CHAPTER 1

Introduction to Mobile Computing

> Where is the life we have lost in living? Where is the knowledge we have lost in information? Where is the wisdom we have lost in knowledge?
>
> T. S. Elliot

1.1 INTRODUCTION

Mobile computing systems are computing systems that may be easily moved physically and whose computing capabilities may be used while they are being moved. Examples are laptops, personal digital assistants (PDAs), and mobile phones. By distinguishing mobile computing systems from other computing systems we can identify the distinctions in the tasks that they are designed to perform, the way that they are designed, and the way in which they are operated. There are many things that a mobile computing system can do that a stationary computing system cannot do; these added functionalities are the reason for separately characterizing mobile computing systems.

Among the distinguishing aspects of mobile computing systems are their prevalent wireless network connectivity, their small size, the mobile nature of their use, their power sources, and their functionalities that are particularly suited to the mobile user. Because of these features, mobile computing applications are inherently different than applications written for use on stationary computing systems. And, this brings me to the central motivation behind authoring this book.

The application development and software engineering disciplines are very young engineering disciplines compared to those such as structural, mechanical, and electrical engineering. Software design and implementation, for the most part,

Copyright 2005. Cambridge University Press.
All rights reserved. May not be reproduced in any form without permission from the publisher, except fair uses permitted under U.S. or applicable copyright law.

EBSCO Publishing : eBook Collection (EBSCOhost) - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE
AN: 181811 ; B'Far, Reza.; Mobile Computing Principles : Designing and Developing Mobile Applications with UML and XML
Account: ns196568.main.ebooks

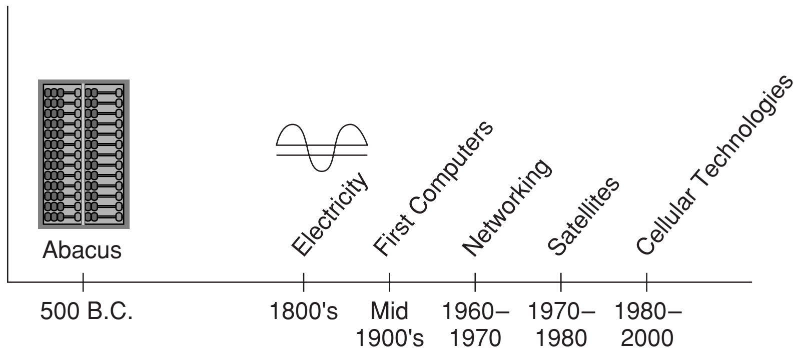

FIGURE 1.1. A Timeline of Mobile Computing.

remain part art and part science. However, there are definite signs of maturation with the development of architectures, metrics, proven tools, and other methodologies that give an engineering discipline its structure. Whereas there are a variety of methodologies, techniques, frameworks, and tools that are used in developing software for stationary systems, there are very few for mobile systems. Although mobile computing systems have existed as long as their stationary counterparts, most of the mature tools, methodologies, and architectures in software engineering today address the needs of stationary systems. One of our goals in this book will be to reflect on the research being done today to help evolve mobile application development and to outline some of the early proven techniques and technologies being tried in the commercial and academic environments.

In this text, we will look at those things that make the functional nature of mobile applications different than their stationary counterparts, take a survey of various development techniques that can be used to address these differences, and look at various basic technologies that allow us, as software developers, to create meaningful mobile applications in an extensible, flexible, and scalable manner.

1.1.1 A Brief History of Mobile Computing

Figure 1.1 shows a timeline of mobile computing development. One of the very first computing machines, the abacus, which was used as far back as 500 B.C., was, in effect, a mobile computing system because of its small size and portability. As technology progressed, the abacus evolved into the modern calculator. Most calculators today are made with an entire slew of mathematical functions while retaining their small size and portability. The abacus and calculators became important parts of technology not only because of their ability to compute but also because of their ease of use and portability. You can calculate the proceeds of a financial transaction anywhere as long as you had an abacus in 500 B.C. or have a calculator today. But, calculating numbers is only one part of computing.

Other aspects of computing, namely storage and interchange of information, do not date as far back as the abacus. Though writing has always been a way of storing information, we can hardly call a notebook a computing storage mechanism. The first mobile storage systems can be traced back only as far as the advent of the age of electronics.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

Figure 1.2. Wireless Communication Systems.

A mobile computing system, as with any other type of computing system, can be connected to a network. Connectivity to the network, however, is not a prerequisite for being a mobile computing system. Dating from the late 1960s, networking allowed computers to talk to each other. Networking two or more computers together requires some medium that allows the signals to be exchanged among them. This was typically achieved through wired networks. Although wired networks remain the predominant method of connecting computers together, they are somewhat cumbersome for connecting mobile computing devices. Not only would network ports with always-available network connectivity have to be pervasive in a variety of physical locations, it would also not be possible to be connected to the network in real time if the device were moving. Therefore, providing connectivity through a wired system is virtually cost prohibitive. This is where wireless communication systems come to the rescue (Figure 1.2).

By the 1960s, the military had been using various forms of wireless communications for years. Not only were wireless technologies used in a variety of voice communication systems, but the aviation and the space program had created great advances in wireless communication as well. First, the military developed wireless communication through line of sight: If there were no obstacles between point A and point B, you could send and receive electromagnetic waves. Then

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

came techniques that allowed for wireless communication to encompass larger areas, such as using the atmosphere as a reflective mechanism. But, there were limitations on how far a signal could reach and there were many problems with reliability and quality of transmission and reception.

By the 1970s, communication satellites began to be commercialized. With the new communication satellites, the quality of service and reliability improved enormously. Still, satellites are expensive to build, launch, and maintain. So the available bandwidth provided by a series of satellites was limited. In the 1980s cellular telephony technologies became commercially viable and the 1990s were witness to advances in cellular technologies that made wireless data communication financially feasible in a pervasive way.

Today, there are a plethora of wireless technologies that allow reliable communication at relatively high bandwidths. Of course, bandwidth, reliability, and all other qualitative and quantitative aspects of measuring wireless technologies are relative to time and people's expectations (as seems to be with everything else in life!). Though most wireless networks today can transmit data at orders of magnitude faster speeds than just ten years ago, they are sure to seem archaically slow soon. It should, however, be noted that wired communication systems will almost certainly always offer us better reliability and higher data transmission bandwidths as long as electromagnetic communications is the primary means of data communications. The higher frequency sections of the electromagnetic spectrum are difficult to use for wireless communications because of natural noise, difficulty of directing the signal (and therefore high losses), and many other physical limitations. Since, by Nyquist's principle [Lathi 1989], the bandwidth made available by any communication system is bound by the frequencies used in carrying the signal, we can see that lack of availability of higher frequency ranges places a limitation on wireless communication systems that wired communication systems (such as fiber optic–based systems) do not have to contend with.

Because the greatest advances in mobile communications originated in the military, it is no surprise that one of the first applications of wireless communication for mobile computing systems was in displaying terrain maps of the battlefield. From this, the global positioning system (GPS) evolved so that soldiers could know their locations at any given time. Portable military computers were provided to provide calculations, graphics, and other data in the field of battle. In recent years, wireless telephony has become the major provider of a revenue stream that is being invested into improving the infrastructure to support higher bandwidth data communications.

1.1.2 Is Wireless Mobile or Is Mobile Wireless?

In wireless connectivity, mobile computing devices found a great way to connect with other devices on the network. In fact, this has been a great source of confusion between *wireless communications* and *mobile computing*. Mobile computing devices need not be wireless. Laptop computers, calculators, electronic watches, and many other devices are all mobile computing devices. None of them use any sort of wireless communication means to connect to a network. Even some hand-held personal assistants can only be synchronized with personal computers through

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

a docking port and do not have any means of wireless connectivity. So, before we embark on our journey in learning about mobile computing, it should be clear that *wireless communication systems are a type of communication system. What distinguishes a wireless communication system from others is that the communication channel is space itself*. There are a variety of physical waveguide channels such as fiber optics or metallic wires. Wireless communication systems do not use a waveguide to guide along the electromagnetic signal from the sender to the receiver. They rely on the mere fact that electromagnetic waves can travel through space if there are no obstacles that block them. Wireless communication systems are often used in mobile computing systems to facilitate network connectivity, but they are not mobile computing systems.

Recently, computer networks have evolved by leaps and bounds. These networks have begun to fundamentally change the way we live. Today, it is difficult to imagine computing without network connectivity. Networking and distributed computing are two of the largest segments that are the focus of current efforts in computing. Networks and computing devices are becoming increasingly blended together. Most mobile computing systems today, through wired or wireless connections, can connect to the network. Because of the nature of mobile computing systems, network connectivity of mobile systems is increasingly through wireless communication systems rather than wired ones. And this is quickly becoming somewhat of a nonmandatory distinguishing element between mobile and stationary systems. Though it is not a requirement for a mobile system to be wireless, most mobile systems are wireless. Nevertheless, let us emphasize that wireless connectivity and mobility are orthogonal in nature though they may be complementary. For example, we can have a PDA that has no wireless network connectivity; however, most PDAs are evolving into having some sort of wireless connectivity to the network.

Also, though it is important to understand that stationary and mobile computing systems are inherently different, this does not mean that they do not have any commonalities. We will build on existing software technologies and techniques used for stationary systems where these commonalities exist or where there is a logical extension of a stationary technique or technology that will mobilize it.

Because of the constant comparison between mobile systems and other types of systems, we will have to have a way to refer to the "other" types of systems. We will use the terms *nonmobile* and *stationary* interchangeably. Although mobile is an industry-wide accepted terminology to distinguish a group of systems with the characteristics that we have just mentioned, there is no consensus on a system that is not a mobile system. For this reason, we will simply use the terms stationary or nonmobile when speaking of such systems. It is also important to note the there is probably no system that is truly not mobile because just about any system may be moved. We will assume that cranes, trucks, or other large vehicles are not required for moving our mobile systems! A mobile system should be movable very easily by just one person.

There are four pieces to the mobile problem: the mobile user, the mobile device, the mobile application, and the mobile network. We will distinguish the mobile user from the stationary user by what we will call the *mobile condition:*

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

the set of properties that distinguishes the mobile user from the user of a typical, stationary computing system. We will wrap the differences between typical devices, applications, and networks with mobile devices, applications, and networks into a set of properties that we will call the *dimensions of mobility*: *the set of properties that distinguishes the mobile computing system from the stationary computing system*. Once we have some understanding of the mobile problem, we will look at some established nonproprietary methodologies and tools of the software industry trade such as Unified Modeling Language (UML) as well as some commercial proprietary tools such as Sun Microsystem's Java, Microsoft's Windows CE, Symbian, and Qualcomm's BREW. Once we have looked at these tools, we will set out to solve the problem of architecting, designing, and implementing solutions for mobile computing problems.

Let us start by looking at some of those variables that create a distinction between mobile and stationary computing systems.

1.2 Added Dimensions of Mobile Computing

It should be obvious that any mobile computing system can also be stationary! If we stop moving it, it is stationary. So, we can say that mobile computing systems are a superset of stationary computing systems. Therefore, we need to look at those elements that are outside of the stationary computing subset. These added dimensions will help us pick out variables that in turn allow us to divide and conquer the problems of mobile computing. The *dimensions of mobility*, as we will refer to them in this text, will be the tools that allow us to qualify our problem of building mobile software applications and mobile computing systems. Although these dimensions of mobility are not completely orthogonal with respect to each other, they are separate enough in nature that we can distinguish them and approximate them as orthogonal variables. Also, keep in mind that some of these dimensions are limitations; nevertheless, they are still added dimensions that need not be considered when dealing with the typical stationary application. These dimensions of mobility (Figure 1.3) are as follows:

1. location awareness,
2. network connectivity quality of service (QOS),
3. limited device capabilities (particularly storage and CPU),
4. limited power supply,
5. support for a wide variety of user interfaces,
6. platform proliferation, and
7. active transactions.

It is absolutely crucial that the reader understands these dimensions of mobility and keeps them in mind throughout the process of design and implementation of the mobile application. Too often, engineers begin with attention to design and get bogged down in details of the tools that they use and small focused problems within the bigger picture of the system, its design, and its architecture. The definition of the word "mobile" reveals the first dimension we will consider: location.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

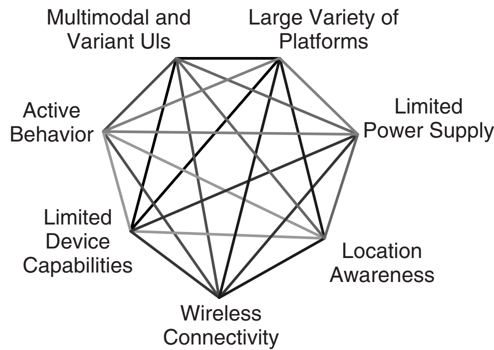

FIGURE 1.3. Dimensions of Mobility.

1.2.1 Location

A mobile device is not always at the same place: Its location is constantly changing. The changing location of the mobile device and the mobile application presents the designers of the device and software applications with great difficulties. However, it also presents us with an opportunity of using the location and the change in location to enhance the application. These challenges and opportunities can be divided into two general categories: *localization* and *location sensitivity*.

Localization is the mere ability of the architecture of the mobile application to accommodate logic that allows the selection of different business logic, level of work flow, and interfaces based on a given set of location information commonly referred to as locales. Localization is not exclusive to mobile applications but takes a much more prominent role in mobile applications. Localization is often required in stationary applications where users at different geographical locations access a centralized system. For example, some point-of-sale (POS) systems and e-commerce Web sites are able to take into account the different taxation rules depending on the locale of the sale and the location of the purchase. Whereas localization is something that stationary applications can have, location sensitivity is something fairly exclusive to mobile applications.

Location sensitivity is the ability of the device and the software application to first obtain location information while being used and then to take advantage of this location information in offering features and functionality. Location sensitivity may include more than just the absolute location of the device (if there is such a thing as absolute location—Einstein must be rolling in his grave now!). It may also include the location of the device relative to some starting point or a fixed point, some history of past locations, and a variety of calculated values that may be found from the location and the time such as speed and acceleration.

There are a variety of methods for collecting and using the location of the user and the device. The user may simply be prompted for his or her location, but this wouldn't make a very user-friendly application. Imagine a system that can only give you directions to where you want to go if you know where you are: It will be useful often, but occasionally, you won't know where you are or it would be too difficult to figure out your location. The device may be reset for a relative location if it has the ability to sense motion and can keep track of the change of location

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

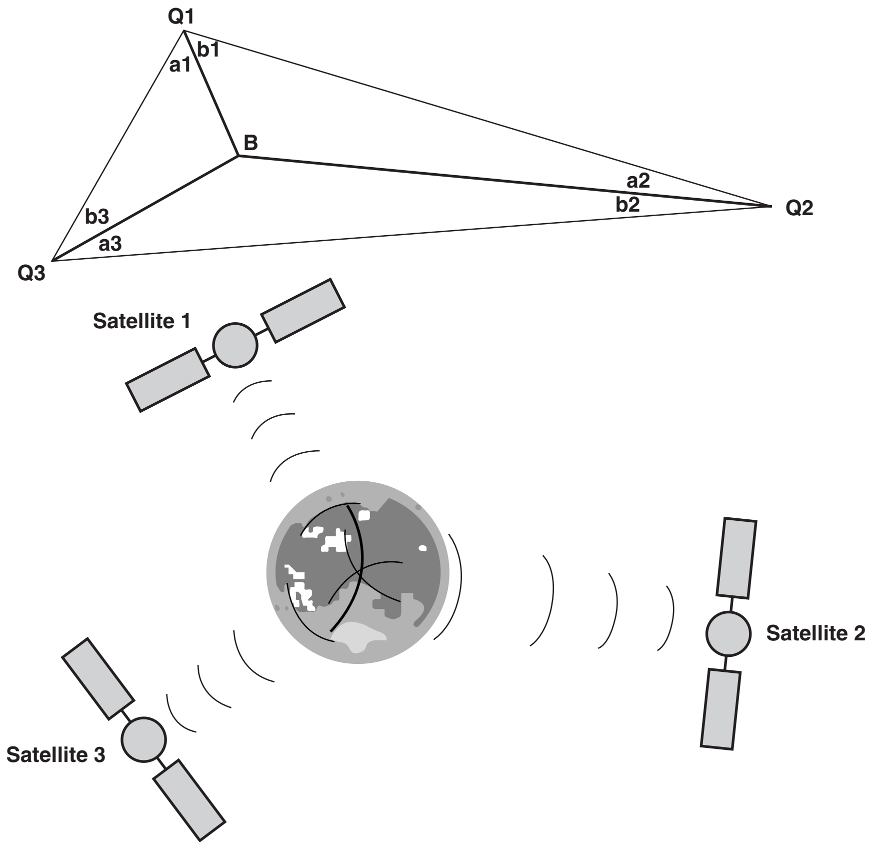

FIGURE 1.4. Determining Position Based on Triangulation.

for some period of time after this reset. Most location-sensing technologies (the particulars of which will be discussed in Chapter 12) use one or more of three categories of techniques: *triangulation*, *proximity*, and *scene analysis* [Hightower and Borriello 2001].

Triangulation (Figure 1.4) relies on age-old geometric methods that allow calculation of the location of a point that lies in the middle of three other points whose exact locations are known. If the distance to each one of the three points is known, we can use geometric techniques to calculate the exact location of the unknown point. *Proximity*-based methods measure the relative position of the unknown point to some known point. *Scene analysis* relies on image processing and topographical techniques to calculate the location of the unknown point based on a view of the unknown point from a known point.

The most well known location sensing system today is GPS. GPS-enabled devices can obtain latitude and longitude with accuracy of about 1–5 m. GPS has its roots in the military; until recently, the military placed restrictions on the accuracy of GPS available for public use. Most of these restrictions have now been lifted. GPS devices use triangulation techniques by triangulating data points from the satellite constellation that covers the entire surface of the earth. If a device does not have GPS capabilities but uses a cellular network for wireless connectivity,

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

signal strength and triangulation or other methods can be used to come up with some approximate location information, depending on the cellular network.

Regardless of how location information is obtained, it is one of the major differences between mobile and stationary systems. Location information can be to mobile applications what depth can be to two-dimensional pictures; it can give us an entirely new tool to automate tasks. An example of a stand-alone mobile software application that uses location information could be one that keeps track of the route that a user drives from home to work every day without the user entering the route manually; this could then be used to tell the user which route is the fastest way to get to work on a particular day or which route may result in the least amount of gas consumed. An example of a wirelessly networked mobile application taking advantage of location could be one that shows a field service worker where to go next, once he or she is finished with a task at one site, based on the requests for work in the queue and the location of the field service worker. It should be noted that acquiring position information requires connectivity to some network-based infrastructure. This infrastructure is typically isolated from the other network-based application infrastructures. Therefore, when we say stand-alone, we mean an application that may use some specific network-based infrastructure, such as GPS, for obtaining location information but is not connected to any other networks as a part of a distributed or network-based application.

Location information promises to be one of the biggest drivers of mobile applications as it allows for the introduction of new business models and fundamentally new methods of adding productivity to business systems.

1.2.2 Quality of Service

Whether wired or wireless connectivity is used, mobility means loss of network connectivity reliability. Moving from one physical location to another creates physical barriers that nearly guarantee some disconnected time from the network. If a mobile application is used on a wired mobile system, the mobile system must be disconnected between the times when it is connected to the wired docking ports to be moved. Of course, it is always a question whether a docking port is available when required let alone the quality and type of the available network connectivity at that docking port. In the case of wireless network connectivity, physical conditions can significantly affect the quality of service (QOS). For example, bad weather, solar flares, and a variety of other climate-related conditions can negatively affect the (QOS). This unreliability in network connectivity has given rise to the QOS field and has led to a slew of accompanying products. QOS tools and products are typically used to quantify and qualify the reliability, or unreliability, of the connectivity to the network and are mostly used by network operators. Network operators control the physical layer of the network and provide the facilities, such as Internet Protocol (IP), for software application connectivity.

Usually, the QOS tools, run by the network operators, provide information such as available bandwidth, risk of connectivity loss, and statistical measurements that allow software applications to make smart computing decisions. The key to designing and implementing mobile applications is that network connectivity and QOS need to be taken into account with an expanded scope. Most software

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

applications, mobile or not, take advantage of networking in some way and, therefore, do have network connectivity features. Stationary applications typically need not worry about the quality of network connectivity as this is handled by lower level layers than the application: the operating system, the hardware (such as the network card in a personal computer), the network itself, and all of the other components that make network computing possible. Stationary software applications typically assume some discrete modes of connectivity mostly limited to connected or disconnected. This works for most applications because most wired network connectivity is fairly reliable.

However, the effect of QOS in designing mobile applications is much more profound. Whereas typical nonmobile applications need to know how to stop operating "gracefully" when suddenly disconnected from the network, mobile applications have to know how to continue to operate even after they are disconnected from the network or while they connect and disconnect from the network intermittently and frequently. For example, let us take the case of a user who is traveling on a train, is using an application on his PDA connected wirelessly to some network, and is downloading a work-related report to look over when the train passes through a tunnel and he loses network connectivity. If the application does not have the ability to stop partway through the download process and restart when connectivity is restored, the user may never be able to retrieve the desired file as he passes through one tunnel after the other and the download process starts over and over again. The application, therefore, must know how to deal with lack of reliable connectivity.

When it comes to taking into account the QOS in most applications, certain functionality is expected of most mobile applications. For example, almost all mobile applications should know how to stop working when the application suddenly disconnects from the network and then resume working when it connects again. Other functionality may be desired but not required. For example, often QOS data are measured and provided by the network operator. For example, the real-time bandwidth available may be part of the data provided and refreshed on some time interval. Such data can be utilized to design applications that dynamically adapt their features and functionality to the available bandwidth.

1.2.3 Limited Device Storage and CPU

No one wants to carry around a large device, so most useful mobile devices are small. This physical size limitation imposes boundaries on volatile storage, nonvolatile storage, and CPU on mobile devices. Though solid-state engineers are working on putting more and more processing power and storage into smaller and smaller physical volumes, nevertheless, as most mobile applications today are very rudimentary, there will be more and more that we will want to do with them. Today's mobile applications are resource-starved. So, although the designers of modern applications designed to run on personal computers (PCs) and servers continue to care less and less about system resources such as memory and processing power, it is a sure bet that memory limitations will be around for a long

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

time for mobile applications because *when it comes to mobile systems and devices, smaller is nearly always better*.

Smaller devices are easier to carry and, consequently, may become more pervasive. This pervasiveness also largely depends on the price of the devices. Making electronic devices very small normally increases the cost, as the research and development that go into making devices smaller are very expensive. But, once a technology matures and the manufacturing processes for making it becomes mostly automated, prices begin to decline. At the point when the device is more and more of a commodity, smaller also means less expensive. This is why a PDA is much less expensive than a PC and yet it is much smaller. So, there is not a simple proportional relationship between size of device and cost of device. Our general rule stands that *when it comes to mobile systems and devices, smaller is nearly always better*. The small size serves the mobile purpose of the device the best. And, we all know that there are physical boundaries on the size of transistors on modern microchips. This means that there is some ceiling for storage and processing power of a device with a limited size bound by the heat produced by the transistors, the number of transistors that can possibly fit into each component, and the many other factors that the microprocessor industry has been studying since the birth of microchips.

Limitations of storage and CPU of mobile devices put yet another constraint on how we develop mobile applications. For example, a mobile calendaring application may store some of its data on another node on the network (a PC, server, etc.). The contacts stored on the device may be available at any time. However, the contact information that exists only on the network is not available while the device is disconnected from the network. But, because the amount of data that can be stored on each type of device varies depending on the device type, it is not possible to allocate this storage space statically. Also, some information may be used more frequently than others; for example, the two weeks surrounding the current time may be accessed more frequently in the calendar application or there may be some contacts that are used more frequently. Mobile applications must be designed to optimize the use of data storage and processing power of the device in terms of the application use by the user.

In this example, the calendaring application may or may not be the only application that uses the storage capacity of the device. So, the first step in designing the application would be designing the appropriate functionality for discovery of other applications on the device, the storage space that they use, and the total storage space available, and then computing the amount of storage available to the calendaring application. The operating system of some devices may offer the available storage space, but this is not guaranteed. So, we need to design with the least amount of assumptions about the hardware capabilities of the device or with all those assumptions valid for all of the devices to be supported by the mobile application.

Storage and processing issues are largely addressed by the various operating systems and platforms on the mobile devices. Therefore, a large part of engineering mobile applications requires first a theoretical understanding of the various

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

types of platforms and operating systems available on mobile devices, then an understanding of the available commercial implementations of the varieties of types of operating systems and platforms and the type of applications best suited for each platform–device combination. We will look at these issues closer in Chapter 2.

This dimension of mobile application design, namely the effect of device limitations, is perhaps the most well known of all dimensions in today's mobile application design. This was the first problem that software developers approached as they tried to port frameworks, platforms, and methodologies of application development of the 1980s and 1990s to mobile applications. It soon became obvious to researchers and developers that existing paradigms and platforms did not suffice. For now, many have simply adopted older methodologies and are building mobile applications as pure embedded applications using assembly language native to the device on which they want the application to run. However, we have already seen, in the evolution of application development for PCs and servers, that developing native applications is cost prohibitive. This is the reason that most of today's complex applications are not written in assembly; rather, they are written in C, C++, or a similar language and then compiled for the platform of need. Virtual machines have given us yet another level of indirection to avoid authoring device- and platform-specific code in languages such as Java, thereby, decreasing the cost of application development even more.

The point is that there is typically some cost involved with layers of indirection in software. Though these layers of abstraction and indirection can have many benefits, we need to balance their use with the single fact that mobile devices are limited in their CPU, memory, and other computing capabilities. And, this muddies solutions to some design and implementation problems that would otherwise be very clear.

1.2.4 Limited Power Supply

We have already seen that the size constraints of the devices limit their storage capabilities and that their physical mobility affects network connectivity. For the same set of reasons that wireless is the predominant method of network connectivity for mobile devices, batteries are the primary power source for mobile devices. Batteries are improving every day and it is tough to find environments where suitable AC power is not available. Yet, often the user is constantly moving and devices are consuming more and more power with processors that have more and more transistors packed into them. For example, a user who walks in New York City and lives in the suburbs may leave work, begin using his or her PDA, get on the subway, and continue using it until returning home. When traveling in Asia, Africa, and South America, users are certain to rely on their batteries more frequently as reliable wired power sources are less pervasive than they are in North America and Europe.

The desirability of using batteries instead of an AC power source combined with the size constraints creates yet another constraint, namely a *limited power supply*. This constraint must be balanced with the processing power, storage, and size constraints; the battery is typically the largest single source of weight in the mobile

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

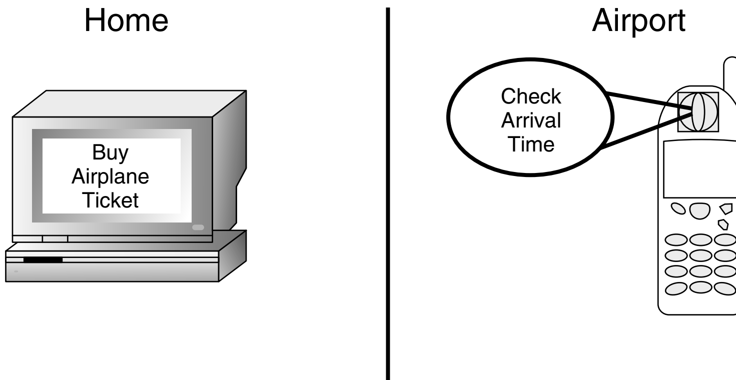

FIGURE 1.5. An Application That Uses Both Voice and Text User Interfaces.

device [Welch 2000]. The power supply has a direct or an indirect effect on everything in a mobile device. For example, the brighter the display, the more battery power is used, so the user interface is indirectly coupled to the power supply.

Most power management functionality is built into the operating system of the mobile device. Therefore, when it comes to device power management, the design focus is more on making the right choice in selecting the proper platform (device, operating system, etc.) and configuring the platform properly. In a typical stationary application, this would suffice. But, in mobile applications, we need to look everywhere we can to save power. Because the operating systems of mobile devices are typically very lean and have as few functionalities as possible, many times the application must carry some burden of awareness of the power supply.

Some platforms allow monitoring of the remaining power and other related power information. Some platforms allow multiprocessing and multithreading, which have an effect on the control over the variation of the CPU activity, which in turn has an effect on the control over the power consumed by the device. Overall, the design and implementation of the application itself is affected less by this dimension of mobility than by any of the others mentioned in this book. This is merely because operating systems and platforms are largely responsible for handling the power consumption issues. However, we will discuss the effects on choice of platform and other architectural and implementation effects that the power supply has on mobile computing systems in a bit more detail in Chapters 15 and 16.

1.2.5 Varying User Interfaces

Stationary users use nonmobile applications while working on a PC or a similar device. The keyboard, mouse, and monitor have proved to be fairly efficient user interfaces for such applications. This is not at all true for mobile applications. Examples of some alternative interfaces are voice user interfaces, smaller displays, stylus and other pointing devices, touch-screen displays, and miniature keyboards. Using a combination of interface types is not uncommon (see Figure 1.5).

For example, drivers who want to get some directions to their destination may use a data-enabled cellular phone, navigate through a simple graphical user

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

interface (GUI) menu to a driving directions application, and then retrieve the desired directions through a voice user interface by saying the address of the source and destination and listening to the directions. Note that navigating to the application may be done much more efficiently on a GUI: It may be as simple as pushing two or three numbers that activate some choices on the screen. However, entering text on the small display of a cellular phone and through the numeric keys of a phone is very cumbersome. It is much easier to say the source and destination and, subsequently, have a voice recognition system translate them, find the directions, and read them to the user by using a text-to-speech system.

A mobile application, based on its device support, the type of users using it, the conditions under which it is used, and many other factors discussed later in this book offers a variety of user interfaces. Perhaps the biggest paradigm shift that designers and implementers of mobile applications must undergo is to understand the necessity of finding the best user interface(s) for the application, architecting the system to accommodate the suitable user interface(s), implementing them, and keeping in mind that a new user interface may be required at any time. Although these user interface advances promise to be one of the main aspects of the next computing revolution, they add much complexity and confusion to the application design as the current application design and implementation methodologies only take into account keyboards, monitors, pointing devices, and sometimes touchscreens. The developer can no longer make any assumptions about the input and output mechanisms to the system; therefore, the development process becomes altogether different, complicating an already complex design process.

User interfaces are difficult to design and implement for the following reasons [Meyers 1993]:

1. Designers have difficulties learning the user's tasks.
2. The tasks and domains are complex.
3. A balance must be achieved among the many different design aspects, such as standards, graphic design, technical writing, internationalization, performance, multiple levels of detail, social factors, and implementation time.
4. The existing theories and guidelines are not sufficient.
5. Iterative design is difficult.
6. There are real-time requirements for handling input events.
7. It is difficult to test user interface software.
8. Today's languages do not provide support for user interfaces.
9. Programmers report an added difficulty of modularization of user interface software.

Meyers recognizes the problems associated with user interfaces of stationary computing systems. These problems are compounded by the *multichannel* requirement of mobile systems. Multichannel systems are systems that use multiple types of user interfaces for input and output such as text, voice, and video (see Chapter 8).

Since the recognition of the complexity of designing user interfaces by Meyers and others, some headway has been made in providing us some tools to reduce this complexity. First, because many GUI-based applications have been developed

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

for stationary systems, the iterative process of design and implementation and feedback from the users have taught us much about what works and what does not. So, we now know more about how to design user interfaces (item 4 in the preceding list).

But, methodologies, tools, and patterns used in the development of stationary applications do very little to separate concerns of user interface from the rest of the application. Sure, there are several design patterns such as the (often misused, abused, and overused) model-view-controller (MVC), but the use of these patterns alone does not take into account the special concerns of various types of user interfaces. They merely make some attempt at separating some of the concerns of the user interface from the rest of the system. They serve us well when we are dealing with a single set of textual inputs and outputs, but today's popular architectural techniques and design patterns are insufficient for a large variety of user interfaces. And this is why much of the research in the area of mobile computing focuses precisely on this problem: How does one separate the concerns of the user interface from the application regardless of the type of user interface?

Today, we also have proven software design and development methodologies, such as that of object oriented programming (OOP) and use of unified modeling language (UML), and the supporting languages and tools, that allow us to gather the requirements of the system more clearly (item 1 in the list of difficulties), to modularize software design (item 9), and to design software without dependence on the language of choice (item 8). But, we have no such methodologies and tools to take into account the multichannel requirements of mobile systems or any of the other added dimensions of mobile application design. Though there is no consensus today how to use these tools to ease the development of multichannel user interfaces, the reader will be presented with what we see as emerging methodologies and tools that leverage existing proven methodologies and tools such as OOP and UML.

Not only do most software applications designed today have large coupling between the user interface and the application, but also very few are designed to render to any desired user interface with few modifications. Most of today's applications need to be massively retrofitted or rewritten altogether every time a new set of user interfaces must be supported. Of course, there is also the special concerns of each type of user interface, such as voice user interfaces, that must be taken into account.

We dedicate several chapters in this book to discussing various issues surrounding software architecture for rendering to any type of user interface, voice user interfaces, and the ways that users communicate with systems. For now, the important factor is to recognize that the user interface design and implementation process has a much bigger effect on an average mobile application than its counterpart nonmobile application.

1.2.6 Platform Proliferation

Because mobile devices are small and there is much less hardware in them than in a PC, they are typically less costly to assemble for a manufacturer. This means that more manufacturers can compete in producing these devices. These cheaper, and

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

typically smaller, devices are often used for special purposes. The sum of these and other similar reasons gives rise to proliferation of the types of devices in the marketplace that an application must support.

Platform proliferation has very significant implications on the architecture, design, and development of mobile applications. Platform proliferation heightens the importance of designing and developing devices independent of the platform. Writing native code specific to the mobile device, unless absolutely necessary because of performance requirements, is not a recommended practice because of the proliferation of devices. For example, it is not wise to write a voice-driven phone book application that runs only on one type of platform. Of course, the platform makers and manufacturers of devices and operating systems of those devices will always try to create restrictions on the developer to prohibit writing platform-independent applications. They may conversely give the developer features that may only be implemented on their platform to tie the developer to that platform. Regardless of the efforts of commercial platform builders, the software architects and developers should be focused on their primary task of meeting the user's requirements. And if these requirements include support of multiple platforms, which happens more frequently than not for mobile computing systems, platform independence should be on the top of the architects' and developers' list when choosing the tools to build an application.

We will try to address the problem of platform proliferation by using nonproprietary methodologies and tools, such as UML, when possible. Throughout the book, we will show our sample code for multiple platforms, alternating from one to the other, so that the reader is exposed to particulars of implementation on several of the most prevalent commercial platforms.

1.2.7 Active Transactions

Most of today's stationary applications have a restriction that can reduce the benefits of a mobile application system enormously: The user of the system must initiate all interactions with the system. We call such systems *passive* systems because they are in a passive state, waiting for some external signal from the user to tell them to start doing some particular thing. With stationary applications, this typically works well. Most people sit down to use a computer because they intend to perform some task. Whatever actions they may perform could signal one or more other passive systems to perform some computing task such as retrieving information or calculating some numbers.

At the same time, during the past two decades, messaging-based systems have been born and have evolved. With messaging systems, any one participant of the system can send a message to the other participant(s), and, if desired, under a specific topic in an *asynchronous* manner. We will discuss both asynchronous and messaging systems later in this chapter and other sections of this book. But, the key idea to take away is that any one participant in the system could send a *message* to another participant in the system. Later came the idea of *push*. In the push model of communication, an information producer announces the availability of certain types of information, an interested consumer subscribes to this information, and

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

the producer periodically publishes the information (pushes it to the consumer) [Hauswirth and Jazayeri 1999]. There is much in common between the concepts of messaging systems and push systems. The principle difference is that messaging systems are asynchronous by definition. This requirement does not exist for push-based systems.

Push systems, by definition, are *active* systems. For example, a particular user could be browsing the Web and, while purchasing some goods online, be notified of the change in the price of a particular stock. In this example, the system has taken an *active* role in starting communication with the user on a particular topic.

Push–pull systems (a more complete name for push systems as the receiver of the "pushing" is said to be "pulling" on a particular topic) can be implemented in a number of ways, including using event-driven systems, messaging middleware, and poll-based systems. Implementation aside, unfortunately, push systems have mostly been a disappointing failure. One of the reasons for this failure has been that most push pull systems have targeted users who are largely focused on the task at hand.

If a user sits at his or her desk and begins using a PC, the user is constantly reminded, in an indirect manner, that he or she can access some piece of information. Even if the user is not performing some exact task, the simple condition of sitting down and using a keyboard and a mouse puts the user in a state where he or she is more likely to remember information processing related tasks. Educators often call this principle being *on-task*: As long as students are sitting at their desks, with their books open, there is a much higher chance of accomplishing tasks related to studying. Based on the same principle, the user who is sitting behind his or her desk working on a PC is *on-task and focused*. For example, if the user sits down and begins to type a memo to a coworker, the chances of the user remembering to check, say, his or her stock portfolio has increased by the mere fact that there are visual reminders, such as the browser icon on the desk top, that will remind the user to perform the task. Even if the user forgets (which is unlikely, particularly if you are sitting on thousands of shares of stock that are worth a tenth of what they used to be after a market crash), if he or she is merely reminded by an e-mail, the user can very easily begin the transaction that performs whatever tasks are needed to retrieve the necessary information and perform the necessary tasks. A reminder system certainly helps mainly because the user is focused on the task of computing and is available to receive the reminder. (We will talk about the lack of focus of the mobile user more in the next section.)

In this book, we will define *active transactions* as those transactions initiated by the system. Active transactions may be *synchronous* or *asynchronous*. *All active transactions are initiated by the system*. Synchronous transactions are time-dependent transactions. Note that the term *transaction* is used in data storage and other systems to indicate boundaries for roll-back and committing of a series of actions that must be successfully executed, in some predetermined manner, for the completion of the transaction. We use the term in a slightly different manner. We use it to refer to a sequence of interactions between the user and the

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

computing system. Synchronous active transactions can be summarized by a set of properties:

1. The transaction is initiated by the system, and during the same transaction, the user is given an opportunity, for a finite period of time, to respond to the action initiated by the system.
2. Synchronous active transactions require a timely response from the user.
3. The interactions between the system and the user work in a sequential and serial manner during a synchronous transaction.
4. Synchronous active transactions are established between the system and a single user. This may be replicated for many users, but at the most elemental level, there is only one user in each active transaction.

Let us look at an example of a synchronous active transaction. One of the tasks often forgotten by the field work force is logging time for tasks. For example, a cable company repair person who forgets to log his or her hours by noon may be called by the system at noon and asked to log these hours, through a voice user interface. The system asks the employee to start telling it, using the key pad or the voice, the time intervals worked and the tasks accomplished during those hours. If the employee does not answer the call, the system logs him or her as unavailable and may try again at a later time. If the employee does answer the call, but does not respond to any one of the questions within some allotted time, the system may record that the transaction has failed because of user irresponsiveness. If the employee answers all of the questions so that the system can successfully log the accomplished tasks, the transaction completes successfully and is logged accordingly by the central system. Of course, there are many reasons for the transaction to fail, including dropped connections, inaccurate interpretation of voice commands, and others. Regardless, the idea is that the user is called by the system and, if the user answers the phone, he or she is asked some questions and expected to respond within some given time frame. Also, the questions are asked in a sequential and serial manner. The system does not ask all of the questions at once and then wait for the user to respond to them one by one. Neither does the system ask questions while the user is answering any one of the questions.

Most of today's active systems are asynchronous. Asynchronous transactions are not time-dependent. Asynchronous active transactions, like their synchronous counterparts, can be described by a set of properties:

1. Asynchronous active transactions work just like messaging systems. They can be established with either 1–*n* receivers or 1–*n* topics to which 1–*m* receivers are subscribed.
2. Asynchronous active transactions may be a composition of 1–*n* messages sent by the system and may require 1–*m* messages back from the users. If 1–*m* messages required as responses from the users are not received within some time frame specified by the system, the transactions may be deemed as failed. Note that we are not defining the semantics of messaging systems (for if that

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

is what we were referring to, we would be wrong). Rather, we are defining the semantics of asynchronous active transactions to be such that they encapsulate a number of messages being sent from the system to the user and from the user to the system and that some messages from the user, marked as responses to the messages from the system, can be required for the successful completion of the transaction.

Now, let us look at the asynchronous version of the same time-logging application that we observed for synchronous active transactions. The system could call the user and wait for the user to answer the phone. Once again, if the user does not answer the phone, it logs the transaction as failed and the user's absence as the cause of the failure. If the user answers the phone, the system reminds the user that he or she has not logged his or her time for the day and needs to do so. At this point, the system asks the user to do this as soon as possible. The user can then call the system back at some later time and log his or her hours, upon which the transaction is considered successful. If the user never calls back to complete the transaction, the system may continue to call the user back with periodic reminders 1–*n* times. Once the *n* limit of times is surpassed, the transaction may be considered as failed. Once again, there are a variety of reasons for the failure of the transaction that can be recorded by the system. However, the main thrust of the example is that the system does not require a timely response. The system may have even asked the user to perform several tasks at a later time and the user may have done each one of those tasks out of order. In this example, the time dependence, sequential order, and serial manner of the tasks of the transactions are irrelevant. This gives the system more flexibility but we lose certainty in when and how a response from the user is going to be received. Also, the serial order of the tasks during the transaction may be desired or undesired.

Choosing whether the active behavior of a system is implemented using an asynchronous active transactional model or a synchronous active transactional model is completely dependent on the user requirements and the available tools (which translates to the available budget).

So, we have now defined the basics of what we will need to treat active transactions. Active transactions are an absolute essential part of mobile application development mainly because of the lack of focus on the part of the user while the user is mobile. The semantics of active transactions are defined only for the purpose of this book. One may argue against these transactions in different contexts. But, for the context of mobile application development, they will serve us well in communicating requirements, architecture, design, and implementation. And why are they less important to stationary applications? Because the stationary user is typically focused on the task of computing while the mobile user is not. We will consider the condition of the mobile user more in the next section.

Finally, it is important to note that active transactions differ from push–pull systems and messaging systems not only because they can be both synchronous and asynchronous but also because they can contain 1–*n* interactions between the system and the user. We will discuss active transactions in much greater detail in Chapter 13.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

We have now looked at the added dimensions that we need in our thinking paradigm to understand mobile application development. Let us quickly look at the root cause of the existence of these dimensions of mobility, namely the environmental effects on the mobile user's requirements.

1.3 Condition of the Mobile User

Any computing system with end users has at least two participants, the computer and the user. We have looked at the computing system in analyzing the dimensions of mobility, those things that make mobile applications different from stationary applications. Now let us look at how the mobile user differs from the stationary user. We will call this difference between the mobile user and the stationary user the *mobile condition*. The elements of mobile condition distinguished here are not necessarily comprehensive as the user studies done and the industry experience with mobile applications are in an infancy stage. However, together, they contain all of the major differences between mobile and stationary users.

The mobile user is fundamentally different from the stationary user in the following ways:

1. The mobile user is moving, at least occasionally, between known or unknown locations.
2. The mobile user is typically not focused on the computing task.
3. The mobile user frequently requires high degrees of immediacy and responsiveness from the system.
4. The mobile user is changing tasks frequently and/or abruptly.
5. The mobile user may require access to the system anywhere and at any time.

Note that the mobile condition is not just about the physical condition of the mobile user but also about the mental state of the user: his or her expectations and state of mind. Note, also, that the differentiating elements between the mobile user and the stationary user are the root causes of the dimensions of mobility. So the relationship between the mobile condition and the dimensions of mobility is one of cause and effect.

Now, recall that we recognized the dimensions of mobility as the difference between mobile and stationary applications. We have now come full circle; we can see that the dimensions of mobility are a byproduct of the requirements of a mobile user to use a mobile application. To complete the chain of logic for our dimensions of mobility, let us look at the differences between the mobile user and the stationary user that comprise the *mobile condition*.

1.3.1 Changing Location

It may seem trivial to state that a mobile user is always, or at least frequently, moving. But, this motion has a significant implication in that the location information can be used to draw conclusions about the context in which the user is using the application. This is the reason location sensitivity and QOS are dimensions of

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

mobility. The location of the user at a given time is a variable. Other variables may be the speed at which the mobile user may be traveling, what network connectivity modes are available to the user, what the quality of that connectivity may be at any given place and time, or how long he or she may stay connected or disconnected. The mobile user also expects the system to have good connectivity coverage. The mobile users will come to also expect the system to know the device's location with fair accuracy as location services become more commonplace. This aspect of mobile computing presents the developers with the opportunity of giving the users functionality not possible with stationary applications. It is a clear differentiator that presents the mobile user with great value that cannot be obtained through a stationary application. Therefore, building applications that take advantage of the location information and that are localized is often a must with commercial mobile applications.

The changing location of the mobile user also forces restrictions on power, size of device, wireless connectivity of the device, and just about every other aspect of the state of the mobile user. In those respects, it creates restrictions that we have already looked at. In using the location information of the application, we have an opportunity to provide functionality beyond that of stationary applications. The moving nature of the mobile user is a physical aspect that gives way to a mental state of *lack of focus*.

1.3.2 Lack of Focus

The primary focus of the mobile user is seldom on the computing task (although, obviously, there are exceptions to this, but we are talking about the majority of time when the user has a device and is mobile). This is the primary reason for the necessity of active transactions. While a user is driving from work to home, the task of driving takes the primary focus. During this time, if the stock price of one of the user's holdings begins to plummet, he or she cannot sell it before it falls too far. The user either does not know of the plummeting price at all or is not focused on checking on the stock price at regular intervals. Mobile users are typically mobile because they are moving between two points with the primary task of reaching the destination.

Another reason for lack of focus is multitasking. Mobile users often multitask. For example, a user may be driving and talking on the phone. Another example could be a user who is entering some data into a PDA out in the field (collecting information on power lines as a field electrician, measuring environmental effects as an environmental engineer, etc.) while doing the primary field work task at the site (such as climbing a pole and paying attention to power lines, finding the right place to measure, and keeping the environmental conditions stable while measuring, etc.). Because of this multitasking nature of the mobile user, a variety of user interface input types such as voice may be needed to take advantage of the senses that are not preoccupied by another task. Also, the user interface to the system must be very user friendly and require as few of the user's senses focused on communicating with the machine as efficiently as possible. For example, voice user interfaces allow users to focus on driving while still getting whatever information they need from the system.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

1.3.3 Immediacy

Mobile users are often in a situation where they need to quickly perform one or more computing tasks, such as retrieving contact information, sending a voice or e-mail message, or triggering some remote process. They don't have the time to go through a long boot sequence or long application setup times. Mobile users normally have higher expectations of performance from their devices than stationary users do. Performance of mobile applications is not an afterthought as it often is in the development of stationary applications. A short delay in application responsiveness can decrease its usefulness enormously. For example, a user who cannot get the necessary contact information from a mobile contact application will eventually become frustrated and use a directory service to find the necessary contact information in urgent situations. It is also important to note that there are different types of immediacy. For example, the user's tolerance, depending on the application, will vary in first connecting to the network compared to the system response time. The types of immediacy depend on the application.

1.3.4 Abrupt Changes in Tasks

As we mentioned before, the mobile user is typically mobile because he or she is focused on something else other than computing. For example, many mobile users will try to use commute time: Whether in a train, in a plane, or in an automobile the user will be distracted by different environmental factors.[†] These factors must be kept in mind in designing and implementing the flow and, once again, the interface of the application. The mobile user needs to be able to stop performing some computing task abruptly, do something that may be completely unrelated, then return to the application after some unknown period of time, and, without much effort to remember what he or she had been doing, continue the computing task. Mobile users expect applications that flow smoothly and do not require complex navigation despite the abrupt nature of their actions.

1.3.5 Anywhere, Anytime

The cliché of "Anywhere, Anytime," along with all of its synonyms or similar clichés ("Everywhere," "Everyplace," or "Anyplace") and other words that refer to this phenomenon such as "Pervasive" and "Ubiquitous" are perhaps the most overused set of words in mobile computing. Nevertheless, this is still one of the most important aspects of mobile computing. The mobile user expects to be able to retrieve data and do computing at any given moment and any given time. And this is precisely why the support for a variety of platforms with a variety of user interfaces is critical for a mobile application: To use an application anywhere and anytime, one may have to use it through whatever device (any device) is available and convenient for that given place and time. Mobile users expect to start a transaction and leave it unfinished on one device at a given place and time

[†] Note that not all of the mobile conditions of the user may coexist at the same time—sometimes users are focused on the task of computing (for example, when they are in a train or a plane); other times, they are not (for example, when a real estate sales person is selling a house and, unbeknownst to that person, another listing comes open that may be a better fit for his or her buyer).

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

and finish the same transaction later on a different device and at a different place and time.

The *mobile condition* of the mobile user should be the primary guiding tool in architecting, designing, and implementing the mobile application. The various problems presented to us by the mobile condition may require solutions that have inherent conflicts. For example, to increase the number of devices and user interfaces supported, we may want to centralize the business logic and use the devices only as thin clients. (Thin clients are discussed in detail in Chapter 16.) However, to make the user interface as friendly as possible and to make the use of application possible even when the device is not connected to the network, we would want to push a significant portion of the application to the client. Obviously, these two aspects are in direct conflict. In another example, we may see that a particular mobile application requires increased CPU to perform a particular task faster. But the increased CPU may mean a considerably larger device, making it more difficult to carry. As with any other engineering problem, while designing mobile applications, we will find that we often need to balance the solutions to problems presented by each mobility dimension. There is no better balancing guide than the mobile condition of the mobile user. Of course, cost in itself can offset the benefits of any solution. Once again, as with any other engineering problem, the solution needs to fit the problem of the customer, in our case the mobile user, within a given budget. With an unlimited budget, nearly anything can be done. But, of course, we all know that there is not such a thing as an unlimited budget.

Therefore the cost and the mobile condition comprise the variables that describe our customer. Every mobile user will have a specific set of needs, but those two are constants. In added dimensions of mobility, we have the major effects of the mobile condition on the requirements for building mobile applications. Once we have gathered the requirements from the user, the first step in building the mobile application is to decide on the architecture. And this is what we will discuss next.

1.4 Architecture of Mobile Software Applications

The first step in building a software application, after the process of gathering requirements, is to lay down a high-level plan of what the application will be like when it is finished. Mobile applications, like any other software application, require such a high-level plan. We call this high-level plan of the mobile application a "mobile software architecture." Our approach to architecture in this text will be bottom up: we will introduce a variety of design patterns, application architectures, and processes with each addressing some specific problem with mobile applications (Figure 1.6).

If you are not familiar with the basic prevalent application architectures in today's distributed Web applications, we recommend that you read Sections 16.1.1 and 16.1.2 of Chapter 16. You will see the terms N-Tier, client–server, mobile agent, and peer-to-peer quite frequently throughout this text and you should have at least a passing familiarity with them. In Chapter 16, we will define

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

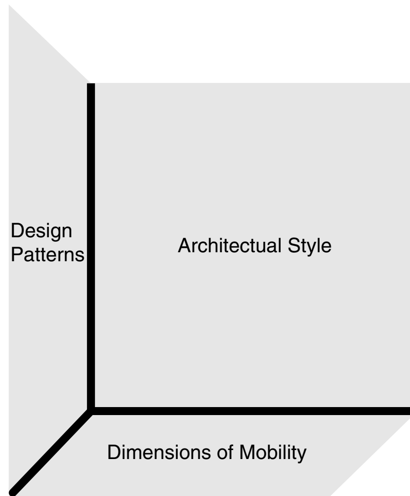

FIGURE 1.6. Mobile Application Development Design Consideration Space.

software architecture to be *a particular high-level abstraction of the system and how its components collaborate*. Then, we will summarize what we will learn in Chapters 1 through 15 to get a feeling for various architectural designs and techniques for mobile applications. For now, let us look at what software architectures will mean to us within the confines of mobile application development.

There are also *architectural patterns*; these are patterns that are recognizable once they are used prevalently in some architectures. Although there are no fully established design patterns, architectural patterns, or even architectures in the field of mobile computing because of its infancy, one of our goals in this text is to outline some techniques that show evidence of such techniques beginning to mature. These patterns exhibit themselves in a variety of families of problems. For example, we will introduce several different architectures for design and implementation of multimodal user interfaces. We will also introduce some lower level design patterns for separating the concerns of building user interfaces from our core application.

Note that the architectural decisions made in a software system are typically the most important during the lifetime of that software system. Architecture is also as much of an art as a skill gained through experience. It is at least partially a stylistic aspect of software. With this said, we will try to lay out the various alternatives made available by commercial vendors and academics. You will need to make the appropriate decisions based on the requirements of the individual project.

1.5 OUR ROAD MAP

In this first chapter, we looked at the dimensions of mobility and the mobile condition. These two helped us understand the fundamental differences between designing a mobile application and a stationary application. Next, we surveyed some high level architectures. As we mentioned, much of what architectures do for us is to lay out a high-level plan of how the components of the system interact with

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

each other and what the general properties of the system are. We will spend the rest of this text discussing the components: the nitty-gritty of how to make things work; but, we will come back to architectural issues periodically and examine how the components fit within the architecture.

To create mobile applications, we will need some tools. Section 1 of this text will give an introduction to those tools. In Chapter 2, we will look at some commercial and open-source frameworks and tools that ease the development process and show some different approaches to creating mobile applications. We will use these frameworks and tools to show examples in the later chapters. In Chapter 3, we will look at Extensible Markup Language (XML) and the nature of XML content. XML is an important piece of the puzzle in distribution of content to any device. In Chapter 4, we will look at UML, the tool we will use in modeling the design of our applications. Though not all mobile platforms use OOP technologies, most do. UML gives us an industry-accepted way of documenting requirements, design, and implementation of the system.

In the second section of this text, we will look at the problems of the user interface. Chapter 5 will show the reader how to separate concerns of particular user interfaces, such as graphical user interfaces, from the concerns shared by all types of user interfaces. In Chapter 6, we will see how to complement the generic user interfaces and render graphical user interfaces for a variety of visual text-driven devices such as PDAs and data-enabled cell phones. In Chapter 7, we will look at VUI (voice user interface) technologies such as voice recognition, text-to-speech technologies, voice transcription, and VoiceXML. At the end of Section 2, we will combine what we have learned in Chapters 5–7 and, in Chapter 8, we will see how to design user interfaces that interface with the user through multiple media types and multiple channels. This section will show us how to design the user interface components to fit the needs of the mobile user. It will also show us how to fit them within the mobile architecture of the system.

In the third section, we will look at a disparate set of topics, each relating to a dimension of mobility or an aspect of the mobile condition. We will start with examining mobile agent and mobile code architectures more closely. These architectures are often neglected in other texts. Because of their importance to mobile computing, we will pay special attention to them. We will then look at various wireless technologies as they are the prevalent means of connectivity for mobile applications; we will also look at the effect of wireless connectivity on architecture, protocols, and other aspects of design and implementation of a mobile system. The disconnected user needs data at the device even when disconnected; for this, we will need a discussion of data replication and synchronization design and implementation issues. We will move on to two key dimensions of mobility, location sensitivity and active transactions, how to incorporate such functionality into the design of the system, and how to implement functionality for some of the frameworks and tools talked about in Section 1. We will finish Section 3 by discussing mobile security issues.

In Section 4, we will see how to put all these aspects together to make a successful system. This section should be a great read for those project managers who want to know what to do differently for a mobile application. And there are plenty

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

of differences, from the requirements-gathering process to testing. In this last section, we will also look at some typical "dos and don'ts" and a case study of implementation of some of the concepts introduced in this text.

It is important to keep in mind that this text is not a text on "how to implement the technology de jour." We are focused on issues of design and engineering that apply across tools. Specific implementations come and go. They evolve based on the demands of the market, economic situation, and many other factors. Though we will use examples from a variety of commercial and open-source specific implementations, we are focused on issues that apply to any and all specific implementations of mobile application platforms; we are concerned with design.

Let us now get started by looking at the tools and frameworks that are available today to create a mobile application.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

CHAPTER 2

Introduction to Mobile Development Frameworks and Tools

The truth of the fact is easier to bear than the truth of the fantasy.

James Hillman

2.1 INTRODUCTION

At its most primitive level, software is a set of instructions for hardware written in machine language. At a higher level, there are assemblers and higher level programming languages. There are frameworks, tools, and other methods of abstracting various aspects of software design that help us achieve one central goal: to handle complexity of software more reliability and faster. The biggest problem with software design and implementation is complexity and it is this complexity that leads into buggy systems, high cost of development, and long development cycles, and the existence of programming languages, frameworks, and other development tools is primarily to solve this very problem of software complexity. In other words, as one of the most fundamental software design concepts, abstraction reduces complexity (at least theoretically).

Today, there are many programming languages, frameworks, and tools designed to develop server-based and desktop applications. These languages, frameworks, and tools have matured through the years, becoming more efficient and more reliable as they get tested in real environments by real users. Along with the maturation of these tools has come the maturation of the process of software design and implementation. Ideas such as OOP, design patterns, and de facto

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

standard software development processes have been developed and have matured with the tools and frameworks in a symbiotic manner. So the question is whether we can take the same methodologies, frameworks, and tools and use them to develop mobile applications. And the answer, just as the answer to all of the other great questions in life, is "Yes and No!"

These notions of abstraction of various concerns in designing and implementing software have been mostly based on reducing the complexity of those systems with the most financial benefits: business systems being used by users of PCs, main frame systems, and other computing systems that require the user to sit in front of a monitor and type. For example, frameworks such as class libraries to write user interface code for Java (AWT, JFC, etc.) or C++ (MFC, Borland, etc.) are all designed around a user interface that allows for data entry through keyboard and mouse and displays information to the user through a monitor. Even at a more rudimentary level, most software is written for PCs and servers without regard to the power consumed by the system, the amount of storage available, and the variety of user interfaces. So, it is fair to say that most of the development tools and frameworks today are designed to facilitate writing software for stationary and non mobile systems.

With that said, though there are many aspects of mobile software design and implementation that are not addressed in today's frameworks and tools, there is much that mobile and stationary software applications share. For starters, most commercial software, whether it is mobile or not, is intended to be run on microprocessors. Developing for mobile or nonmobile applications includes similar processes of requirements gathering, design, implementation, and testing. But, we repeat the question, "Can we or can we not use the same methodologies, frameworks, and tools for mobile application development?"

The answer is more of a "Yes" as the software gets closer to the hardware and more of a "No" as it gets farther from the hardware. The frameworks that help us when writing software that is "closer" to the hardware such as compilers and assemblers focus on easing the process of programming granular tasks such as moving bits and bytes between memory locations and performing additions, subtractions, and multiplications; these are all very basic operations when looking at software applications from the bird's-eye view. However, high-level frameworks and tools such as user interface development tools (HTML, JFC, Visual Basic, etc.) and other component development tools (COM/DCOM, EJB, etc.) that try to solve high-level business logic problems do not lend themselves well to mobile application development. The layers of abstraction in most of today's frameworks and tools have been done with a strong bias toward developing software for stationary applications.

These tools do not take into account the concerns, mentioned in Chapter 1, that make mobile software development inherently different from software development for stationary systems. With this in mind, an entirely new market is expanding around developing software tools and frameworks for mobile application development. Most of what exists today is in the infancy stage; therefore, we can expect a significant amount of organic evolution in these frameworks and tools: The weak will die and the strong will evolve and improve. Nevertheless,

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

we can start to see families of frameworks and tools as well as the features that create the taxonomy of this space. In this chapter, our focus will be on the feature sets and the taxonomy while using real tools and frameworks in the market today. The reader should focus on the concepts of the frameworks rather than the implementations. Although some will die and others will evolve, the reasons for their formulation, design concepts, and feature sets will remain applicable.

Frameworks and tools for mobile application development are evolving based on the growth of architectural techniques and innovations that accommodate the dimensions of mobility. Although the purpose of any significant tool and framework used in mobile application development should be to reduce the complexity of the mobile application, all tools, regardless of their implementations, attempt to address the same issues. However, depending on the architectures that each may support, their implementation and usage significantly vary. Therefore, it makes sense to create the taxonomy of these tools based on the architectures. Let us begin by looking at the frameworks and tools that address mobile application development in a fully centralized architecture.

2.2 Fully Centralized Frameworks and Tools

Developing fully centralized mobile applications differs from other fully centralized applications by virtue of QOS, limited power supply, active transactions, and location awareness (four of the dimensions of mobility mentioned in Chapter 1). Fully centralized mobile applications typically have custom-designed clients to perform specific tasks. So, the user interface on the devices used to access the centralized system is optimized to the task being performed. The software on such devices is typically embedded in nature and is designed to do only one thing. Also, because of this embedded nature of fully centralized mobile systems, resources of the device are not a concern in software development: The abilities of the client are known beforehand. Platform proliferation, once again for the same reason, is not a concern: Software systems in fully centralized mobile systems are all about the software on the fully centralized host; the client devices are dumb with little or no ability to perform dynamic computing tasks and what little software exists on them is embedded. Therefore, three of the dimensions of mobility—namely platform proliferation, limited device capabilities, and support for a variety of user interfaces—do not apply to fully centralized applications.

Location sensitivity, in most fully centralized systems, is achieved as an integral part of the network system or hardware-based location information on the client device (such as GPS modules). Call centers are a prime example of what *can* be a fully centralized mobile application. A cell phone user may call a call center to access the system. The call center may approximate the location of the user through receiving information from the cellular telephony system, from a GPS module on the cell phone communicating with the system through the same or different channel. (Circuit-switched phone calls carry only voice whereas packet-switched calls can contain multiple channels of data and voice.)

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

The application at the central host as well as the embedded software on the client must be designed with QOS issues in mind. Because all of the software on the device is embedded, the module handling the communications piece can be considered tightly coupled to the other modules on the client; therefore, taking into account that QOS issues become natural.

In summary, fully centralized mobile applications are about a monolithic layer of software from the client to the server with very little software on the client. What software resides on the client is typically embedded, or at least highly coupled to the device, in nature. Fully centralized mobile applications are the right solutions for applications that require little to no flexibility in changing the requirements of the client over the lifetime of the application and that have large development and deployment budgets allowing for custom-designed hardware and embedded software. Some good examples of such systems are battlefield systems used in determining the location of a target and sending it to a centralized system, which then relays it to another system responsible for launching a missile. Another good example is the kind of system used in grocery stores for inventory tracking as stock personnel track and refill the on-the-shelf inventory. In this case the mobile devices are customized to record information about groceries and relay them to some centralized inventory management system.

This is seldom the case in the world of commercial application development. If mobile applications are to be pervasive, the same agile economic models that surround the stationary applications on the PC and servers must succeed. Precisely for this reason, we will not spend much time dwelling on issues surrounding embedded software. For those interested, there are a variety of resources for embedded software development. Our focus in this text will be mobile applications that can be used on at least a small variety of devices and ones that do not require custom-designed hardware. With this said, let us look at N-tier client–server applications and the corresponding tools and frameworks.

2.3 N-Tier Client–Server Frameworks and Tools

As we discussed in Chapter 1, client–server architectures allow us to enable communication between two applications with one application acting as the server and the other acting as the client. For mobile applications, the server may have special needs, but it is typically powerful enough to run a wide range of applications. For mobile applications, there may be special logic that treats the dimensions of mobility. Client applications, in the case of mobile development, are typically those being run on mobile devices. Writing large applications for the devices to serve as the client is typically not possible, primarily because of the limited resources on the devices and the large variety of them. So, more often than not, mobile applications are distributed. The state of the art, as of the date of authoring this text, in proven distributed computing systems are the N-tier client–server architectures.

One of basic problems of application development that is magnified in mobile environments is code portability and mobility. The varieties of the so-called platforms (combination of hardware and operating systems) have prompted the

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

creation of tools and frameworks such as Sun's Java Virtual Machine and Microsoft's Common Language Run-time. The primary goal of these tools is to give code more portability across platforms. The problem is magnified when considering the added factors that the variety of mobile devices dwarfs the variety among PC and server operating systems, that virtual machines tend to be large and require lots of memory and CPU cycles, and that, once more, they are designed primarily with the primary task of designing applications for stationary computing systems.

Here, there are two factors that are inherently opposite in nature. First, we need a layer of software, be it a virtual machine or otherwise, that abstracts us away from the specificity of hardware. This is the only practical way to write software rapidly for mobile systems. But then, as software layers are added, performance is hampered and system requirements go up. This tension between these diametrically opposed factors has given rise to the creation of numerous frameworks and tools for mobile application design. More than ever, selection of the frameworks and tools depends on the requirements of the application.

We can address this problem in three ways:

1. *Thin-Client Wireless Client–Server*: We can have some homogeneous browser specifications and implement the browsers for each device in a client–server environment. The browser can then load markup code and render it or even load plug-ins. This approach would be similar to the Web-model approach where the browsers are implemented for a variety of operating systems so that Web developers do not have to worry about the environment in which those browsers run. As we saw in Chapter 1, this would require a persistent and stable connection to the network and only allow for the lowest common denominator feature set among the various platforms and devices. So, at least today, this model is implemented by having different families of devices and platforms with one corresponding browser for each. We will look at various techniques for serving the right type of content to each type of browser. Such tools and techniques focus on building a server-side structure that serves up the right type of markup language to the browser that interprets it on the client. The Wireless Access Protocol (WAP) and its user interface markup language of WML give us a framework for building thin-client wireless applications with an N-tier client–server architecture.
2. *Thick-Client Wireless Client–Server*: The client application on the mobile device may be a custom application. If so, this thick client may communicate with the server, with the client executing some tasks and the server executing the others. Stationary client–server architectures using thick clients typically use the client as a means of storing a small subset of the data for use of the application when disconnected from the network and performing business logic that does not need to be centralized. Having thick clients for mobile devices is a bit more difficult. For one thing, as we have mentioned time and time again, mobile devices have very restricted resources. There are those who say that Moore's Law will eventually eliminate any practical restrictions that affect the application developer; however, there are other problems. There is the deployment and provisioning problem: How do you distribute software to such a wide range of

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

devices? How do you even write software for such a large variety of platforms? The platforms that allow thick-client development for mobile devices address this in two ways:

a. Some provide an operating system or a virtual machine that provides the application programmers with a platform that lessens the number of permutations for writing code. J2ME (Java 2 Micro Edition) allows this through a small virtual machine that sits on top of the hardware (or the operating system that is run on the hardware). Microsoft requires an installation of some flavor of Windows on the device (such as Windows CE) that allows the application programmer to write programs for Windows. Symbian also provides an operating system for mobile devices. Both Sun Microsystem's Java and Microsoft technologies, despite their differences, allow developers to create applications on top of an *operating environment*. These tools are typically products of software vendors who want to sell software and do not want to limit themselves to a given hardware platform.

b. Hardware manufacturers, such as Qualcomm and Texas Instruments, provide programming environments directly on top of hardware (ASIC, EEPROM, etc.). We will look at Qualcomm's BREW as an example of this.

Client–server architectures that rely on a thick client require a full-blown development platform for the device. Such platforms, however, may be used in environments other than just thick-client client-server-based systems. For example, we can use J2ME to build stand-alone applications for small mobile phones. Typically, many of the same programming environments that are used for building client applications on the devices for a client–server system are those same environments used to build applications for the devices in a peer-to-peer or mobile agent–based system. In the case of J2ME and Symbian for example, the development tools provided by the platforms can be used for building applications for a variety of architectures.

3. *Stand-alone Applications*: Lastly, we can build stand-alone applications for the devices using those same platforms that we mentioned for the thick-client client-server-based systems. The only difference here is that stand-alone applications do not really need networking components. For example, many of the first applications for the Palm operating system were only downloadable through the cradle that attaches the Palm to the serial port of the device. From there, you can download an application and run it with no network connectivity. Building stand-alone mobile applications is somewhat of a novelty as the mobile user needs to be able to at least synchronize the application with some external system periodically. There are few applications, such as stand-alone games, that just need to be downloaded and executed on the device.

But, in the mobile world, the manufacturer's of devices want to differentiate their hardware from their competitors. One way of doing this is by allowing the developers to write programs very specific to the device platforms in platform-specific languages such as C or C++, as in the case of BREW and BREW-like environments.

Figure 2.1 shows some of the more popular platforms at the date of authoring this text and their ability to provide functionality based on connectivity to the

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

Connectivity / Platform	Stand-alone	Networked	
		Wired	Wireless
Mobile Platforms			WAP
	Symbian		
	BREW		
	Java		
	.NET		

Figure 2.1. Some Products in Various Categories of N-Tier Client–Server Frameworks and Solutions.

network. Today's popular operating systems allow applications to be written without a lot of low-level programming to access hardware. They also allow multiple applications to use the same hardware simultaneously and have standard functionality such as accessing permanent storage (such as disk IO), volatile storage (such as RAM), and interface peripherals such as the monitor and the keyboard. But, traditional operating systems are typically large and take up considerable permanent storage. They also typically require quite a bit of volatile storage to get started. For this reason, embedded software development will always be around. Platforms such as Qualcomm's BREW present another alternative in writing applications for the device. Developing in such environments as BREW represents the opposite end of the spectrum to Java: The applications are written specifically for a given hardware platform without the traditional notion of the operating system. Such platforms as BREW allow for developing software that is optimized for a "chip set" or specific hardware. The code is then compiled and then "burned" onto the device. Depending on the type of hardware used, this "burning" process can be repeated *n* times. Because these types of tools and frameworks are specific to the device itself, they focus on solving the problem of writing applications for devices. So, the problem of transporting data back and forth between the network and these devices as well as transforming them to the proper formats used by each type of device remains unsolved.

There is yet another family of tools and frameworks written to handle processing of data on the server and communicating with a wide variety of devices. Typical tasks solved by these tools include support for messaging as a means of asynchronous communication; support for HTTP or a similar protocol as a means of synchronous communication; and the ability to transform different types of XML by accommodating some complex set of rules that include workflow, device-type recognition, and multichannel rendering of content. Apache's Cocoon project

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

and IBM's Wireless Transcoding Publisher are examples of frameworks that try to fulfill some of these goals.

Whereas Java offers an open and relatively mature environment to program in the same language on any platform, Microsoft is trying to take advantage of its large lead in the software development market to extend its technologies to include application development tools with its .NET and Windows CE technologies. Java allows the developer to program in Java and run the code anywhere. In contrast, .NET promises to allow the developer to program in any language and run it on any .NET-based environment (various Microsoft Windows family of products). Of course, this means that any device running the applications written using the CLR (Common Language Runtime) has to run an operating system that supports that CLR. Such operating systems are limited to the Windows family of operating systems. So, although Java is language bound and *cross-platform*, .NET is platform bound and *cross-language*.

2.3.1 Mobile Operating Systems and Virtual Machines

Although Java tries to solve the proliferation problem by making the code portable between different platforms, there are other plausible approaches. One of these approaches is to create tools that make the applications native to one platform. Microsoft's .NET framework deploys such a strategy. The tools provided by the .NET framework allow the programmer to develop the application in a variety of languages supported by the framework. The individual applications are then compiled to code that can be executed on the same platform. Microsoft's creation of the .NET platform is spurred by economic reasons, namely to keep Windows as the dominant computer operating system. However, this does not imply that the tools provided by the .NET framework are either superior or inferior. It is simply a different technical approach whose merit should be judged by the implementing developers and the users of applications that use this platform.

The principal technical difference between the .NET and Java approaches is that .NET generalizes by operating system and Java generalizes by programming language. So, with .NET, every device, be it a PC or any other type of computing device, is required to run some flavor of Microsoft Windows as its operating system.

Something important to remember as we go through various tools is that developing applications for mobile devices typically involves use of an emulator provided by the platform or device manufacturer. In this way, the unit testing and quality control of mobile applications differs from that of stationary applications: Everything is typically finished and tested on the emulator first and, then, tested on the actual device.

2.3.2 Hardware-Specific Tools and Frameworks

One way to deal with device proliferation is to avoid it! Device manufacturers can allow the application programmers to develop code that directly takes advantage of the device features and functionality. The notion of an operating system, in such case, is much different than what we typically think of as an operating system. The services offered by the operating system are few and very low level. The downside here is a tight coupling to a platform that, in turn, can translate to

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

heavy reliance on the manufacturer of that platform. This is the approach that Qualcomm offers in its BREW platform. BREW is a framework designed to allow application developers program applications for devices based on Qualcomm's CDMA technology. We will look at CDMA further in Chapter 9. It is a physical layer communication protocol that offers very efficient use of the bandwidth available in a segment of the spectrum.

In this chapter, we will address the development tools and frameworks in a client-server context. In Chapter 9, we will look at some mobile agent tools as well as seeing how the tools that we look at in this chapter apply to mobile-agent architectures. We will look at the various families of frameworks and tools that may be used to develop mobile software applications and some commercial platforms that fall into each family. We will select the most common environment as opposed to the most elegant environments. There are many reasons for this, the most obvious of which is that the commercial success of products, often, does not have a direct relationship with the elegance of the technical solution. Also, as engineers, we often have to select popular platforms to build systems for economic and other business reasons. Once we have selected our frameworks and tool sets, we will use them, later on in the book, to develop sample applications.

Let us start with Java, as it is today's most popular application development programming environment.

2.4 Java

Today, it is widely accepted that Java as a programming language offers the most portable commercial environment for writing software applications. The success of Java has been mostly in providing standard Application Program Interfaces (APIs), a very thoughtfully designed infrastructure for OOP that prohibits many bad design and implementation habits such as multiple inheritance. Standard and open APIs offer a process of evolving a language that is open to many vendors. Furthermore, there exist implementations of the virtual machine and the native dependencies of the APIs for most popular operating systems. There are three major categories of Java APIs and virtual machines, namely J2ME, J2SE, and J2EE.

Java offers three distinct features as a mobile application platform:

1. Java is an object oriented programming language. As any other programming language, it can be used to write applications.
2. Java offers complete code mobility and weak mobile agent ability. Java allows for platform-independent programming.
3. Java is a platform.

We will assume that the reader has at least an understanding of what Java is as a programming language and will discuss the code mobility aspects of Java further in Chapter 9.

First, Java, as with any other programming language, is just that: a programming language. It allows us to program a set of instructions. Perhaps just as importantly,

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

Java is somewhat of a *vendor-neutral language-based platform*." Java seems to have solved the problem that has plagued many other programming languages in the past: the lack of standardizing libraries. With C++ and many of the other programming languages, one of the biggest problems has been the lack of industry-wide standards in APIs, components, and tools. Different vendors have offered similar components and frameworks with no uniformity among them in their APIs and interfaces. Vendors have done this to differentiate their products; however, this forces developers to rewrite code when moving from one component set or framework to another or even to completely redo the architecture of the system. Java has solved this problem by enforcing standard API interfaces to the components and frameworks and allowing for vendors to compete on the basis of the implementation of the APIs. For example, Java Database Connectivity (JDBC) APIs present the same interface to the developers regardless of what database is being used

Java, as a platform and programming language, offers mobile code. But, the standard Java Virtual Machine was designed for desktop computers and requires far too many resources for the typical cell phone, PDA, or mobile device. The standard Java Virtual Machine is packaged, along with accompanying tools and class libraries, into Java 2 Standard Edition (J2SE). A smaller version of the virtual machine, along with a subset of classes and tools of J2SE plus a few additional tools, forms J2ME designed for small devices.

2.4.1 J2ME

J2ME is a specification for a virtual machine and some accompanying tools for resource-limited devices. J2ME specifically addresses those devices that have between 32 kB and 10 MB of memory. J2ME addresses the needs of two categories of devices [Sun Micro J2ME Spec 2000]:

1. *Personal, mobile, connected information devices*. This portion of J2ME is called CLDC for Connected, Limited Device Configuration. These types of devices include cell phones, PDAs, and other small consumer devices. CLDC addresses the needs of devices with 32 to 512 kB of memory. The virtual machine for the CLDC is called KVM for K-Virtual Machine.
2. *Shared, fixed, connected information devices*. Internet-enabled appliances, mobile computers installed in cars, and similar systems that have a total memory of 2 to 16 MB and can have a high bandwidth and continuous connection to the network are in this group. CDC, or Connected Device Configuration, is the part of J2ME that addresses such devices. CDC is a superset of CLDC.

Let us look at both CDC and CLDC and how we can use them to develop mobile applications.

CLDC and MIDP

Figure 2.2 shows how J2ME components, and other parts of Java as a platform, stack up. Figure 2.3 shows the breakdown of the J2ME MID Profile stack. As we mentioned previously, CLDC is mainly intended for devices that are

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

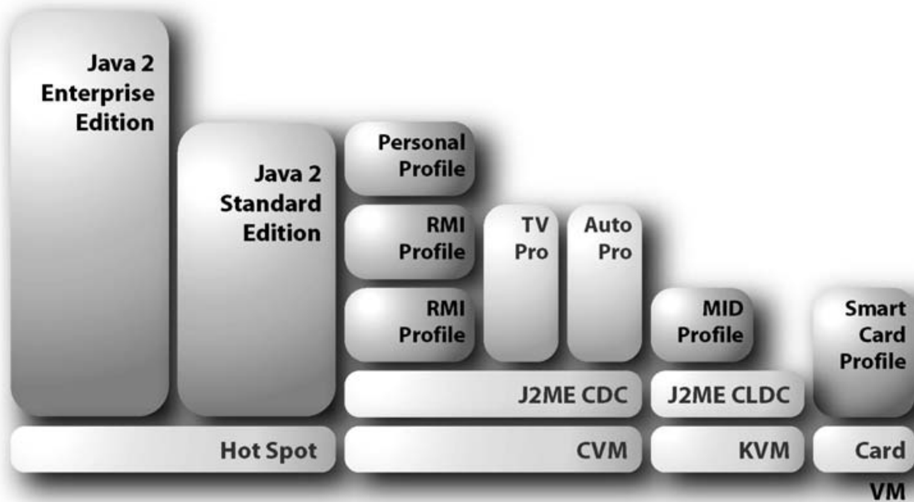

Figure 2.2. J2ME Stack (CLDC/CDC and MIDP).

resource-starved such as mobile phones and PDAs. CLDC addresses the following features:

1. *Providing a virtual machine for providing language features*. Perhaps the most important thing to keep in mind for those who have built applications using the Java Virtual Machine on desktops and servers is that the J2ME/CLDC Virtual Machine is not at all like the version that comes with J2SE. To cut down on the required resources for running it, the KVM does not provide many of the advanced features that the J2SE Virtual Machine does. The KVM is based on the Spotless project, which started at Sun Labs. The KVM takes up anywhere from 40 to 80 kB depending on the device. The KVM is written in C (as are most other Java Virtual Machines). Some features not offered on the KVM are the following:
   a. *Floating point arithmetic*: Floating point operations are expensive or require the chipset on the device to have specific implementations for them. Many of

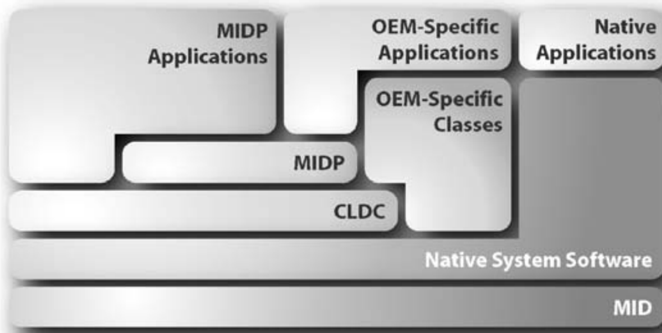

Figure 2.3. Layering of Functionality between CLDC and MIDP.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

the resource-starved mobile devices either do not have floating point specific features on the chip set or do not expose them for use by applications software running on the device.

b. *Support for JNI*: Java Native Interfaces (JNI) allow developers to write applications that use C/C++ programming languages along with Java in providing Java APIs to modules or applications not written in Java.
c. *Thread grouping*: Advanced threading features are not offered on the KVM and CLDC. Multithreading requires a baseline amount of resources to be dedicated to creating, maintaining, and destroying threads. Each thread takes up a certain amount of resources by simply existing, even if it never does any actual work. Because the KVM is intended for resource-starved devices, it is natural to assume that doing lots of advanced multithreading is not something that makes much sense on such devices.
d. *Full-blown exception handling*: Exception and error handling seems to be one of the first places that platform providers trim when building frameworks and tools for limited devices. Although this makes more work for the application developer, it allows the framework and the applications to be linear.
e. *Automatic garbage collection of unused objects*: Though the KVM does offer some of the memory management features of the J2SE Virtual Machine, it does not offer *finalization* of objects. This means that you have to tell the KVM when you are done with an object. The KVM is not capable of finalizing based on the scope of methods, etc.
f. *Weak references*: An object is said to be weakly referenced if it is necessary to traverse the object that refers to it to reach it. The J2SE Virtual Machine does not allow finalization of an object until all weak and strong references to that object are cleared. The KVM does not provide this functionality for weakly referenced objects. The elimination of weak references and finalization in the KVM make programming for the KVM more like writing C and C++ applications than writing a typical J2SE application. Much of the automatic memory management benefits of Java are in its ability to manage memory based on weak references and to automatically finalize. These features have been eliminated to shrink the virtual machine. Although they allow the applications to be faster, the static size of the applications grow as memory management is more manual and there is a higher probability for typical C/C++ memory management type bugs in the applications. This is not to imply that there is no garbage collection. Indeed, there is a garbage collector in the KVM. However, the garbage collector has to be manually notified when to discard objects.

2. *Providing a security framework for tasks such as downloading MIDlets (J2ME CLDC/MIDP applications)*. Security is one of the most troublesome and complicated features for providers of mobile application frameworks and tools. J2ME builds on the experiences of Java applets in creating a security paradigm for mobile applications. It should be noted that CLDC does not provide the full J2SE security model, though it does provide enough low-level virtual-machine security to guarantee that the application can not harm the device in any way.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

It also provides a sandbox model, though it is different than the J2SE sandbox model.

The security sandbox of CLDC is provided by removing the ability to write JNI code to access native functions on the device, providing a very limited set of APIs (which we will look at next), taking away the ability to write custom classloaders (there are no custom classloaders in CLDC), and a class file verification process that assures that the files called to be executed are Java class files. The verification of a class file is also different from its counterpart in J2SE. CLDC class file verification is a two-step process that offloads some of the task of verification from the device. The CLDC verifier needs about 10 kB to execute. But, because of the offloading of some of the verification process from run-time, the size of the class files is slightly larger (about 5%).

3. *Providing a reasonable amount of functionality for input and output*. Most programs need a persistence mechanism. CLDC provides a very limited and yet sufficient set of APIs to read and write to the nonvolatile memory provided by devices. It should be noted that the persistence of data on the device is hardware dependent.
4. *Providing some internationalization capabilities*. CLDC's input/output (I/O) package (see the next section) provides input and output stream readers that can handle different character encoding schemes. This allows internationalization in two ways:
   a. *Dynamic*: The program can determine the required character set dynamically and use the proper character set at run time. Programmatically, this is the more elegant option. However, it requires additional code to implement the rules for discovery of the required character set. This approach works well for small applications where the device resources are not taken to their limit.
   b. *Static*: There can be multiple versions of the J2ME application ready to be loaded onto the device. Provisioning of the application can take care of the version of software that is distributed to the application. Though this approach is less elegant, both the amount of code downloaded by the device and the amount of logic executed at run time can be reduced. The flexibility of having different character sets for the same device is still available as different versions of the application are available for download on the network.
5. *Providing a reasonable amount of networking capabilities*. CLDC provides a connection framework to provide basic networking capabilities. Profiles such as MIDP build on top of this framework and can introduce more advanced networking capabilities.

As we saw, there are some features whose support was eliminated to shrink the CLDC to a manageable size on the device. Some features have been intentionally left out to be handled by "profiles" that are built on top of the CLDC. Profiles address features that can be addressed, in the same manner, for a group of devices but whose implementations vary because of the differences among those devices. The best example of a feature set falling into a profile is the user interface capabilities. Because various devices have different methods of entering data, different

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

screen sizes, etc., the best place for the user interface functionality is in the profile. The areas addressed by profiles are the following:

1. download and installation of applications,
2. life-cycle management of applications,
3. user interface feature,
4. database functionality, and
5. event handling.

The Mobile Information Device Profile (MIDP) is currently the only widely known and accepted CLDC profile. There are other profiles, such as Personal Digital Assistant Profile (PDAP) designed for PDAs (typically assumed to have more memory, processing power, and other resources than MIDs), that extend CLDC. MIDP is designed for devices that are assumed to have the following characteristics:

1. Small displays of approximately 96×24 of 1:1 shaped pixels with a depth of 1 bit.
2. A minimum of 128 kB of nonvolatile memory (for storing information that is not lost when the device is shut off and turned back on). This is mainly intended for storing the application itself.
3. Wireless connection to the network (with all of the implications of what wireless connectivity is at the time this text is being written: low-bandwidth, intermittent connectivity, no standard protocol such as TCP/IP, etc.).
4. A minimum of 8 kB of nonvolatile memory for use by the application. This 8 kB refers to information that the application should be allowed to store on the device.
5. An ITU-T phone keypad (this is the standard alphabet mapping to the ten digits on a phone keypad) or a QWERTY keyboard (such as those available on Palm, RIM, or Handspring devices).

Now, let us quickly look at the Java APIs for CLDC and MIDP so that we can write a simple application.

Overview of the CLDC and MIDP Java APIs

There is a core set of APIs that every implementer of CLDC (device manufacturers and hardware integrators) must implement. These APIs fall within two groups:

1. J2SE-like APIs: There are three packages, namely java.lang.*, java.io.*, and java.util.*, that are inherited from the J2SE environment. It is important to note that only a small subset of the classes available with J2SE in each package is available for CLDC. Also, those classes available in these packages are not identical to their J2SE counterparts in interface or implementation (though the designers have done their best to keep the interfaces as similar as possible to ease the task of porting).
2. CLDC-specific APIs: In the current version of CLDC (1.0.2) a small set of classes provides I/O and networking capabilities particularly needed by small and mobile devices. The package holding these classes is javax.microedition.io. The

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

main class that the J2ME application developers must familiarize themselves with is the *connector* class. J2SE networking facilities assume the availability of a TCP/IP connection. Obviously, this assumption is not a valid one for mobile applications as a variety of communication protocols and schemes may be used to allow the device to communicate with the network. So, CLDC defines a connection framework in its Java API, providing a method for various network providers, device manufacturers, and protocol designers to offer the application developers options other than TCP/IP for communicating with the network. For example, it is possible that a vendor provides WAP-style connections (WDP/UDP) that can be invoked by CLDC connection objects by passing the right parameters to it. An example could be the following:

```
Connection c = Connector.open("http://www.cienecs.com");
```

As we mentioned previously, MIDP builds on the top of CLDC to offer the functionality required to build a real application. Let us review the MIDP APIs quickly.

1. *Timers*: Two classes, java.util.Timer and java.util.TimerTask, allow developers to write MIDlets that are started, one time or at some specified interval, at a given time.
2. *Networking*: Whereas CLDC provides a generic connection framework that can be built upon by the device manufacturers and network providers, MIDP provides HTTP implementation, a high-level application networking protocol, that hides the lower layer implementation of networking between the device and the network (TCP/IP, WAP, etc.). The javax.microedition.io.* package holds the lone class of HttpConnection that allows connecting to network resources through HTTP.
3. *Storage*: javax.microedition.rms.* (where rms stands for record management system) provides a very simple API for storing and retrieving data. The query capabilities provided by this package, though extremely rudimentary, are invaluable as they provide the basics of database-like access to nonvolatile persistence on the device.
4. *User Interface*: javax.microedition.lcdui.* offers a set of rudimentary user interface APIs to build interfaces for MIDlets. Like the storage package, the user interface package is very simple. However, it accomplishes much by offering an interface that is fairly generic, leaving the mapping of the interface to the implementation to the MIDP implementers. This increases the portability by allowing authoring of user interfaces without worrying about a great amount of detail on the implementation of MIDP on a particular device (though it still does not mean perfect portability).

Now, let us look at a simple J2ME/CLDC application.

Hello MIDP

CLDC applications only make sense as an application of a profile. Because the user interface of the J2ME application is reserved for the profiles, writing a CLDC Hello

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

World application really does not make that much sense. The profile of choice for our example, obviously, will be MIDP. Applications for MIDs (Mobile Information Devices) are appropriately called MIDlets (like their counterparts of server-side applications, which are called servlets, small browser-based applications called applets, etc.).

As in applets and servlets, MIDlets are treated as components controlled by a framework under the inversion of the control principle to which we refer to frequently in this book. For a J2ME class to qualify as a MIDlet, it has to do the following:

1. Extend the MIDlet class.
2. Implement the following methods:
   a. startApp(): This method gets called after the class is instantiated. Think of this like the run() method of a thread in Java.
   b. pauseApp(): This method is called if the application has to be suspended for some reason. Suspension of the application can be required for power saving, an incoming phone call, or a series of other reasons.
   c. destroyApp(boolean b): This is used to do any maintenance necessary before the application is discarded. This method is necessary mainly because finalization and weak references are not available in J2ME. (It can be used for release of other resources as well depending on the type of the application.)

Figure 2.4 shows a simple MIDP application that simply shows a message on the screen and allows the user to exit the application.

A variety of vendors, such as Borland and Sun, offer J2ME development tools. Sun Microsystems has a free tool kit that offers the following components for development of J2ME applications:

1. *KToolbar*: This is the overtool that provides a GUI to manage collecting the classes that are put into the MIDlet, any name-value property sets that are used by the classes, and any resources such as icons used by the MIDlet. It also provides GUI control over build and bundling of the MIDlet into a deliverable package to the device.
2. *Preverifier*: As we mentioned previously, preverification of classes allows J2ME to offload some work from the device.
3. *Compiler*: The J2ME compiler compiles the classes. Remember that J2ME classes need to be preverified before they are ready to be used.
4. *Emulators*: There is a series of emulators that ship with any development kit. Mobile device and mobile software vendors provide other emulators for J2ME.
5. *Emulation of Performance*: The Preferences tool allows the developers to adjust for the virtual machine proficiency, network performance, storage monitoring, and network traffic monitoring. These features have only been available in the latest version of the tool kit. Though they may seem secondary, they actually provide a huge leap over the previous versions of the tool as, for the first time, some of the dimensions of mobility are treated within the tool kit. These are namely limited devices resources and QOS.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

```
import javax.microedition.midlet.*;
import javax.midroedition.lcdui.*;

public class HelloMIDP extends MIDlet implements
  CommandListenter {

    public static final String HELLO = "Hello MIDP";

    private Display mDisplay;
    private Command mExit;

    public HelloMIDP() {
        mDisplay = Display.getDisplay(this);
        mExit = new Command("Exit", Command.SCREEN, 1);
    }

    public void startApp() {
        TextBox myMessage = new TextBox(HELLO, HELLO, 256, 0);
        myMessage.addCommand(mExit);
        myMessage.addCommand((CommandListener) this);
        mDisplay.setCurrent(mDisplay);
    }

    public void pauseApp() {
        //Our application is very simple and does not really
        //require any manual finalization or other actions if
        //the application is suspended. The implementation of
        //this method is not trivial for more complicated
        //applications.
    }

    public void commandAction(Command aCommand, Displayable
      aDisplayHandle) {
        if (aCommand == mExit) {
            destroyApp(false);
        }
    }

    public void destroyApp(boolean b) {
        notifyDestroyed();
    }
    }
}
```

FIGURE 2.4. Hello MIDP.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

Though the Java community is working on it, unfortunately, J2ME still does not treat multimodal user interfaces and location sensitivity at all. These are two dimensions of mobility that have gone nearly completely neglected in J2ME.

Using the KToolbar to generate an application is fairly intuitive once you have your source code in. Tools such as Borland's JBuilder and Websphere Anywhere Suite offer editors specially customized for J2ME code. Once the code is compiled, you will run a *.jad file in one of the emulators. The *.jad files encapsulate information about MIDlets.

To deploy a J2ME application, everything is bundled into a JAR file. A JAR file can have one or more MIDlets (classes that inherit from the MIDlet class and implement the appropriate methods). The JAR manifest file (a text file that specifies the classes that are in the JAR file along with some attributes for those classes) is used by the MIDP environment (implemented by the J2ME device) to recognize and install the applications. There are a set of required attributes in the manifest file needed for the environment to run an application. The J2ME tool kit provides a GUI to create the attributes. Although the JAR manifest contains a set of attributes for all of the MIDlets in the JAR, there is a JAD file for every MIDlet. The JAD file acts as an application descriptor. The JAD file must have the name, version, vendor, JAR URL, and JAR size of a MIDlet. It may contain other information such as a description and an icon.

Treatment of Dimensions of Mobility by CLDC and Profiles

Because of the way profiles are layered on the top of CLDC, dimensions of mobility are treated in a peculiar way. Let us look at the dimensions individually.

1. *Location Awareness*: To date, there is no treatment of location awareness in J2ME. However, this is being treated. JSR 179, Location API for J2ME is defining an optional package to build on top of CLDC version 1.1 and higher. This JSR is intended to work with various positioning methodologies such as GPS or cell-based triangulation; however, it is explicitly intended to hide the implementation, and complexities thereof, of the positioning system. Therefore, the API will be agnostic to the method of finding the location. Currently, the package name is proposed to be javax.microedition.location.
2. *Network QOS*: During the development, as we mentioned earlier, various development tools offer emulation of QOS conditions for wireless devices. J2ME's connection framework addresses this issue, in an extremely elegant manner, after deployment of the application on the device. Because the connection framework is able to create any type of connection, network providers and device vendors can provide their own APIs on the device. The connection framework also provides the flexibility to use datagrams of various protocols such as WAP. Obviously, standard connections of TCP/IP and HTTP are available as well.
3. *Limited Device Capabilities*: KVM takes away large chunks of functionality, which is helpful but not necessary, for development of applications to shrink the size of the virtual machine. This was obviously done with limited device capabilities in mind. The tools provided with the J2ME tool kit also provide settings to emulate the behavior of the limited device such as a setting that allows one

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

to account for the performance of the virtual machine (KVM) on the various devices.

4. *Limited Power Supply Management*: This is one of the areas left virtually untreated by J2ME. But, this lack of treatment seems to be common throughout the application development platforms. Next-generation mobile platforms and tools will include intuitive techniques to take into account the power supply levels, at run time, to optimize the use and performance of the application.
5. *Support for a Large Variety of User Interfaces*: Though J2ME takes into account the variations in simple graphical user interfaces in CLDC/MIDP, there are no parts of that to treat multichannel user interfaces (e.g., mixtures of audio, video, text, etc. for input and output to the system). This lack of treatment of voice and other nontextual channels exists in both development and deployment environments.
6. *Platform Proliferation*: By allowing one to select from a different sets of emulators, J2ME provides fair support, at least in CLDC/MIDP stack, for developing applications for various devices. In its architectural design, by breaking down various devices into families of devices supported by CDC, CLDC, etc. and creation of layers such as MIDP, J2ME is perhaps the most well designed application development framework in treating platform proliferation. Furthermore, though we do not address various embedded Java technologies outside of J2ME in any considerable depth, Java, as a platform, offers the most comprehensive treatment of the variation of hosts for software applications.
7. *Active Transactions*: Because CLDC/MIDP applications are components run by a virtual machine and within a tightly controlled sandbox, writing an active application is a difficult task. The components do not control their own life cycles (they are controlled by a state machine that calls predefined methods depending on the events that are sent to it), thereby making J2ME applications inherently passive. It is possible to achieve a limited amount of active behavior by polling. Supporting active transactions (sometimes referred to as *push* if it is between two different hosts on a network) is something that the Java community is actively discussing.

Overall, J2ME offers a very good treatment of dimensions of mobility. Although some aspects are currently neglected, the Java community is continually working on treating them. Though it may take a long time, it is comforting to know that they will eventually treat each dimension and that the treatment will be vendor neutral, creating an environment of competition where new products will flourish and the better products will survive.

XML and J2ME

XML is the document format of choice when it comes to ubiquitous applications; we will look at this and related XML issues in detail in the next chapter. However XML is not only text, but it also requires considerable horse power to parse it. Although XML-based technologies such as XML-based Web services are ideal for providing ubiquitous content for mobile devices, they are tremendously

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

troublesome for a resource-starved platform that has to save every bit of memory and every cycle of CPU.

As you have noticed in our discussion of CLDC and the profiles that accompany it, there is currently no special treatment of XML (though such are being discussed at the time of writing this text in the Java community). There are three types of parsers [Knudsen 2002]:

1. *Model Parsers*: These parsers go through the entire XML document and create some representation of the document in a programmatic model. DOM (Document Object Model) parsers are model parsers. Model parsers use considerable memory and processing power because, regardless of what you need out of the XML document, the entire document is parsed and represented in some other format in memory.
2. *Push Parsers*: These parsers emit events as they parse through the document. Once again, they go through the entire document; however, the advantage they offer over the model parsers is that they do not keep a representation of the document in memory.
3. *Pull Parsers*: Pull parsers do not go through the entire document. Rather, they leave the control on how much of the document is parsed to the client.

Selecting the parser is somewhat of a balancing act that often requires some knowledge of the average and maximum size of documents. The application always knows what information it needs from the document. Putting together what needs to be extracted from the document, the typical size of the document, and the size of the application depending on the parser used tells us what the best fit for our need is. For example, whereas pull parsers are typically larger in size (kXML is a pull parser for CLDC/MIDP), they have a simple interface allowing for a small application to do all the necessary work without taking up a lot of memory to store the entire document. This works well for scenarios involving larger documents and straightforward data extraction from XML. However, if the documents are going to be small, the cost of storing them in memory is less, so a smaller model or push parser does the job effectively.

We will have some samples later in this text that parse XML on the device with J2ME.

Using UML to Model J2ME Applications

As we mentioned in Chapter 1, one of our objectives in this text is to tie the entire development cycle into UML and use it as a tool to facilitate the development process. Java and UML have been married during their evolution. As part of the Java platform, it is natural that we think about modeling J2ME applications with UML.

There are two aspects to modeling J2ME applications with UML. First, J2ME applications have a great deal in common with all other Java applications: They are written in Java, which is an object-oriented language. UML is designed to model object-oriented languages. Second, there are features of desktop and server-side

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

virtual machines (e.g., J2SE Virtual Machine) that are not available in J2ME, such as finalization and weak references. The elimination of these features forces the developer to take care of some tasks manually. Most Java developers are not used to managing memory semimanually or worrying about weak references; therefore, UML gives us a great visual tool to track down weak references, make sure objects are finalized in a proper way, etc.

Let us enumerate the various uses of UML in a J2ME application:

1. *Class Diagrams*: As with any other Java application, we can model the classes and the relationships among them with UML class diagrams. The class diagrams present us with an invaluable tool to see where weak references may be. To do this, however, we need to be very explicit in specifying association types and life-cycle controls within our UML diagrams. When it comes to modeling J2ME classes with UML class diagrams, the more detail, the better. J2ME applications are typically not very large (remember the resource restrictions), so a significant amount of detail added to the class diagram does not create an unmanageable situation.
2. *State Diagrams*: State diagrams can be used in representing the life cycles of the various objects. With J2ME, having numerous state diagrams can be invaluable in giving developers a visual tool to analyze the life cycle of various objects that may need to be finalized and to reveal bugs that are caused by the lack of support for weak references. State diagrams can also be used to represent the effect of various events. Because CLDC applications, and most other J2ME applications, are components used in an environment of inversion of control, the driver component (for example the MIDlet) implements a particular set of methods and/or inherits from some class with some default behavior. State diagrams can help in clarifying the behavior of the components as various events, driven by the user interface or otherwise, change the state of objects.
3. *Component Diagrams*: Though most J2ME applications are fairly small, component diagrams can come in handy too. One of the techniques used in creating multifunction J2ME applications is to divide them into smaller applications, each represented by a component in a component diagram, and to make the user interface hide the disparateness of the small applications, disguising them as one large application.
4. *Sequence Diagrams*: As we will see in later chapters, these diagrams can be extremely useful in representing user interfaces. The profile layer (MIDP) encapsulates the user interface implementation, and the MIDP APIs are designed in such a way that user interface actions are specified generically and the specific functionality is delegated to the MIDP implementation. Because of these features, sequence diagrams help in documenting the exact various interactions on various implementations on MIDP on devices that are all CLDC/MIDP compliant but vary slightly in specifics such as the number of buttons on the keypad, extra buttons, the number of lines on the screen, etc.

We will discuss using UML for various parts of the development process of mobile applications throughout this text. Keep in mind that UML is a general tool and

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

its use can be subjective when applied to specific things like various APIs and platforms.

2.4.2 CDC

We have looked at CLDC, one of the two parts of J2ME intended for devices (Java Card and other embedded technologies being somewhat tangent to our discussions). The other part of J2ME is CDC, which is targeted at environments, where more than 512 kB (usually about 2 MB) of memory is available for the Java environment and the application [Laukkanen 2002]. Whereas CLDC can have a variety of profiles built on top of it, CDC profiles are built on top of the so-called Foundation Profile. Like CLDC's KVM, the CDC has its own virtual machine, the CVM (C Virtual Machine).

Unlike the KVM, the CVM supports all of the features that the J2SE Virtual Machine does. The CDC is smaller than J2SE by the virtue of its lack of many of the class libraries that are shipped with J2SE. The CVM also offers some changes to improve performance on resource-starved devices. These include lower memory usage (about 60% less than the J2SE virtual machine), an extensible CVM architecture (to add functionality), and a design that accommodates real-time operating systems (RTOSs). Because the CVM has been implemented mostly in C, it can be ported to, and between, real-time operating systems easily. (The more assembly-level code exists in the implementation of a software application, the more difficult it becomes to port to, and between, RTOSs because assembly code is specific to platforms—hardware and operating system combinations).

The most significant classes eliminated from the CDC/Foundation Profile are the GUI classes. To date, CDC implementations exist for several handheld operating systems, including Windows CE, Linux, and Symbian.

In his paper [Laukkanen 2002] Laukkanen looks at the performance aspects of CDC versus J2SE under a variety of conditions. For those planning on implementing CDC applications, this paper is a must read. Laukkanen's testing results show that although CDC performs nearly as advertised with smaller applications (fewer objects, threads, etc.), as the application gets larger, it begins to underperform. Keep in mind, though, that in a resource-starved mobile device, we should not have large applications anyway. Although CDC minimizes the use of memory resources, as Laukkanen puts it, "the fact is that without Foundation Profile, the CDC is quite useless." This is because the architecture of CDC simply modularizes the functionality of J2SE into multiple profiles, allowing the vendors and application developers to only use the part of the Java platform that they need while still having the full functionality of a full-blown Java Virtual Machine in CVM.

We will not be using CDC-based examples in this text. Although CDC increases in its relevancy to mobile application development because of the increasing resources on the mobile devices, the programming paradigm of CDC is not much different than that of J2SE. So, writing CDC-based J2ME applications is much the same as writing any J2SE application. Also, there is no special treatment of dimensions of mobility in CDC as, to date, it is mostly used for network appliances (e.g., TVs) that are always connected and fairly stationary (though this does not

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

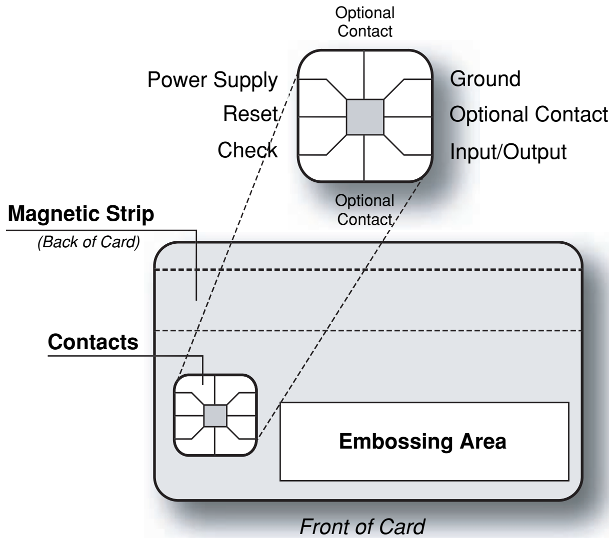

FIGURE 2.5. Java Card.

mean that CDC has any limitations that prohibit it from being used for mobile applications).

2.4.3 Java Card

Smart cards have been around for a long time. A smart card is a card that has an embedded processor or some type of electronic memory device able to store data, interface with some known set of devices, and allow the stored data to be retrieved. Most smart card technology, prior to Java Card, has been based on proprietary technologies. So, interoperability between different cards and card readers/writers has only been possible if the manufacturer of the card or the reader/writer offers an open API and the counterpart implements that open API. Obviously, with every manufacturer having its own API, managing smart cards and their readers/writers has been one of the most technically challenging tasks in creating smart cards. It has also created economic scaling problems in using smart cards across different businesses, locations, languages, etc.

The Java Card (Figure 2.5) specification is designed to solve these two problems. The Java Card API provides an API that, when abided by, allows for interoperability between different card readers/writers and cards regardless of the manufacturer and Java Card API implementer.

Today, there are three types of smart cards [Ruuskanen 2000]:

1. *IC (Integrated Circuit) Memory Cards*: This is the most common type of smart card. These types of cards hold a small amount of data (less than 4 kB) and have no processing power. These cards are used as debit cards, security cards, and others.
2. *IC Microprocessor Cards*: These cards typically have 16 kB or less of read-only memory and half of kilobyte of random-access memory. Java Card falls into this family. These types of cards provide a very small amount of processing power

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

that can be used for things like encryption and decryption of the user profile information on the card.

3. *Optical Memory Cards*: These cards provide the largest amount of storage of all smart cards. Though they do not provide any processing power, they can be very useful since they hold up to 4 MB of data.

As in the case of CDC and CLDC, the Java Card has its own virtual machine, the Java Card Virtual Machine (JCVM). But, the Java Card Virtual Machine is fundamentally different from the other virtual machines we have discussed. The JCVM never stops! The JCVM's state is permanently persisted into the electronically erasable PROM (EEPROM) when the card is removed from the reader. It is restored when it is inserted back into the reader.

Smart card technologies such as the Java Card offer a very unique and innovative approach to problems solved by mobile applications. Smart cards are one of the smallest devices in the range of mobile devices. Though they do not offer much in the way of input/output or processing power, they introduce a different paradigm of mobile computing where the user depends on card readers to exist everywhere he or she goes. Though this paradigm is not as flexible as a device that is available to the user all of the time, smart cards are smaller and less intrusive. The smart card of the future may even offer things such as a small display for receiving messages and wireless access to the network.

We will not discuss the Java Card much during the remainder of this text. Smart card technologies promise to be a sizable part of the solution set to the mobile computing problem; however, the applications for smart cards are very small, passive, and typically not applicable to anything that is represented by UML.

Now, let us look at another key Java technology that can help us in tying the network of mobile devices together.

2.4.4 JINI

The Java Naming and Directory Interface (JNDI) allows various resources to be identified in a generic manner on the server side; however, it is far too heavy for implementation on mobile devices. But, we already know that one of the necessary pieces of functionality to write mobile applications is discovery of devices and services. In Chapter 3 and the remainder of the book, we will look at platform-independent discovery mechanisms such as RDF, CC/PP, and UAProf. Java, however, gives us Java Intelligence Network Infrastructure (JINI), a base technology for ad-hoc networking. JINI provides lookup services and its own discovery protocol. Let us go through the basic transactions that JINI provides:

1. *Lookup*: This is a JINI service that maps interfaces indicating the functionality offered by a service to sets of objects that implement the service [Hashman and Knudsen 2001]. Lookup functionality of JINI provides the basic foundation for a federated service in which a variety of services cooperate and various processes can offer each other various services.
2. *Discovery*: Before a given process begins using a service found by the lookup process, it must find that service. The act of finding lookup services is called

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use

discovery. This is typically done by the underlying infrastructure that offers the JINI implementation.
3. *Events*: The various JINI participants can register to listen to the various events emitted by the other JINI participants. Any so called JINI device (anything that can become a participant in a JINI network) can register events with any other JINI device. In this way, the architectural communication model is more like peer-to-peer than it is client–server.
4. *Leasing*: JINI devices share resources through a process called leasing. The term leasing is used because the amount of time for which the service is available to the lessee is known in advance, at the time of the lease. This is a distinct requirement of JINI. Although the amount of time for which the service is being used by the lessee has to be known at the time of the lease, this time can be dictated by the leaser (the device whose service is being used) or through a negotiated process between the leaser and the lessee.
5. *Joining*: For a JINI device to offer its services to other devices, it first has to join the JINI federation. This is done through a process called joining.
6. *Transaction Management*: Interactions between the various JINI devices may be compound, being made of several simple atomic interactions. Because of this, transaction management is needed to ensure the proper semantics are provided to avoid partial results and bad data.

JINI specification merely provides us with a set of rules on how JINI devices must behave. Most implementations that exist today are not designed for mobile devices because they take up too many resources; however, there are some that offer "mobilized JINI." PSINaptic, for example, offers an implementation of JINI suitable for mobile devices in its JMatos. A clear advantage the JINI and other ad-hoc networking technologies offer is that they allow mobile devices to roam through a variety of networks. This promise, however, is difficult to fulfill primarily for two reasons. First, the network operators of different networks roamed by a JINI device may be operated by different entities, thereby having closed boundaries to the JINI devices. Second, even if these network operators open up their networks for interoperability, a JINI implementation would have to live on the top of a quilt of different low-level communication protocols implemented by each network.[†]

As Eronen recognizes [Eronen 2000], the biggest downfall of JINI today is its requirement of a virtual machine: "JINI's Java dependency, while enabling most of JINI's best features, is at the same time the most limiting aspect of the technology. A Java Virtual Machine that is required for each JINI service is not a light piece of software." Today, JINI and J2ME on the same device is not widely available. The Java community is working on making JINI a more usable technology for mobile devices with real implementations.

[†] As a side note, a group of JINI devices that are aware of one another are often called a JINI Federation. The word "Federation" is frequently used in cooperative and ad-hoc networking environments to indicate participation in a distributed computing system that requires some level of autonomous behavior on first joining the federation, then allowing others to discover the device and the services on the device, and finally interacting with the other members of the federation.

EBSCOhost - printed on 4/16/2021 2:31 AM via INFORMATION AND LIBRARY NETWORK CENTRE. All use subject to https://www.ebsco.com/terms-of-use