PAMD: Developing a Plug-In Architecture for Palm OS-Powered Devices Using Software Engineering

Hernan Eguiluz
Venkat Govi
You Jung Kim
Adrian Sia

August 2002

Product Line Practice Initiative

Technical Note
CMU/SEI-2002-TN-020

Contents

Abstract ..... **vii**

1 Introduction **1**
1.1 About This Document 1

2 Current State of Plug-Ins in Industry **2**
2.1 General Concept of Plug-Ins 2
2.2 Plug-Ins for Mobile Devices 2
2.2.1 HackMaster Framework 3
2.2.2 MagicText—a HackMaster Framework Extension 4
2.2.3 DiddleBug Framework 4

3 Plug-In Architecture for Mobile Devices (PAMD) **5**
3.1 Scenario 5
3.2 PAMD Requirements 5
3.3 PAMD Architecture 6
3.3.1 PAMD Architecture Qualities 6
3.3.2 PAMD Components 7
3.3.3 PAMD System Architecture 8
3.3.4 Tradeoffs 10

4 PAMD Implementation **13**
4.1 Plug-In Manager Implementation 13
4.1.1 Communication Between the Plug-In Manager and an Application 13
4.1.2 Communication Between a Plug-In Manager and a Plug-In 13
4.1.3 Recovery Mechanism from Malicious Corruption to the Plug-In Registry 14
4.2 PAMD Plug-In Implementation 14
4.3 PAMD Application Implementation 15
4.4 Design Limitations 15
4.4.1 Single Input and Output Parameters 15
4.4.2 Validity of a Plug-In 16
4.4.3 Circular References 16
4.4.4 Platform Limitations 16

4.4.5 Single Service 16

5 A Programmer's Perspective of PAMD 17
5.1 PAMD Data Types 17
5.2 Abstract Data Types 18
5.3 Writing a PAMD Plug-In 18
5.4 Writing a PAMD-Aware Application 20

6 Use of Software Engineering Tools 22
6.1 Architecture Tradeoffs Analysis Method 22
6.2 Software Risk Evaluation 23
6.3 Software Process 24

7 Future Directions 26

8 Conclusions and Lessons Learned 27

Appendix A Plug-In Framework Code Structure 29

Appendix B Sample Plug-In Body 30

Appendix C Sample Application Code 32

Appendix D Acronyms and Terms 34

References 35

List of Figures

Figure 1: General Architectural View 9
Figure 2: Component Interaction Diagram 9

List of Tables

Table 1: Architectural Qualities 7
Table 2: Plug-In Launch Codes 13
Table 3: Plug-In Run Modes 15

Abstract

This technical note describes a plug-in architecture for Palm Operating System devices developed by the authors, a team of graduate students from Carnegie Mellon's Master of Software Engineering program. The note highlights the architecture's three most important aspects: the product (a plug-in architecture) created from a software architecture point of view; the implementation details that made this a unique project; and the software engineering facets of the project. This note also shares lessons learned and suggests possible avenues that could be pursued in the future to make the plug-in architecture for mobile devices (PAMD) more universal.

1 Introduction

As Portable Digital Assistants (PDAs) have limited memory and processing power, they can only run small applications with limited functionality. A common solution to this problem is to create pieces of functionality that can be shared by many applications—commonly called plug-ins. Currently, plug-ins implemented for the Palm Operating System (OS) are application dependent. As each application that implements a plug-in architecture does it in its own proprietary way, plug-ins are not interchangeable between applications. Another characteristic of Palm OS applications is that they function independently; very few of them share data. Those that do tend to be highly coupled.

To investigate developing a plug-in architecture, we participated in a project conducted by the Software Engineering Institute (SEI). At the beginning of this project, there was no simple way for two unrelated applications to exchange related data in a Palm OS device because most Palm OS applications are not aware of other applications and their capabilities.

Hence, there was a need to develop a nonproprietary application-level plug-in architecture for mobile devices (PAMD) for the Palm OS. This architecture had to allow applications to transparently extend their functionality beyond what was conceived by their designers. PAMD intends to bridge the communications gap between applications and plug-ins by providing a communications gateway. Using PAMD, applications and plug-ins can communicate with each other by passing data and control.

1.1 About This Document

This technical note targets a technically savvy audience. At the very least, the reader is expected to have a basic understanding of the PALM OS and software architecture.

This document describes this plug-in architecture development project from the initial concept of a plug-in to the architectural details of the proposed solution and how software developers interact with it. Our experience using software engineering tools is also documented. Finally, this document ends with a series of appendices containing sample code that a software developer can use when programming with PAMD.

2 Current State of Plug-Ins in Industry

To understand the derivation of software requirements and the resulting architecture associated with plug-ins, you should understand the workings of plug-ins and the limitations of some existing plug-in designs.

2.1 General Concept of Plug-Ins

A plug-in is a separate module of code that behaves like an extension to the application that invokes it. Plug-ins allow the features of an existing application to be extended beyond its original design. For example, plug-ins are frequently used in Web browsers to increase the flexibility of the browser by handling one or more data types (through Multipurpose Internet Mail Extensions [MIMEs]), thereby extending the capabilities to a wide range of interactive and multimedia capabilities. The details of the plug-in mechanisms may be different from each other due to the various platforms and semantics; nevertheless, the concept can be translated from desktop to mobile devices. A browser, such as Netscape Communicator, stores a list of plug-ins in a plug-in directory that acts like a registry. When the browser encounters data of a particular MIME data type, it searches its registry for a matching plug-in that can handle that data type. Once located, the plug-in is loaded into memory and initialized, and an instance of it is created. The life cycle of the plug-in is controlled by the browser that invokes it. When the browser window is closed, the plug-in instance is deleted, and the code is unloaded from memory.

One important point to note is that the plug-in is platform (browser) dependent, that is, a Netscape Communicator plug-in will not work on Internet Explorer because the mechanism (handshaking protocol) between the interacting modules is different. Hence, it is the responsibility of the user to download the appropriate plug-in. A plug-in that is compatible with the platform would be launched when its service is requested, whereas an incompatible plug-in would not.

2.2 Plug-Ins for Mobile Devices

Porting the plug-in conceptual model for resource-limited mobile devices seems to be an ideal way of extending the capabilities of applications. Although the plug-in concept has been applied to Palm OS devices by third-party application developers, there is no standardized framework for plug-in development. The framework refers to a specification where plug-ins can be reused by different applications even though both applications and plug-ins are developed independently.

The current approach for writing plug-ins is to develop the program as a code resource. This method has been used to develop large applications with over 64KB of code. However, Palm OS devices don't provide a standard service and data description to allow applications to discover plug-ins and make use of their services—thereby enabling a plug-in's reusability. Hence, the plug-ins created do not have the ability to be reused by different applications at the binary level.

However, using a similar plug-in concept and code resource mechanism, others have come up with their own frameworks (both proprietary and open standard) to support the extension of their applications with plug-ins. However, the research conducted for this project shows that in most of these frameworks, the plug-ins are tightly coupled to either the applications or the OS platform, greatly limiting their reusability. The following frameworks, described further below, were researched using information from public sources:

- HackMaster [DaggerWare]
- MagicText (a HackMaster framework extension) [Synergy Solutions Inc.]
- DiddleBug [SourceForge.net]

2.2.1 HackMaster Framework

HackMaster is a framework that manages *hack* extensions to the Palm OS system software. Hacks are similar in concept to a plug-in except that they use an OS mechanism. Specifically, hacks make use of the system trap vectors, a table pointing to system routines maintained by the OS, to alter a system routine with another non-system routine to provide the desired function when invoked. To change the behavior of the original system, hacks hook themselves into one of the many system-routine traps. In this approach, HackMaster provides a framework for managing hacks that is similar to that for plug-in models, so that hacks behave in a standard way and do not conflict with each other. To write hacks, programmers need to identify the appropriate system trap, so that it can be rerouted to itself. HackMaster takes care of patching and unpatching the traps.

HackMaster limits the portability and reliability of plug-ins because its platform needs to be tightly integrated with the Palm OS. The patching and unpatching of vectors is necessary to provide new functions and restore system functions. Moreover, traps are not guaranteed to remain static,[1] so there is no way to ensure that all versions of the Palm OS will support HackMaster. In addition, hacks can potentially cause conflicts among themselves that can lead to system instability. Also, the number of plug-ins that can be supported is limited by the number of available system traps. It is unclear whether multiple hacks can be chained together to achieve a particular goal.

[1] This is because traps are undocumented features of the Palm OS.

2.2.2 MagicText—a HackMaster Framework Extension

MagicText is a plug-in that can be extended by other plug-ins. It uses the HackMaster framework to provide text enhancement and management functions for applications. Its main limitation is that it is suitable only for text-entry enhancements—it is not a general-purpose plug-in architecture. Thus, it would not be easy to adapt the framework to accept various kinds of data types.

2.2.3 DiddleBug Framework

DiddleBug is a freeware graphical application for the Palm OS that provides a sketching capability. It comes with IntelliBooger™, an embedded extension or plug-in framework that allows the creation of entries in applications such as ToDo, DateBook, and other built-in Palm OS applications from within DiddleBug.

DiddleBug is an example of a tightly coupled plug-in model in which plug-ins are tightly integrated with Palm applications. DiddleBug uses IntelliBooger plug-ins as a bridge between itself and built-in Palm applications to allow graphical data to be sent to those applications. This is essentially a closed framework targeting only the built-in applications, although it may be possible to target other applications. This closed framework has limited the usability of plug-ins.

[TM] IntelliBooger is a trademark of Handspring.

3 Plug-In Architecture for Mobile Devices (PAMD)

PAMD is an architecture framework[2] for the exchange of data and control between applications. It details the architecture, syntax, and semantic behavior of applications to support application-level data exchange.

3.1 Scenario

Currently, there is no easy and standard way for applications to communicate inside a PDA running the Palm OS. The following end-to-end scenario explains how PAMD can help achieve this and assumes that PAMD-aware applications and the plug-ins used are already installed in the user's PDA.

1. A user launches the PAMD-aware Notepad application.
2. The user enters text in the Notepad application.
3. The user decides to email the text that he just entered in the Notepad application. Since that application has no email-sending capabilities, he decides to see if there are any plug-ins that can be used to send email.
4. The user selects the Plug-Ins menu item to locate the available plug-in services installed in the Palm device.
5. PAMD provides all the plug-ins available for the Notepad application to use. In this case, plug-ins such as email, uuencode, and print are displayed.
6. The user selects the email plug-in.
7. The text is sent from the Notepad application to the mail database.
8. The user switches to the Mail application, and in the Outbox, he finds an email containing the emailed text.

3.2 PAMD Requirements

The scenario described above was the original vision for the project and led the project team to identify the following requirements of PAMD:

- It must use nonproprietary plug-ins. Current Palm OS plug-in technology is application or content specific. Any application that complies with the PAMD framework will access

[2] *Framework* is a rather overloaded word in software. For our purposes, it is a supporting structure around which something can be built. In the context of this technical note, it refers specifically to the communication between applications and plug-ins.

PAMD plug-ins to extend its capabilities. From the other point of view, any plug-in that complies with PAMD shall be used by any PAMD-aware application.[3]

- It uses bi-directional communication between applications and plug-ins: Currently, plug-ins and applications, although loosely coupled, resemble a "pipe and filter" architectural style. Therefore, communication between them is one-way (with the exception of the operation's success or failure indicator). PAMD-aware applications shall be able to pass and receive complex information to and from plug-ins.
- Communication is allowed between plug-ins. Typically, communication is allowed only between applications and plug-ins; it is restricted between plug-ins. PAMD shall provide a communications mechanism between plug-ins that allows one plug-in to extend its functionality through the use of another.
- The hardware and software used must be compatible. There should be a reference implementation of PAMD that runs on devices that are powered by Palm OS V3.1 and higher. In particular, devices manufactured by Handspring, Sony Clie and Palm need to be supported.
- It should be at least on a par with other applications in terms of performance. PDA users are used to fast response times from applications, which imposes a need for PAMD to be lean and efficient. Possible memory limitations of 2MB make this goal more difficult to achieve.
- It must be programmer and user friendly. Otherwise the stakeholders may not adopt it, regardless of how good it is.

3.3 PAMD Architecture

3.3.1 PAMD Architecture Qualities

The following architectural qualities were deemed essential to satisfy the requirements outlined above. Each attribute is explained and accompanied by one or more metrics that were used to measure how well the quality was achieved in the final system. The following table is ordered by the importance of each quality, starting with the highest.

[3] Depending on the context, a *PAMD-aware application* could be either an application that is communicating via PAMD to execute a plug-in, or a plug-in that is communicating via PAMD to execute another plug-in.

Quality Attribute	Description	Metric
Programmer-friendly	The application programming interface (API) provided by PAMD should be easy to learn and use for both plug-in and application developers. Otherwise, a steep learning curve may discourage software developers from using PAMD.	• Days required to adapt an application to be PAMD aware • Days required to create a plug-in
Performance	PAMD has to provide its services to Palm users in a reasonable amount of time. If a PDA is sluggish when using PAMD, the users will stop using it.	• Seconds to retrieve a list of compatible plug-ins • Seconds to execute a plug-in
Usability	PAMD has to provide user-friendly interfaces for end users so that they can use PAMD plug-ins with their applications. It also needs to provide various features in the PAMD plug-in manager for maintaining PAMD plug-ins.	• User success rate[4] in using PAMD • Time required to complete a task • Error rate • Users' subjective satisfaction
Availability	PAMD will be available for service every time a PDA is turned on, because it is the only way in which applications can communicate with plug-ins.	• MTF[5] / (MTF / MTR[6])
Portability	PAMD, PAMD-aware applications, and PAMD plug-ins must work on Palm OS V3.1 and higher. More than one company offers products that utilize extended versions of the Palm OS. PAMD shall work on any of these, but PAMD shall not work on competing PDA operating systems like Windows CE.	• Number of Palm OS versions supported • Number of Palm OS PDA brands supported

Table 1: Architectural Qualities

3.3.2 PAMD Components

The PAMD architecture is composed of three entities: a PAMD plug-in manager, a PAMD-aware application, and a PAMD plug-in.

Plug-In Manager

The principal role of the plug-in manager is to coordinate communication between plug-ins and applications. When an application wants to execute a plug-in, it requests the plug-in

[4] Percentage of tasks that users complete correctly
[5] Mean time to failure
[6] Mean time to repair

manager to execute the plug-in on its behalf. At this point, the plug-in manager calls the specified plug-in and passes control to it. Once the plug-in executes, the plug-in manager regains control and then returns the result and control back to the calling application.

In addition to coordinating communication, the plug-in manager is responsible for maintaining the plug-in registry, where information about each installed plug-in, such as the data it can handle, is stored. The PAMD system uses that information to determine the state of availability of every registered plug-in. The type of information that can be processed by a given plug-in serves as selection criteria for matching plug-ins and applications.

PAMD Plug-In

The role of a PAMD plug-in is to provide a service to applications or other plug-ins. A plug-in doesn't store any information about which applications will use it and doesn't need to be aware of PAMD as long as it conforms to the PAMD specification. This achieves the desired separation between applications and plug-ins.

PAMD-Aware Application

PAMD-aware applications are aware of the availability of the plug-in manager; request and receive services from PAMD plug-ins; and can get services through the provided PAMD APIs and abstract data types (ADTs). PAMD-aware applications know about plug-ins vis-à-vis PAMD, and PAMD-aware application developers must not assume that a given plug-in is present. Therefore, the use of PAMD APIs and ADTs and the indirect access of plug-ins via the plug-in manager help satisfy the requirement for nonproprietary plug-ins.

3.3.3 PAMD System Architecture

PAMD-aware applications and PAMD plug-ins interact via the plug-in manager. When the application sends a request for services to the plug-in manager, the plug-in manager delegates the request to a plug-in. The assumption behind this process is that applications and plug-ins should know what kind of data types they exchange. Note that data exchange is done through direct access to the shared memory. To maintain up-to-date PAMD plug-in information, the plug-in manager queries the system database. Faster use of this information is accomplished through a private database called a plug-in registry.

For the sake of simplicity, the following architectural diagrams show only one PAMD-aware application and one plug-in acting as a service provider. Figure 1 describes the basic components and connectors in PAMD. Note that a plug-in can act both as a consumer and provider of services.

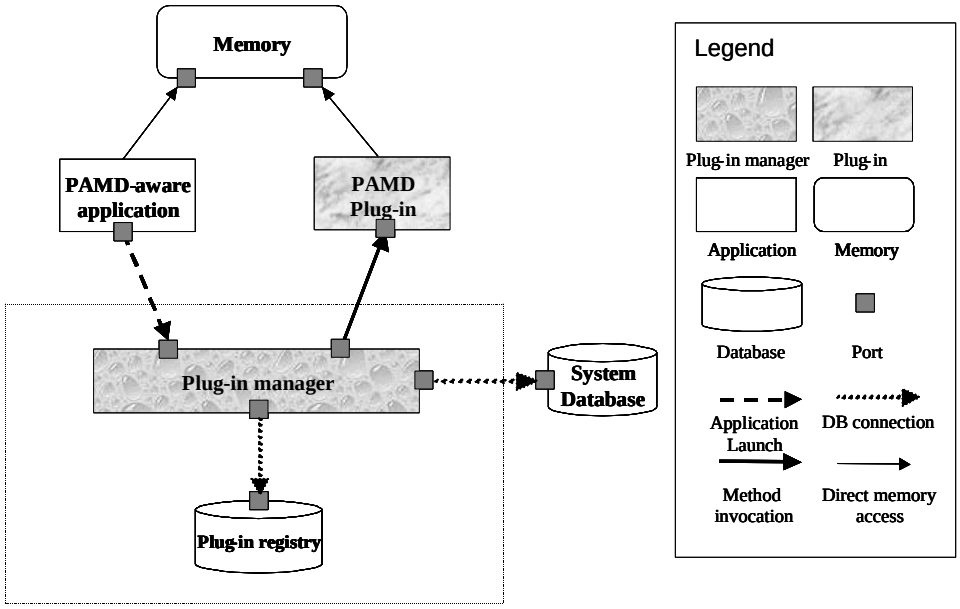

Figure 1: General Architectural View

Figure 2 shows how the components interact with each other when an application tries to execute a plug-in. The steps shown are described below.

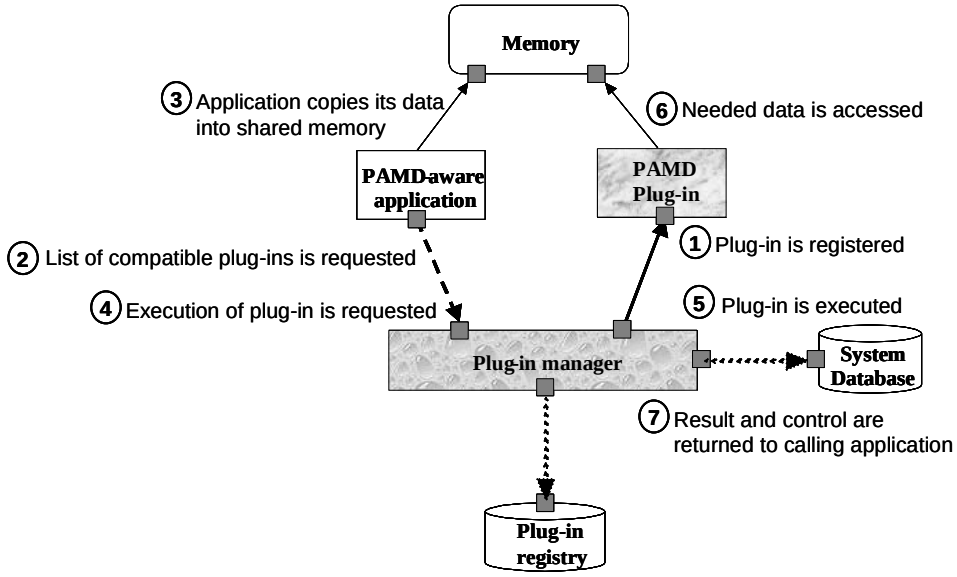

Figure 2: Component Interaction Diagram

1. The plug-in is registered.
   The plug-in manager searches the system database to detect a newly installed plug-in. If one is found, the plug-in manager registers it by asking it for its name and description, and the data types it supports.

2. A list of compatible plug-ins is requested.
   A PAMD-aware application requests a list of compatible plug-ins that can process the input and output data types handled by the application. Upon receiving this request, the plug-in manager searches the plug-in registry and returns the list of compatible plug-ins to the application.
3. The application copies its data into shared memory.
   This allows the plug-in to access required input data.
4. The execution of the plug-in is requested.
   A PAMD-aware application requests the plug-in manager that a specified plug-in be executed.
5. The plug-in is executed.
   The plug-in manager calls the plug-in to perform its service and yields the control thread to it.
6. Needed data is accessed.
   If a plug-in needs to access the input data during execution, it accesses it from shared memory. At the end of its execution, the plug-in can save its output data in shared memory.
7. After the plug-in finishes execution, control and the result are returned to the plug-in manager. The plug-in manger then returns the result and control to the calling application.

3.3.4 Tradeoffs

In developing the PAMD system architecture, the following three key architectural tradeoffs were made:

1. performance vs. availability
2. performance vs. security
3. architectural complexity vs. flexible inter-application communication

Each tradeoff is described below.

Performance Vs. Availability

This tradeoff involves two approaches that provide communication between an application and a plug-in. The first approach requires that an application and a plug-in communicate with each other via the plug-in manager. The plug-in manager invokes and runs a plug-in upon an application's request. The application cannot call a plug-in directly to extend its functionality. Because the PAMD architecture uses an indirect invocation method to call plug-ins, the performance associated with plug-in execution will degrade. However, since the plug-in manager coordinates the communication, it can provide a more secure runtime environment (e.g., an application will not be able to call plug-ins that are not in service).

In the second approach, an application can access a plug-in directly. In this architecture, the role of plug-in manager is restricted to maintaining a plug-in registry and passing compatible

plug-in information to an application. Therefore, the PAMD manager doesn't participate in the execution of a plug-in. Due to the direct invocation of plug-ins from an application, the performance regarding plug-in execution will be better than in the first approach, since there is no need to route plug-in invocations through the plug-in manager. However, in contrast with the previous approach, the appropriate execution of a plug-in cannot be guaranteed, because each application is responsible for handling illegal executions.

Considering the fact that the Palm OS isn't robust enough to handle the kind of fault outlined above gracefully (often leading to system crashes), the first approach can increase the availability of PAMD by ensuring that nonexistent plug-ins will not be executed.[7] These checks can be handled transparently as a service to all applications that request the execution of a plug-in. This also increases usability from the perspective of application developers because they will not need to care about the exception handling of these situations. Hence, the first approach was selected for PAMD.

Performance Vs. Security

This tradeoff involves caching plug-in information about installed plug-ins using two approaches. Which approach is used depends on whether a persistent database or volatile memory space is used.

The first approach is to use a database into which data collected from plug-ins is persisted. The use of this database reduces the workload of building the PAMD registry when the plug-in manager is opened. However, it exposes security problems, since the database can be deleted or edited easily without PAMD's knowledge, using Palm utilities. This data corruption may lead to a system crash, thus lowering system availability.

The second approach uses an internal data structure to save plug-in information, which prevents users from tampering with the registry information. In this case, the information needs to be rebuilt, requiring more processing time, especially when many plug-ins exist in the system. Since the number of plug-ins is the most critical factor in how quickly the list of plug-ins can be built, the system may not meet performance requirements when the number of plug-ins increases.

Assuming that an average user is unlikely to delete or corrupt the database, the first approach is chosen. When a database is deleted or corrupted, the plug-in manager can rebuild the registry, a process that is expensive, but unlikely to recur. And the performance gained by this approach outperforms the reduction in security and availability.

[7] Among other possible checks

Architectural Complexity Vs. Flexible Inter-Application Communication

This tradeoff involves applications acting as plug-ins. Two popular applications in the Palm OS are the phone book and email applications. One limitation of the email application is that its users need to type email addresses every time an email is composed—no contacts list facility is provided. Consider the scenario of a user who wants to send an email to a person in the phone book application. PAMD could allow the user to select the person from the phone book application, and then it could pass the email address and control transparently from the phone book to the email application.

Under the current architecture and without a plug-in, this scenario can't be achieved, since the current plug-in manager assumes that services can be provided only by a plug-in. The plug-in manager executes a plug-in as a part of its code under the same thread, not as a separate application. Therefore, it's impossible to change the phone book application, which acts as both an application and a plug-in within the existing PAMD.

To support this scenario, the definition of a PAMD plug-in needs to be extended to include the passive invocation of a code resource and standalone applications that can provide services. Although this extension provides more flexible application-level communications, it also makes PAMD plug-ins more complicated, thereby increasing implementation complexity and steepening the programmer's learning curve.

4 PAMD Implementation

This section concentrates on how the plug-in manager and the other components are implemented. It ends with a discussion of the limitations of the current implementation.

4.1 Plug-In Manager Implementation

4.1.1 Communication Between the Plug-In Manager and an Application

As a means of communication between Palm applications and plug-in managers, a Palm OS application launch mechanism is used. Typically, an application launches when it receives the launch code `SysAppLaunchCmdNormalLaunch` from the PALM OS. However, developers can also create user-defined launch codes that can be used as communications mechanisms between applications. As long as both sending and receiving applications know the user-defined launch codes and how they work, they can communicate using them.

In PAMD, two launch codes, `PAMD_CMD_GET_COMPATIBLE_PLUGINS` and `PAMD_CMD_RUN_PLUGIN`, are defined for communications between the plug-in manager and an application. An application sends those launch codes to the plug-in manager when it requests services, and the plug-in manager responds to those launch codes appropriately.

Launch Code	Description
`PAMD_CMD_GET_COMPATIBLE_PLUGINS`	Get a list of compatible plug-ins that support the data types handled by an application. The input and output data types need to be passed together.
`PAMD_CMD_RUN_PLUGIN`	Execute the plug-in of interest with parameter blocks. The plug-in manager executes the plug-in and returns the result and control to the calling application.

Table 2: Plug-In Launch Codes

4.1.2 Communication Between a Plug-In Manager and a Plug-In

A plug-in is a simple Palm code resource that the plug-in manager executes as a normal function call. However, since the code is external to the plug-in manager at compile time, the plug-in manager needs to load the plug-in's code into memory for the execution and then call the plug-in with the parameter block, holding the data passed from a calling application.

4.1.3 Recovery Mechanism from Malicious Corruption to the Plug-In Registry

As mentioned in Section 3.3.2, the current architecture uses a database to maintain a list of plug-in information called the PAMD registry. However, the database is visible to users, who can delete it. Doing so may lead to a system crash, thus lowering system availability.

Therefore, as a means to protect the registry from intended data corruption, the plug-in manager checks the "modification number" in the database, which is incremented every time a record in the database is added, modified, or deleted. Whenever the application updates the plug-in registry, it keeps the last modification number in the application preference, where a Palm application can save its state information. Then when it starts the update, the plug-in manager compares the current modification number of the plug-in registry and the modification number saved in the application preference. If those values are different, the plug-in manager assumes that other applications or users have modified the plug-in registry and starts the recovery process by retrieving information about the plug-ins in the system and updating the plug-in registry with the collected information.

4.2 PAMD Plug-In Implementation

The plug-in manager loads and executes the plug-in as a function call using a parameter block. Based on information stored in this parameter block, different sections of the plug-in's code will be executed. These separate sections correspond to modes in which a plug-in can execute. The mode is located in the parameter block and specifies how a plug-in should respond to the call from PAMD. Table 3 describes the modes currently used. PluginHelp and PluginConfig are optional modes in the sense that, although the functions they call for need to be present, their implementation could be empty. The plug-in may not respond to these modes but has to be able to handle them gracefully. The other three modes must be handled by the plug-in.

Mode	Description	Implementation
PluginRun	When in this mode, the plug-in provides its service. For example, the WordCounter plug-in should count the number of words in its input and store the result in the output.	Mandatory
PluginInfo	This part is executed when PAMD detects a newly installed plug-in and needs to gather information about it. The code to handle this mode returns basic information such as the input/output data type it will process later. No forms are to be displayed inside this block.	Mandatory
PluginCleanup	This mode is used to give a plug-in the opportunity to clean up after itself before being removed from the system.	Mandatory
PluginHelp	This mode is used to signal a plug-in about which an end user is asking for more information. Therefore, a plug-in displays copyright, help, and “about” information.	Optional
PluginConfig	In this mode, a plug-in shows its configuration form, if it has one.	Optional

Table 3: Plug-In Run Modes

4.3 PAMD Application Implementation

There are no restrictions on how an application should be implemented. An application only needs to use the PAMD API and ADTs to communicate with PAMD. See Section 5 for more information about these topics.

4.4 Design Limitations

In the current design, there are some limitations resulting from the data semantics used. Note that to address them, considerable overhead would have to be added to the system. As a consequence, the system’s usability and performance would have been reduced.

4.4.1 Single Input and Output Parameters

The use of single input and output parameters limits the granularity of the data. Even though a complex data type can always be used to encapsulate what would require many arguments, input and output information are limited to a single field. Handling data that contains multiple parameters may not be achieved easily. Further, heavy pointer de-referencing will be used, requiring more data manipulations that are error prone. Also, flexibility may be limited when more than one type of data needs to be processed by a plug-in.

4.4.2 Validity of a Plug-In

It is assumed that the plug-in's data type and service description describe it accurately. Even though there is a mechanism in place to check whether the plug-in has been tampered with since it was last accessed, that mechanism is limited to the operating environment. Currently, there are no checks on the validity of the plug-in to be installed. This could raise an issue if secure operations are desired, because a plug-in can be tampered with to implement other services, hence breaching security; or the data type can be tampered with to achieve similar goals, causing the system to crash.

4.4.3 Circular References

Circular references can be interpreted in two different ways in the context of PAMD. The first scenario refers to the use of one plug-in by another plug-in through PAMD. This scenario can be extended to more than two plug-ins. The second scenario refers to a plug-in calling itself recursively, which is similar to a recursive function call. In such a call, there is always the possibility that the recursion level will exceed the OS stack limit. In the case of the Palm OS, this situation will bring the system down. Such circular references would cause the system to be depleted of memory very fast and may also cause some data to be overwritten, crashing the system as a result. A possible solution would be to implement some checking mechanism within PAMD. However, doing so would add considerable overhead to running a plug-in. Therefore, handling this situation is now left up to the plug-in programmer.

4.4.4 Platform Limitations

The current architecture is suitable only for the Palm OS environment, even though it may not work on Palm OS V3 or lower due to the limitation of Palm APIs, or on Palm OS V5, since this version of the Palm OS is multithreaded. Further, PAMD is not designed to work on the Windows CE environment, as that was not a requirement. The dependency on the Palm OS API to remain portable across various Palm OS versions makes it impractical to address portability to an entirely different platform. Though it can be argued that the architecture is portable as long as similar functions can be provided through a layer of abstraction, much work is required on the API or kernel that limits PAMD's portability.

4.4.5 Single Service

To provide multiple services within a single plug-in, several factors must be considered:

- "Single input and output" limits the type of service that can be provided to a specific data type.
- Multiple services working on the same data types need to be differentiated.
- It is unclear how multiple services can be represented and managed, since plug-ins are currently designed as single-code resource databases that can be deleted or added at will.

5 A Programmer's Perspective of PAMD

This section concentrates on those aspects of PAMD that are going to be more interesting to software developers. This is just a fraction of the developer documentation that is available in the form of a programmer's manual.[8]

The main elements that compose a programmer's experience while working with PAMD are

- abstract data types, used to hide implementation details and provide a safer environment for software developers
- an API, provided for applications to interact with PAMD
- a plug-in framework, designed to simplify the creation of plug-ins

5.1 PAMD Data Types

PAMD data types act in a way similar to MIME types in a browser. Although the same concept is used, the types usually associated with browsers are not prescribed. This brings up a very important point: how an application or plug-in determines which data type to support, which data types exist, and what those types are like is outside the scope of PAMD.

From an implementation point of view, PAMD data types contain four bytes and can be any combination of printable characters except spaces.

Data types are used to determine compatibility between the data that an application needs to share and is expecting in return, and what a plug-in will take and produce. The following rules are followed to this end:

- The input and output data types specified by a plug-in need to match those specified by the plug-in caller.
- A plug-in that receives `BLOB` data types will match any input data type provided by the plug-in caller.[9]
- A plug-in caller that expects a return type of `BLOB` will match any output data type provided by a plug-in.[10]

[8] Eguiluz, H.; Govi, V.; Kim, Y. J.; & Sia, A. *Z Specification for PAMD.* To be published on the Web.

[9] Except for the data type `VOID`, which means that the application is not going to provide any input data

[10] `VOID` is an exception here too.

5.2 Abstract Data Types

The IEEE defines an abstract data type as

> "a data type for which only the properties of the data and the operations to be performed on the data are specified, without concern for how the data will be represented or how the operations will be implemented"[IEEE 99]

PAMD relies heavily on ADTs. This was a conscious decision to encapsulate data structures that are shared by applications, plug-ins, and PAMD. The interfaces to these ADTs should ease programming and provide a more secure environment for applications and plug-in developers.

Given the limitations imposed by the C programming language, it is impossible to prevent a programmer from directly accessing the information held in the ADTs. Yet we have strongly discouraged direct access in the programmer's manual on the basis that the ADTs keep track of internal information used to diagnose problems. The ADTs also provide consistency checks and a very detailed set of error messages that can give software developers valuable feedback when errors occur. The team believed that this last point was particularly important because of the added complexity involved in developing embedded applications.

ADTs have encapsulated three main data types:

1. PAMDPluginParamBlock, which encapsulates parameters passed between applications and plug-ins. These parameters primarily carry input and output data to and from plug-ins.
2. PAMDPluginInfo, which encapsulates the information that a plug-in gives to PAMD at registration time. This is a simple form of introspection.
3. PAMDPAMDPluginList, which represents a list of compatible plug-ins passed to an application by PAMD

Without these ADTs, the implementation effort would have been smaller. The team believed that the ADTs would make the implementation developer friendly and were therefore worth the added effort.

5.3 Writing a PAMD Plug-In

A plug-in is a code resource with a single point of entry through which PAMD passes control to it. A plug-in framework, provided as part of the PAMD developer's kit, takes care of routing the information received from PAMD to the appropriate user-defined functions. Each function maps to one of the modes described in Table 3 above. Therefore, a plug-in developer needs to provide appropriate implementation for these functions.

So, writing a plug-in becomes a matter of writing those five functions that are required by the framework.

Every plug-in follows a very simple set of steps when executed:

1. It extracts the input information provided by its caller (if such information is required).
2. It does any required processing.
3. It sets its result as the output information to be passed back to its caller (if required).

When a plug-in is asked to describe itself, it needs to pass the following information back to PAMD:

- its input data type, which corresponds to the type of data it is expecting from a caller
- its output data type, which corresponds to the type of data it will provide to its caller, most likely, based on the input that is passed to it
- its name, which corresponds to what will be displayed to the user on the screen. The name has to be short, yet descriptive enough to convey which service the plug-in provides.
- its description, which is displayed only on PAMD's control panel

Whether a plug-in is being created from scratch or as an adaptation to existing code, a few things need to be kept in mind:

- No global variables can be accessed from within a code resource.
- If string constants must be used, the compiler has to be instructed to make them PC-relative. Otherwise, at least with the CodeWarrior environment, they are made global.
- If the plug-in is potentially going to be called by another plug-in or by itself, it should be made reentrant, and no state should be kept from invocation to invocation (like a database).
- Plug-ins can have their own graphical user interfaces (GUIs) as normal applications.
- Plug-ins are not applications, so they cannot receive launch codes from the Palm OS.
- Plug-ins can send launch codes to applications.
- A plug-in that returns information to its caller must set the output parameter of the parameters structure that is passed to it. This must be done even when the plug-in's result is stored in the input data.
- If the plug-in being developed is algorithmic intensive and doesn't rely much on the Palm OS, developing and testing it in a PC-based development environment may shorten its development time.

For sample code that illustrates a complete plug-in implementation, see Appendices A, B, and C.

5.4 Writing a PAMD-Aware Application

In general, an application that is going to use PAMD to extend its functionality needs to follow these steps:

1. Determine which plug-ins are compatible with the data types that are passed and returned by the plug-in user.
2. Select a compatible plug-in to run.
3. Run the plug-in.
4. Get the plug-in's result.

Steps 1 and 3 make use of the PAMD API mentioned earlier. Each step is explained below. For sample code that illustrates these steps, see Appendices A, B, and C.

Step 1: Determine which plug-ins are compatible.

To generate a list of compatible plug-ins, the `PAMDGetCompatiblePlugins()` function needs to be called. This API function takes as an argument a plug-in list with specified input and output data types. PAMD uses these data types to determine plug-in compatibility.

Step 2: Select a compatible plug-in to run.

Once the list has been generated, the plug-in caller needs to select a plug-in for execution. The caller determines how the plug-in is selected—either by a user (through a GUI) or by the plug-in caller. In either case, the list of plug-ins that was returned by PAMD needs to be searched using the ADT that encapsulates plug-in lists.

Step 3: Run the plug-in.

To run a plug-in, PAMD requires the following information from the plug-in caller:

- the plug-in identification, which is found in the plug-in list generated in Step 1 and used by PAMD
- the data that the plug-in requires (if any). For plug-ins that expect data from their callers, this should be specified as the input argument to the plug-in.
- the input and output data types that the application believes the plug-in expects and will return. These data types are used by PAMD to double-check the plug-in invocation.

Once this information is collected in the form of a parameters structure, the plug-in caller can pass it to PAMD. Then PAMD, after making the necessary verifications, will execute the plug-in on behalf of the caller.

Step 4: Get the plug-in's result.

Once PAMD finishes executing a plug-in, it returns control to the caller. At this point, it is the caller's responsibility to extract the information, if any, that was returned by the plug-in. Any data that the plug-in returns to the application becomes the application's responsibility to return to the system.

6 Use of Software Engineering Tools

This section discusses the use of various software engineering tools in the development of this project. The following sections describe each tool and the team's experience using it.

6.1 Architecture Tradeoffs Analysis Method

The Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) is used to understand the tradeoffs inherent to the architecture of a software system by evaluating various conflicting quality attributes such as performance and modifiability [Kazman 98]. "The purpose of the ATAM is to assess the consequences of architectural decisions in light of quality attribute requirements and business goals. The ATAM process is a short, facilitated interaction between multiple stakeholders, leading to the identification of risks, sensitivities, and tradeoffs. The purpose of an ATAM evaluation is not to provide precise analyses; rather, it is to discover risks created by architectural decisions [Clements 01]. The ATAM helps to provide an in-depth understanding of the software system's design, while highlighting tradeoffs and potential risks that may violate desired qualities of the system. The analysis can also improve an already developed architecture and help align the participating stakeholders' understanding of it.

Specifically for this project, ATAM techniques were used to identify important qualities of the system and prioritize them using a utility tree. Various qualities and their sensitivity points were determined, leading to the identification of conflicting qualities that required design tradeoffs and analysis. Although the ATAM was not conceived for deriving an architecture, this did not prevent the team from using its analysis techniques to derive an improved architecture from an initial attempt. In fact, it was possible to derive an initial architectural pattern using the prioritized qualities derived from the ATAM's utility tree and known architectural styles [Shaw 96]. It is our experience that deriving the utility tree from an initial architecture, while keeping in mind business goals and functional and nonfunctional requirements, focuses the architecture team on producing an architecture that truly responds to stakeholders' needs. Lastly, it served as an information source for spreading architectural knowledge across the development team and for creating more comprehensive documentation.

For this project, the ATAM was used to probe the architectural design using various questions and scenarios. This allowed the project's various stakeholders to fully understand the extent

[SM] Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

and limit of the design. This is interesting due to the exploratory nature of the project. On first impression, the ATAM seems like a tool useful only for large software systems where the number of stakeholders is very large. However, our experience with small projects indicates that the ATAM is suitable for them too.

6.2 Software Risk Evaluation

According to the Department of Defense (DoD), software risk management is a proactive approach for minimizing the uncertainty and potential loss associated with a project. Its aim is to provide insights that support decision making through a continuous assessment of risks and opportunities, focusing on the day-to-day operational risks that a project may face.

The Software Risk Evaluation (SRE) is a method that enables the identification, analysis, tracking, mitigation, and communication of risks in a software-intensive project over its entire life cycle. The SRE is used to identify and categorize specific risks that result from a development project's management, resources, and constraints. The SRE is particularly useful in the early phases of a project for assessing risks from all stakeholders' (including developers and clients) perspectives. In doing so, misunderstandings related to requirements can be rectified early in the development process before they propagate and become costly mistakes. In addition, it helps improve the communication between clients and developers, since it highlights each one's concerns. Through constant monitoring, potential risks can be identified and mitigated, which helps in the management and control of a software project [SEI].

A condensed version of the SRE (mini-SRE) was conducted approximately one-third of the way into this project. Several different stakeholders assisted in this mini-SRE; of particular importance was the fact that the project's clients were present. From the risks that were identified during the mini-SRE, the following two are the most important:

- The team didn't clearly understand what the requirements for the project were. This prompted a reaction from the clients that led to the most serious risk at the time,
- The team had unknowingly complicated the requirements, which had almost stalled the development process.

These were major risks that could have caused the project to fail if they were not identified early in the development cycle. Once these and other risks were identified, the team settled on mitigation strategies for them. In particular, for the two risks mentioned above, the team reacted very quickly (within a week) by creating prototypes that confirmed the direction for the project and reestablished the clients' confidence in the team's ability to deliver a system that satisfied their vision.

We believe that the SRE techniques and the collaboration of our clients were invaluable in taking the team out of analysis paralysis.

6.3 Software Process

A software process defines the different stages that a software product goes through during its lifetime and provides the necessary guidelines, including management functions and roles, for carrying out each stage. The success of adopting any specific software-development life-cycle model depends on the type and scale of the software project, the organization's business goals and structure, and the people involved in the software project.

As the project was inherently exploratory and team members could work only part-time, the Team Software Process[SM] (TSP[SM]) [Humphrey 00] was tailored and evolved to meet the project's objectives. The TSP-prescribed techniques that were useful included the following:

- Divide the project into several cycles.
- Plan each cycle and track its progress using earned value.
- Assign team members specialized management roles to coordinate different areas of the development process while team members also handle engineering roles. This was especially beneficial during the early stages of the project because it helped organized the team.

Our tailoring of the TSP included the following:

- Add new roles to the development team, such as client liaison. Other roles, such as quality assurance and development managers, were not initially employed and were only added later when required.
- Remove TSP templates and scripts that would add significant overhead to the team members, as well as those that require a steep learning curve.
- Remove a conventional design phase from the project, given that the end product is relatively small. Instead, the team concentrated on specifying the interfaces between PAMD, applications, and plug-ins. Different team members developed these pieces individually while complying with agreed-upon interfaces. Although integration was not perfect, only six defects were found during this phase.

The team initially used questionnaires and use cases to elicit requirements, but found that they were insufficient to fully understand the project's requirements. As a consequence, end-to-end scenarios and a risk-reduction prototyping were used to complement the original techniques. By developing toy prototypes that focused on achieving a single goal, critical requirements were identified and understood [Wallnau 01]. As this was an exploratory project, it was sometimes difficult to find the right goals for the prototypes because the team was discovering requirements. Knowledge of just the technology or just the requirements doesn't help to resolve this kind of situation—knowledge of both is necessary. A better understanding of requirements improves our knowledge of the technology—what is and what is not possible—and this, in turn, helps to determine new, unnecessary, or refined requirements. Eventually, what is learned from these iterations becomes marginal, and the project's requirements become clear and stable.

[SM] Team Software Process and TSP are service marks of Carnegie Mellon University.

Given the time constraints experienced by the team, the development of prototypes was divided among various team members. Informal code reviews were used to share the information that each individual gained while developing prototypes, including knowledge about Palm OS technologies. Once requirements were clear, the team evolved to a more conventional process and implemented PAMD from scratch.

It is well known that a software process is a set of guidelines that needs to be tailored for each project. However, the team learned that tailoring is not limited to the beginning of a project, but rather can and sometimes needs to be (as in our case) done throughout the life of the project.

7 Future Directions

To make PAMD more universal and provide a larger benefit to its users, we think that the following avenues should be explored:

1. Plug-ins could be enhanced to provide multiple services instead of only one, as the current implementation allows. This could allow plug-in developers to aggregate closely related functionality in one plug-in.
2. PAMD could support communication between multiple applications rather than just between applications and plug-ins and between multiple plug-ins. This would allow the creation of meta-applications.
3. PAMD could support data-type compatibility beyond MIME matching by storing a plug-in-supplied list of data types with which a given service is compatible. So, if a service that converts strings to uppercase can receive both normal ASCII and UNICODE strings, this could be advertised by the plug-in and recorded by PAMD.
4. PAMD could support remote procedure calls if the Palm-OS-compatible device has a network connection of some sort.
5. PAMD could be extended to support the verification/authorization of plug-in use. This could help the creation of a plug-in “marketplace” where developers can license their plug-ins to PAMD users.
6. PAMD could support the association of an icon with a plug-in.
7. PAMD could allow users to attach a symbolic name to each service and then search the list of plug-ins using that name.
8. PAMD could be ported to Windows CE devices via an abstraction layer that provides mechanisms similar to those used by PAMD.

All of these proposed extensions are feasible.

8 Conclusions and Lessons Learned

The outcome of this project was a reference implementation of PAMD that runs on multiple versions of the Palm OS and on hardware provided by different vendors. It proved the feasibility of the idea of an open and generic application-level plug-in architecture for Palm OS-based devices.

This project demonstrated that its objective could be achieved using software engineering techniques, therefore disputing the general idea that an exploratory software project cannot be served by software engineering practices. The question should not be *if* software engineering techniques should be used for a project of this nature, but rather *which ones*.

It appears that nonfunctional requirements are as important as functional requirements, or perhaps even more so, due to the increasing requirements for security, performance, and portability. As with any software project, tradeoffs were made. However, contrary to usual practices, these tradeoffs were evaluated and made using software engineering techniques.

From a technical point of view, the team learned that, at the very least, the following topics need to be addressed when building a plug-in architecture:

- how to determine data type compatibility
- whether the roles of applications and plug-ins can or should be interchangeable
- how to describe a service provided by a plug-in
- how many services shall be provided by a single plug-in
- the security implications of the architecture
- mechanisms for selecting one plug-in from a set
- mechanisms for passing data between applications and plug-ins
- the amount of information that the system is going to store about each application and plug-in
- The cost of having an application use a plug-in should approach that of a function call. Although this is never really achievable, the architecture and design have to strive to get as close as possible to this ideal.
- The architecture needs to be both transparent and visible, depending on its users' situations and desires.
- The architecture needs to survive both malicious usage and usage lapses.

From a process point of view, the team learned that

- Simple, very focused throwaway prototypes can serve many purposes—they can teach a new technology, increase the engineer's confidence, and, when shared with the client, increase the client's confidence in the team.
- It doesn't always make sense to follow a very strict software process until requirements are well understood. A team should concentrate on identifying those requirements rather than on a software process.
- Software techniques that are believed to work only on large projects can help a small project too.
- The team confirmed that the testing process is greatly improved when engineers who don't have an investment in the code participate.
- Addressing nonfunctional requirements that are abstract and nontrivial is hard. Software architecture can be used to provide a template for reasoning about them. Such reasoning allows the design to be scaled in terms of complexity and size.
- ATAM was effective when reasoning about nonfunctional requirements and highlighted necessary tradeoffs, as well as risks. It also works as a way to spread architectural knowledge between team members.
- Formal methods, such as Z, can be used to reason about the requirements and testing of a design. Although such methods are usually of particular importance in the design of safety-critical systems, they can also be useful in less critical systems where accuracy is important. [11]

In conclusion, the use of software engineering techniques has made this project closer to an engineering practice than to an art. We say this because there is a well-defined methodology approach that can be followed and improved. Exploratory processes are an art in the sense that they can't be codified. Software engineering tools, such as toy prototypes, ATAM, the SRE, and formal methods, serve to bound such an exploratory project.

[11] Eguiluz, H.; Govi, V.; Kim, Y. J.; & Sia, A. *Z Specification for PAMD.* To be published on the Web.

Appendix A Plug-In Framework Code Structure

This appendix provides actual code that implements the plug-in framework. Although it is very simple, it is powerful in the sense that it frees plug-in developers from a good part of the complexity of implementation. Another advantage is that it makes plug-ins easier to understand.

```
PAMDErr PlugInMain( PAMDPluginParamBlockPtr parama
                    PAMDpluginInfoPtr pluginInfo )
{
   PAMDErr   pluginResult;

   switch ( params->mode )
   {
     case pluginRun:
        pluginResult = PluginRun(params);

     case pluginAbout:
        pluginResult = PluginAbout(void);
        break;

     case pluginConfig:
        pluginResult = PluginConfig(void);
        break;

     case pluginInfo:
        pluginResult = PluginInfo(pluginInfo );
        break;

     case pluginCleanup:
        pluginResult = PluginCleanup(void);
        break;

   }
   return pluginResult;
}
```

Appendix B Sample Plug-In Body

The following code shows the case of a plug-in transforming a string into uppercase. Error handling has been omitted for the sake of clarity.

```
PAMDErr PluginRun( PAMDPluginParamBlockPtr params)
{
   Uint32   theResult = 0;

   MemHandle   mhToText;
   Uint32   size;
   Char*    textToConvert;

   PAMDErr  e;
   int         I;

   //Output data.
   MemHandle   mhOutput;
   Char*    txtOutput;

   // Get hold of input data.
   e = PAMDGetInputArg(params, &mhToText, &size);
   textToConvert = MemHandleLock(mhToText);

   // Get and prepare output buffer.
   mhOutput = MemHandleNew(size * sizeof(Char));
   txtOutput = MemHandleLock(mhOutput);

   MemSet(txtOutput, size, 0);

   // Conver the string.
   for (i = 0; i < size; i++)
   {
      // Don't change anything that is not a letter.
      if ((textToConvert[i] < 97) || (textToConvert[i] > 122))
         txtOutput[i] = textToConvert[i];
      else
```

```
            txtOutput[i] = textToConvert[i]-- 32;
        }

        MemHandleUnlock(mhToText);
        MemHandleUnlock(mhOutput);

        e = PAMDSetOutputArg(params, mhOutput);
        if (e != PAMD_SUCCESS)
            return e;

        return PAMD_SUCCESS;
}
```

Appendix C Sample Application Code

This appendix introduces sample source code that follows the four steps described in Section 5.4.

```
#include “PAMDAPI.h”
#include “PAMDParamBlockADT.h”
#include “PAMDPluginInfoADT.h”
#include “PAMDPluginListADT.h”

// Variable definitions
PAMDErr        e;
PAMDPluginInfoPtr    pinfo;
PAMDPluginListPtr    plist;
PAMDPluginParamBlockPtr params;
MemHandle         inputData;
MemHandle         outputData;
uInt32         outputDataLength;

// STEP 1: Getting the list of compatible plug-ins
// Create a plug-in list and then populate it with the
// desired input and output types.
e = PAMDCreatePluginList(&plist);

e = PAMDSetInputDataType(plist,""TX"");
e = PAMDSetOutputDataType(plist,""TX"");

// Now get the list of plug-ins that are compatible.
e = PAMDGetCompatiblePlugins(plist);

// STEP 2: Selecting a plug-in to run
//…
// Somehow one plug-in gets selected.
//…

// STEP 3: Running the plug-in
// Create the parameters block needed to call the plug-in.
e = PAMDCreateParams(&params);
```

```
// Set the minimum required data to call the plug-in.
e = PAMDSetCreatorId(params, pinfo->creatorID);
e = PAMDSetInputDataType(params, pinfo->inputDataType);
e = PAMDSetOutputDataType(params, pinfo->outputDataType);
e = PAMDSetInputArg(params, inputData);

// Now run the plug-in.
e = PAMDRunPlugin(params);

// STEP 4: Getting the plug-in's result
// Once the plug-in returns, get its output.
e = PAMDGetOutputArg(params, &outputData,
                         &outputDataLength);

// …
// Use the data
// …

// STEP 5: Free any used resources
// Once the parameters and the list of plug-ins are no
// longer needed, free them.
e = PAMDDestroyPluginList(&plist);
e = PAMDDestroyParams(&params);
```

Appendix D Acronyms and Terms

ADT	Abstract Data Type
API	Application Programming Interface
ATAM	Architecture Tradeoff Analysis Method
DoD	Department of Defense
GUI	Graphical User Interface
MIME	Multipurpose Internet Mail Extensions
MSE	Master of Software Engineering
MTF	Mean Time to Failure
MTR	Mean Time to Repair
OS	Operating System
PAMD	Plug-in Architecture for Mobile Devices
PDA	Portable Digital Assistant
SEI	Software Engineering Institute
SRE	Software Risk Evaluation

References

[Clements 01] Clements, P.; Kazman, R.; & Klein, M. *Evaluating Software Architectures: Methods and Case Studies*. Boston, MA: Addison Wesley, 2001.

[DaggerWare] <http://www.daggerware.com/hackmstr.htm>

[Humphrey 00] Humphrey, Watts S. *Introduction to the Team Software Process*. Reading MA: Addison Wesley, 2000.

[IEEE 99] Institute of Electrical and Electronics Engineers. *IEEE Standards Software Engineering, Volume 1: Customer and Terminology Standards*. New York, NY: Institute of Electrical and Electronics Engineers, 1999.

[Kazman 98] Kazman, R.; Klein, M.; Barbacci, M.; Lipson, H.; Longstaff, T.; & Carrière, S. J. "The Architecture Tradeoff Analysis Method," 68-78. *Proceedings of the ICECCS*. Monterey, CA, August 10-14, 1998. Los Alamitos, CA: IEEE Computer Society, 1998.

[Palm 00] Palm, Inc. *Palm OS Programmer's API Reference*. Santa Clara, CA: Palm, Inc., 2000.

[Potter 96] Potter, B.; Sinclair, J.; & Till, D. *An Introduction to Formal Specification and Z. Second Edition*. Harlow, Essex, England: Prentice Hall Europe, 1996.

[SEI] Software Engineering Institute. Software Risk Evaluation Service. <http://www.sei.cmu.edu/products/services /sw.risk.eval-service.html>.

[Shaw 96] Shaw, M. & Garlan, D. *Software Architectures Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Prentice Hall, 1996.

[SourceForge.net] <http://sourceforge.net/projects/diddlebug>

[Synergy Solutions Inc.] <http://www.synsolutions.com/software/magictext/>

[Wallnau 01] Wallnau, K.; Hissam, S.; & Seacord, R. *Building Systems from Commercial Components*. Boston, MA: Addison Wesley, 2001.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2002	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
PAMD: Developing a Plug-In Architecture for Palm OS-Powered Devices Using Software Engineering	F19628-00-C-0003

6. AUTHOR(S)

Hernan Eguiluz, Venkat Govi, You Jung Kim, Adrian Sia

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2002-TN-020

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

This technical note describes a plug-in architecture for Palm Operating System devices developed by the authors, a team of graduate students from Carnegie Mellon's Master of Software Engineering program. The note highlights the architecture's three most important aspects: the product (a plug-in architecture) created from a software architecture point of view; the implementation details that made this a unique project; and the software engineering facets of the project. This note also shares lessons learned and suggests possible avenues that could be pursued in the future to make plug-in architecture for mobile devices (PAMD) more universal.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Palm OS, plug-in, software architecture, Architecture Tradeoff Analysis Method, ATAM, Software Risk Evaluation, SRE	46

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL