sensor. In this way, other applications can expand the set of available options that Pennyworth uses to "perceive" the surrounding environment.

Theoretical considerations

The overall design and implementation of Pennyworth is driven by the practical goal of creating a robust and usable system that implements some of the major approaches found in research toolkits in a manner that encourages mainstream adoption by end users and third party developers. However a focus on overcoming practical barriers to widespread adoption does not preclude anticipating and addressing theoretical concerns. This section further situates Pennyworth in the context-aware literature and explains how its design and implementation addresses points advanced by members of the HCI community.

In a discussion of the architecture of the Context Toolkit , Winograd (2001) identifies three major architectures for context-providing software components: *widgets*, *service infrastructures*, and *blackboards*. Widgets are the approach used in Abowd and Dey's system where developers include software components that directly interact with sensors to achieve context-awareness. Services decouple the application and the context gathering by defining a standard interface that applications use to retrieve higher-level information from a centralized context service. Winograd's blackboard model mirrors the Observer design pattern (described above), where applications subscribe to a shared space where context messages are posted.

The widget approach silos the context-aware functionality within a single application. The service and blackboard models decouple the inference and sensing components from the application, freeing the developer to focus on how context should be used in his application. Pennyworth rejects the widget model in favor providing both the service and blackboard interfaces. The service architecture is implemented using the system's scripting interface. Any application on the local system can use this interface to retrieve the current context or any of the sensor readings:

```
tell application "Pennyworth"
  set activity to the value of the prediction named "Activity"
  set power to the value of the observation named "Power Status"
  return activity & " - " & power
  -- Returns something like "Writing – Wall"
end tell
```

Pennyworth implements the blackboard model by using the system's distributed notification center (illustrated in the previous section). By exposing two types of interfaces to other

applications, Pennyworth harnesses the strengths of Winograd's blackboard as well as Hong and Landay's (2001) infrastructure approach.

By default[9], the applications that may retrieve context information from Pennyworth are those running on the host system. This is a conscious design decision made to address concerns about the user's privacy and control over the contextual information (Ackerman, Darrell & Weitzner 2001). In its current incarnation, Pennyworth focuses on making user's desktop applications context-aware. This focus currently limits the extent that the system can be used to share context among devices in a ubiquitous computing environment.

In their discussion of privacy in relation to context-aware systems, Ackerman and colleagues identify four privacy requirements imposed by several regulatory regimes:

1. *Notice*: The individual should have clear notice of the type of information collected, its use, and an indication of third parties other than the original collector who will have access to the data.
2. *Choice*: The ability to choose not to have data collected.
3. *Access*: The ability for the data subject to see what personal information is held about him or her, to correct errors, and to delete the information if desired.
4. *Security*: Reasonable measure taken to secure (both technically and operational) the data from unauthorized access.

Pennyworth partially fulfills the notice requirement by providing a transparent view into the information collected and the inferences generated. Pennyworth can restrict access to context predictions by encrypting context change notifications. Users must provide a password to third party applications seeking to use for the protected updates[10]. Pennyworth fulfills the choice requirement in several ways: the user can decide to not run Pennyworth at all, he can disable sensors that collect overly sensitive information, and he can choose the level of detail that the system provides through a strategic choice of context labels. The system's transparency and user configuration features meet the access requirements, and Pennyworth relies upon the host system to secure any information collected from the user. That is, Pennyworth's data enjoys the same level of protection as the user's local e-mail messages, web browsing history, stored passwords, and other personal information.

Similarly, Bellotti and Edwards (2001) argue that context-aware systems must also support *intelligibility* and *accountability*. These requirements stem from the basic limitation that machines will never interpret context as well as humans. To address this issue, Bellotti and Edwards distill their intelligibility and accountability requirements into four major principles:

---

[9] The system may be configured to broadcast context information to applications not residing the on the current host system. This is currently an experimental feature not discussed in detail in this thesis.

[10] There are currently no similar protection mechanisms in place for information retrieved using the scripting interface. However, extending the password protection to those interfaces is a straightforward implementation.

1. *Inform* the user of current contextual system capabilities and understandings.
2. Provide *feedback.*
3. Enforce *identity and action disclosure* particularly with sharing nonpublic (restricted) information.
4. Provide *control* (and defer) to the user, over system and other user actions that impact him, especially in cases of conflict of interest.

Pennyworth fulfills the first requirement by making the sensing and generated models transparent and understandable. At any time, the user may review the data being collected and review how inferences are generated from that information. Similarly, the system provides feedback using those mechanisms as well as the panel that displays the system's real-time context predictions.

Pennyworth largely sidesteps the identity and action disclosure requirement by delegating that responsibility to the applications using context. However it does grant users the ability to limit which applications receive context information using the password protection mechanism. At any time, the user may revoke a suspicious application's context privileges by changing the password.

Finally, Pennyworth provides control through mechanisms such as the ability to pause the system, selectively enable specific sensors, override the machine learners using rules, view and modify action scripts, and manage context labels through the system's preferences. Furthermore, the user may correct the system at any time by using a system keyboard shortcut to invoke the training interface.

The delegation of the identity and action disclosure highlights the point that Pennyworth is only one component of a larger context-aware platform. In the next chapter, we discuss the other major component of the platform – applications.

Three context-aware applications

The prior chapter discussed Pennyworth from the perspective of an end-user and application developer. In real-world deployments, Pennyworth has proven itself as a usable and robust context inference engine. However, without any applications to apply that context, the system is little more than a novel tool that explores how a user's context relates to surrounding environment.

This chapter describes three applications that use Pennyworth to solve existing problems in real-world environments.

Context-aware notifications

As our lives become more interconnected and we remain available to others through electronic means, dealing with interrupts has become an increasingly taxing part of our electronic lives (Horvitz, Koch, & Apacible, 2004; Iqbal & Bailey, 2006). A context-aware system for managing notifications (Ho & Intille, 2005) provides a potential solution. They identify eleven factors that mediate this interaction. Among these factors, *activity of the user*, *utility of the message*, *modality of interruption*, and *social engagement of the user* are addressable using a context inference engine in conjunction with a centralized notification system.

On the Mac platform, Pennyworth can be combined with the open-source Growl system (Forsythe & Hosey 2008) to create a context-aware notification system. Growl is a centralized system that runs on the user's own machine. Applications submit notifications to Growl in order to alert users about events occurring within the system. Growl then displays these messages using presentation settings that the user previously selected. Growl allows users to standardize how interruptions are presented using variety of styles and configurations. Users benefit from using this system by replacing a myriad of application-specific notification schemes with a single consistent behavior and interface.

Growl was an ideal candidate for implementing a context-aware notification system because it enjoys wide adoption among developers and users on the Mac platform. This ubiquity allowed Pennyworth to improve an existing system already deployed by a significant number of users. Since Growl already exposes an open scripting interface, making it context-aware is a simple matter of constructing an action script that maps a predicted context to a specific configuration. A script that controls when an application may interrupt uses code similar to the following:

```
on prediction(type, prediction)
  if (type is "Activity") then
    tell application "GrowlHelperApp"
      if (prediction is "Socializing") then
        select style with name "Smoke" for application "Mail"
      else if (prediction is "E-Mail") then
        select style with name "Nano" for application "Mail"
      else
        select style with name "Null" for application "Mail"
      end if
```

```
    end tell
  end if
end prediction
```

In this example, the `Smoke` style is the most intrusive. The `Nano` style has a smaller footprint and unobtrusively slides into the user's display before sliding out of view a few seconds later. The `Null` style is one where notifications do not appear at all.

In this configuration, e-mail notifications are most obtrusive when the user is socializing. The rationale for this choice may be that the user is not engaged in a high-value activity, so more noticeable notifications are tolerable. When the user begins reading or writing e-mail, Pennyworth reconfigures the Growl system to use a less obtrusive style. The rationale behind this choice may be that since the e-mail application is already open, the need for highly visible alerts is no longer necessary. However, the user may not be looking at a list of messages, so some notification may still be useful. The final clause of the script instructs the system to turn off all mail notifications for other activities. In this case, e-mail notifications may be more of a nuisance than a benefit, thus are suppressed entirely when not socializing or using e-mail.[11]

Other users may have other priorities and the mapping between their contexts and notification preferences will differ from the script above. Pennyworth allows users to define these preferences using the scripting mechanism without *any further involvement needed from the software's developer*.

Smart homes and environments

There are fewer examples of flashy context-aware applications than the smart home. In this scenario, the environment monitors its inhabitants and adjusts itself accordingly. For example, when an inhabitant is present, the software activates devices in the environment. When the inhabitant leaves, it shuts down the devices to reduce power consumption and wear.

Using an approach similar to that of the notification system, Pennyworth can drive an environment equipped with computer-addressable devices and other components. A script that activates a lamp when the user is detected looks like this:

```
on prediction(type, prediction)
  if (type is "Activity") then
    tell application "Shion" -- Other apps may be used as well
      if (prediction is "Away") then
        deactivate device named "Overhead Light"
      else
        activate device named "Overhead Light"
      end if
    end tell
  end if
```

[11] These rationales are provided simply as an example that one user might use. In other cases, social activities may be less interruptible than solitary activities. The user may configure the system to match the rationales that they employ.

```
end prediction
```

When the system infers that the user is away, it turns off the overhead lamp. When it infers the user doing anything else, the light comes on.

The smart environment scenario is almost identical to the notification system, save for one important detail: the environment itself can become a source of information for Pennyworth. Using the technique described in the previous chapter where third-party applications broadcast sensor readings to Pennyworth, a home automation application can contribute the state of the devices to the context inference engine. If the user listens to a radio while cleaning the local area, the device's state may be the crucial clue that lets the system accurately infer the user's context when away from the computer. When several key devices are active in a household, the system may infer that the user is not alone.

Aggregating and sharing group context

The theoretical discussion above about the privacy implications of context-aware software highlights the default Pennyworth configuration of not sharing any contextual information beyond the local host system. While this default configuration is intended to protect the privacy of the user, there may be instances where the user wishes to breach this wall in order to achieve a greater good.

The group context service that we implemented consists of a centralized web service where instances of Pennyworth can post each users' context changes. In this scenario, several users run Pennyworth on their own computers and the local software uploads context changes to the online service using an action script.

This configuration may be useful to improve the communications and coordination of a workgroup. If all members' context is made available to each other, this may serve as the electronic equivalent of strolling past someone's cubicle to unobtrusively assess whether they are interruptible (Nichols, et al., 2002). In our lab, we have made this information visible in a variety of ways: Dashboard widgets, mobile phone interfaces, and Chumby widgets.

In the previous chapter, we discussed how Pennyworth sidesteps Bellotti's identity and action disclosure requirements by delegating that responsibility to the application using the context. The group context system is the kind of deployment where this delegation becomes important. In our testing of the group context system, we use reciprocity and social translucence (Erickson & Kellogg, 2000) as the governing design principles. In order to gain access to the group context information, a new member must share their context as well. This sharing allows others in the group to easily assess who is monitoring their context, as their updates are also visible.

Furthermore, users are encouraged to modify the action script that handles uploads to protect private information. For example, if a member doesn't want the context "visiting the doctor" visible to others, they can update the script to implement whitelists ("upload context when context is *Y* or *Z*") or blacklists ("upload context unless context is *Y* or *Z*") using simple combinations of conditional statements. Since the user can craft these scripts to match their

preferences, they can implement the optimal privacy policies that match their situation and preferences.

Other context-aware applications

These three examples are intended to showcase how Pennyworth enables context-aware applications in three very different situations. These examples are not intended to define the scope of applications that can work with Pennyworth. We have omitted discussing a variety of other applications that work well with Pennyworth (including context aware IM clients, media players, and security mechanisms) because using context to drive these applications is a straightforward exercise of the techniques discussed in this chapter.

Conclusion

> *After a busy day of work, our hero begins to pack up for the day. As he rises from his chair, the blinds to the outside open, revealing a sunny late afternoon. He looks outside, satisfied that he'll finally get around to taking that walk around the neighborhood. As he looks outside and ponders which route he'll take, his computer saves the state of his desktop and locks the workstation. He returns to packing his bag and heads out the door. As the door closes, the lights in the office dim to off, and the room patiently waits for our hero to return for another day.*

This thesis began by telling a short story and asking why, for the typical user, the story sounded like science fiction as opposed to something typical and unremarkable. We argue that all of the building blocks for constructing robust context-aware technologies are available now and the fundamental problem is the knowledge gap between researchers constructing context-aware prototypes and the developers who transform ideas into concrete technologies.

We present Pennyworth as a technology that bridges this gap. Its design and implementation advances the state of the art in research systems by taking a holistic approach to context-aware architectures that acknowledges the importance of three major classes of stakeholders: end-users, developers, and researchers. Pennyworth gives end users the mechanisms needed to begin immediately constructing a context-aware environment using existing software. It grants developers a flexible and powerful tool set for constructing novel applications that use context in interesting ways. For the researchers, its implementation addresses a variety of theoretical concerns that are often ignored by other systems, research and commercial alike.

We demonstrated the practical utility of the system by describing three distinct applications that each highlights a unique aspect of developing with the system. The context-aware notification system shows the power of using Pennyworth to actively manage the configuration of another application on the computer. The discussion of the home automation application built upon the dynamic configuration by additionally demonstrating how applications themselves contribute useful information to the learning system. The group context system showed how aggregating individual users' context using a web service might improve interactions within a team.

While there are still many technologies that still remain trapped in the pages of pulp magazines and paperback books, this thesis demonstrates that context-aware technology is not one of them. The only thing holding back widespread context-enabled technologies is the lack of availability of robust tools to construct them. Pennyworth solves that problem in a solid and approachable way.

References

Ackerman, M., Darrell, T., and Weitzner, D. (2001). Privacy in Context. *Human-Computer Interaction, 16(2)*, pp. 167-176.

Bellotti, V., and Edwards, K. (2001). Intelligibility and Accountability: Human Considerations in Context-Aware Systems. *Human-Computer Interaction, 16(2),* pp. 193-212.

Cooper, A. (2004). *The Inmates Are Running the Asylum*. Sams Publishing.

Dey, A., Abowd, G., and Wood, A. (1998). CyberDesk: a framework for providing self-integrating context-aware services. In *Proceedings of the 3rd International Conference on Intelligent User Interfaces (IUI '98)*, pp. 47-54. NY: ACM Press.

Erickson, T. and Kellogg, W. A. (2000). Social translucence: an approach to designing systems that support social processes. *ACM Transactions on Computer-Human Interaction (ToCHI), 7,* pp. 59-83.

Fogarty, J., Au, C., and Hudson, S. (2006). Sensing from the basement: a feasibility study of unobtrusive and low-cost home activity recognition. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology (UIST '06)*, pp. 91-100. NY: ACM Press.

Fogarty, J. and Hudson, S. (2007). Toolkit support for developing and deploying sensor-based statistical models of human situations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*, pp. 135-144. NY: ACM Press.

Fogarty, J., Hudson, S., and Lai, J. (2004). Examining the robustness of sensor-based statistical models of human interruptibility. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '04)*, pp. 207-214. NY: ACM Press.

Forsythe, C., and Hosey, P. (2008). *Growl*. Available online at http://www.growl.info/.

Ho, J., and Intille, S. (2005). Using context-aware computing to reduce the perceived burden of interruptions from mobile devices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '05)*, pp. 909-918. NY: ACM Press.

Hong, J. and Landay, J. (2001). An Infrastructure Approach to Context-Aware Computing. *Human-Computer Interaction, 16(2),* pp. 287-303.

Horvitz, E., Koch, P., and Apacible, J. (2004). BusyBody: creating and fielding personalized models of the cost of interruption. In *Proceedings of the 2004 ACM Conference on Computer Supported Cooperative Work (CSCW '04)*, pp. 507-510. NY: ACM Press.

Iqbal, S. T. and Bailey, B. P. (2006). Leveraging characteristics of task structure to predict the cost of interruption. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '06)*, pp. 741-750. NY: ACM Press.

Lederer, S., Hong, I., Dey, K., and Landay, A. (2004). Personal privacy through understanding and action: five pitfalls for designers. *Personal and Ubiquitous Computing, 8(6)*, pp. 440-454.

Munguia Tapia, E., Intille, S., Larson, K. (2004). Activity recognition in the home setting using simple and ubiquitous sensors. In *Proceedings of PERVASIVE 2004*, pp. 158-175.

Nichols, J., Wobbrock, J. O., Gergle, D., and Forlizzi, J. (2002). Mediator and medium: doors as interruption gateways and aesthetic displays. In *Proceedings of the 4th Conference on Designing interactive Systems: Processes, Practices, Methods, and Techniques (DIS '02)*, pp. 379-386. NY: ACM Press.

Quinlan, J. (1993). *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc.

Salber, D., Dey, A., and Abowd, G. (1999). The context toolkit: aiding the development of context-enabled applications. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '99)*, pp. 434-441. NY: ACM Press.

Schilit, B., Theimer, M. (1994). Disseminating active map information to mobile hosts. *IEEE Network, 8(5)*, pp. 22-32.

Symonds, D., (2006). *MarcoPolo: Context-aware computing for Mac OS X.* Available online at http://www.symonds.id.au/marcopolo/ .

Winograd, T. (2001). Architectures for Context. *Human-Computer Interaction, 16(2),* pp. 401-441.