Introduction

Z47 is a distributed operating system providing the infrastructure for the execution of Z++ distributed applications. In order to accomplish this demanding requirement, **Z47** comprises of a large number of components, some of which are described in this document.

Despite its complexity **Z47** is slightly more than 500 Kilobytes in size. Furthermore, **Z47** is self-contained and mainly uses standard services of the host OS. This makes it readily available for all platforms.

The Graphical User Interface (GUI) is presented in an abstract object-oriented fashion in the Z++ language. However, at **Z47** level the design requires system library calls for speed. This is the only reason **Z47** is not universally available, yet. Its availability only needs a small team of engineers per platform for the implementation of the drawing of graphic elements for that platform. Otherwise, the **Z47 Graphic Manager** is complete.

Chapter 1 introduces **Z47** in the context of other technologies, and its relative place.

Chapter 2 presents all major components of **Z47** as an operating system.

Chapter 3 discusses how **Z47** performs IO.

Table of Contents

Chapter 1. Languages and Virtual Machines

- Evolution of languages
- Virtual Machines
- Generalizations
- Z47 Threads and Processes

Chapter 2. Z47 Components

- Z47 Processor
  - Instruction Unit
  - Memory Unit
  - Arithmetic/Logic Unit
- Loading an Application
  - Loader Unit
- Managing Components
  - Module Manager
- Managing Processes
  - Process Manager
- Managing Threads
  - Thread Manager
  - Task Manager
- Managing Signals
  - Signal Manager
- Handling Exceptions
  - Exception Unit
- Database Sessions
  - Database Manager

Chapter 3. Hardware Connections

- Graphical User Interface
  - Graphic Manager

Chapter 1. Languages and Virtual Machines

In this chapter we take a brief look at languages, virtual machines and operating systems in order to properly place **Z47** in the list of abundant wealth of technologies. **Z47** solves serious open issues of distributed computing.

Evolution of Languages

Initially, the notion of operating system appeared as a set of routines along with a macro language or a job processing language for using the available routines. The invention of BASIC virtual machine is historically a major progress as a generalization of the notion of a macro language. At the same time, FORTRAN invented another historical notion, namely linkage with the libraries written in the same language. This made it possible for everyone to use extensive libraries written by researchers and continue to extend the contents of FORTRAN library for mathematical computations.

Backus, the inventor of FORTRAN, also introduced the concept of grammar for a programming language and the language ALGOL. The next historical event is the ALGOL like language C and the operating system UNIX. The language C was used in the writing of the operating system. This allowed C to use the services of UNIX as a macro language through the notion of linkage. In addition, C could link with any library written in the language.

The fundamental concept of **process** was introduced with the unveiling of UNIX. The language C could create processes and manage communication among them by using the facilities of the operating system. The notion of C system call is a significant historical event that will persist for an unforeseeable future. At the time of writing an OS its facilities are unknown and certainly do not exist. However, after its completion the language C can simply link with the facilities provided by the OS via system calls. Thus the language C must remain a low-level language.

The purpose of C++ is not quite clear yet, though the important notion of **class** was known long before C++ was introduced. The language C++ cannot be used where C is needed. It only slightly extends C making it easier to write larger system programs such as compilers. Several important programming concept introduced by Eiffel and ADA were known at the time but were ignored in the design of C++.

Nonetheless, there is a need for a language like C++ and someday it will take its final form. Some system programs are large enough to benefit greatly from a properly designed object-oriented language. On the other hand, applications, in particular distributed applications, are abstraction independent of the platform on which they are developed. A system language like C++ cannot include the necessary abstractions for developing platform-independent applications.

Virtual Machines

The notion of virtual machine has been reintroduced in various forms since the invention of BASIC. Smalltalk language makes everything globally available to the code being written next. On the other hand, Java language has scopes and provides a form of linkage with classes written by others.

In designing languages the focus so far has been the ease and reliability of developing applications as understood for a few decades. The notions needed for distributed computing have been introduced in an ad hoc manner. A deeper research reveals the fact that the notions of distributed processes and distributed inter-process communication are fundamental for distributed computing.

The main hurdle in generalizing the notion of process to distributed process is that a process is an entity of an operating system. Thus, we need a distributed operating system in order to speak of a distributed process. The focus of designing a virtual machine is the language it supports. The notion of process does not directly appear in a language. Hence in designing a virtual machine there is no point in including the notion of process.

The notion of an operating system, such as UNIX is an abstraction. However, a hardware OS must deal with real resources and their management. Processes are tied to the resources they need. Nonetheless, it is possible for a distributed application to run as a simultaneous set of UNIX processes spread over several nodes. The problem is the lack of abstractions for a reliable development of such an application.

Furthermore, a hardware process, that is a process created by a hardware OS like UNIX, is effectively tied to the node it is running on. In practice it is formidable to send an executing process over to another node for continuation of its execution. The notion of traveling process will require fundamental changes to the core of a hardware OS that will interfere with many of its desirable characteristics.

The solution to these issues requires a more general abstraction of the notion of operating system, and a generalization of the concept of process, without interfering with the operations and functions of host hardware OS.

Generalizations

The generality of mathematical abstractions, from simple geometrical concepts, to algebraic structures of groups and rings and to the abstract spaces of analysis of Freshet and Sobolev, rests on the freedom of these notions from inessential details.

For a generalization of the notion of operating system to a distributed operating system we begin by dropping certain responsibilities of an OS. For instance, we drop physical and virtual memory management, interaction with device drivers, and so on. However, we continue this reduction of responsibilities only to a point that will still allow us to speak of processes and threads.

In generalizing the notion of process we reduce its responsibilities to exchanging data and signals with other processes and maintaining its private data. Having done that, we are able to extend the notions of inter-process communication for the purposes of distributed computing.

In practice, however, our generalized notion of process will need to be aware of opened files, sockets etc. Therefore, when traveling from node to node a process must relinquish its control of such resources by closing them. But that seems logical anyway. After all, a particular file opened on the source node does not necessarily exist on the destination node. Furthermore, such mistakes can be avoided by properly designing the language.

Unlike mathematics, notions of computation require computational engines. The generalized notion of distributed operating system needs to be made available for all hardware OS. Otherwise, it will not be general enough for distributed computing. Specifically, an important characteristic of a distributed application is its ability to distribute its child processes among nodes of any kind. This is the only way that a designer and the engineers can view a distributed application as a **set of processes executing on a single familiar operating system**.

Much of the research, including autonomous agents, had to be postponed because the notion of virtual machine was inadequate. There was a need for a distributed operating system hence **Z47** was born.

Software engineering needs to be broken down into branches, and engineers should be divided into categories. An assembly-like language, such as C is indispensable for writing hardware OS. A direct extension of C, like C++, is very useful in creating system programs for assisting engineers who design and develop user applications in an abstract language like Z++. We started with arithmetic, moved up to algebra and then to calculus.

When an idea takes a well-understood abstract form, just like mathematical notions, it can be studied as an entity in isolation and continue to evolve independently. This remains true for the future of **Z47**.

Z47 Threads and Processes

With regard to Z++ language, the terms component, module and application are indistinguishable. A Z++ program is an application when used on its own. However, the same application can be used as a component in another larger application **without rebuilding**. Thus the terms module and component are only relative to how a Z++ program is used, **but not how it is written**.

A Z++ program becomes a **Z47** process at execution. When a Z++ application is made up of several components (applications), the startup component becomes the starting process and other components become its child processes. Clearly, this continues recursively for those components that are themselves made up of other components.

Each component is a process and maintains its own context including its open files, objects, instruction pointer, etc.

In Z++ language, a larger program can use another program either as a **class** or as a **task**. When **task** is used, first a thread is created and then the component is created in that thread.

When a component is included in a larger program using **class** as opposed to **task Z47** does not create a thread. Instead, the thread of control moves between components. Nonetheless, each component is a process and a **context switch** takes place when control enters another component. Note that even in this case, a child process (component) could be multi-threaded in which case each of its threads will receive their time-slice.

When a child component is remotely loaded **Z47** creates a shadow process corresponding to the remote component.

The smallest unit of execution managed by **Z47** is a process. The smallest unit that gains control of **Instruction Unit** of **Z47** for the duration of its time-slice is a thread.

Chapter 2. Z47 Components

In this chapter we describe major components of **Z47** distributed operating system. Like any operating system **Z47** comprises of a number of well-known components. However, **Z47** abstracts away a large amount of complexity with regard to distributed computing.

Unlike the language of mathematics, a programming language must be designed around an engine. Unless the engine possesses the necessary abstractions, the language designed around it will not be able to support the notions needed for expressing solutions in a particular category of problems.

The problem solved by **Z47** cannot be overcome by designing a language around the existing operating systems. A simple criterion for a distributed operating system is the ability of its processes to **execute on a different node the next time they receive their time-slice**.

Distributed Inter-Process Communication requires the participation of the operating system. A language cannot provide the abstractions needed for communicating distributed applications by simply relying on such notions as Remote Procedure Call. The run-time library of such a language will need to maintain a context and continually run as a system process. However, it is not clear if even this scenario will in fact provide sufficient computational notions for a language for distributed computing.

Z47 Processor

The **Processing Unit** of **Z47** Distributed Operating System is called **Z47 Processor**. Usually, for simplicity we refer to **Z47** as the **Z47 Processor**.

The processor comprises of **Instruction Unit**, **Memory Unit** and the **Arithmetic/Logic Unit**.

Instruction Unit

The **Instruction Unit** remains in idle state until an application is loaded, or a remote node contacts **Z47**. It then begins the usual cycle of fetch, decode and execute so long as there is a process.

The **Instruction Unit** has a number of registers and an instruction pointer. Processor instructions for Jump to Subroutine, Return, Register and Branching instructions are in this unit.

Memory Unit

Z++ language is multiple-inheritance and provides a number of new type constructors such as collections and semi-dynamic arrays. In addition, Z++ provides an extended form of the type constructor enumeration. Furthermore, it supports C language pointers. Thus, locating objects is not straightforward.

The **Memory Unit** provides a set of instructions for retrieval of objects and delivering them to the **Instruction Unit**.

The **Memory Unit** is a complex component consisting of several smaller components, which will not be presented in this writing.

Arithmetic/Logic Unit

This is the simplest component of **Z47**. The Z++ compiler breaks down the extensive set of Z++ language arithmetic and logical operators to the capabilities of this unit.

This unit is essentially a set of processor instructions.

Loading an Application

A Z++ application can have any number of **entry** points. Moreover, an application may be loaded as a start up program, or as a component of a program in execution for local or remote execution. Furthermore, the component to be loaded may reside on the local node or a remote node.

Loader Unit

Depending on the location of the component to be loaded and the location of its execution the following scenarios are possible.

The **Loader Unit** performs the following operations for a local load intended for local execution.

- Puts the binary image of a Z++ component into memory
- Collects the **entry** points of the component
- Makes a **Z47** process and sets the instruction pointer
- Passes the information to the **Module Manager**

When the component needs to be downloaded from a remote node for local execution, the **Loader Unit** first downloads the binary image of the component to the local node and then proceeds with the operations listed above. The **Loader Unit** on the local node follows a protocol in communicating with the **Loader Unit** on the remote node for getting the binary image of the component.

When the component is intended for execution on a remote server, the **Loader Unit** of the local node makes sure that the **Loader Unit** of the remote node completes its operations, as listed above. The remote node may have to download the component from another remote server.

The **Loader Unit** on local node then creates a **Z47** **Shadow Process** on the local node to correspond to the remote component and passes the information to the **Module Manager**.

The shadow process takes a tiny amount of resources for channeling function calls to its corresponding remote process.

Managing components

A component has two aspects. The module aspect of a component is, handled by the **Module Manager** and its process aspect is, handled by the **Process Manager**. Here we are discussing the module aspect of a component.

Module Manager

The **Module Manager** ensures that only one copy of the binary image of any number of components using the same image is loaded into the memory. It accomplishes that by communicating with the **Loader Unit**.

The **Module Manager** maintains the list of **entry** points to each component and manages the calls entering and leaving the component.

Since a component is also a process it has its own context of execution. Therefore, whenever execution enters a component the **Module Manager** performs a context switch. This context switch is not related to time-slice. Instead the **Thread Manager** handles the context switch when a thread completes the duration of its time-slice. Thus, two forms of context switch take place in **Z47**.

The **Module Manager** cleans up after a component when it is removed. For instance the component may have opened files but not closed them due to an exception. In **Z47** the **Module Manager** rather than the **Process Manager** closes all files left open prior to removing the component.

When removing a component, unless there is another component still using the binary image associated with the component being removed, the **Module Manager** unloads the code.

Managing Processes

A Z++ application executes as a **Z47** process. In addition, a component loaded by a process has certain aspects that relate to its execution as a process. For instance, a component may create several threads.

Process Manager

Z47 processes communicate, locally and remotely. Older forms of communication, such as Remote Procedure Call, are not visible at Z++ language level. Though transparent within Z++ source code these forms of communication do take place among **Z47 Processors**.

Some forms of communication are only partially visible to an engineer. For instance when making a database query the engineer knows a communication will have to take place with a database server.

Tell/hear remote signaling is an example of a form of **Z47** communication mechanism quite visible at Z++ language level.

There are several other forms of communication taking place among **Z47 Processors** running on different nodes. For instance, when an exception occurs on a remote node it is delivered to the local node waiting for the completion of a service.

One important responsibility of the **Process Manager** is the delivery of messages of any type to the correct processes. The collection of communication mechanisms of **Z47** provides a set of abstractions for Distributed Inter-Process Communication. These abstractions are exploited by, the Z++ compiler so the language can furnish a simple and yet powerful set of communication abstractions for distributed computing.

All threads of a process/component share the same global stack as apart from their individual stack for function calls. Therefore, the **Process Manager** must know the component that owns a thread for proper context switch associated with time-slice expiration. Thus, the relationship between a process and a component is that the process owns a component. However, a process cannot own more than one component.

A **Z47** process maintains a list of threads, open sockets and other items as maintained by a process on any reasonable operating system. A process owning a component is also aware of other pieces of information such as the list of open files maintained by the component.

Managing Threads

In Z++ language one can create global threads and object-oriented **task** threads. Here, the term thread means either kind while the term task specifically refers to threads creates via Z++ **task**.

Each thread, independent of the process it belongs to, receives a time-slice during which it will have the control of the **Z47 Processor**.

Thread Manager

A thread maintains its own instruction pointer, stack of function calls, local objects, signals and exceptions. The global stack, open sockets and so on are maintained by the process the thread belongs to.

A thread can be in one of several states. Generally, a thread is in active state meaning that it can take control of the processor. However, a thread may be waiting for the completion of an input/output operation, a socket transmission, the return of a call made to a remote module and so on. A thread may also be in a graphic state waiting for user input. Other states include signaled and exception states.

The **Z47** **Dispatcher** checks the state of the thread before transferring control to it. In some cases the dispatcher assigns a small time-slice to a waiting thread. In most cases the thread is skipped until the next visit.

A thread in one of the wait states, depending on the type of its state, may have already changed its state to active prior to the dispatcher considering it for execution. For instance this happens when a thread is waiting on the completion of a socket transmission.

In other forms of wait state the dispatcher tells the thread to verify and adjust its state. For instance this happens when a thread is in a graphic wait state. If the state of the thread changes to active, the dispatcher will let the thread use the processor for its time-slice.

A thread can receive various kinds of signals. Some kinds of signals must be queued by the thread. The dispatcher will transfer the signals that need queuing to the queue of the thread before considering it for execution. In fact, the thread may not receive its time-slice this time around perhaps because it is in a wait state. Nonetheless, the thread will receive its pending signals for processing at a later time when it gains its time-slice.

Task Manager

Tasks are threads but they have needs beyond global threads. The **Z47** component **Task Manager** manages operations specific to tasks.

When a global method of a Z++ **task** is invoked, the request for its execution is queued by the **Z47 Processor**. Eventually when the task thread is considered for its time-slice, all waiting requests for the task object are transferred to its private queue prior to submitting the control of the processor to it.

When a task thread receives its time-slice it responds to the waiting requests in its queue in the order in which they arrived. If it finds its queue empty and that it has nothing else to do, it immediately relinquishes the control of the processor.

Z++ tasks also receive requests for executing certain methods called **Handlers** in Z++ language. These requests arrive in the form of signals. When a signal for a task handler arrives the task thread executes the code of the handler associated with that signal.

Signals intended for the execution of task handlers are queued by the **Z47 Processor** and are delivered to it just before transferring the requests for the execution of its regular methods. Thus, when the dispatcher considers a task thread for execution it first transfers all handler signals, followed by all calls to regular methods to the queues of the task. Other types of signals that need queuing, as for regular threads, are transferred at this time. All these transfers take place prior to checking the state of the task thread for execution.

When a task thread gains its time-slice it begins by responding to the signals sent to it and follows that with the requests for the execution of its regular methods.

In Z++ language a **task** can have an **Idler** method. If a task object has an idler, the method is invoked when there are no further requests of any kind waiting in the queue of the task.

As mentioned earlier, a **task may not necessarily use up its entire time-slice**. Once all requests have been responded to, and in the absence of an idler, a task thread voluntarily relinquishes its control of the processor.

Managing signals

The signaling system of Z++ language is quite extensive. Please refer to **Z++ Reference Language** for illustrations of various forms of signals and their use.

Z++ signals travel among heterogeneous nodes. The component of **Z47** that manages different types of signals is the **Signal Manager**.

Signal Manager

The **Signal Manager** has a different queue for each kind of signal. The **Signal Manager** is the recipient of all signals either arriving from a remote node or generated locally. The exchange of some signals between two nodes, like tell/hear signals, begins with an initial negotiation. The **Signal Manager** of the local node negotiates with the **Signal Manager** of the remote node prior to accepting such signals for queuing.

The delivery of any kind of signal removes it from the queue of the **Signal Manager**. The signal may simply be passed to the thread requesting it, or transferred to its private queue depending on the kind of the signal.

The delivery of signals that require queuing also depends on their kind. Some signals are delivered to the private queue of the first thread that requests them. For other kinds of signals, threads must register the signals with the **Signal Manager** in order to receive them in their private queue.

A signal that does not need queuing is delivered to a thread during the execution of the Z++ statement that is requesting the signal. The dispatcher does not deal with these kinds of signals. However, the dispatcher informs the **Signal Manager** to transfer all signals that require queuing to the queue of a thread before the thread is allowed to gain control of the processor.

As an example, a **hear** signal must be registered and requires queuing but is consumed during the execution of a Z++ statement. On the hand, a task handler signal needs queuing but is not involved in a Z++ statement. It is only used as a specification for invoking its associated handler method. In both these cases the dispatcher instructs the **Signal Manager** to transfer the signals to the queue of the thread.

Handling Exceptions

Z++ exception mechanism provides resumption. In addition, exceptions raised in any programming unit of Z++ language are not lost until they reach the final thread of a process without being handled. At that point, the application is terminated.

Exception Unit

The **Exception Unit** is responsible for keeping track of the point of resumption. When a program attempts to resume, the **Exception Unit** resets the instruction pointer to the correct statement depending on the type of resumption.

The **Exception Unit** keeps track of a raised exception until it is either handled, or delivered to the next layer. A raised but not handled exception in a thread is transmitted to its parent. An exception occurring in a module without being handled is transmitted to the application that loaded the module, either locally or remotely.

The **Exception Unit** records user-defined exceptions per process thereby avoiding the possibility of collisions. However, most of the work with regard to exceptions is the work of the Z++ compiler. For instance, an exception layer is a linguistic construct.

In other cases, the compiler attempts to equip the language with constructs to correspond to the operations of the **Exception Unit**. For instance, the **Exception Unit** transfers un-handled exceptions to the parent thread or module. In Z++ language, the **throws** specification attempts to ensure during compile time that all thrown exceptions are eventually caught. In other words, these two activities are parallel but independent.

Database Sessions

Within one Z++ application, whether on a desktop or a handheld device, there can be any number of database sessions with different database systems. Z++ language hides a large amount of coding complexity. However, Z++ is able to do so by relying on **Z47**.

Z++ statements are object-oriented extensions of familiar SQL statements. For examples and illustrations please see **Z++ Language Reference**.

Database Manager

Database Manager starts, maintains and ends each database session. **Z47** assigns a number to a session, which Z++ libraries use in a manner similar to a file descriptor.

The manager maintains a list of sessions for each thread, rather than the entire process. This helps Z++ language with some of its features for scopes of database objects.

The compiler does the work of breaking down Z++ object-oriented SQL statements and the separation of database requests from Z++ language extensions. However, the manager carries out the mapping between the fields of a database table and the members of a Z++ object. This is natural because the manager is the line between the Z++ language and a database system.

The **Database Manager** does not load a database system library on a handheld device. For a handheld device, the manager contacts a **Z47** running under **Z++ Internet Server**. The two managers follow a protocol for their operations.

Note that, the **Database Manager** of the remote **Z47** running under the **Internet Server,** that is the proxy manager, will not be running in the context of a **Z47** process. It simply establishes a session with the requested database system, and in case of failures it reports exceptions to the **Database Manager** on the handheld device.

The remote proxy manager also allocates a buffer for results of queries and sends them to the manager on the handheld device in requested sizes.

In a proxy scenario, the local **Database Manager** instead of a system database object creates a client-proxy database object. For instance, if the database system is mySQL, the manager will create an instance of its type for mySQL for its operations. But in a proxy scenario, the manager creates a client-proxy object. The remote proxy manager also creates a server-proxy database object.

In a proxy scenario, the **Database Manager** on the handheld device maintains the programming session associated with a thread, and performs the mapping between database table-fields and program object-members.

Database exceptions are raised by, the **Database Manager**.

The **Database Manager** consists of a number of components for communication with a database system, the execution of proxy protocol, maintaining the context of a session, error reporting and exceptions, etc.

Chapter 3. Hardware Connections

Applications are developed for interaction with users. Interaction with an application simply means that users can provide input, and the application can produce output. Indeed, applications perform extensive IO without directly engaging with a user. In fact, much of the user IO is transformed to several other forms of IO.

IO is tied to hardware devices, even when a device is attached in a virtual form. IO capabilities are constantly evolving and IO devices are growing. **Z47** must support the very best of available IO capabilities for each and every platform.

For a developer all IO must come in the form of abstractions in the language Z++ and free of any system calls. This includes the forms of IO that an engineer needs, but are not used in interactions with users.

Some of the IO abstractions can be turned into linguistic constructs, and others should be provided in the form of libraries. For instance, Z++ includes linguistic constructs for graphical user interface, but consol IO is presented as libraries.

Not all forms of IO are supported by every platform. For instance consol IO is not suitable for a handheld device. Well, not every branch of mathematics is used in all applications of mathematics. The language should cover the entire spectrum of platforms.

The Z++ GUI has a corresponding managing component in **Z47**. Other forms of currently supported IO use library calls. For details of each form of IO please refer to the **Z++ Language Reference**.

Each form of IO presented as libraries has a small unit in **Z47** for uniformity. Consider file operations. Z++ file streams are similar to C++ file streams. If C++ streams are supported for a platform, the file unit of **Z47** lets C++ libraries do the work. Otherwise, the file unit will contain the necessary code to simulate C++ libraries.

When a thread requests an IO operation, but needs to wait, the IO unit sets the state of the thread to waiting for IO completion. The state of a thread is checked by, the dispatcher.

The IO unit raises exceptions related to IO.

Graphical User Interface

Z++ provides an object-oriented Graphical User Interface. For illustrations and examples please see **Z++ Language Reference**.

Z47 has a fairly complex unit for managing GUI operations. The platform for dealing with GUI is determined by the **Z47 Processor** built for the platform, rather than the language Z++. The role of the language is to facilitate the building of GUI entities for each platform via tools. Furthermore, Z++ GUI operations, including dealing with the devices such as mouse are uniform and independent of platform.

Graphic Manager

Corresponding to each Z++ language **frame**, the **Graphic Manager** creates a **Z47** graphic entity. Roughly, a graphic entity corresponds to a top-level window along with all graphic elements, such as fields and buttons drawn in it.

The **Graphic Manager** maintains all graphic entities. All graphic input from the host OS is received by the manager.

Each graphic entity is endowed with an input queue. When the dispatcher visits a thread in graphic wait state, it informs the **Graphic Manager** to check its queue for any input associated with the thread. If any input has been received the **Graphic Manager** takes the following steps.

- Transfer the input to the queue of the entity associated with the thread
- Inform the entity to respond to the input

The entity performs the following operations when instructed by the manager.

- Construct the argument to pass to instinct method of **frame** (Z++ language)
- Set the instruction pointer to the start of the code of instinct method.
- Change the state of thread to active.

At this point, the dispatcher is informed of the completion of operation. The dispatcher then gives the thread its time-slice and lets it use the processor.

Unless there is an input available for a thread in graphic wait state, the dispatcher will not consider it for execution.

When all input in the private queue of a graphic entity are served, the entity changes the state of its associated thread to graphic wait state.

The **Graphic Manager** is responsible for all the drawing and erasing operations, and raising exceptions related to graphic operations. The manager uses system calls as much

as it can. When a platform does not support a needed graphic element it will put it together as a compound element using as few system calls as possible.

The manager becomes more involved in the operations of a compound element. For instance, consider putting together a container from a field, a button and a drop list. Sometimes the list must be shown, and its current element must be highlighted. The user may click, the button, enter text in the field, or make a selection from the list. In these operations it is the **Graphic Manager** that is creating a single view of the compound element and moving data between the field and the list.