Java™ API for USB (javax.usb)

JSR-80 Specification

Version Code:
Version 0.9.0

Date:
Wednesday, April 04, 2001

Address questions to:

E. M. Maximilien
(919) 301-7014
maxim@us.ibm.com

D. D. Streetman
(919) 301-5672
ddstreet@us.ibm.com

B. Dimmock
(919) 301-5713
bkd@us.ibm.com

Java™ is a registered trademark of Sun Microsystems, Inc. http://www.sun.com

Authors, Reviewers and Contributors

IBM

E. Michael Maximilien
Boyd Dimmock
Dan Streetman
Brian Weischedel

Sun Microsystems

Paul Klissner
Sunil Dusankar
Ron Kleinman

Fujitsu-ICL

Harry McKinlay

Wincor-Nixdorf

Peter Duellings

Independent

Roger Lindsjö
Steve Turner
Paul Gay
Boris Dainson

Table Of Contents

Abstract 4
1 Introduction 5
2 Audience, Motivation and Requirements 6
2.1.1 Intended Audience 6
2.1.2 Motivation 6
2.2 Requirements 7
2.2.1 Java editions targeted 7
2.2.2 Functionality 7
2.2.3 Performance 7
2.2.4 USB Version 8
2.2.5 Internationalization (i18n) 8
2.2.6 Abstraction 8
3 Architecture 9
3.1 Overview 9
3.1.1 Future 10
4 Design 11
4.1 UML Package Diagram 11
4.2 USB OS Services Object Model 12
4.3 USB Device Object Model 14
4.4 Descriptor Hierarchy 16
4.5 Utility Classes/Interfaces 17
4.6 USB Pipe Object Model 18
4.6.1 USB Pipe Input and Output 20
4.6.2 USB Pipe Types 20
4.6.3 USB Pipe State Model 21
4.6.4 USB Pipe Class Diagram 22
4.6.5 USB Pipe Event Class Diagram 23
4.7 I/O Request Packets (IRPs) 24
4.8 Device Event Object Model 27
4.9 Request/StandardOperations Object Model 28
4.9.1 Requests and USB Device Operations Usage - Dynamic Model 29
5 Sample Usage: Client Application Examples 30
5.1 USB View Application 30
5.2 UsbPipe Test Application 30
6 Conclusion 32
7 Appendix A: Design Considerations and Future Releases 34
7.1 J2ME 34
7.2 USB 2.0 34
7.3 Claiming Interface using Policies 34
7.4 Support for Isochronous Transfer 35
8 Appendix B: Frequently Asked Questions (FAQs) 36
9 Appendix C: Change Summary 42
10 References 44
10.1 Web 44
10.2 Books 44

Abstract

Universal Serial Bus (USB) technology was demanded by the general IT market to simplify the connectivity of peripheral devices while adding simplicity, flexibility, and capacity to device attachment. This proposal describes a Java interface to USB: javax.usb. It includes the architecture and design for this interface, which allows JVMs to access the USB host controller with the full capabilities of the USB architecture. Similar to Sun Microsystems' javax.comm API for RS232 serial and parallel connectivities, javax.usb is a Java-based API which is dependent on an implementation to map to the native OS interface for USB. Availability of this interface implementation on Java enabled platforms allows for true cross platform access to USB from Java software.

1 Introduction

The javax.usb is a Java API giving full access to USB on a host system enabled for the Java 2 platform. The API is similar in intent but very different in architecture and design to Sun's javax.comm. The javax.comm API is simple API that gives access to RS232 ports and standard parallel ports from Java applications. The javax.comm architecture and design, though sufficient for simple serial ports such as RS232, is completely inadequate for USB, which is a tree based, dynamic, expandable, plug-and-play, high bandwidth, powered bus. To correctly provide access to all the functionality of USB it was clear that a new API was needed. Javax.usb is such an API, and gives full access to USB from Java on any platform that supports USB 1.1 or later specification. For each USB platform the JVM or OS provider will need to implement the javax.usb specification by adding some native code that plugs in javax.usb and binds it to the host OS USB driver architecture. To prove the validity of javax.usb, we have developed an open source reference implementation of the specification for the Linux OS.

2 Audience, Motivation and Requirements

2.1.1 Intended Audience

The following are the intended audiences of this document:

1. JSR-80 expert group: the expert group will ultimately decide the definition of the Java API for USB; namely the contents of this document
2. JVM and/or OS providers: any platform that intends to support the JSR-80 specification will need to use this document to expose this API to access USB
3. USB device manufacturers: who will create Java middleware drivers (or services) to allow access to their devices from Java applications and applets[1]
4. Java application and applets developers: who want to enrich their applications with access to locally attached USB devices

2.1.2 Motivation

Implementation of device services (i.e. drivers) in Java is a long-range strategy for portable device support. This allows providers to move towards a 'common' set of device services, which execute on all Java enabled platforms. Basically this API is extending the "Write once, run anywhere"™ mantra to services (or drivers) of USB devices and consequently to applications/applets needing to access these devices.

We recognize that there are additional evolving technologies for device attachment (i.e. Firewire™, aka IEEE 1394) which may suggest that there should be provision in this API to support USB and other technologies seamlessly. However, our intent is to expose USB uniqueness in this API in order to provide full access to the bus. Trying to provide a generic communications API for the Java platform is beyond the scope of this document[2]. We plan instead to provide a separate API that can focus on the particular communication transport at hand: USB. It is our opinion that seamless access to devices, independent of the attachment technology, is correctly implemented via a layer of abstraction (an abstract device model) above this API. In fact, for the Retail industry[3], there are extensive specifications for device interfaces for point of sale devices, which eliminate the application's need to implement to a specific attachment technology (i.e.. RS232, USB, RS485). It is worth noting that these retail specifications do not deal with the device connectivity in a generic way, but with the functionality of the device.

---

[1] Since Java applets have very stringent security restrictions, access from applets to any API that is not part of the Java core API that needs to load native libraries will necessarily need to be trusted (i.e. signed). Please see the details on trusting applets for the Java 2 platform at http://java.sun.com/security

[2] For an analysis of why having a generic communications API could be problematic and why we have chosen to have a separate API, please see section 8.2 of this document.

[3] The Java Point of Sale or JavaPOS specification can be found at http://www.javapos.com. Also the Financial industry has a similar specification for devices attaching to financial point of service (e.g. used by Bank tellers) at http://www.jxfs.com

2.2 Requirements

2.2.1 Java editions targeted

This API is targeted at the J2SE (Java 2 Standard Edition) and J2ME (Java 2 Micro Edition) platforms. The requirement for J2SE stems from the fact that USB is becoming the primary connectivity for client personal platforms and is also in demand for vertical markets' client platforms (e.g. Retail Point of Sale). The requirement for J2ME as a targeted platform is based on USB becoming an increasingly popular connection for portable/personal devices (i.e. PDAs, etc.). As we move forward in the JCP process, the expert group will refine the requirements for the particular profile required for J2ME[4].

2.2.2 Functionality

USB is a more complex and feature rich bus than standard RS232 and parallel ports. USB has a multiplex, high-bandwidth, dynamic, tree-based, powered, plug-and-play architecture. The requirement for javax.usb is to expose the full architecture of USB to Java software. This implies that:

1. The API should closely follow the official USB specification. (See: http://www.usb.org/)
2. The API must support the dynamic nature of USB, in that devices can be attached and detached at runtime (hot plugging). The API must provide a mechanism to determine what is attached when a change occurs
3. The API must support multiplex operations such that USB devices can be attached to USB hubs that multiplex a USB port into more ports. Each port can then accommodate another hub which can have hubs or devices attached. The current limit is 5 hub-levels deep and with up to 128 devices (including root hub) attached per "bus" (per USB controller)
4. The attached devices form a tree instead of a list (devices attached to hubs which are themselves devices). A device list via topological sort, breadth first search or depth first search of the tree should also be possible
5. The API provides access to transporting data and communication signals. USB also transports power, such that devices can be powered from the USB. Although not a dependency on the API, USB power information should be provided to the user
6. USB devices can usually be dynamically configured and can contain more than one configuration. Drivers for such devices are also typically loaded and unloaded dynamically.

2.2.3 Performance

To allow creation of device services for all categories of devices, the API should minimize overhead. The API should allow for implementations that are quick, lean and expose all device functionality in a manner that fits the type of device attached. The

[4] It is worth noting that this first specification of the javax.usb does not include any dependency that would prevent it from being implemented on the Personal Java platform or superset of the J2SE platform (i.e. J2EE).

benchmark is for the javax.usb API to not limit the data capacity throughput or response time expectation as compared to the native OS interface.

2.2.4 USB Version

It is acceptable for an initial reference implementation to provide an API that will support the USB 1.1, which is already prevalent in the marketplace. The USB 2.0 design point is to be a transparent superset from a functional basis, which means that USB 1.1 devices and drivers should work in a USB 2.0 platform (see the USB 2.0 specification for details http://www.usb.org/). USB 2.0 specific details can be addressed in a follow-on revision of this API.

2.2.5 Internationalization (i18n)

There is the requirement to support worldwide USB solutions. This may require management of the translation between USB LANGID codes and the i18n locales of Java. USB devices have string descriptors that describe them and their functionality (e.g. Manufacturer string, Product string, etc.) These strings are in UNICODE, so they can be in any language. This must be supported.

2.2.6 Abstraction

To maximize flexibility in the use of this API, it is important that the application be independent from the actual underlying implementation and platform. The API specification will use mostly Java interfaces with a minimum of classes to allow flexibility in implementation.

3 Architecture

The Java API for USB or javax.usb is architected to meet the requirements set forth in the Audience, Motivation and Requirement section of this document. To reiterate, below are the general architectural goals for javax.usb:

1. The API specification should be in accordance to the USB 1.1 specification.
2. The API specification should be easily implemented on any platform that supports the J2SE and J2ME and USB.
3. The API specification should use mostly Java interfaces with a minimum of classes to allow flexibility in implementation.
4. Clients should be able to use any platform supporting javax.usb to create their device services (device drivers) and be guaranteed that their services should work on all platforms supporting the javax.usb specification.

3.1 Overview

The USB 1.1 specification's chapter 5 (Data Flow Model) and chapter 9 (Device Framework) define a logical model for all USB hosts and devices. Chapter 9 also describes in detail the generic functionality and operations that are supported for all USB devices. Using these as a guide, we have defined javax.usb to include the following subsystems:

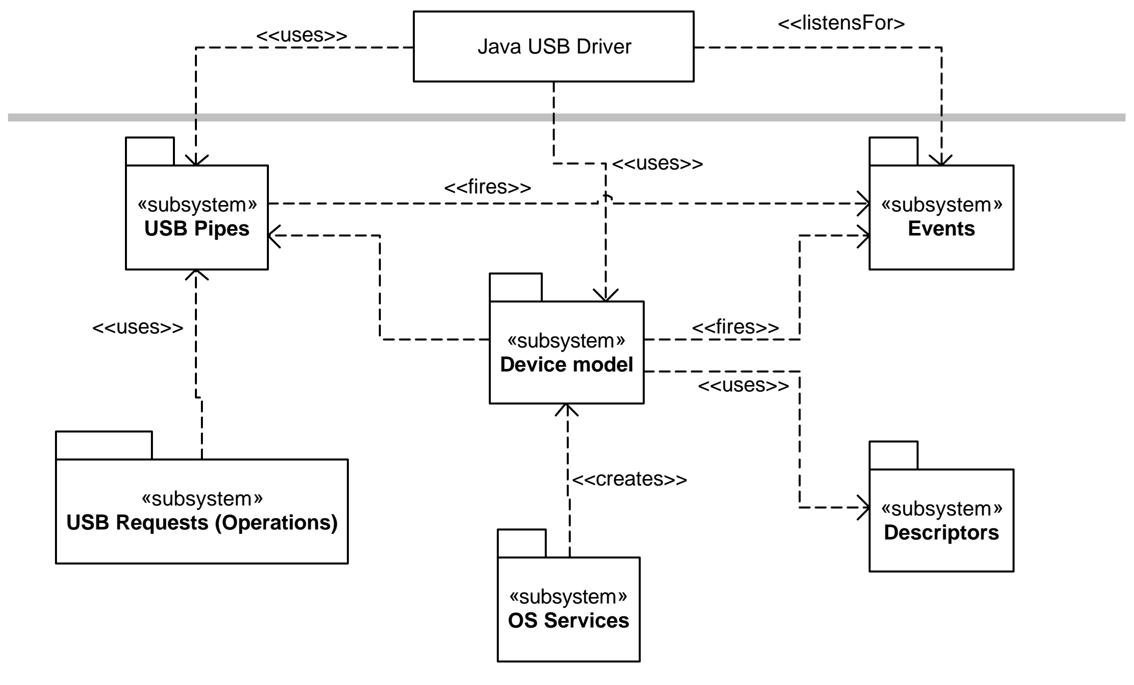

Figure 3.0 High-level overview of the javax.usb architecture[5]

[5] All subsystems are a logical set of interfaces/classes and do not always map to a Java™ package

Together the above subsystems form the core of the javax.usb specification. They should meet the goals and requirements already set forth.

Name	Description	Java Package
Device model	Defines an object model for describing USB devices (including hubs) and all of their characteristics. This is the main facade for users of javax.usb. In the device model we also define a hierarchy for the different descriptors that describe USB devices.	javax.usb
Events	Defines an event model (similar to the JavaBeans event model) for USB devices and pipes.	javax.usb.event
USB Pipes and USB I/O Request Packets (IRP)	Models the communications to and from USB devices and their components. These pipes are logical and are fashioned after the USB 1.1 specification chapter 5 USB pipes.	javax.usb
Request and USB Operations	Models the type of requests that clients can send to USB devices. These requests or operations are fashioned after the USB 1.1 chapter 9 requests.	javax.usb
OS Services	Defines the services needed from the underlying OS and serves as part of the bootstrap that implements javax.usb.	javax.usb.os

Table 3.0 javax.usb subsystems definition

3.1.1 Future

Although by using the current javax.usb one can access and communicate to any kind of USB device, javax.usb does not directly address USB class device specifications (e.g. HID, Audio, Storage, ...). This is intentional. Class devices such as HID can be addressed specifically as extensions to javax.usb or in a future revision of this specification. We intend that an API for these class-specific USB devices would be architected as a layer on top of the javax.usb[6].

[6] This may of course possibly add some new requirements to the javax.usb but we believe that the current architecture will stand firm, as we are able to communicate and access class-specific devices with the current javax.usb reference implementation.

4 Design

To specify in detail the design of javax.usb we will make heavy use of the Unified Modeling Language (http://www.rational.com/uml). We will start with a package diagram that shows the different packages used by javax.usb and their interdependencies. For each logical grouping of classes/interfaces that implement the architecture subsystems, we show detailed static and dynamic structures in the form of class, statechart and sequence diagrams. Where appropriate we also describe the contracts of key interfaces.

4.1 UML Package Diagram

javax.usb is specified using 4 main packages:

1. *javax.usb*: the core interfaces and classes modeling USB devices, descriptors, communication pipes and requests/operations.
2. *javax.usb.util*: various utility classes and interfaces that are used by other classes and interfaces in the javax.usb specification.
3. *javax.usb.event*: classes and interfaces implementing the Event subsystem for USB devices and pipes.
4. *javax.usb.os*: classes and interfaces that specify the services that need to be provided by the underlying OS to accommodate an implementation of javax.usb.

This UML package diagram shows the different javax.usb packages and their inter-dependencies.

The javax.usb package is the central package where all the Device Object Model exists. The javax.usb.os package is the bootstrap to the underlying javax.usb implementation and the javax.usb.event and javax.usb.util are the event/listener and utility packages respectively.

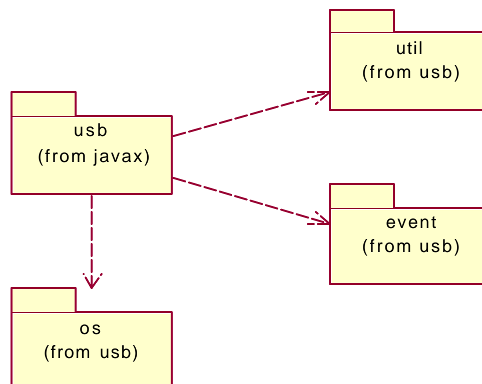

Figure 4.1 javax.usb package diagrams

The javax.usb API interfaces and classes are heavily documented with JavaDOC. Please refer to the javax.usb API release.

4.2 USB OS Services Object Model

UsbHostManager is a Singleton [Gamma95, 127] class (that is only one instance exists per JVM) that bootstraps the implementation (the UsbServices object). This is shown in the class diagram by an aggregation relationship between the UsbHostManager and the UsbServices interface. The UsbHostManager uses the UsbServicesUtility to query the UsbProperties (loaded with the contents of the jusb.properties file) for the current implementation class for the UsbServices interface. The implementation of the UsbServices interface must have a default constructor. The Java reflection API is used to create the UsbServices object. The following diagram shows the Java interfaces/classes in the javax.usb.os package:

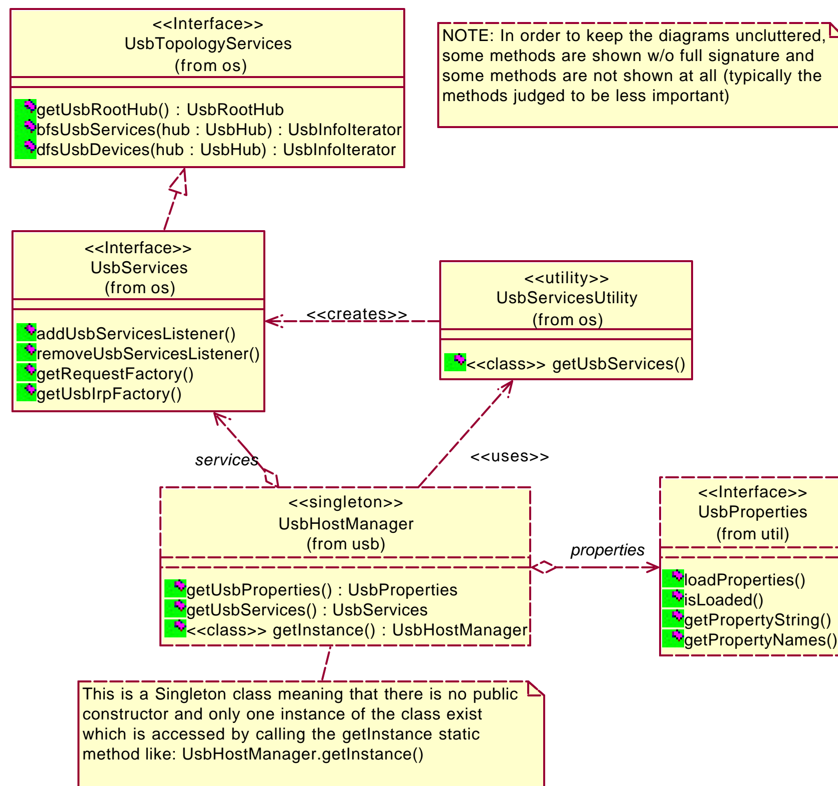

Figure 4.2.0 javax.usb.os class diagram

The UsbHostManager.getInstance() static method is used by clients to access the only instance of the UsbHostManager. The current UsbHostManager implementation uses lazy initialization whereby the first client that calls it pays the price of the bootstrap as well as creating the instance itself. During creation of the UsbHostManager instance the UsbServices is created as mentioned above. Once the client gets the UsbHostManager reference it typically does the following:

1. Registers for events via the UsbHostManager.getInstance().getUsbServices().addUsbServicesListener(listener) to get notified when new devices are attached or current devices are detached.

2. Asks the UsbServices for the root USB hub, which it uses to find the devices that it cares about.
3. Uses other utility methods that return UsbInfoIterator of the UsbDevices in breadth-first-search and depth-first-search orders.

The following sequence diagram shows typical interactions of clients with the UsbHostManager and UsbServices objects:

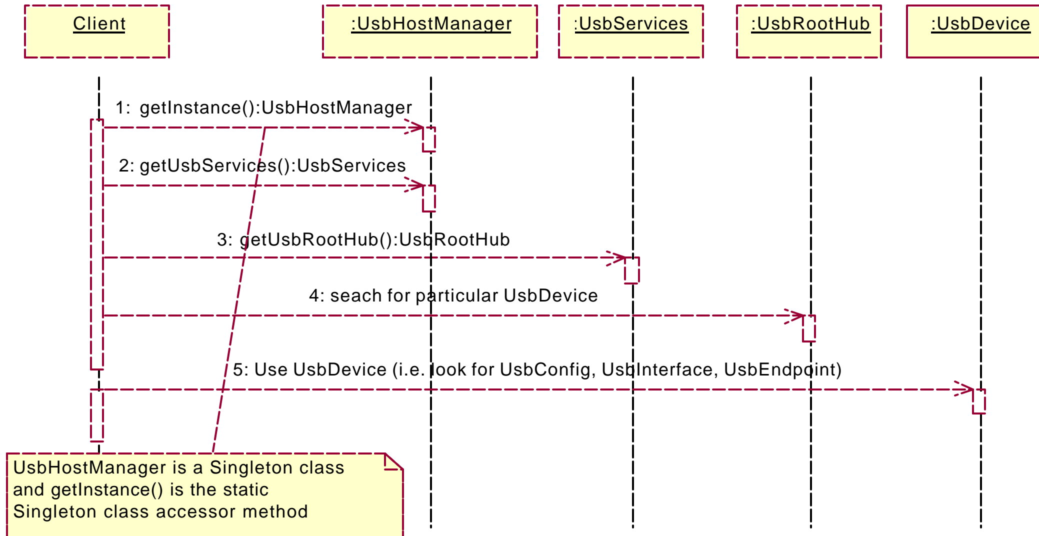

Figure 4.2.1 javax.usb.os typical usage sequence diagram

Since there could be more then one USB controller hardware per USB host, the UsbRootHub object returned from the UsbServices is always a virtual UsbHub. This UsbHub has attached to it all actual root hubs that are attached to this host. For instance, if the host has two USB host controllers, then the virtual root hub returned by calling getUsbRootHub() will have two UsbHub objects attached to it.

Once the root hub is returned from the UsbServices object, the client may need to search for a the particular device. This can be done in one of two ways:

1. Iteratively searching through all UsbDevices attached to the UsbRootHub or other UsbHub
2. Using a UsbInfoVisitor filter class. An example of this method of searching for a device is shown in section 8.5 of this document (FAQs)

4.3 USB Device Object Model

Each USB device is composed of various components. And the different components of a device are well structured. The USB device model shown in the class diagram below tries to model a generic USB device with its component. It can be understood as follows:

1. A USB device is an object whose class implements the UsbDevice interface. Since USB hubs are also USB devices, then USB hubs are modeled with the UsbHub interface which extends UsbDevice and also contains a set of UsbDevices (or UsbHubs, since UsbHubs are UsbDevices). In essence, this is an application of the GoF Composite pattern [Gamma95, 163]
2. A USB device has one or more configurations. This is modeled as an aggregate relationship between UsbDevice interface and the UsbConfig interface. The UsbConfig interface contains getter methods describing it. This information is part of the descriptor for that configuration.
3. Each USB configuration for a device has a set of interfaces associated with it. The interfaces define the exposed functionality of the USB device for that configuration. This is model with an aggregate relationship between UsbConfig and UsbInterface.
4. Each USB interface for each USB configuration on a USB device can have endpoints associated with it. This is modeled again with an aggregate relationship between UsbInterface and UsbEndpoint. A USB endpoint is a source or sink of data to the USB device.

The following UML static class diagram shows the different Java interfaces and classes that make up the device object model:

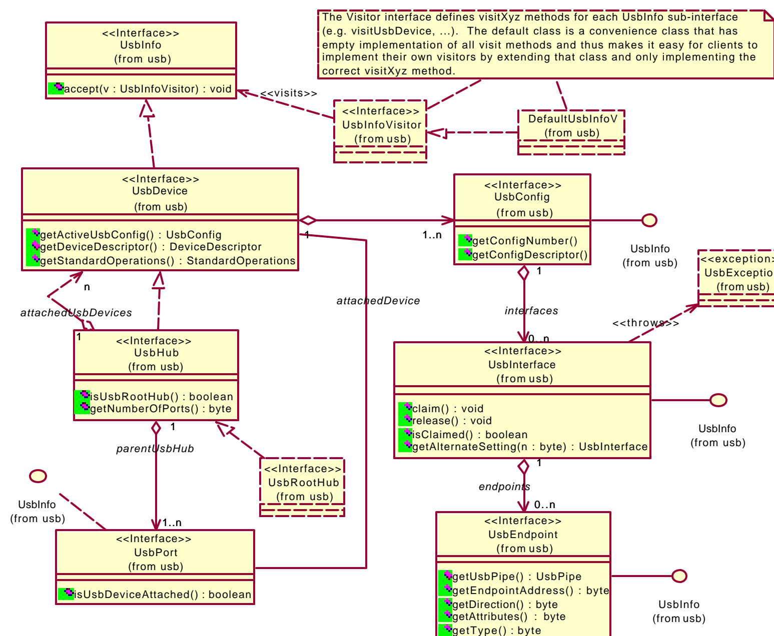

Figure 4.3.0 javax.usb device model class diagram

All of the different device model interfaces extend UsbInfo, which is a token interface that facilitates keeping a typed checked list of the different interfaces[7]. Also with this interface, all of the interfaces on the device model (since the hierarchy is stable) allow Visitors [Gamma95, 331]. The UsbInfoVisitor is defined with visitXyz method for each of the interfaces in the model. With this Visitor pattern, one can add functionality (i.e. Methods) to the model without having to change the interfaces[8]. This also allows easy traversal of the model objects. Note also that the model can still be traversed and functionality added without using the visitors. Visitors just add flexibility and are somewhat justified because the model is stable (e.g. no new device model Java interfaces are anticipated to support version 2.0 of the USB specification).

To show how clients typically access UsbConfig, UsbInterface and other objects, we show a simple sequence diagram accessing a particular endpoint for some interface on the current active configuration of a UsbDevice.

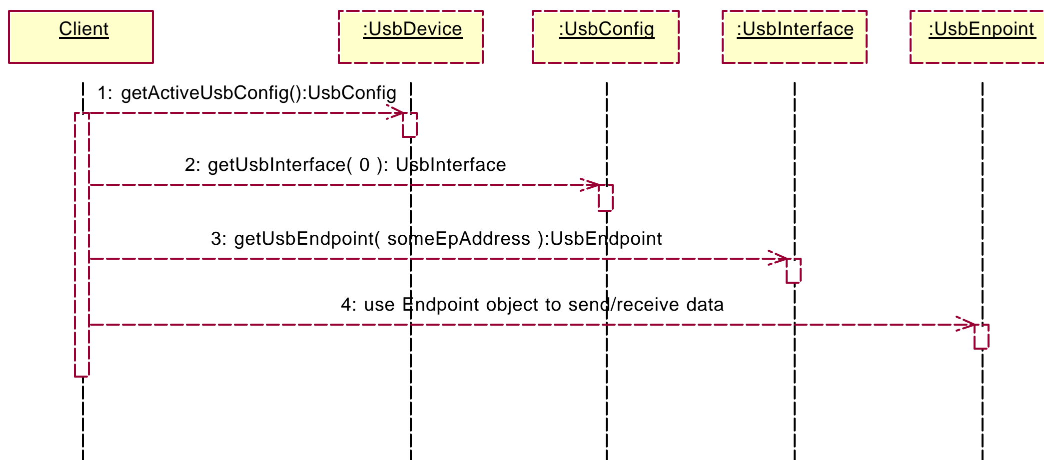

Figure 4.3.1 diagram showing access to a UsbEndpoint from a UsbDevice object

Alternate settings of any UsbInterface object can be obtained by calling getAlternateSetting(byte) method of the UsbInterface object. This method will return the UsbInterface for the alternate setting selected.

[7] Java's lack of generic support means that typical Java list or collections keeps list of java.lang.Objects which is somewhat unsafe. This is not a very significant issue but we felt that having UsbInfoList and UsbInfoListIterator would keep the object model more consistent.

[8] See section 8 of this document (FAQs) for an example on how to use the UsbInfoVisitor to search for a particular device.

4.4 Descriptor Hierarchy

The descriptor extends the device model by adding a set of interfaces that separates the information associated with each of the USB descriptors. By themselves, these descriptors can be used as a way to collect the device's descriptor information and search for specific information[9].

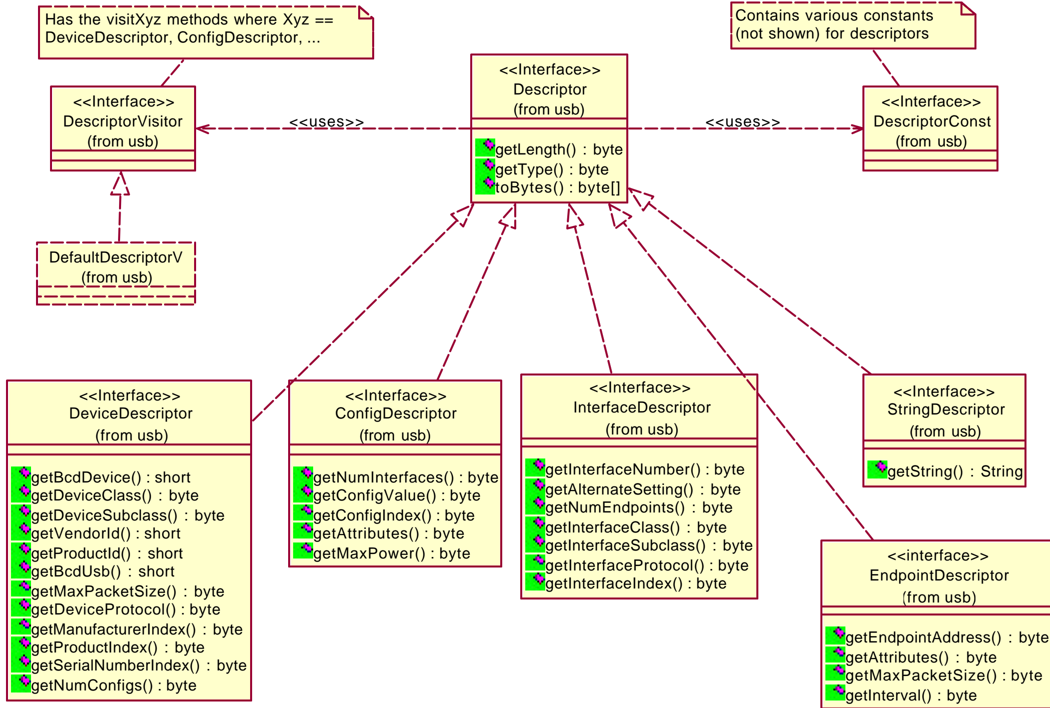

Figure 4.4.0 javax.usb descriptors class diagram

Descriptor objects for each type of the USB model objects can be accessed by calling get<Type>Descriptor() method on the actual model object. For example, to access the InterfaceDescriptor object of some interface of the active configuration of a UsbDevice the following snippet of code can be used:

```
InterfaceDescriptor iDescriptor =
someDevice.getActiveUsbConfig().getUsbInterface( iNumber ).
                                getInterfaceDescriptor();
```

Similar code can be used to access other Descriptor objects.

[9] In the current release of this specification Descriptor objects are immutable. This might change to allow users to modify descriptor information by accessing these objects and using them to submit via the Request/StandardOperations. See section 4.9 for details about Request/StandardOperations.

4.5 Utility Classes/Interfaces

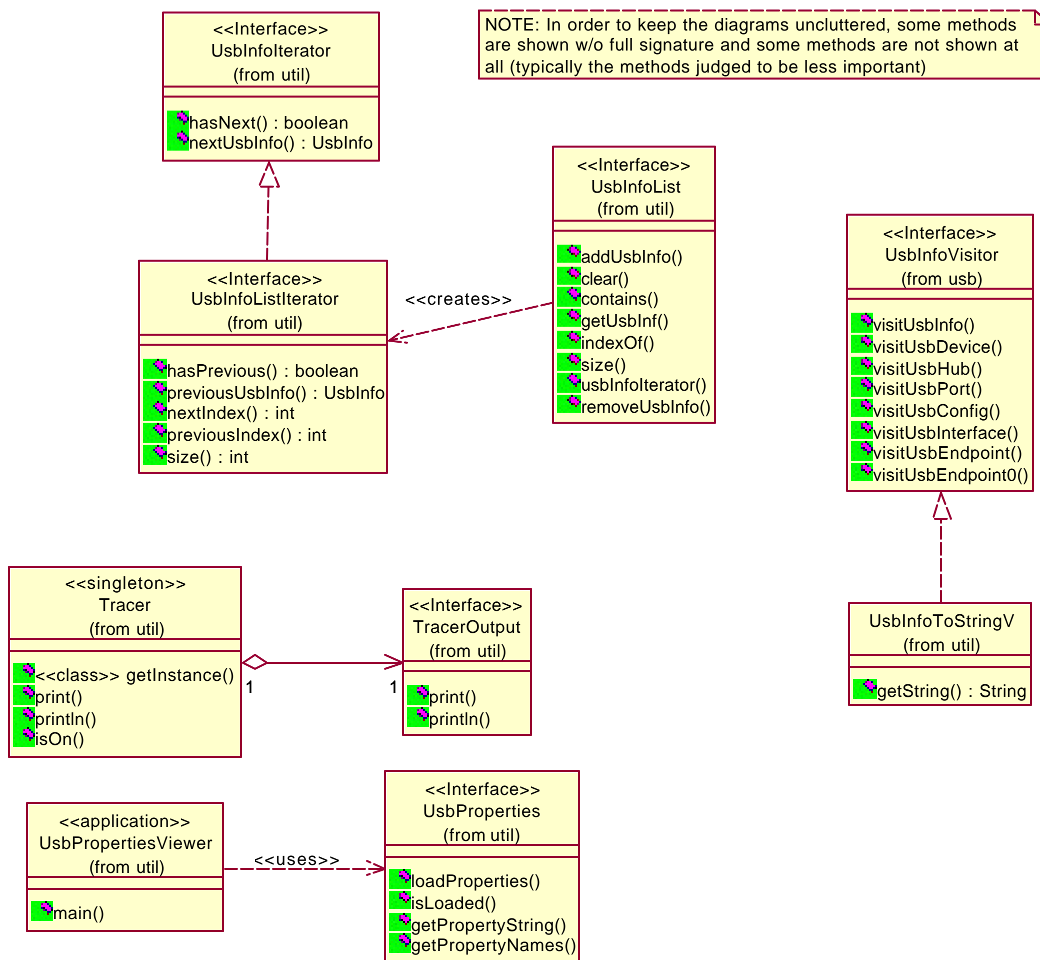

Figure 4.5.0 javax.usb.util utility class diagram

As mentioned before, we have created simple List interfaces and Iterator interfaces for all UsbInfo objects. These interfaces collect all types of UsbInfo objects. The device model makes use of these utility interfaces/classes. Since the implementation of these interfaces should be reusable, the javax.usb.util package contains default implementations of both the list and iterator interfaces using the Java collection API. The UsbProperties interface allows the OS services subsystem to capture runtime information (such as turning on tracing or the level of tracing, etc...).

An example of using the UsbInfoVisitor is illustrated in UsbInfoToStringV, which shows how to create a String representation of any UsbInfo object without having to modify the toString() method for each class implementing the interfaces or having to write code doing heavy conditional statements (i.e. `if( object instanceOf UsbInterface )` {...}).

4.6 USB Pipe Object Model

Pipes are the only method of communication between client software (the host) and a device's endpoints. In this specification, pipes are modeled as 'logical' pipes; they are objects which belong to a specific endpoint[10] and exist for as long as the device model exists. The special Default Control Pipe, which is present on all devices, is not directly accessible by client software; instead, applications should use Requests. See the USB specification section 5.3.2 for details on USB pipes.

Pipes are accessed through their associated endpoint[11]. There are several conditions and actions that must be met or performed before using a pipe. First, the pipe must be in an *active* state. Pipes belonging to an endpoint on an active interface setting (and active configuration) are active; pipes on inactive interface settings (or inactive configurations) are inactive. Active pipes must be opened before use. The diagram below shows how to prepare a UsbPipe for use.

1. The UsbInterface that owns the UsbPipe's UsbEndpoint must be claimed via claim(). This call may fail if the UsbInterface is claimed by any other client (or anything else goes wrong).
2. Get the UsbPipe object from its associated UsbEndpoint via getUsbPipe().
3. Call open() on the UsbPipe. If opening the pipe does not fail, the UsbPipe is now ready for data submission. If the call fails, it will throw a UsbPipeException that indicates the problem.

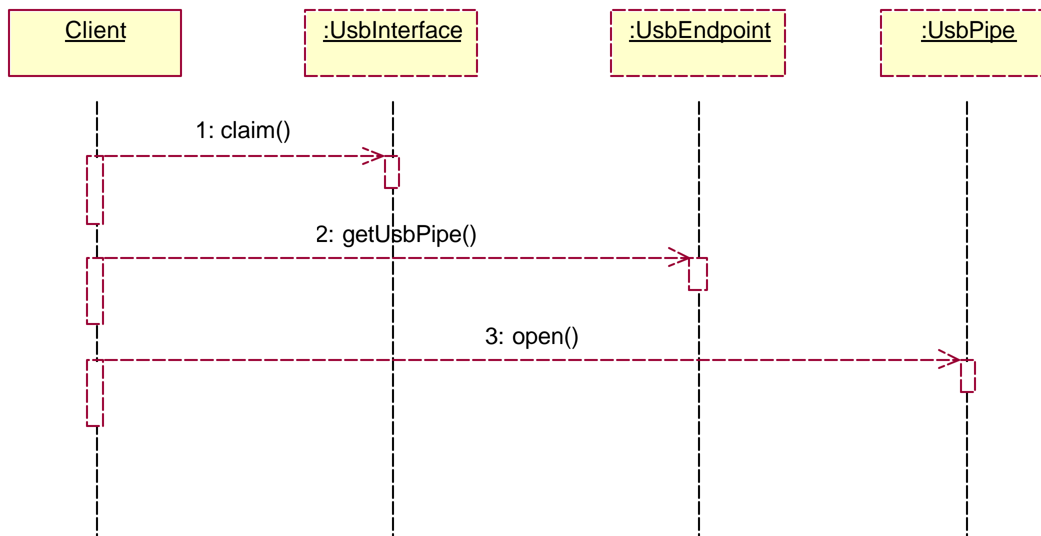

Figure 4.6.0 Preparing a UsbPipe for communication

[10] Most pipes belong to a specific endpoint. Control pipes are slightly different. The USB specification, section 5.3.2, states: "A USB pipe is an association between *an endpoint*..." and software on the host, then in a later paragraph states, "the pipe that consists of the *two endpoints* with endpoint number zero is called the Default Control Pipe". This specification assumes a Control pipe is a single pipe that is bi-directionally connected to its single associated endpoint. The direction bit (which is part of the endpoint bEndpointAddress) should be ignored.

[11] Except the Default Control Pipe, which is not accessed directly; instead, Requests are used.

The diagram below indicates how to submit data synchronously and asynchronously using byte[]s. See section 4.7 for more complicated methods.

1. Create a byte[] which will be the required data buffer.
2. If the direction of communication is out (host to device), fill the byte[] with the data you wish to send.
3. For synchronous communication, call syncSubmit(byte[]). This will block until the submission is complete.
4. After the submission is complete, all of the UsbPipe's listeners will receive a data or error event (depending on whether the submission was successfully completed or an error occurred).
5. The syncSubmit(byte[]) method will either return the number of bytes transferred, or throw a UsbException.

Asynchronous communication is similar to synchronous, except asyncSubmit(byte[]) is used and the call returns a SubmitResult object that can be used to track the submission. The SubmitResult is returned immediately after the subsystem accepts the submission. The client may call waitUntilCompleted() on the SubmitResult to block until the submission completes.

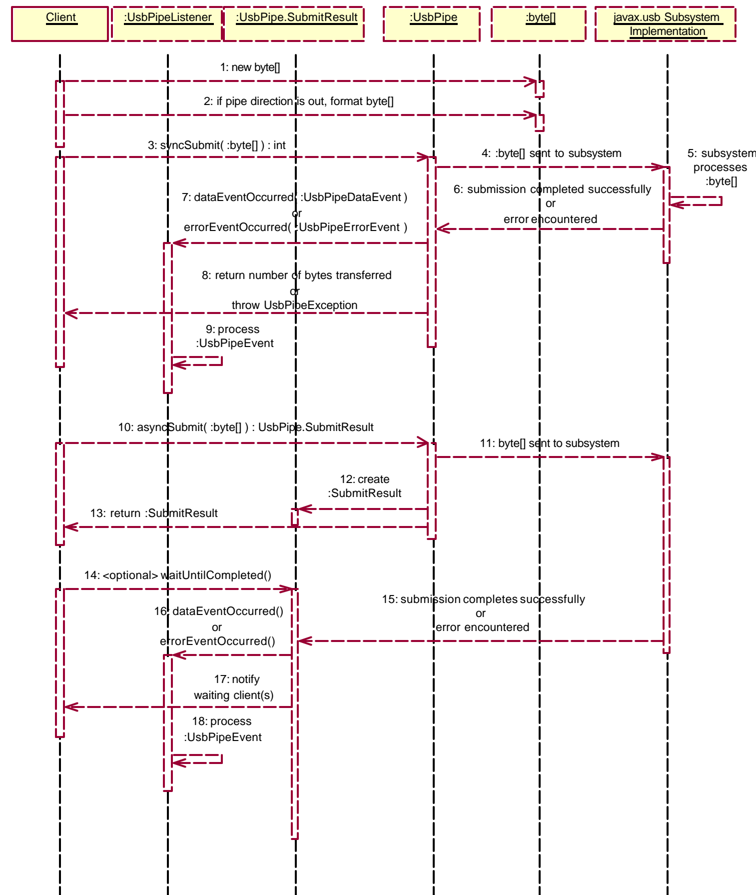

Figure 4.6.1 UsbPipe communication using a byte[]

4.6.1 USB Pipe Input and Output

A UsbPipe's endpoint's direction determines whether it may be used for input or output[12]. Only pipes with a host-to-device direction endpoint may be used for output, and only pipes with a device-to-host direction endpoint may be used for input. For input, the provided data buffer is filled with data received from the endpoint, and for output the provided data is sent to the endpoint. There is no minimum data size[13], nor is there a maximum data size[14]. If the data size is greater than the UsbPipe's maximum packet size the data will be sent in segments[15] as outlined in the USB specification section 5.3.2.

4.6.2 USB Pipe Types

There are four transfer types defined in the USB specification section 5.4 :

1. Control
2. Bulk
3. Isochronous
4. Interrupt

Except for Control pipes, the data format is determined by the client and the data direction is determined by the endpoint.

4.6.2.1 Control Pipes

There are two types of Control pipes, normal Control pipes and the Default Control Pipe. The Default Control Pipe is not directly accessible. Instead, Requests must be used. Normal Control pipes may be directly accessed, but they require a specific data format. The first eight (8) bytes of the provided data buffer is the Setup packet. The direction of data flow is determined by the bmRequestType direction bit. Be aware that all fields are little-endian according to the USB specification section 8.1. This means the word-sized fields in the Setup packet must be provided by the application in little-endian order.

1. Byte 0 is the bmRequestType
2. Byte 1 is the bRequest
3. Byte 2 is the LSB of the wValue
4. Byte 3 is the MSB of the wValue
5. Byte 4 is the LSB of the wIndex
6. Byte 5 is the MSB of the wIndex
7. Byte 6 is the LSB of the wLength
8. Byte 7 is the MSB of the wLength.

---

[12] Control pipes can be used for input and output, i.e. they are bi-directional. See this specification sec 4.6.2.1
[13] Control pipes require a 8 byte setup packet which is embedded in the data. See this specification sec 4.6.2.1
[14] Isochronous pipes impose a maximum data size of 1023 bytes per packet. See the USB specification section 5.6.3.
[15] Isochronous pipes will not segment data; i.e. one packet per submission. See this specification section 4.6.2.2

Also see the USB specification section 9.3 for more information on the format of the Setup packet. The format of the actual data portion is determined by the client and outside the scope of this specification.

4.6.2.2 Isochronous Pipes

Isochronous pipe direction is determined by its associated endpoint. Isochronous transfers are time-sensitive and more complicated than other transfers. The USB specification section 5.12.6 states that “when an isochronous transfer is presented to the Host Controller, it identifies the frame number for the first frame”. Currently, this specification does not provide a way for the application to indicate the starting frame number; instead, each submission represents a single packet, and the implementation should schedule the packet for the earliest possible frame, and maintain proper scheduling of subsequent packets as long as the pipe is busy. If the pipe becomes idle the implementation drops frame synchronization and starts over by scheduling the next packet for the earliest possible frame. The application should provide as many packets as possible so the implementation can maintain proper frame-synchronized packet scheduling. Composite submissions may be used in an ‘optimized’ way by the implementation. See the USB specification section 5.10.2 for details on isochronous optimization of multiple submissions.

Synchronization and Feedback types [USB specification section 5.12.4.1, 5.12.4.2, and 5.12.4.3] are not addressed in this version of this specification.

4.6.3 USB Pipe State Model

A pipe exists in one of two conditions: active or inactive An inactive pipe belongs to an inactive configuration and/or interface setting. An active pipe belongs to an active configuration and interface setting. Only active pipes are of interest, since inactive pipes cannot be used. For active pipes, there are two superstates : the closed state and open state. The open state has three substates : the idle, busy and error states. All active pipes start in the closed state[16]. No submissions can be made on the pipe in the closed state. When the pipe is opened it will change into the open state.

4.6.3.1 Idle/Busy State

The idle state is the first state of the open state. Submissions can be made in this state, and will cause the state to change to the busy state. The busy state indicates a submission is in progress on the pipe. When there are no more submissions in progress the state will change back to the idle state. If the pipe is closed from the idle state, it will change to the closed state; the pipe cannot be closed from the busy state. Any persistent pipe errors (e.g. a stalled pipe or removed device) that occur in any of the open states cause the state to change to the error state. The pipe can only be closed from the idle state or error state, not the busy state.

[16] The Default Control Pipe is always active and open; the client may pass Requests at any time. If the Default Control Pipe enters an error state, the device must be reset; the subsystem in some cases may automatically do this. A device’s Default Control Pipe starts in the open (idle) state, and can never change to the closed state.

4.6.3.2 *Error State*

This state indicates there is a persistent error on the pipe. No submissions are possible in this state. When changing to this state, any submissions in progress will be aborted with an appropriate UsbException. Action must be taken appropriate to the original error to change from this state. Any attempts to use the pipe while in the error state will result in a UsbException indicating the current error condition. To begin using the pipe again, the application must close the pipe and re-open the pipe (as well as correcting whatever error occurred). Note that if the error originated from a device disconnect, the error is uncorrectable and the pipe will never exit this state.

4.6.4 USB Pipe Class Diagram

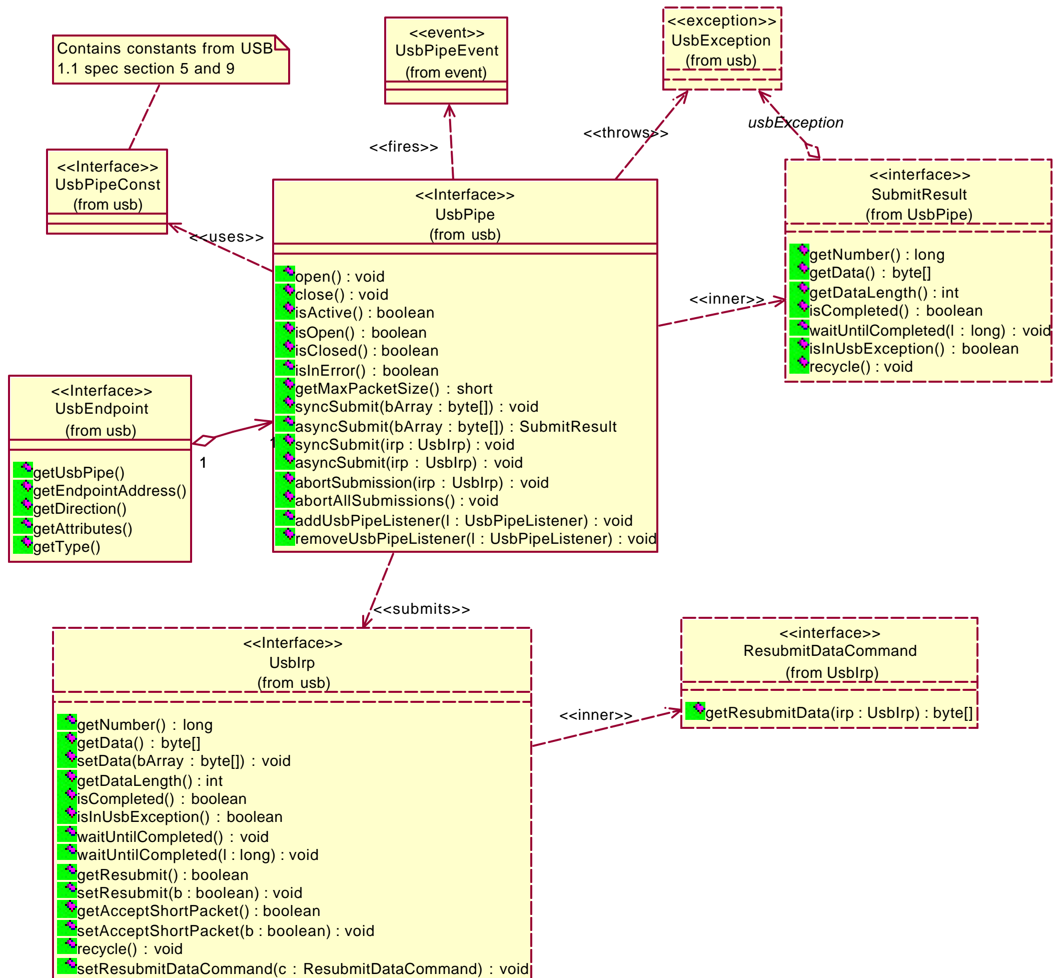

Figure 4.6.4.0 USB pipe specification class diagram

4.6.5 USB Pipe Event Class Diagram

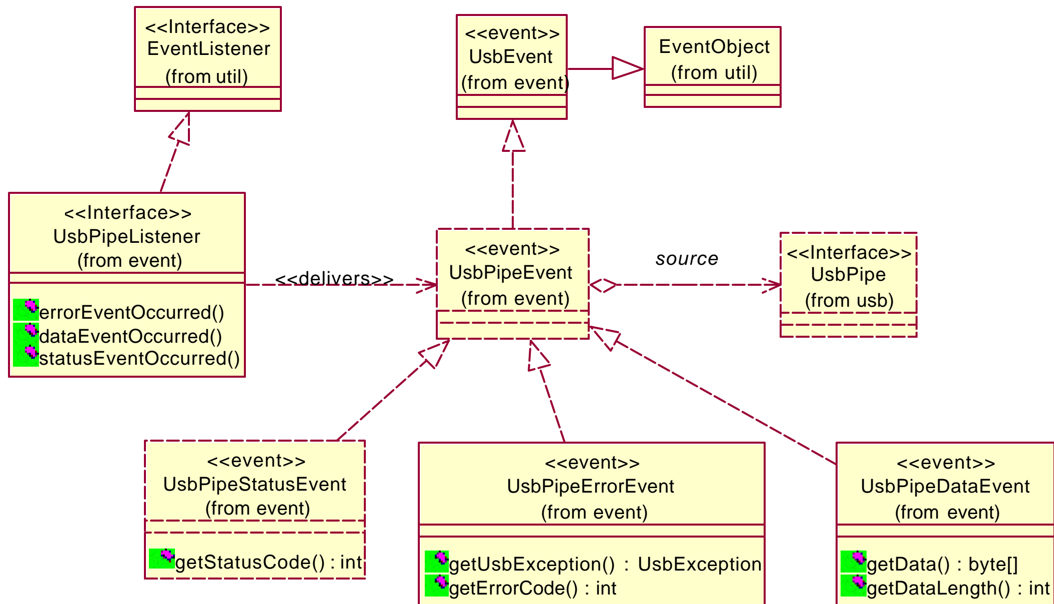

Figure 4.6.5.0 USB pipe events class diagram

Like UsbDevice, UsbPipe allows for clients to register for asynchronous events. Modeled also after the JavaBeans event model, UsbPipeEvents form a hierarchy with two kinds of events: Error and Data. A UsbPipeErrorEvent indicates that an error has occurred in a submission on the pipe. A UsbPipeDataEvent indicates asynchronous data is available for the client. Pipe states are described in section 4.6.3.

4.7 I/O Request Packets (IRPs)

Pipes provide different methods of communication. For simple communication, a data buffer may be provided which will be used in the communication (see section 4.6.0). For more complicated submissions, I/O Request Packets (IRPs) may be used. The USB specification section 5.3.2 describes IRPs. In this specification, an IRP consists of the data buffer, communication policy information, and other meta-data, in a single object. IRPs provide much more control over the submission process than using a byte[] data buffer. The diagram below indicates how to use UsbIrps for asynchronous communication.

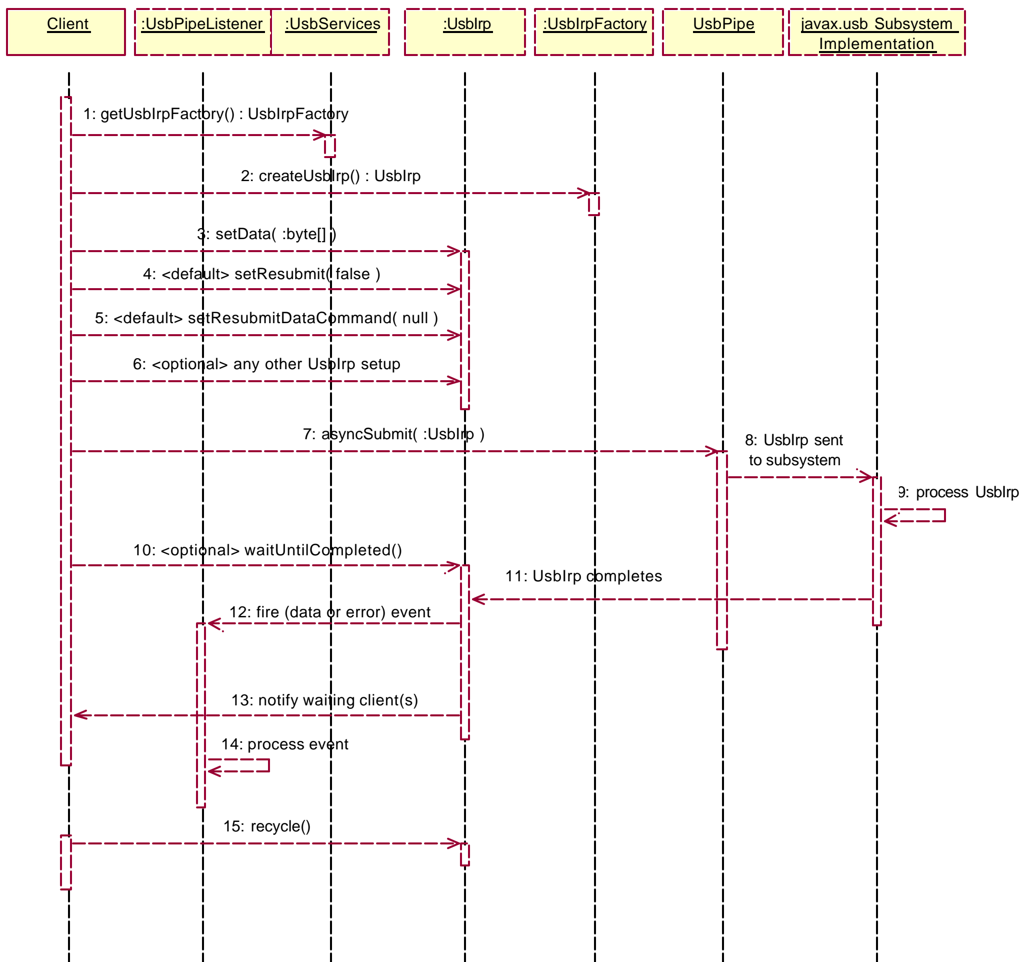

Figure 4.7.0 UsbPipe communication using a UsbIrp

IRPs may be set to automatically resubmit themselves upon completion. This is especially useful for input pipes (e.g. input interrupt pipe). If the IRP is set to automatically resubmit, it will resubmit itself immediately after completion. Its status will remain 'active', and any clients waiting for it to complete will not be notified. It will fire a data event (if appropriate) or an error event (if appropriate) through its associated pipe. Resubmission will continue until the IRP is set to not resubmit (by whatever means).

If an IRP is set to resubmit itself, it will ask the current ResubmitDataCommand[17] for the new data buffer. The default for this Command is to create a new data buffer of equal size and copy the contents into the new buffer. Clients may set the ResubmitDataCommand so they can handle processing the completed data buffer and providing the new data buffer. The ResubmitDataCommand may turn off resubmission (if the client chooses to do so). Events will still be fired normally.

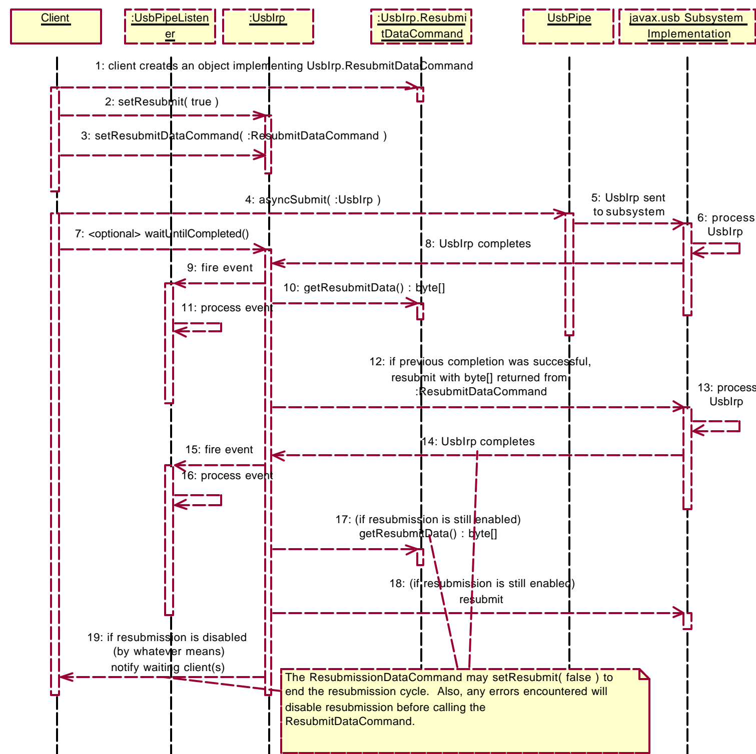

Figure 4.7.1 UsbPipe communication using a resubmitting UsbIrp

UsbIrps also allow the user to specify whether short packets should be accepted. The details are explained in the USB specification section 5.3.2. If the client specifies not to accept short packets for this UsbIrp, the subsystem will treat a short packet like a communication error, and the UsbIrp will not complete successfully. The default is to accept short packets.

4.7.1 Composite UsbIrps

A special type of UsbIrp is the UsbCompositeIrp. This object can be submitted just like a normal UsbIrp. However, instead of containing its own data buffer, it instead contains a list of individual UsbIrps. When submitted, those individual UsbIrps are used for submission. The implementation ensures that those UsbIrps are submitted uninterrupted, i.e., no other data is submitted in between two of the composite's UsbIrps. Additionally, the implementation may use optimization to handle the UsbIrps more

[17] GoF Command pattern

efficiently. The UsbCompositeIrp inherits all the methods present in a normal UsbIrp, and additionally has a UsbIrpList of its individual UsbIrps. It also contains a CompositeErrorCommand, which is executed only when one of the individual UsbIrps encounters an error. The result of the command determines whether the remaining UsbIrps should continue with their submissions. It also determines the status of the UsbCompositeIrp; if the command stops submission of the remaining UsbIrps, the UsbCompositeIrp completes with an error.

Normal operation of a UsbCompositeIrp is identical to that of a UsbIrp. The exception is instead of setting the data on a UsbCompositeIrp, the data should be set on UsbIrps, which are then added to the UsbCompositeIrp's list. Once the UsbCompositeIrp has UsbIrps added to it, it is ready for submission.

4.8 Device Event Object Model

Each USB device can be dynamically attached and detached. Because of this dynamic behavior, clients of a USB device must be able to receive asynchronous events associated with a USB device. To accomplish this goal, USB devices have an event model (patterned after the JavaBeans event model). Each USB device can have a set of UsbDeviceListeners that register to receive UsbDeviceEvent objects. These events are delivered asynchronously.

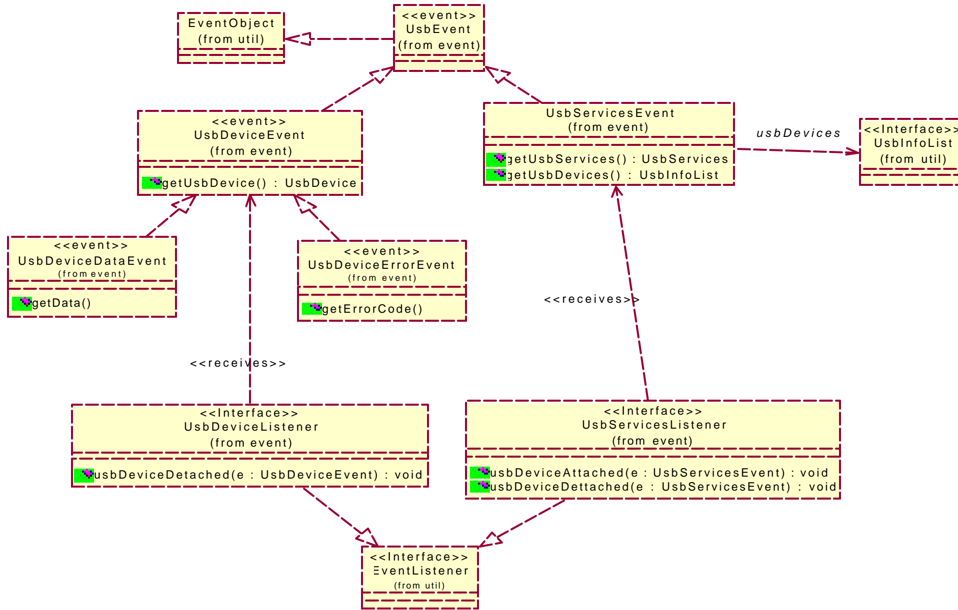

Figure 4.8.0 UsbDevice and UsbServices event class diagram

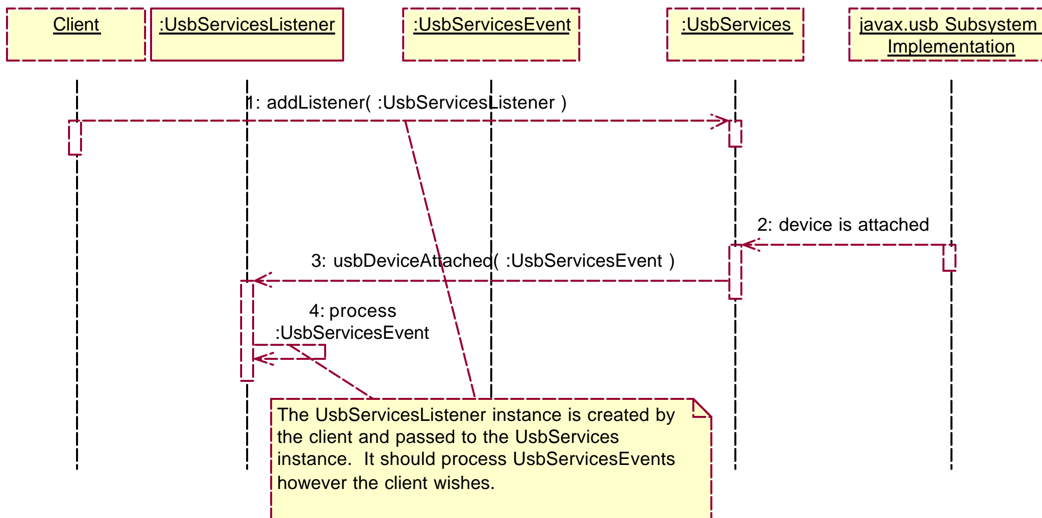

Figure 4.8.1 typical event sequence diagram

4.9 Request/StandardOperations Object Model

The Request and USB operations provide a simple mechanism for performing USB standard device operations as well as USB class and vendor defined operations or requests using javax.usb. The USB specification section 9.4 specifies a series of standard requests that all USB devices must support. These are modeled in the Request interface. The operations that these requests represent are sent to the device using the StandardOperations interface. The request results are made available using getter methods from the Request objects. Since the different type of requests have different data encoding and required values, a RequestFactory is provided to create Request objects. If the data passed to the factory or set directly on the Request object is invalid, a RequestException is thrown. Similarly, vendor and class specific request are created using the RequestFactory and submitted via the appropriate UsbOperations object. In cases where more than one Request object need to be submitted, these can be aggregated into a RequestBundle and submitted via the UsbOperations object.

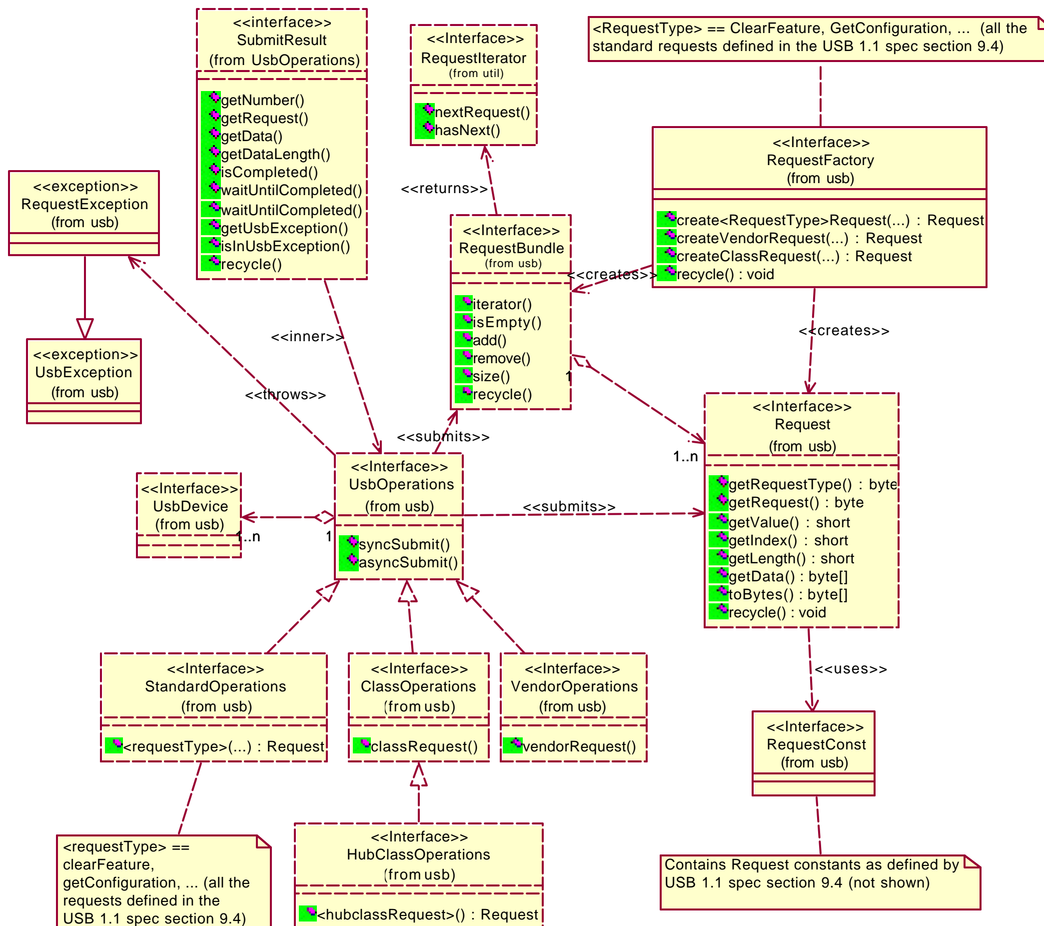

Figure 4.9.0 javax.usb Request and StandardOperations class diagram

4.9.1 Requests and USB Device Operations Usage - Dynamic Model

To execute standard USB device operations, use the Request/Standard Operations mechanism. This is accomplished by the following steps (also illustrated below in the UML sequence diagram):

a. Get the UsbServices object from the UsbHostManager.
b. Get the RequestFactory object from the UsbServices object.
c. Create the appropriate Request object via the factory in step b (above).
d. Fill in the required data for the request. Setting data that is not appropriate for the Request object might result in a RequestException to be thrown. See the USB specification section 9.4 for details on data required for each Request type.
e. Get the StandardOperations object from the UsbDevice.
f. Submit the Request object via the StandardOperations object.
g. Read the result of Request using the appropriate getter methods in the Request object.
h. Recycle the Request object via the recycle() method. Note: a Request object should no longer be used after being recycled. A new object should be obtained from the factory (which could be recycled objects) if new Requests are required.

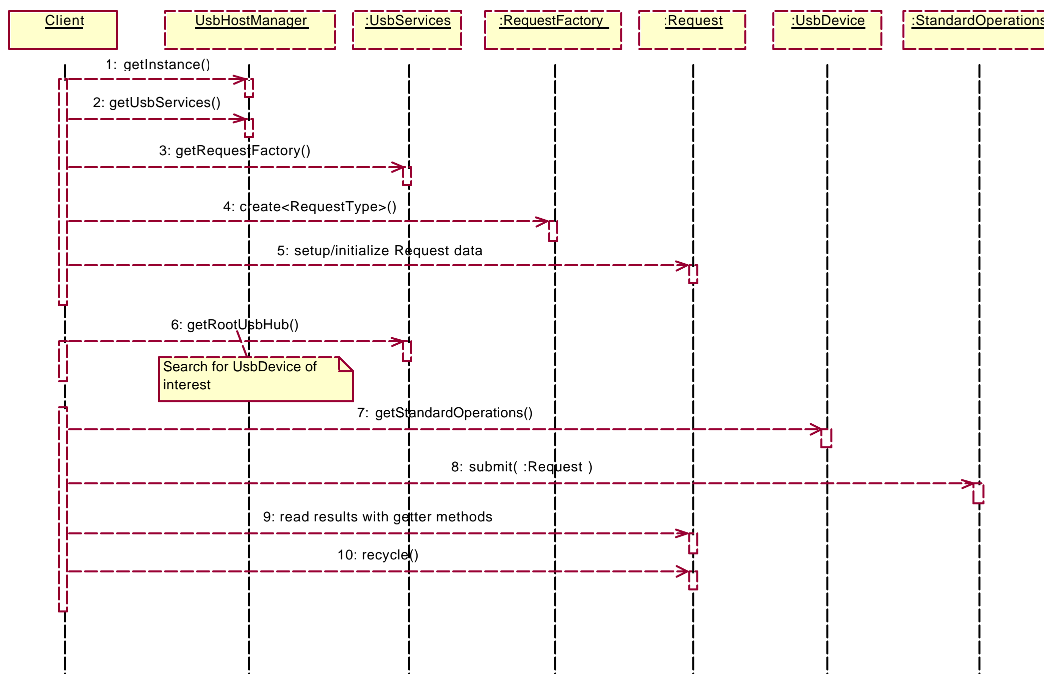

Figure 4.9.1 javax.usb Request and StandardOperations sequence diagram

5 Sample Usage: Client Application Examples

5.1 USB View Application

The javax.usb USB View application is a pure-Java application that showcases the javax.usb API. Just as other USB view applications on native platforms (like the USB View Windows application that ships with the Windows DDK), you can use it to view the current USB topology on the host and see configuration(s), interfaces and endpoints of the attached devices.

The application is written using the Swing API. The "Topology" tab shows the current USB topology on the host and the "Hub list" and "Device list" tabs filter out the hubs or devices. This is done by creating and using the sample javax.usb.util Visitor classes. Press the "Refresh" button to refresh the topology or run it in "auto-refresh" mode.

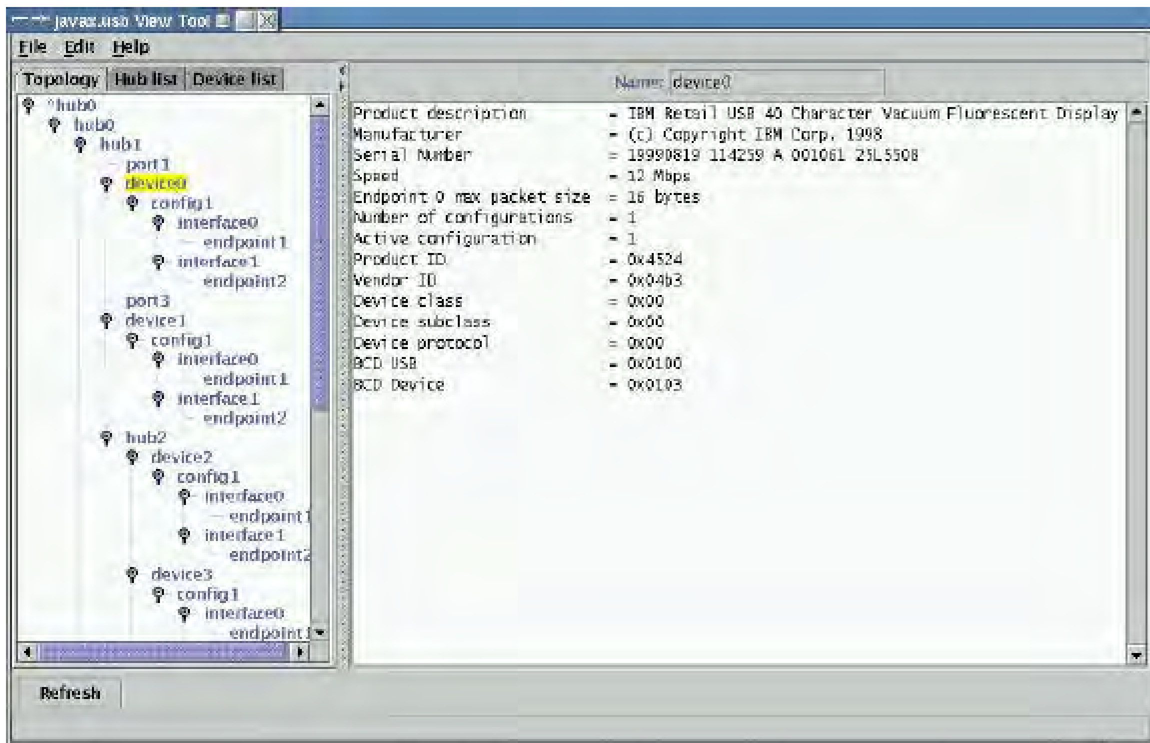

Figure 5.1.0 javax.usb USB View application main window

5.2 UsbPipe Test Application

The UsbPipeTest is an add-on tool to the USB view application, written in 100% Java, using javax.usb, that allows clients to create and use UsbPipes. It allows the user to graphically select a UsbInterface and UsbEndpoint, create and open a pipe to it and submit any USB requests to that opened pipe.

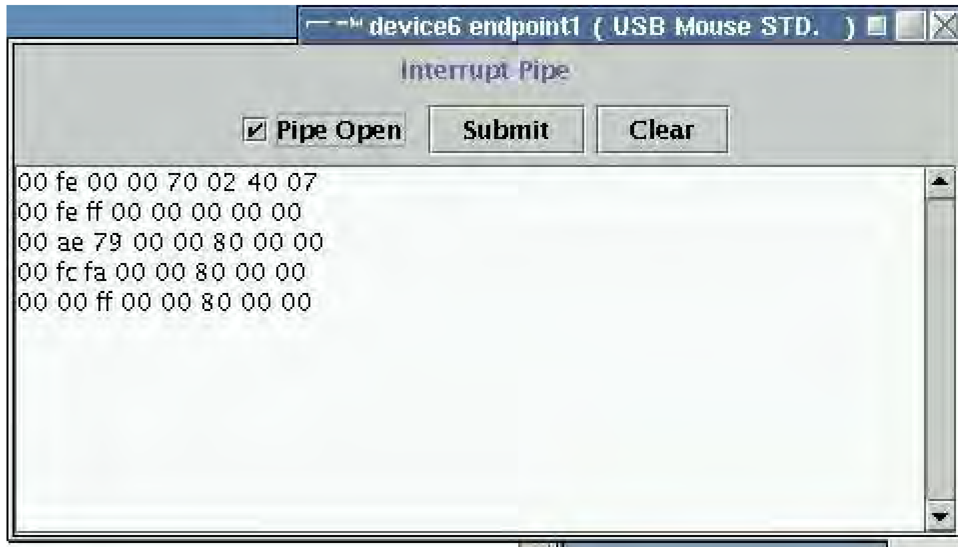

Figure 5.2.0 UsbPipe Test application showing an Interrupt pipe

6 Conclusion

The javax.usb specification is meant to enable USB on the Java platform. This specification is targeted at the J2SE and J2ME platforms. This specification only assumes that the underlying platform fully supports the USB 1.1 specification. Using the javax.usb specification, developers should be able to create services (or drivers) for their USB devices. These would allow third party applications and applets to use their USB devices. Both the device services and third party application should be portable to all javax.usb enabled platforms. The following diagram illustrates this concept:

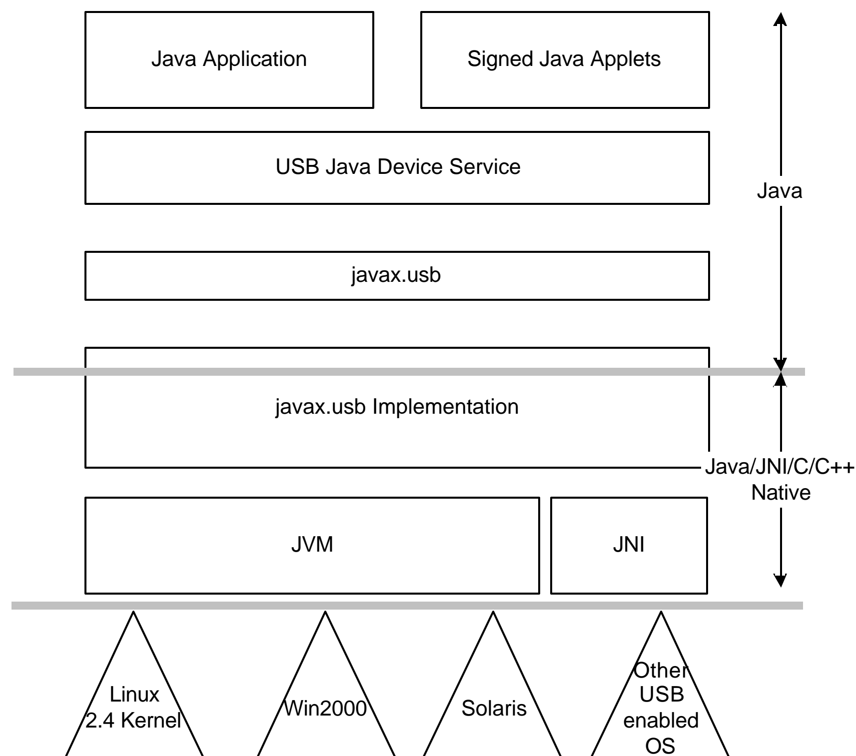

Figure 6.0.0 Java application/applets, device service and javax.usb layers

As part of the JCP process, this specification is provided as the underlying architecture and design for the javax.usb. The following are (or will be) provided as part of the JCP process:

1. The full set of classes and interfaces for this specification with intensive JavaDOC documentation.
2. A reference implementation for the Linux OS platform (originally by IBM).
3. Various utilities and tools that use javax.usb. These tools, like device services for javax.usb, will work on all Java enabled platforms with a javax.usb implementation.
4. A full set of test cases that can verify that an implementation follows the semantics of this specification.

We believe that this specification, in the spirit of the Java "Write Once, Run Anywhere™" mantra, will enable USB for the Java platform and thus open the Java community to a whole new set of applications that otherwise would not be possible.

7 Appendix A: Design Considerations and Future Releases

7.1 J2ME

The Java Micro Edition (J2ME) is the Java platform targeted for personal, portable devices. There are two configurations in J2ME, namely CLDC (Connected Limited Device Configuration) and CDC (Connected Device Configuration), and multiple profiles defined for J2ME to allow the system integrator to choose the minimum specific function required to use for a particular solution. Javax.usb will run on the CDC profile[18].

The Foundation Profile is a set of Java APIs, which, together with the CDC, provides a J2ME application runtime environment for consumer electronic and embedded devices. More study is required to identify any specific requirements that javax.usb may have on the classes in the Foundation Profile.

From a pragmatic standpoint, at this time PersonalJava is a prevailing standard for companies developing Java enabled consumer devices. Since J2SE requires more resources than these devices provide, EmbeddedJava is suitable for closed systems only, and the complement of J2ME profiles are just beginning to be defined to provide for the full matrix of requirements. More study is required to determine if the CLDC profile can be supported as well.

7.2 USB 2.0

The current installed base of system units supporting USB and the USB peripherals themselves are largely ones conforming to the USB 1.1 specification. The follow-on USB 2.0 should support this prevalent standard. The USB 2.0 design point is to be a transparent superset from a functional basis, which means that USB 1.1 devices and drivers should work on a USB 2.0 platform. So we therefore expect that the current specification and implementations of javax.usb should be applicable to USB 2.0 hosts and stack. USB 2.0 specific functions can be addressed in follow-on revisions of this API.

7.3 Claiming Interface using Policies

The javax.usb 1.0 release provides a simple API for controlling access to a device so that only one client can use a particular device interface at a time. The advantage to this claim() and release() interface is that it is simple to use and effectively protects an application's access to a claimed USB interface from interference from another application running in another JVM. This mechanism does not, however, help the application to control access if multiple applications are started in a single JVM. Any application in the JVM will be able to access the USB interface of a device claimed by any other application in the same JVM.

[18] http://www.sun.com/software/communitysource/j2me/cdc

This issue has been considered and there is a proposal under consideration to expand the claiming mechanisms to use a sort of Policy object that would give more control to the application. With this API extension, the application would provide its own policy object thus customizing the control of claiming and releasing of USB interfaces by the client application. This will allow finer granularity of control to the client application.

7.4 Support for Isochronous Transfer

Isochronous Data Transfers occupy a pre-negotiated amount of USB bandwidth with a pre-negotiated delivery latency (also called streaming real time transfers). Isochronous transfers are complicated, time-sensitive, and may not be applicable to all hardware environments.

The API currently provides support for Isochronous data transfer types for platform implementations that can provide this level of capability. It is considered by the Expert Group, that a platform can be compliant to the javax.usb API without providing this function, if it adequately responds to such API requests with the appropriate exceptions.

8 Appendix B: Frequently Asked Questions (FAQs)

8.1 What Diagnostic Capabilities are provided for use with this API?

Various utilities and tools that use the javax.usb interfaces are also provided with the reference implementation. PipeView and the UsbView utilities are available in a package named javax.usb.tools. These utilities are written in Java for use with any reference implementation. They are indispensable as example code for reference implementations as well as for diagnostics for applications using the API. UsbView can be used to identify the current USB ports that are attached and the endpoints and interfaces for the devices present.

There are tab selectors for "Topology" showing the current USB topology on the host, the "Hub list" to list the attached hubs, and "Device list" filtering out the connected devices. Also supported are the "Refresh" function to refresh the topology and an "auto-refresh" mode. The UsbView is also designed as a JavaBean allowing integration of its functionality into other tools.

The PipeView tool allows the user to create USB pipes and exercise the device access. It allows the user to graphically select a USB device interface and endpoint, create and open a pipe to it, and submit any USB requests to that opened pipe. Since the tool actually communicates to the device, the input/output stream delivery can be tested via the specific device responses. The PipeView also shows how to integrate the UsbView into another application.

8.2 Why not create a Java communication API for all device communications via the Java 2 platform that would include, for instance, Bluetooth, rather than a specific API for USB?

The most direct answer to this question is in the realm of performance and pragmatism. An API to talk to communication devices needs to have the least amount of overhead. Many (like us at IBM) will be creating middleware that is used by application writers to talk to attached devices. This API must have minimal path overhead and at the same time expose all device functionality in a manner that fits the type of device attached. In addition, there has already been an expert group (JSR 82) formed to create a low level Java API for bluetooth technology. See http://java.sun.com/aboutJava/communityprocess/jsr/jsr_082_bluetooth.html for details on the work in progress for bluetooth.

It is also notable that it would be possible to create a middleware layer by providing a wrapper class to abstract all of the device accesses to the different APIs to meet this need. This would be useful to applications that want to use a middleware driver without caring about the underlying connectivity issues. In other words, the middleware driver would use the connectivity specific API and expose another API to the application.

The javax.comm design point was RS232/Parallel and it is not a good design base for other bus architectures. It would most likely need to be changed to come up with a unified serial approach. This would not be acceptable to the industry since there is extensive javax.comm device support already in place.

As far as a unified serial API, we have given it some thought and here are some issues with that idea. The issues are captured in the list of alternatives given below and the problems that occur with each approach. This analysis supports our conclusion that each connection type (e.g. RS232/Parallel, USB, FireWire, Bluetooth, etc) will most likely need its own API. There are 3 possible approaches that could be used to make a unified serial API:

1. Design to the lowest common denominator (which could be RS232). This would limit the features exposed by the generic API. For example, USB devices can have multiple sub-devices (called USB interfaces) and each interface can have multiple endpoints, which can all have communication channels (called USB pipes) open to them. There is no adequate mapping of this structure of interfaces and endpoints to serial-like communication.
2. Design a complex API that supports all serial communications and enables the correct functions for the type of device attached. There are various issues with this:
    a. This is a questionable design approach since the API for a particular bus would only implement a subset of the whole API, which clients need. For instance, USB devices are dynamically configured but RS232 devices are statically configured.
    b. Since clients would talk to a small subset of the big API, why not separate them? Clients would need to know what type of device they are talking to and select the correct API functions.
    c. This complex API would take time to reach agreement and would delay the implementation.
    d. It is not clear that we would have a workable solution in a reasonable amount of time. USB is just now being supported on most platforms (e.g. GNU/Linux, Solaris, AIX, etc). FireWire is also in its infancy on many platforms other than the Mac OS and Win32, and it would be premature to implement an API for it now.
3. Take the opposite approach of (2) and design a very abstract API that each particular bus API would extend to add its particular features. The approach is actually the most feasible; however, there are a few issues that makes it undesirable:
    a. This would require incompatible changes to the javax.comm, which is already in use.
    b. The various buses are different enough that this abstract API would not be useful in and of itself, and users would need to access the particular bus API. If that is the case, then why not have separate APIs?

c. This would also mean that in creating this abstract API we would need to foresee the future and support all future communication APIs. This would be impractical.

This line of reasoning leads to the conclusion that a separate API is the most practical and likely to succeed. The JSR 80 specification and working reference implementation will attest to the validity of this approach.

8.3 What are the expected performance characteristics of a Java API for USB*?*

USB's actual throughput is a function of many variables. One of these is certainly the efficiency of the API implementation; however other significant ones are the target device's ability to source or sink data, the bandwidth consumption of other devices on the bus, and the efficiency of the underlying operating system's USB software stack. In some cases, PCI latencies and processor loading can also be critical. Performance of device handling software written in Java should approximate the speed of the same function written in C++ on the same platform. This is based on the assumption of an appropriate JIT (Just in Time) compiler for the platform and adequate memory to minimize paging.

Assuming that only the target endpoint consumes a significant amount of bus bandwidth and both the target and the host are able to source or sink data as fast as USB can move it, the maximum attainable bandwidth is a function of the transfer type and signaling rate. These bandwidths are given in chapter 5 of the USB specification. In practice, most hosts can reach the maximum isochronous and interrupt bandwidths with a single target endpoint. With bulk transfers, typical transfer rates are around 900 kb/s to a single endpoint, increasing to near ideal transfer rates with multiple endpoints.

8.4 What is the design language used in this specification?

The UML used in this specification was done using Rational Rose 2000e.

Two main categories of UML diagrams are used in this specification:

1. Static structure diagrams: these are typically class and package diagrams. They show the static relationships (that is relationship that are constant with time) of classes/interfaces with other classes/interfaces.
2. Dynamic diagrams: these are sequence diagrams and state chart diagrams. These diagrams show the dynamic relationship (that is relationship over time) of objects with other objects or within one object.

Interfaces are either shown using the lollipop symbol or a box symbol adorned with the <<interface>> stereotype. A UML interface maps directly to a Java interface.

All other boxes are classes. They typically are plain classes (i.e. not adorned) or adorned with an <<event>> or <<utility>> stereotype. The <<event>> stereotype indicates an event class and the <<utility>> indicates a class that typically has static methods used as services by other classes. Classes typically do not show attributes since

that would violate encapsulation. Showing attributes also gives too much information about the class implementation; and furthermore, the attributes of a class can be inferred from the relationships of a class with other classes. The operation compartment of a class or interface typically shows critical operations only. This is done to keep the static structure diagrams simple.

Classes show an inheritance relationship (for instance, a relationship) with a open arrow pointing from the subclass to the superclass. This is a straightforward “extends” in Java. Implementation of interfaces is shown just like inheritance except that the line of the arrow is dashed.

Classes and interfaces show aggregation relationship (i.e. containment, whole-part relationship) with a line between the two classes. The “whole” class has its side of the line adorned with a diamond. The other class (the “part” class) either has an arrow or nothing. Multiplicity of containment is indicated by numbers or * to indicate 0 or many.

Relationships between classes can also be shown by a dashed line arrow between the classes/interfaces. This is a dependency relationship. This means that the class/interface on the start of the arrow depends on the other. That dependency can be further refined by indicating a stereotype like <<uses>> or <<creates>> which as the name suggests, means that one class/interface uses the other (in some way, like for instance a Const interface, an interface defining constant objects, is typically used by various classes/interfaces) or means that one class/interface creates instances of the other. This could indicate that the class/interface is a factory for the other.

There is a general philosophical note by Booch, Rumbaugh and Jacobson in the various UML books listed in the References section of this document. UML is a modeling language, it is not meant to execute on the machine, it just allows quick, efficient overview and creation of complex models to a level of abstraction that is higher than looking at source code. That said, it is a great tool to create, explain, specify, introduce a complex (or non obvious) piece of software to other engineers. In other words, it is a way to communicate. However, UML cannot (and is not meant to) replace code. Once the big UML picture is communicated, engineers should refer to the source files or the JavaDOC (which is the closest thing to source files).

In sequence diagrams, method calls between objects that terminates in : <Type> implies that a instance of <Type> is returned to the caller.

For a quick, gentle and short introduction to the UML, see Martin Fowler’s book listed below “UML Distilled”.

8.5 How do I use the UsbInfoVisitor interface?

The Visitor pattern allows an external client to add methods to a hierarchy of classes/interfaces without having access or modifying the sources. The hierarchy has to be stable otherwise the Visitor code will need to change. For a complete and thorough discussion of the Visitor pattern including advantages/disadvantages as well as where its use is applicable, please refer to Design Pattern book [Gamma95, 331].

The best explanation of how to use the UsbInfoVisitor is to look at a simple example of filtering the UsbRootHub for a particular device that one is looking for. For that we will:

1. Create a FilterUsbDeviceV visitor. This class will extend DefaultUsbInfoV and add two setter methods (set the vendor and product ID) and one getter method to return the UsbDevice found (if any)
2. Show a code snippet that uses that visitor by:
    a. Create an instance of FilterUsbInfoV
    b. Set the vendor and product ID of the device that needs to be searched for (filtered)
    c. Get the UsbRootHub from the UsbServices
    d. Visit the UsbRootHub to filter out the device
    e. Get the resulting filtered device if found

```
import javax.usb.*;

public class FilterUsbDeviceV extends DefaultUsbInfoV
{
  public UsbDevice getUsbDevice() { return usbDevice; }

  public void setVendorId( short vId ) { vendorId = vId; }

  public void setProductId( short pId ) { productId = pId; }

  public void visitUsbDevice( UsbInfo usbInfo )
  {
    UsbDevice device = (UsbDevice)usbInfo;

    if( device.getVendorId() == vendorId && device.getProductId() == productId
)
      usbDevice = device;
  }

  public void visitUsbHub( UsbInfo usbInfo )
  {
    UsbHub hub = (UsbHub)usbInfo;

    UsbInfoIterator iterator = hub.getAttachedUsbDevices().usbInfoIterator();

    while( iterator.hasNext() )
      iterator.nextUsbInfo().accept( this );
  }

  private UsbDevice usbDevice = null;

  private short vendorId = 0;
  private short productId = 0;
}

<client-code-snippet>
//...

FilterUsbDeviceV visitor = new FilterUsbDeviceV();

visitor.setVendorId( 0x1234 );
visitor.setProductId( 0x5678 );

UsbRootHub rootHub = UsbHostManager.getInstance().
  getUsbServices().getUsbRootHub();

rootHub.accept( visitor );

UsbDevice device = visitor.getUsbDevice();

//...

</client-code-snippet>
```

Of course this is a simple example which assumes, for instance, that there is only one such <vendorId, productId> device attached to the hub, but nevertheless, it clearly shows the power of visitors.

9 Appendix C: Change Summary

Changes resulting in document revisions will be summarized in this table in reverse chronological sequence. Revision bars (|) will highlights the text changed in the new document versions.

Version	Date	Change Description
0.9.0	6/29/2001	1. Some minor improvements and clarifications on the Pipe section 2. Request section has be improved (diagrams and a bit of text)
0.8.0	4/5/2001	1. Added more illustrative UML static structure diagrams 2. Added more dynamic structure diagrams 3. Expanded the FAQ section 4. Added a "Future Consideration" section 5. Various improvements to existing paragraphs to make them flow better
0.0.3	12/15/2000	1. More spelling/grammar errors fixed (when the technical content of paragraphs have not changed due to spelling/grammar error, revision bars "|" are not used) 2. Using Dewy dotted decimal notation in entire specification 3. Some UML diagrams improvements, especially showing key method names and signatures 4. Added info and UML diagrams about USB request and operations
0.1.0	02/19/2001	1. Spelling/grammar errors fixed 2. Added more English description for each UML diagrams 3. Added an appendix for explaining the usage of UML in this document 4. Added explanation on the UsbPipe model and how to use it

		5. Added various description on the different sections
0.0.1	10/2000	Original

10 References

10.1 Web

[W1] "JSR-80, Java API for USB"
http://java.sun.com/aboutJava/communityprocess/jsr/jsr_080_usb.html

[W2] "JavaPOS Specification" Sun, IBM, NCR, Epson, Fujitsu-ICL, NRF, Datafit, MGV, RCS, et al., http://www.javapos.com

[W3] "UML Document" http://www.rational.com/uml/

[W4] "USB 1.1 Specification" http://www.usb.org/developers/data/usbspec.zip

[W5] "USB 2.0 Specification" http://www.usb.org/developers/data/usb_20.zip

[W6] "Linux USB Project" http://www.linux-usb.org

[W7] "Java Communication API" http://java.sun.com/products/javacomm/index.html

10.2 Books

[Gamma95] E. Gamma et al. "Design Patterns: Elements of Reusable O-O Software" Addison Wesley 1995.

[Lea00] D. Lea "Concurrent Programming in Java: Design Principles and Patterns" second edition, Addison Wesley 2000.

[Booch98] Booch, G. et al "The Unified Modeling Language: User Guide" Addison Wesley 1998.

[Rumbaugh99] Rumbaugh, J. et al "The Unified Modeling Language: Reference" Addison Wesley, 1999.

[Fowler99] Fowler, M. et al "UML Distilled" Addison Wesley 1999, second edition.