17 The Luther Architecture

A Case Study in Mobile Applications Using J2EE

with Tanya Bass, James Beck, Kelly Dolan, Cuiwei Li, Andreas Löhr, Richard Martin, William Ross, Tobias Weishäupl, and Gregory Zelesnik

God is in the details.
— Ludwig Mies van der Rohe

Workers involved in the maintenance or operation of large vehicles (such as tanks and aircraft) or portions of the industrial infrastructure (such as bridges and oil rigs) have great difficulty using computers to support their tasks. Because the object being maintained or operated is large, work on it must be in situ, outdoors or in special structures, neither of which is conducive to desktop computing. In particular, a computer solution usually involves a wireless infrastructure and either a handheld or a hands-free computing device.

Inmedius is a company that was established in 1995 as an outgrowth of Carnegie Mellon University's Wearable Project (see the sidebar History of Wearable Computing) to provide support for front-line maintenance and operation workers. Initially producing one-of-a-kind solutions for its customers, as the company grew it realized the necessity for general solutions that could be quickly tailored to a customer's needs.

The front-line worker does not work alone but requires a great deal of back-office support. Problem reports must be collected and work must be scheduled to enable repairs to be made, replacement parts must be taken from inventory and re-ordered, and maintenance records must be analyzed. All of this work-flow management requires integrating the front-line worker with the back-office worker who has access to a desktop computer.

Note: All of this chapter's contributors work for Inmedius Corporation in Pittsburgh.

The Luther architecture was designed to provide a general framework within which Inmedius could provide customized solutions for the maintenance problems of its customers. It is based on the Java 2 Enterprise Edition (J2EE) architecture, so becomes an application of the general J2EE/EJB framework (discussed in Chapter 16) to an environment where the end user is connected over a wireless network and has a device with limited input/output capabilities, limited computational capabilities, or both.

History of Wearable Computing

Arguably, the first wearable computer was the wristwatch. It was invented around 1900 and at first was unable to compete with the pocket watch. Why would someone wear a watch on his wrist when his existing pocket watch kept good time and could be accessed quite freely? However, during World War I, the British Army issued wristwatches to its troops so that they could synchronize attacks while keeping their hands free for weapons. Suddenly, it became fashionable in Britain to show support for the "boys in the trenches" by wearing wristwatches. Now, of course, you rarely see a pocket watch.

By the early 1990s, technology had begun to support the wearing of digital, full-function computing devices. One organization investigating the use of these devices was the Wearable Group of Carnegie Mellon University headed by Dan Siewiorek. They viewed a wearable computer as a tool to support workplace functions, with the workplace epitomized by locales where aircraft and other large vehicles were maintained—out of doors or within large buildings such as hangars or railroad roundhouses.

The focus on use in a workplace meant that ease of use and design sophistication were primary. The Wearable group conducted experiments with computers designed and constructed by students in actual workplaces. The success of these experiments created the demand that Inmedius was organized to exploit.

A second group, operating at the same time and centered at the Media Laboratory of the Massachusetts Institute of Technology, styled themselves "borgs." They viewed the wearable computer as a consumer product designed to change the lives of those who wore it. They wore their computers all of the time and were interested in innovative uses of them and in memory support applications. One example was using the conductivity of the skin as a network medium and having two computers exchange business cards when their wearers shook hands.

By the late 1990s, the two groups were collaborating to make wearable computers a viable academic discipline. Various commercial companies had begun to offer computers and head-mounted displays, and large commercial concerns had begun to show interest. Now, with the increasing miniaturization of hardware and the increasing sophistication of software (as evidenced by this chapter), wearable computing can only become more prevalent.

— LJB

17.1 Relationship to the Architecture Business Cycle

Figure 17.1 shows the Architecture Business Cycle (ABC) as it pertains to Inmedius and the Luther architecture. The quality goals of re-usability, performance, modifiability, flexibility of the end user device, and interoperability with standard commercial infrastructures are driven, as always, by the business goals of the customer and the end user.

INFLUENCES ON THE ARCHITECTURE

The next sections elaborate on the things that influence the Luther architecture.

End Users. Inmedius's business is providing computer support for front-line workers. Figure 17.2 shows such a worker utilizing one of the hardware configurations supported by Luther applications. The worker is performing an industrial process, the steps of which are displayed on the head-mounted display apparatus that he is wearing. The computer is worn on the user's chest and uses a dial as its primary input device. The process is described in a manual stored on the back-office computers, and the manual pages are served to the worker as various steps of the process are completed, which can number more than 500. The worker

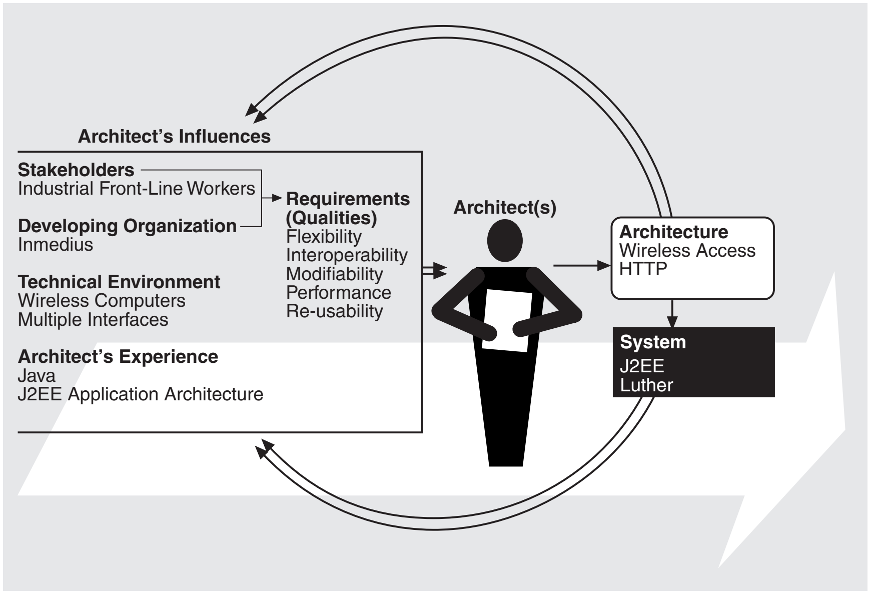


FIGURE 17.1 The ABC as it pertains to Inmedius and Luther

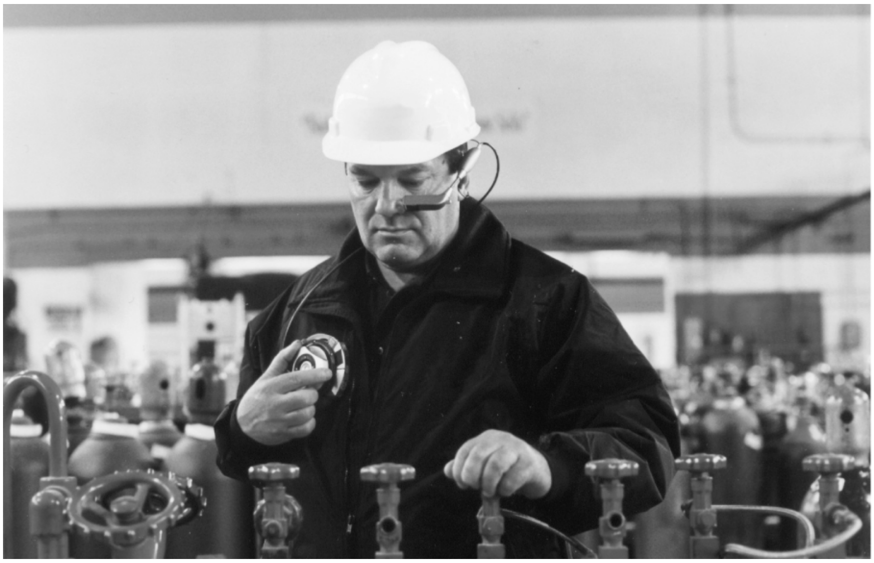


FIGURE 17.2 A field service worker using the Inmedius solution. Courtesy of Inmedius Corporation.

reports the results of portions of the process to the back office via the system. A part may be replaced, for example, and the part number is reported so that the inventory can be adjusted and any quality-control ramifications analyzed.

The workers may need to use one or both hands to perform the process, so those hands are not available for computer input. Further, workers may need to be mobile to carry out the tasks.

Different processes and customers may require different hardware configurations because requirements, such as mobility and the number of hands available for computer input, can vary.

Developing Organization. If the Luther architecture can facilitate the development of complex enterprise solutions in a fraction of the time they take to develop as standalone, "stovepipe" systems, Inmedius gains a significant competitive advantage. To achieve this, the company must meet increasingly shorter time-to-market for enterprise solutions. The development cycles for these solutions have to be in the low single-digit months for Inmedius to remain competitive in its target markets.

Solution development must be performed quickly and frugally by a few tens of engineers. The quality of the delivered solution must be high to ensure customer satisfaction. Also, the delivered software artifacts must be easily modifiable so that corrections and enhancements require little effort by Inmedius and do not compromise the integrity of the original solution's architecture.

Technology Environment. Luther has been influenced by developments in both software and hardware. As we discussed in Chapter 16, J2EE provides enterprise solutions for commercial organizations. It was a good fit with the Luther requirement to interoperate with back-office processes. J2EE also facilitates the packaging of domain-specific application capabilities into re-usable components that can be combined in different ways.

In addition to software influences, emerging hardware technology has influenced Luther—specifically, in the need to support small wireless computers with voice input capabilities and high-resolution, head-mounted displays. On the other hand, differing environments may require different types of devices, each with its own set of capabilities. This imposes a requirement that Luther be flexible with respect to the types of user interfaces supported.

INFLUENCES ON THE ORGANIZATION

The influences of Luther on the organization are in the areas of organizational structure, software developers' experience, and business approach.

Organizational Structure. Prior to Luther, Inmedius was a solution factory, with each solution developed as a stovepipe application for a specific customer. Organizationally, the Solution Group was the largest engineering group in the company. Luther's development created the need for a Products Group (containing a Component Development Group) to engineer and maintain the domain-specific component capabilities the Solution Group uses to create its solutions for customers. The Product Group is concerned with generalized capabilities for markets, whereas the Solution Group is concerned with specific applications for individual customers. This is an instance of a two-part organizational structure for software product lines, as described in Chapter 14 and illustrated by CelsiusTech case study in Chapter 15.

Software Developers' Experience. Prior to Luther, Inmedius was staffed with experienced and sophisticated software developers, who nonetheless had a number of new criteria to satisfy in Luther's development:

- Learning the Java programming language
- Becoming Sun Java Programmer Certified
- Learning the J2EE application architecture
- Learning how to package capabilities as J2EE/EJBs
- Learning how to create Java servlets and JavaServer Pages
- Learning how to use the various J2EE services provided by J2EE implementations

Business Approach. The Luther architecture has had a dramatic effect on the way Inmedius does business. As we said in Chapter 14, single-system solutions require a large amount of resources, and this resource drain and the stovepipe mentality associated with single system development inhibits global thinking. The move to a product line based on Luther enabled Inmedius to begin thinking about product lines instead of focusing on individual systems. Furthermore, as we saw with CelsiusTech, new markets became available to Inmedius that could be seen as generalizations of existing markets, not only in a business sense but also in a technical sense.

17.2 Requirements and Qualities

The Luther architecture was designed to meet two sets of complementary requirements. The first set governs the applications to be built—namely, enterprise applications for field service workers. These requirements are directly visible to customers, since failure to meet them results in applications that do not perform according to expectations—for instance, an application that may work correctly but perform poorly over a wireless network. The second set of requirements involves introducing a common architecture across products. This reduces integration time, brings products to market faster, increases product quality, eases introduction of new technologies, and brings consistency across products.

Overall, the requirements can be separated into six categories:

- Wireless access
- User interface
- Device type
- Existing procedures, business processes, and systems
- Building applications
- Distributed computing

Wireless Access. Field service workers must move about while performing their tasks. Furthermore, they must move about in an environment rich in machines, hazards, and other people. In order to interact with back-office systems, the devices used by workers must access remote servers and data sources without being tethered by a landline to a local area network. Because of the variety of Inmedius customers, these wireless networks may need to be of varying capacity and availability.

User Interface. Part of the Inmedius competitive advantage is its high-fidelity user interfaces, which allow a worker to focus on the task at hand without being hindered by the interface or the access device. Different devices have different screen footprints, and the Luther architecture must facilitate the display of meaningful information on each of them. This does not mean constructing a single user interface and adapting it to all device types. Instead, Luther must support the rapid construction of interfaces that filter, synthesize, and fuse information in ways that are displayable on a particular device and useful to its user.

Variety of Devices. Field service workers use a variety of computing devices in the field. No one device will suffice for all field applications, and each has limitations that must be addressed by the Luther architecture. Inmedius must engineer performance-enhancing solutions to run on all of these devices, which include:

- Personal data assistant (PDA) devices such as Palm Pilot, Handspring Visor, vTech Helio, IBM WorkPad, and Apple's Newton and MessagePad 2000
- Pocket PC devices such as Compaq iPAQ, Casio EM500, HP Jornada, and Phillips Nino
- Handheld, pen-based tablets running Windows CE such as Fujitsu Stylistic and PenCentra and Siemens SIMpad SL4
- Handheld Windows CE PC devices with pen and keyboard such as Vadem Clio, HP Jornada 700 series, NEC MobilePro, Intermec 6651 Pen Tablet Computer, and Melard Sidearm
- Wearable computing devices such as Xybernaut MA-IV, Via family of products, and Pittsburgh Digital Greenhouse's Spot

Different classes of device have different memory footprints, processor speeds, and user input devices that can radically affect a user's interaction style from one class to another. For example, a wearable computer can bring the power of the desktop computer into the field, making client applications as sophisticated there as they are in the office. Users in this case also have a plethora of input devices to choose from, including keyboard, voice, pen, and custom devices.

On the other hand, the processor speeds, memory footprints, and available input devices for the PDA class are severely limited, which means that user interactions that can be engineered for these devices are also constrained. Still, PDAs are extremely important in the various contexts in which field service workers perform their tasks. The Luther architecture must address the variability of the users' interaction styles, which are limited by differences in hardware capability among the device classes.

Existing Procedures, Business Processes, and Systems. Field service workers are only one part of most enterprises. Information gathered by them must be stored in the back office; instructions for them come, partially, from outside the field; and many applications already support existing business processes.

To respond to these needs, the Luther architecture must intergrate its functions with a worker's existing procedures and processes, enable applications to be hosted on servers and databases from many vendors, and simplify the integration of applications with legacy systems

Building Applications. Enabling faster construction of applications is one of the main motivations for Luther. There are a number of aspects to this goal, including:

- Encouraging software re-use and making it easier for applications to work together. This avoids wasting valuable resources to "re-invent the wheel."
- Enabling a build-first, buy-later strategy for enterprise functions (e.g., work flow).

- Providing a stable platform for adoption of new features and emerging technologies that span applications, such as location sensing, automatic detection and identification of nearby physical objects and services, and advanced user interface features like synthetic interviewing.

Distributed Computing. The Luther architecture must provide enterprise application developers with a framework and infrastructure that even out the differences in client device capabilities and provide application servers with the following distributed application features.

- *Scalability.* The Luther server framework must facilitate scalability with no impact on performance. That is, the addition of any number of domain-specific components over time must have no impact on the performance of the application software, nor must it cause the re-engineering of client applications. In addition, client applications must be easily reconfigurable to make use of added capability. The framework must also support the ability of applications to discover new capability and to dynamically reconfigure themselves to make use of it.
- *Load balancing*. The Luther architecture must support load balancing in a distributed environment. Most of the computation in its applications will be performed on the server side, with the results sent to the client. As more and more clients access the capability from a given server, the application server infrastructure will have to detect heavy loads on a given server and offload processing to application server components located on different server nodes within the enterprise. Similarly, the enterprise environment application must be able to detect a node failure and shift to another application server in the enterprise to continue execution. In both cases, load balancing must be transparent to the user, and in the first case it must also be transparent to the client application.
- *Location independence*. To support load balancing, domain-specific application capability must be distributed, and the Luther architecture must support this. To be able to change locations dynamically, applications must be location independent.
- *Portability.* Enterprise application environments invariably comprise a set of heterogeneous server hardware platforms. The Luther architecture framework will have to allow the software to run on myriad platforms in order for enterprise applications to work.

17.3 Architectural Solution

The main architectural decision made in response to requirements was that Luther would be constructed on top of J2EE, which has the following advantages:

- It is commercially available from a variety of vendors. Components, such as work-flow management, that may be useful in Luther are being widely developed.
- HTTP becomes the basis of communication because it is layered on top of the TCP/IP protocol, which in turn is supported by a variety of commercial wireless standards, such as the IEEE 802.11b. Any Web-based client can be made mobile given the appropriate wireless LAN infrastructure. Most of the devices that must be supported by Luther can support HTTP.
- It separates the user interface and allows the *user experience* paradigm to be implemented. This paradigm proposes that the computer and its application be another, noninvasive, tool for the field service worker. It must be a natural extension of the way that tasks are performed, yet provide performance-enhancing benefits for both the field service worker and the organization.

  The paradigm goes on to say that multiple views of an enterprise application should be developed, each for a particular field service worker's role. A view is tailored to that role to enhance performance and job satisfaction, and filters, fuses, synthesizes, and displays the appropriate information for it. The view includes the use of role-appropriate input devices.

  For example, if a keyboard is not appropriate, perhaps voice input can be used. If the environment is too noisy, perhaps a custom input device like a *dial* is used, which a user can turn (the dial is mounted on the user's uniform as shown in Figure 17.2) to navigate through links, buttons, radio buttons, and other similar UI widgets in the client application to make them hot. In the middle of the device, the user can tap an "enter" key to select the link, click the button, and so forth. This device can be used in the most rugged environments, for example, even when a worker is wearing thick gloves.

  "Separating the user interface" is a tactic we saw for usability in Chapter 5. In Luther it brings the flexibility to change the user interface and adapt it to different devices and needs as well, which is a kind of modifiability. Again we see that some tactics apply to achieving more than one kind of quality attribute.
- It supports the separation and abstraction of data sources. The user experiences require the filtering, fusion, synthesis, and display of data that comes from multiple, disparate data sources. Some of these data sources are database management systems, others are legacy applications built on enterprise resource planning systems that encapsulate corporate data. Inmedius realized that by abstracting and separating data sources from the applications that use them and by providing them with well-defined, standard interfaces, the applications remain true to their defined abstractions and thus are re-usable. Additionally, some interfaces are industry standards, such as JDBC/ODBC, which allow the data sources themselves to be treated as abstract components that can be swapped in and out of the enterprise application at will.

Figure 17. 3 shows how a Luther application interacts with its environment. (It does not show the J2EE elements; we will discuss the mapping of the application to J2EE shortly.) First, note the (n:1:m) relationship among user interfaces,

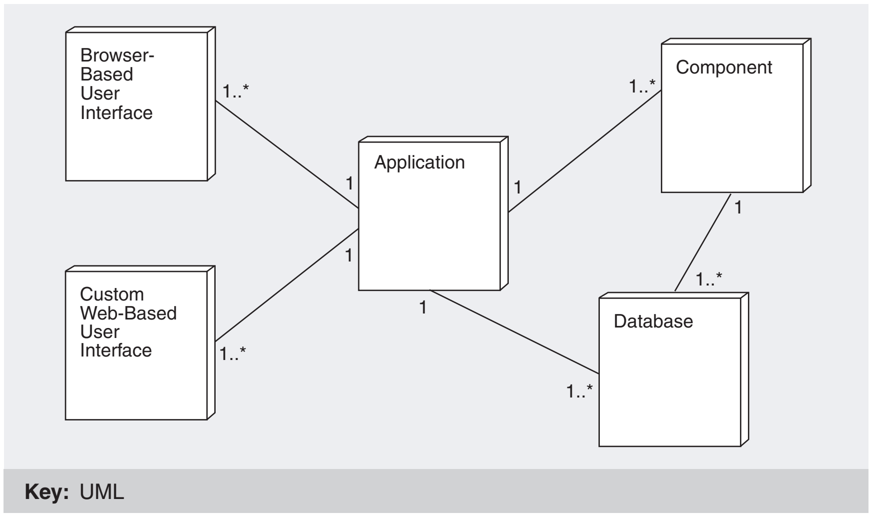


FIGURE 17.3 Deployment view of a Luther application

applications, and what Inmedius calls "components," that is, building blocks for application functionality. A Luther application is thin; much of its business logic is assembled from existing components, and it is not tied to any specific user interface. Essentially, the application code contains these three things:

- Session state definition and management
- Application-specific (i.e., nonreusable) business logic
- Logic that delegates business requests to an appropriate sequence of component method invocations

The application does not have a main method; it has an application programming interface (API), which represents the features and functions available from the application to its user interfaces. The user interface is independent of the application. It may expose any subset of features appropriate for the target interface device. For instance, if a user interface is created for a device with a microphone and speaker but no display, it does not expose features of the application that require graphics.

Now we turn to an in-depth discussion of the three main elements shown in Figure 17.3: the user interface (UI), the application, and the components.

USER INTERFACE

The strategy for developing user interfaces in the Luther architecture is as follows. First, a combination of domain experts, cognitive psychologists, and graphic artists work with a client to understand the various workers' tasks and roles, the work environments, and the necessary interface characteristics of the desired access devices. Next, they craft the user experience based on these constraints, with the result being a storyboard, screen shots, and a prototype. The point is that the result of the design process must be a high-quality, high-fidelity user experience, as described before. This is essential, since the application is meant to augment the user's existing work procedures and be a natural extension of the work environment. Consequently, the task of developing the user experience is delegated to the people best suited for it—domain experts who understand the task and the work environment; cognitive psychologists who understand how people think, reason, and absorb information; and graphic artists who are skilled at presenting information in an effective and appealing manner.

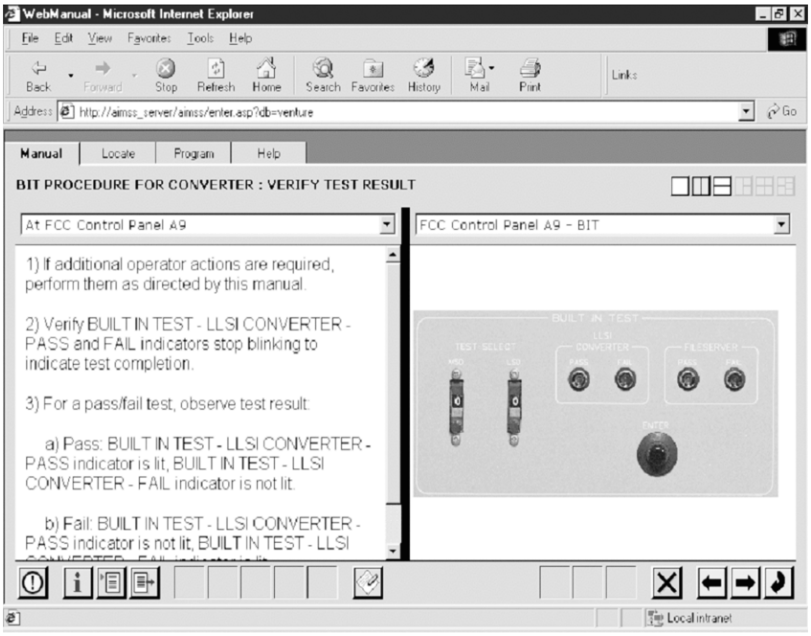
The next step is to take the output of the design process—the storyboard, screen shots, and prototype—and quickly convert this to a working user interface on real devices. Here, the architecture must support the integration of custom user experiences. Integration must be rapid, and it should enable creation of common portions and re-use of software to the greatest extent possible, all the while preserving the integrity and fidelity of the original user experience design.

Turning a user experience design into a working user interface is complicated by many factors. First, a variety of client devices must be supported. This includes an assortment of mobile devices with varying screen sizes, operating systems, and input devices. A user interface that performs well on a desktop PC is severely limited by the smaller screen, less memory, and less functional support on a mobile device. Some mobile devices, for example, have no keyboard or mouse support, rendering user interfaces that require them useless. A second factor is the limitations introduced by technology. For instance, certain types of user interaction or information display are cumbersome over HTTP and may lead to poor performance.

In the end, there may be multiple client devices and user interfaces for any given application. The software architecture must be flexible enough to deal with multiple clients that differ greatly from one another. In Figures 17.4 and 17.5, the two types of user interface implementation supported by Luther are shown—namely, browser-based clients (Figure 17.4) and custom, Web-based clients (Figure 17.5). Figure 17.6 refines the view given in Figure 17.3 and illustrates the structure of each type.

Browser-Based Clients. Browser-based user interface clients correspond simply to browser-based clients in J2EE. They are not restricted to Web browsers, however, but equally support other forms of markup such as a Wireless Markup Language (WML) over a Wireless Application Protocol (WAP) for cellular phones. While the markup language is different in this case (i.e., WML), the same mechanisms for delivering the content can still be employed—that is, a combination of servlets and JavaServer Pages (JSPs).

Browser-based clients use standardized methods for the exchange of information (i.e., commercial Web browsers on the client side, HTTP over TCP/IP as the network protocol, and JSPs and Java servlets on the server side), and use common data formats (i.e., hypertext documents and style sheets). To make the


WebManual - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print
Links
Address http://aimss_server/aimss/enter.asp?db=venture
Go
Manual Locate Program Help
At FCC Control Panel A9
FCC Control Panel A9 - BIT
BUILT IN TEST
TEST SELECT
LLSI CONVERTER
FILESERVER
PASS
FAIL
ENTER
Local intranet

BIT PROCEDURE FOR CONVERTER : VERIFY TEST RESULT

1) If additional operator actions are required, perform them as directed by this manual.

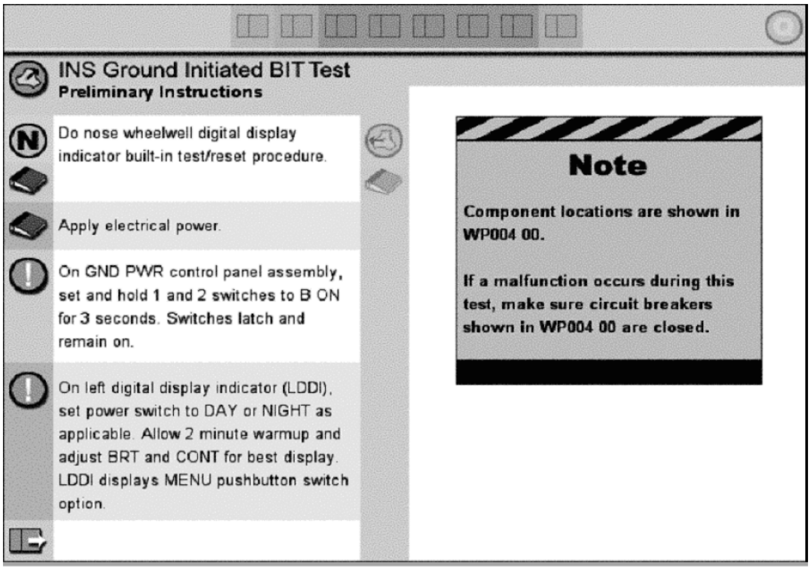
2) Verify BUILT IN TEST - LLSI CONVERTER - PASS and FAIL indicators stop blinking to indicate test completion.

3) For a pass/fail test, observe test result:

a) Pass: BUILT IN TEST - LLSI CONVERTER - PASS indicator is lit, BUILT IN TEST - LLSI CONVERTER - FAIL indicator is not lit.

b) Fail: BUILT IN TEST - LLSI CONVERTER - PASS indicator is not lit, BUILT IN TEST - LLSI

FIGURE 17.4 Browser interface for maintenance procedure



INS Ground Initiated BIT Test

Preliminary Instructions

Do nose wheelwell digital display indicator built-in test/reset procedure.

Apply electrical power.

On GND PWR control panel assembly, set and hold 1 and 2 switches to B ON for 3 seconds. Switches latch and remain on.

On left digital display indicator (LDDI), set power switch to DAY or NIGHT as applicable. Allow 2 minute warmup and adjust BRT and CONT for best display. LDDI displays MENU pushbutton switch option.

Note

Component locations are shown in WP004 00.

If a malfunction occurs during this test, make sure circuit breakers shown in WP004 00 are closed.

FIGURE 17.5 Custom Web-based user interface

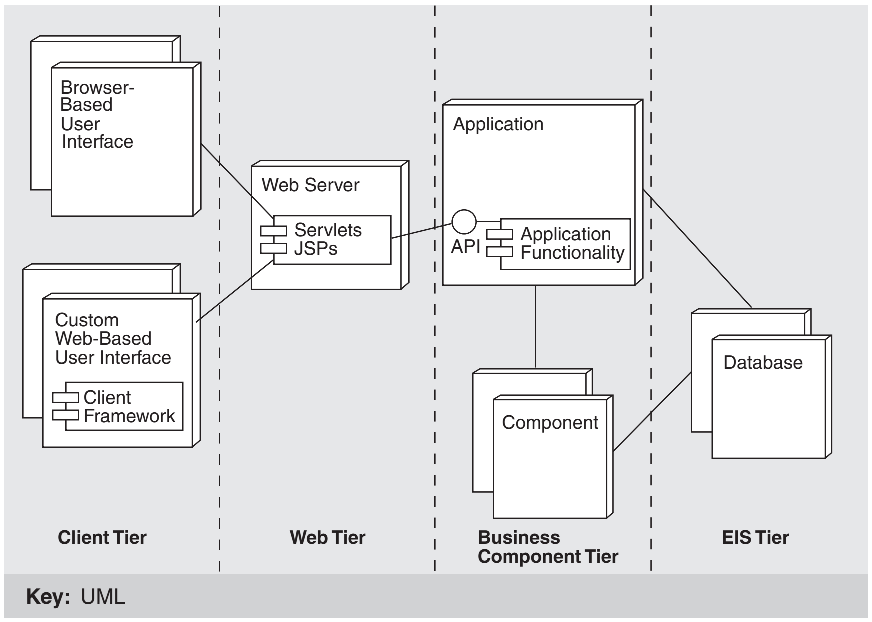


FIGURE 17.6 User interface as a C&C view overlaid onto a deployment view

client thin, most of the presentation logic is implemented on the server, which increases the chance of creating an interface that is portable across browser vendors and versions.

Browser-based clients are primarily intended for

- devices that support browsers and have traditional input devices such as pens, keyboards, and mice.
- applications that display content easily representable with markup languages and renderable by a browser, perhaps augmented with plug-ins.

Browsers were originally designed for desktop computers—making PCs their optimum target device—but today's mobile devices also support them.

Certain restrictions limit the use of browser-based interfaces. In design, for instance, they do not always make the best use of valuable resources, such as the available screen real estate, and the browser model supports only limited types of user interactions built around the HTTP request/response cycle. Also, browser-based interfaces are not suitable for all mobile devices because no browsers exist for certain ones; when they do, they may lack support for essential features such as frames, graphics, and JavaScript.

Custom Web-Based Clients. Custom Web-based user interfaces are more complex. This type is different from a custom client, which in J2EE is a standalone program that implements all of the presentation logic and uses the remote invocation method (RMI) over the Internet Inter-ORB Protocol (IIOP) to interact directly with the business logic (i.e., EJBs). A custom Web-based client is also a standalone program but, unlike a custom J2EE client, it uses HTTP to communicate with the server and interacts with Web-tier entities, such as servlets and JSPs, in the same way as a browser-based client does.

Custom Web-based clients are written in a native development environment for a specific device or class of devices. Since the user interface is a standalone program, this gives the UI designers the most freedom in terms of user interactions that can be supported, and can lead to the best use of resources such as screen real estate. The downside is higher cost of development.

The Luther architecture has tried to minimize the amount of native code that must be written to create a custom, Web-based client, with a client framework that supports interfaces of this type, as shown in Figure 17.6. Basically, the framework standardizes elements that are needed across applications, including session management, authentication, and support for creating and sequencing presentation logic on the client, the Web container, or both. In essence, the client is a thin, standalone program that creates and lays out the native UI widgets. It also implements a small portion of the presentation logic such as input validation and sorting of tabular displays. Just as with browser-based clients, the bulk of the presentation logic is implemented on the Web tier in components managed by the client framework.

Custom, Web-based clients have advantages over other types of custom user interfaces. First, they are thin. In other words, compared to a fat client (i.e., a custom program where all of the presentation logic is implemented in the client tier), they are smaller, easier to maintain, and easier to port across devices. Second, they use HTTP to interact with the Web tier, unlike J2EE custom clients that use RMI over IIOP. This makes them more appropriate for non-Java implementations and simpler to implement over wireless networks.

Creating a custom, native user interface for each application on each device is too costly, even for a small number of devices. This is avoided by sorting interface devices into classes by characteristics. For each device class, a high-fidelity interface is designed and implemented as described previously. The client framework eases the burden of implementing this interface across a device class. Likewise, by implementing a significant portion of the presentation logic in the Web tier, client devices in the same class can use this software and thus share a significant portion of their implementation. Finally, the client framework introduces features that allow a device to advertise its interface characteristics. This information is made available to the presentation logic on the Web tier so that small adaptations can be made to the content before it is delivered to the client.

APPLICATIONS

In the Luther architecture, the application is responsible for uniting the system into a single functional entity and exposing an API for interacting with it. The user interfaces call into this API to provide these features to an end user.

Applications reside between any number of user interfaces and any number of components. An application ties together *m* components and exposes the aggregated “application” functionality to *n* user interfaces. The applications are “user interface agnostic,” meaning that they expose functionality that any user interface can use. Each interface can expose all or a subset of this functionality as appropriate. For example, a user interface running on a mobile client like a Windows CE device cannot expose the administrative features you would expect to find in a desktop version. The idea is to expose all functions that can be performed in the system; each user interface decides which of these functions to expose to the user and how to expose them.

The requirement for rapid development and deployment leads to designing the application to be as thin as possible. This is achieved by delegating the bulk of the business work to components (discussed in the next section). The criterion for moving application code into a component is simple: Is the functionality re-usable? If so, it should be generalized (to increase re-usability) and implemented as a component. On the other hand, if a piece of functionality is not likely to be re-used, it is incorporated into the application.

The essential elements of an application include the following:

- *Application programming interface.* A façade for the functions exposed by the system to the user interfaces. Note that data passed through the API is generic (e.g., XML) rather than presentation specific (e.g., HTML).
- *Session state.* Initialized when a user authenticates, a session state exists until the client program terminates. J2EE simplifies state management, since the containers support authentication and authorization along with storage and retrieval of the session state. The application simply determines what data needs to be persisted across requests and makes the appropriate calls to store and to retrieve it.
- *Application-specific business logic.* Any logic that is unique to this application and that cannot be re-used in other applications.
- *Delegation to components.* Code for delegating work to components. In general, this is achieved via the *Business Delegate* design pattern.[1]

[1] A business delegate acts as a façade for a component—it locates the component and makes its functions available to the rest of the application. In this way, only the business delegate need be concerned with how to locate and access the component, hiding these details from the rest of the application. For instance, if a component is implemented as an EJB, the business delegate performs the necessary Java Naming Directory Interface (JNDI) look-ups and narrows the EJB remote interface; the fact that the component is implemented as an EJB remains hidden. The application is *not* responsible for component lifecycle management because the J2EE containers perform this function. However, since it does the delegating, it has to choose which component(s) to use. The application also includes logic that manages component interactions and inter-relationships. Clearly such logic belongs in the application. Following this rule simplifies the implementation of the components and minimizes inter-dependencies.

These elements result from application of the "anticipate expected changes" tactic and the associated "separate user interface" tactic for modifiability.

A new user interface can be created without changing the application layer or components at all. A new implementation of a component can be integrated into the system without affecting the application layer or the user interfaces. New functionality can be added to the system by incorporating another component, adding the necessary API methods to the application layer, and adding (or not) new features to each user interface to expose the new functions.

COMPONENTS

The intention behind a component is that it represent an element for re-use. The strategy is therefore to create a library of components from which applications can be easily and quickly synthesized to create specialized solutions for customers. The library contains *core* components related to the client and server frameworks; *domain-specific* components for domains, such as maintenance, repair, and overhaul; and *generalized capability* (i.e., utility) components that applications might need to round out functionality, such as security, authorization, and user management.

Inmedius's strategy is to evolve a large library of core, domain-specific, and generalized capability components for the Luther architecture framework and for specific customer domains. Application development therefore becomes an exercise in creating business logic that composes the necessary set of capability components into a customized solution for the customer.

Crafting common components is a central theme in the construction of software product lines and represents an intense application of the "abstract common services" tactic for modifiability—in this case, the ability to produce new solutions.

Component Design. The strategy for designing components is to use design standards, wherever possible, for the component's API and behaviors. For example, the Inmedius work-flow component (described later) is an instantiation of the Workflow Management Coalition's specification for work-flow functionality and behavior. This design strategy allows Inmedius to replace its own components with any other vendor's components that adhere to the same capability specifications. It facilitates the expansion of the Inmedius component library to include such components.

Capability Partitioning. It may be that the library does not contain a capability component required by a given application under development. A decision must be made as to whether to design and implement the capability as part of the application itself or as a new, re-usable component.

The key design heuristic is whether the capability is a part of the application's business logic for this specific solution or an instance of a more general capability that might be used in other applications.

Component Packaging. Any application in Luther uses the J2EE environment and its services. Given this constraint, components in that environment can be packaged as EJBs; Java bean components; individual Java class libraries, applets, servlets, or some combinations of these. In other words, a component is *not* synomous with an EJB, but rather can be packaged in a variety of ways.

The strategy for packaging a given capability depends on the J2EE services used as well as the tradeoffs among a number of key factors (e.g., frequency of inter-object communication, location of object instances, and need for J2EE services such as transactions and persistence of object state over multiple user sessions). For example, communication with an EJB is via RMI, a heavyweight communication mechanism. In some J2EE containers, communication with EJBs is optimized (into local method calls) if the communication is within the same Java Virtual Machine (JVM). However, since optimization is not required of a J2EE container, communication between EJBs always has the potential of being costly, so must not be taken lightly if performance is an issue. An alternative is to create a Java class library to avoid the need (and overhead) for RMI. However, this also forces the component to take on additional responsibilities previously handled by the container, such as creation and deletion of component instances.

Objects associated with a component must be made accessible to a user for the extent of a session. They may change during that time but the data must persist and be consistent across sessions. Consequently, components often require transactions. Multiple users may be accessing the same objects simultaneously, potentially for the same purpose, and this has to be handled gracefully. Supporting transactions also makes graceful recovery from failure easier by leaving the database in a consistent state.

As described in Chapter 16, the EJBs model supports several bean types, including entity beans, session beans, and stateless session beans. The different types are intended to support different forms of business logic, and they are handled differently by the container. For instance, an entity bean allows the choice of managing persistence yourself via callbacks supported by the container (i.e., bean-managed persistence) or having the container do it for you (i.e., container-managed persistence). In either case, a significant amount of overhead is involved, which limits the practical use of an entity bean to long-lived business entities characterized by coarse-grained data accesses.

What the J2EE Container Provides. There are several capabilities that applications require, such as transaction support, security, and load balancing. These capabilities are very complex (indeed, many corporations organize their entire business around offering them) and are outside the scope of a given application or application domain. One of the main drivers in Inmedius's decision to build Luther using J2EE was the fact that commercially available J2EE-compliant containers provide these features, so Inmedius does not have to implement them.

Many of these capabilities can be configured for an individual EJB at application deployment time, or they are provided to the EJB transparently by the

J2EE container. In either case, the EJB developer does not have to embed calls to them directly into the code, so they can be easily configured for a given customer. This not only facilitates the creation of application-independent EJB components but also guarantees that the components will successfully run within all J2EE-compliant containers.

- The EJB container provides transaction support both declaratively and programmatically. The component developer can programmatically interact with the container to provide fine-grained, hard-coded EJB transaction support. The developer may also declaratively specify, via the deployment descriptor, how EJB methods should behave within transactions. This allows transactions to behave differently in different applications without the EJB having to implement or configure them directly in the code.
- J2EE provides an integrated security model that spans both Web and EJB containers. Like transaction support, security features can be used either declaratively or programmatically. If methods are written to include definitions of the permissions required to execute them, the developer can specify which users (or groups of users) are allowed method access in the deployment descriptor. Otherwise, entries in the deployment descriptor can be used to declaratively associate access rights with methods. Again, this allows the component methods to have arbitrary permissions determined by the application, without having to rewrite the component.
- The EJB container also provides transparent load balancing. EJB instances are created and managed by the container at runtime; that is, they are created, activated, passivated, and removed, as necessary. If an EJB has not been accessed recently, it may be passivated, meaning that its data will be saved to persistent storage and the instance removed from memory. In this way, the container effectively performs load balancing across all of the instances in the container to manage resource consumption and to optimize system performance.

What the Component Developer Provides. The component developer provides the client view, or API, of the component, as well as the component implementation. With a simple EJB, this amounts to writing only three classes: the home interface, the remote interface, and the implementation class.

The component developer also provides definitions of the data types exposed to clients through the API. These are implemented as additional classes, and often take the form of value objects that are passed back and forth to an EJB through the API.

EXAMPLE OF A RE-USABLE COMPONENT: WORK FLOW

In this section, we will look at one of the re-usable capability components developed for the Inmedius component library, the issues it raised, and the decisions made. The work-flow component, the largest of the capability components thus far created, is an example of the how a generalized capability is engineered and packaged for inclusion in the Luther architecture.

Design Rationale. The primary responsibility of the work-flow component is to allow a client to model a work flow and then move digital artifacts through it. The component must also allow clients to define resources and assign them to work-flow activities. Naturally, the component must be highly re-usable and extendable, which means that it should provide general work-flow capabilities; provide a clear but generic model of operation to the applications that will use it; and be agnostic with respect to the digital artifacts that may move through a particular work-flow instance. The creation of a full-functionality work-flow component requires complex idioms such as branching, merging, and looping. Generally implementing a work-flow capability is a very large, complex task.

Inmedius faced a dilemma in that there was a legitimate need for work-flow capabilities in its applications but many factors, such as the following, prevented their complete implementation:

- The size and complexity of a complete work-flow capability was beyond Inmedius's resources.
- Complete work-flow capability was not a core business objective or a core competency.
- Other companies had built far more complete solutions.

The long-term solution was to form alliances with organizations that provide componentized work-flow capability for J2EE applications. Until that happened, however, Inmedius had to implement a subset of capability in order to deploy solutions.

Thus, the strategy was to design a component that could be easily swapped with a more complete one from another organization at a later time. This created the need for a standardized work-flow component interface. Notice how the ABC works in this case. The design of the Luther architecture opened up a new business opportunity (work-flow management) and Inmedius had to make an explicit business decision to enter this market. Inmedius decided that it was outside its core competence.

The Workflow Management Coalition has developed of a set of functional and behavioral work-flow specifications that have been recognized by the work-flow community. Inmedius architects built its component to those specifications, yet implemented only the functionality that is necessary for use by the current applications.

This strategy leveraged the knowledge and experience of the work-flow community and all of its activities. The community had already defined business objects and relationships between objects, so Inmedius did not have to reinvent them. Second, by adhering to Workflow Management Coalition specifications, Inmedius could now replace its work-flow component with that of another vendor, with minimal effort if a customer required a certain degree of functionality not provided in the Inmedius component.

Two Workflow Management Coalition specifications describe the two primary elements: the definition of a work-flow model and the representation of its runtime instances (see Figure 17.7). The work-flow model definition is made up of one or more process definitions, each of which consists of activity definitions and transitions between those activities and all participating resources. In each process definition, a process manager oversees all runtime instances of a specific process definition; each runtime instance maintains state as to which activities have been completed, which are active and who is assigned them, and context data that the work-flow component needs to make decisions while the process is active.

One issue of concern to Inmedius was concurrency. Should more than one user be permitted to modify a work-flow model definition at one time? If active runtime instances exist, should a user be permitted to modify a work-flow model definition? Should a user be permitted to start a new work flow if its definition is being modified? Given the implementation, a yes answer to any of these questions posed a significant problem because of the relationship between a definition

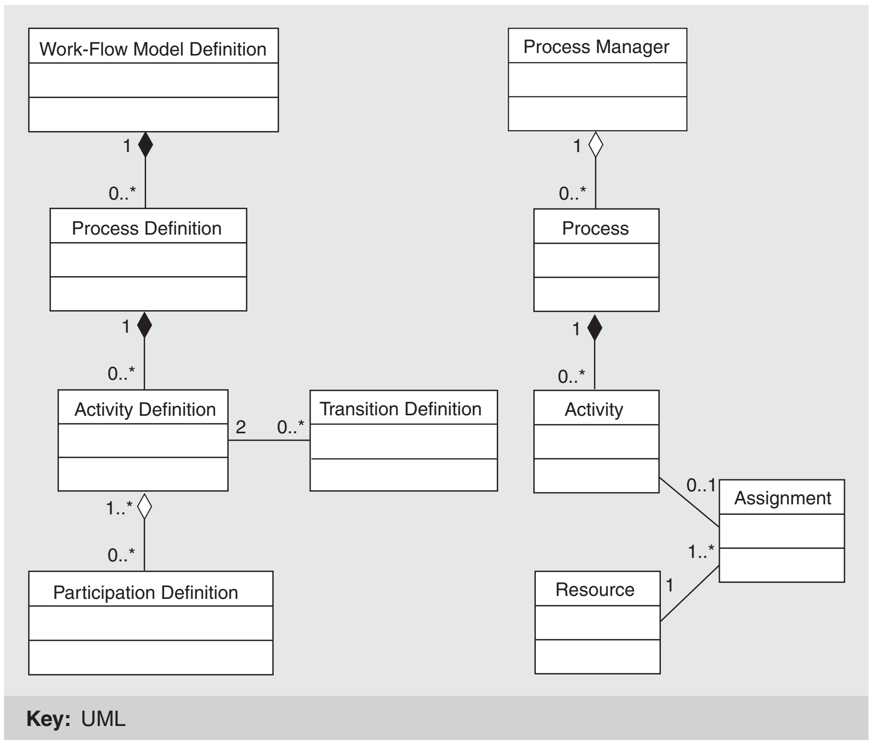


FIGURE 17.7 Class diagram for the work-flow component

and its runtime instances. As a result, any solution would have to prohibit these situations from occurring.

Because the underlying problem in each of the situations described before revolved around modifying the work-flow model definition, the solution was to associate a lock with it. In order to modify a definition, a user must obtain a lock. Only one lock can exist for a given definition and it cannot be obtained if the definition has any associated active runtime instances. In addition, a new runtime instance cannot be started if the work-flow model definition is locked.

Packaging. The work-flow component is packaged as two EJBs: a stateless session bean for managing instances of work-flow model definitions and a single entity bean for managing the definition itself (see Figure 17.8). The decision to package the component this way was based strongly on the characteristics of the different EJBs.

Entity EJBs implement abstractions in an application that represent shared resources, where persistent object data is shared among many components and users. The work-flow model definition represents just such a single shared resource—namely, a definition of a process that can be instantiated many times. In Inmedius applications, any user in any location can start a new process based on this single work-flow model definition and participate in its activities.

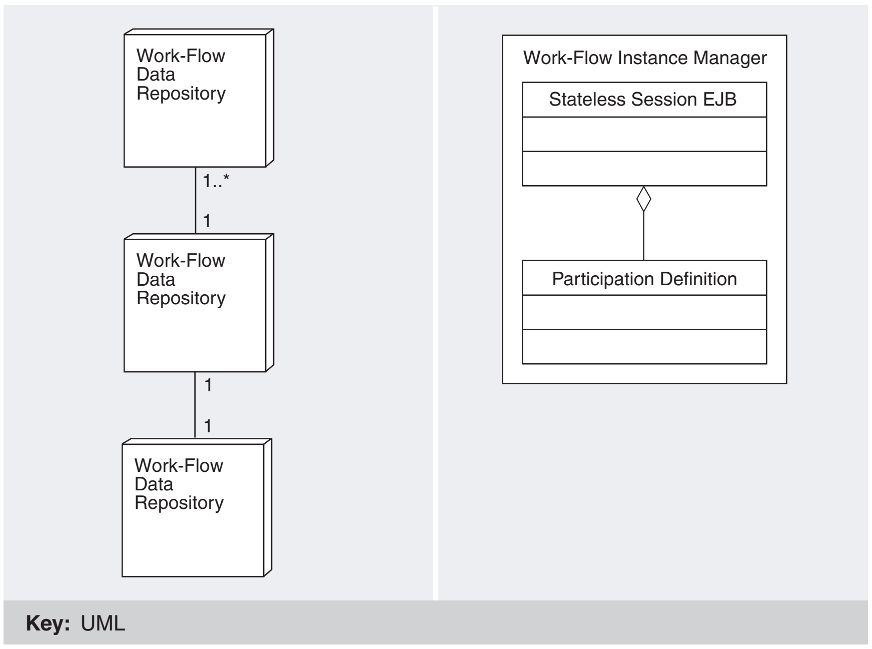

Work-Flow Data Repository
1..*
1
Work-Flow Data Repository
1
1
Work-Flow Data Repository
Work-Flow Instance Manager
Stateless Session EJB
Participation Definition
Key: UML

FIGURE 17.8 Work-flow component packaging diagram

Session EJBs model state and behavior. The definition of new work-flow models, the creation of work-flow model instances, the creation of activities, the assignment of resources to activities, and the completion of activities, for example, are all services provided to users over the course of a work-flow instance life cycle or session. Therefore, work-flow instances are most naturally implemented by session EJBs.

Once it was decided to make the work-flow instance manager a session EJB, a decision had to be made as to whether to make the session EJB stateful or stateless. This depended on the characteristics of the state to be maintained. Typically, a stateful session EJB maintains state for a single client with whom it is having a dialog. However, the state of a runtime work-flow instance is not manipulated by just a single client but is updated by many clients, including those who participate in the actual work-flow process and managers who want to monitor the process and analyze its results. As a result, the work-flow instance manager was implemented as a stateless session EJB, which is more lightweight and scalable than a stateful session EJB and which persists the state in a database on behalf of a given client, where all the other clients have access to it.

Another design tradeoff concerned how to package the individual objects within a work-flow model definition. Should they be packaged as entity EJBs, or should they comprise Java classes packaged using some other structure, such as a library? Because these objects interact with and are dependent on each other, to package them as entity EJBs would constantly require locating and retaining multiple EJB handles in the application, creating much overhead. In addition, recall that any method invocation on an EJB is essentially an RMI call and can be quite costly. While most J2EE containers can determine if the method invocation is in the same Java Virtual Machine and therefore optimize it into a local method call, this is not guaranteed. For these reasons, the design decision was to create entity EJBs for coarse-grained abstractions in the application, such as the work-flow model definition, and to implement the finer-grained abstractions in the entity EJB itself as libraries of Java classes—all to reduce the overhead associated with the heavyweight entity EJB relationships.

An example of this type of design decision in the work-flow component was deciding where to locate the logic that determines whether to grant a request for a lock on the work-flow model definition. Originally, that logic was placed inside the entity EJB implementing the work-flow model definition. A request to lock the definition would be made directly to the entity EJB, which would determine if the lock could be granted (and, if so, lock it).

A problem became apparent when it came time to enhance the business logic so that a lock could be granted only if no active runtime work-flow instances existed. The methods that provided runtime work-flow instance information were defined on the stateless session EJB, the object interacting with the entity EJB. It did not seem right to pass a reference to the stateless session EJB into the entity EJB—first, because the entity EJB would be aware of the environment in which it exists (thus, hampering re-use); second, because any method invocations made by the entity EJB on the stateless session EJB would be RMI calls.

Another option was to use the data access objects of the entity EJB directly in order to retrieve the necessary information from the database. However, this would break the abstraction implemented by the entity EJB, forcing it to be responsible for something that it should not be responsible for and that is already the responsibility of another object. Lastly, there would be a duplication of code that would create maintainability problems.

The solution was to place the logic (i.e., that determines whether a request for a lock on the work-flow model definition is granted) in the stateless session EJB. The entity EJB now simply knows how to persist and retrieve locks to and from the database. When a request for a lock is received, the stateless session EJB determines if it can be granted and, if so, instructs the entity EJB to lock the work-flow model definition. This solution maintains the integrity of the abstractions implemented by the objects and eliminates unnecessary inter-EJB relationships.

Distributed and Detached Operations. When designing the component to support distributed and detached operations, a number of interesting issues arose, primarily about whether to support distributed concurrency of work-flow activities. Consider a scenario in which a work-flow model definition and its runtime instances are located across multiple servers. While J2EE transaction support can guarantee that no two users can violate work-flow rules if they access the same data in the same database, it cannot guarantee that rules will not be violated if two users access replicated data for the same work flow in different databases.

In this scenario, one user could lock a work-flow model definition in one location for the purpose of modifying it while another user was creating a new runtime instance of the same definition in another location. During data replication and synchronization among the distributed servers, conflicts might arise that could corrupt the work-flow data in the enterprise environment if not resolvable. To guarantee that work-flow rules would not be violated across multiple databases, additional functionality would be needed to resolve every type of conflict. Implementing this level of functionality was outside the scope of Inmedius's initial release. To meet the requirement, distributed and detailed operation scenarios had to be supported.

The system architecture and environment dictated the two scenarios of distributed and detached operations initially supported. In a distributed operation, a common repository is shared that itself supports transactions (e.g., a database). In other words, multiple instances of the application server may exist in several locations but each must access the same data repository that contains the work-flow model definitions and runtime instances. This is because the information used by the application server to determine whether work-flow rules have been violated is stored in the data repository. In detached operations, one installation (i.e., application server and data repository) is designated as the master installation and all others as subordinate instances. The work-flow model definition must be created and defined via the master and then replicated to all subordinates. Once a definition is distributed, it cannot change other than specifying who can participate in the defined activities. As runtime work-flow instances at the subordinate installations are created and eventually closed, these are replicated back to the master for historical purposes.

RAMIFICATIONS OF USING J2EE

This section discusses the rationale for several Luther decisions regarding the use of J2EE.

Decisions Made by Design versus Those Dictated by J2EE. When designing a system using the J2EE runtime environment, some decisions are left up to the designer and others are constrained by the J2EE rules and structure. For example, J2EE mandates where servlets, JSPs, and EJBs reside within a container—servlets and JSPs in the Web tier and EJBs in the EJB tier.

However, the Java 2 Enterprise Edition environment also provides the designer with some flexibility—for example, in implementing security (declarative versus programmatic), transaction support (declarative versus programmatic), and data access (container-managed versus bean-managed).

When designing a component, the designer has total control over functionality to allocate to a servlet, JSP, or EJB, and here the obvious choices might not always be the best. For instance, one of Inmedius's components supports collaboration between two or more users. Since this component represents re-usable business logic, the rules of component selection specify that it should be packaged as an EJB. Unfortunately, further analysis proved that this was not the correct design. Additional factors must be considered when determining how to map a component design onto the four logical tiers provided by J2EE, as shown in Figure 16.2.

Issues Introduced by the Multiple Tiers in the J2EE. One issue is performance. A major contributor to poor performance is the number of calls made from one J2EE entity (e.g., servlet, EJB) to another within a given transaction. Technically, each EJB method call is an RMI call, which can be very expensive. The implementation of coarse-grained EJBs and the elimination of inter-entity EJB relationships are two ways to address this issue and thereby ensure good component performance.

Another issue is transactions, which may be managed programmatically or declaratively. Obviously, managing transactions declaratively is somewhat easier because code does not have to contain begin and end transaction statements. However, developers must be mindful of how their J2EE entity will be used. The easy course is to require transactions for all methods. Unfortunately, this creates unnecessary runtime overhead if transactions are not truly needed. Another problem arises when methods on a J2EE entity do not require transaction support and the deployment descriptor enforces this. If another container involved in a transaction uses the J2EE entity, the transaction it has created will fail. Instead, the deployment descriptor should declare that the method supports transactions. Careful thought must be given to what aspects of a component require transactions to ensure correct operation, and these decisions must be mapped to a combination of the declarative and programmatic mechanisms supported by J2EE.

17.4 How Luther Achieved Its Quality Goals

All but one of Luther's quality requirements came from its customers: wireless access; flexibile user interfaces and devices; support for existing procedures, business processes, and systems; and for distributed computing. The only one that came from Inmedius was ease of building applications.

The primary decision in achieving these requirements was to use J2EE, but only in a particular fashion. The user interface was clearly and cleanly separated from the applications, standards were used whenever possible, and a re-usable library of components was to be constructed opportunistically. Table 17.1 shows the strategies and tactics used in this effort.

TABLE 17.1 How Strategy Achieves Goals

Goal	Strategy	Tactics
Wireless Access	Use standard wireless protocols	Adherence to defined protocols
Flexible User Interface	Support both browser-based and custom interfaces through HTTP	Semantic coherence; separate user interface; user model
Support Multiple Devices	Use standard protocols	Anticipate expected changes; adherence to defined protocols
Integration with Existing Business Processes	Use J2EE as an integration mechanism	Abstract common services; component replacement
Rapid Building of Applications	Use J2EE as a basis for Luther and construct re-usable components	Abstract common services; generalize module (in this case, J2EE represents the generalized module)
Distributed Infrastructure	Use J2EE and standard protocols	Generalize module; runtime registration

17.5 Summary

Inmedius develops solutions for field service workers. Such workers require high mobility with untethered access to computers. These computers are typically highly portable—sometimes with hands-free operation. In each case, systems require integration with back-office operations.

Luther is a solution that Inmedius constructed to support the rapid building of customer support systems. It is based on J2EE. A great deal of attention has been given to developing re-usable components and frameworks that simplify the addition of various portions, and its user interface is designed to enable customer- as well as browser-based solutions.

Reliance on J2EE furthered the business goals of Inmedius but also introduced the necessity for additional design decisions in terms of what was packaged as which kind of bean (or not). This is an example of the backward flow of the ABC, emphasizing the movement away from stovepipe solutions toward common solutions.

17.6 For Further Reading

The reader interested in wearable computers is referred to [Barfield 01] as well as the proceedings of the annual IEEE-sponsored International Symposium on Wearable Computers (*http://iswc.gatech.edu/*).

The business delegate pattern used in Luther can be found in [Alur 01]. The Workflow Management Coalition reports its activities on *http://www.wfmc.org*.

17.7 Discussion Questions

1. Many of the case studies in this book feature architectures that separate the producers of data within a system from the consumers of data. Why is that important? What kind of tactic is it? Compile a list of the tactics or design approaches used to achieve separation, beginning with the ones shown in this chapter.

2. A great deal of attention has been given to separating the user interface from the remainder of the application both in Luther and in our other case studies. Why is this such a pervasive tactic?