Proposal for a Drag and Drop subsystem for the Java Foundation Classes

(Draft: 0.95).

Laurence P. G. Cable.

THIS IS A DRAFT SPECIFICATION, IT IS THEREFORE SUBJECT TO CHANGE, AND FURTHERMORE IMPLIES NO INTENT ON BEHALF OF JavaSoft TO DELIVER SUCH BEHAVIOR

Send comments to java-beans@java.sun.com.

Note:

The API described herein is partially implemented in JDK1.2 Beta2 but will not be completely available until Beta3.

1.0 Requirements

This proposal is based upon an (incomplete) earlier work undertaken in 1996 to specify a Uniform Data Transfer Mechanism, Clipboard, and Drag and Drop facilities for AWT.

The AWT implementation in JDK1.1 introduced the Uniform Data Transfer Mechanism and the Clipboard protocol. This draft proposal defines the API for the Drag and Drop facilities for JDK1.2 based upon, but extending these 1.1 UDT API's.

The primary requirements that this proposal addresses, are:

1. Provision of a platform independent Drag and Drop facility for Java GUI clients implemented through AWT and JFC classes.
2. Integration with platform dependent Drag and Drop facilities, permitting Java clients to be able to participate in DnD operation with native applications using:
    - OLE (Win32) DnD
    - CDE/Motif dynamic protocol
    - MacOS
3. Support for 100% pure JavaOS/Java implementation.
4. Leverages the existing *java.awt.datatransfer.** package to enable the transfer of data, described by an extensible data type system based on the MIME standard.
5. Does not preclude the use of "accessibility" features where available.
6. Extensible to support diverse input devices.

The proposal derives from the previous work mentioned above, but incorporates significant differences from that original work as a result of the advent of the JavaBeans event model, Lightweight Components, and an increasing understanding of the cross-platform integration and interoperability issues.

2.0 API

2.1 Overview

Drag and Drop is a direct manipulation gesture found in many Graphical User Interface systems that provides a mechanism to transfer information between entities associated with a presentation element in the GUI. Normally driven by the physical gesturing of a human user, Drag and Drop provides both a mechanism to enable feedback regarding the possible outcome of any subsequent data transfer to the user during navigation over the presentation elements in the GUI, and the facilities to provide for any subsequent data transfer.

A typical Drag and Drop operation can be decomposed into the following states (not entirely sequentially):

- A *DragSource* comes into existence, associated with some presentation element (*Component*) in the GUI, and some potentially *Transferable* data.
- 1 or more *DropTarget*(s) come into/go out of existence, associated with presentation elements in the GUI (*Components*), potentially capable of consuming *Transferable* data.
- A human user gestures to initiate a Drag and Drop operation on a *Component* associated with a *DragSource*.

  Note: Although the body of this document consistently refers to the stimulus for a drag and drop operation being a physical gesture by a human user this does not preclude a programmatically driven DnD operation given the appropriate implementation of a *DragSource*.
- The *DragSource* initiates the Drag and Drop operation on behalf of the user, perhaps animating the GUI *Cursor* and/or rendering an *Image* of the item(s) that are the subject of the operation.
- As the user gestures navigate over *Components* in the GUI associated with *DropTarget*(s), the *DragSource* receives notifications in order to provide “Drag Over” feedback effects, and the *DropTarget*(s) receive notifications in order to provide “Drag Under” feedback effects.

  The gesture itself moves a logical cursor across the GUI hierarchy, intersecting the geometry of GUI *Component*(s), possibly resulting in the logical “Drag” cursor entering, crossing, and subsequently leaving *Components* associated *DropTarget*(s).

  The *DragSource* object manifests “Drag Over” feedback to the user, in the typical case by animating the GUI *Cursor* associated with the logical cursor.

  DropTarget objects manifest “Drag Under” feedback to the user, in the typical case, by rendering animations into their associated GUI *Component*(s) under the GUI *Cursor*.
- The determination of the feedback effects, and the ultimate success or failure of the data transfer, should one occur, is parameterized as follows:

- By the transfer “operation”: **Copy**, **Move** or **Reference**(link**)**.
- By the intersection of the set of data types provided by the *DragSource* and the set of data types comprehensible by the *DropTarget*.

- When the user terminates the drag operation, normally resulting in a successful Drop, both the *DragSource* and *DropTarget* receive notifications that include, and result in the transfer of, the information associated with the *DragSource* via a *Transferable* object.

The remainder of this document details the proposed API changes to support this model.

2.2 Drag Source

The *DragSource* is the entity responsible for the initiation of the Drag and Drop operation:

2.2.1 The *DragSource* definition

The *DragSource* and associated constant interfaces are defined as follows:

The *DnDConstants* class defines the operations that may be applied to the subject of the transfer:

```
public class java.awt.dnd.DnDConstants {
     public static int ACTION_NONE= 0x0;
     public static int ACTION_COPY= 0x1;
     public static int ACTION_MOVE= 0x2;
     public static int ACTION_COPY_OR_MOVE= ACTION_COPY |
                                           ACTION_MOVE;
     public static int ACTION_REFERENCE = 0x40000000;
}

public class java.awt.dnd.DragSource {

     public static Cursor     getDefaultCopyDropCursor();
     public static Cursor     getDefaultMoveDropCursor();
     public static Cursor     getDefaultLinkDropCursor();

     public static Cursor     getDefaultCopyNoDropCursor();
     public static Cursor     getDefaultMoveNoDropCursor();
     public static Cursor     getDefaultLinkNoDropCursor();

     public static DragSource getDefaultDragSource();

     public void
            startDrag(Component          c,
                      AWTEvent           trigger,
                      int                actions,
                      Cursor             dragCursor,
                      Image              dragImage,
                      Point              dragImageOffset,
                      Transferable       transferable,
                      DragSourceListener dsl)
            throws InvalidDnDOperationException;

    protected DragSourceContext
         createDragSourceContext(
              DragSourceContextPeer dscp,
              Component             c,
              int                   actions,
              Cursor                dragCursor,
              Image                 dragImage,
              Point                 dragImageOffset,
              Transferable          transferable,
              DragSourceListener    dsl
         );

    public FlavorMap getFlavorMap();
}
```

The *DragSource* may be used in a number of scenarios:

- 1 default instance per JVM for the lifetime of that JVM. (defined by this spec)
- 1 instance per class of potential Drag Initiator object (e.g *TextField*). [implementation dependent]
- 1 per instance of a particular *Component*, or application specific object associated with a *Component* instance in the GUI. [Implementation dependent]
- some other arbitrary association. [implementation dependent]

A controlling object, the Drag Initiator, shall obtain a *DragSource* instance either prior to, or at the time a users gesture, effecting an associated *Component,* in order to process the operation.

The initial interpretation of the users gesture, and the subsequent starting of the Drag operation are the responsibility of the implementing *Component*, or associated controlling entity.

When a gesture occurs, the *DragSource*'s *startDrag()* method shall be invoked in order to cause processing of the users navigational gestures and delivery of Drag and Drop protocol notifications.

In order to start a drag operation the caller of the *startDrag*() method shall provide the following parameters:

- The *Component* that received the *AWTEvent* that was interpreted as the starting gesture.
- The *AWTEvent* itself that was interpreted as the starting gesture.
- The Drop actions that may be performed; the union of ***ACTION_COPY***, ***ACTION_MOVE***, and ***ACTION_REFERENCE***, as appropriate.
- A *Cursor* representing the initial "Drag Over" feedback for the operation(s) specified. (This shall be a *Cursor* that provides "No Drop" visual feedback to the user).
- An (optional) *Image* to visually represent the item, or item(s) that are the subject(s) of the operation.

  On platforms that can support this feature, a "Drag" image may be associated with the operation to enhance the fidelity of the "Drag Over" feedback. This image would typically be a small "iconic" representation of the object, or objects being dragged, and would be rendered by the underlying system, tracking the movement of, and coincident with, but typically in addition to the *Cursor* animation.

  Where this facility is not available, or where the image is not of a suitable type to be rendered by the underlying system, this parameter is ignored and only *Cursor* "Drag Over" animation results, so applications should not depend upon this feature.
- Where an *Image* is provided; a *Point* (in the co-ordinate space of the *Component)* specifying the initial origin of that *Image* in the *Component* for the purposes of initiating "Drag Over" animation of that *Image*.
- A *Transferable* that describes the various *DataFlavor*(s) that represent the subject(s) of any subsequent data transfer that may result from a successful Drop.

  The *Transferable* instance associated with the *DragSource* at the start of the Drag operation, represent the object(s) or data that are the operand(s), or the subject(s), of the Drag and Drop operation, that is the information that will subsequently be passed from the *DragSource* to the *DropTarget* as a result of a successful Drop on the *Component* associated with that *DropTarget*.

  Note that multiple (collections) of either homogeneous, or heterogeneous, objects may be subject of a Drag and Drop operation, by creating a container object, that is the subject of the transfer, and implements *Transferable*. However it should be noted that since none of the targeted native platforms systems support a standard mechanism for describing and thus transferring such collections it is not possible to implement such transfers in a transparent, or platform portable fashion.
- A *DragSourceListener* instance, which will subsequently receive events notifying it of changes in the state of the ongoing operation in order to provide the "Drag Over" feedback to the user.

As stated above, the primary role of the *startDrag*() method is to initiate a Drag on behalf of the user. In order to accomplish this, the *startDrag*() method must create a *DragSourceContext* instance to track the operation itself, and more importantly it must initiate the operation itself in the underlying platform implementation. In order to accomplish this, the *DragSource* must first obtain a *DragSourceContextPeer* from the underlying system (usually via an invocation of *java.awt.Toolkit.createDragSourceContextPeer()* method) and subsequently associate this newly created *DragSourceContextPeer* (which provides a platform independent interface to the underlying systems capabilities) with a *DragSource-Context*.The *startDrag*() method invokes the *createDragSourceContext*() method to instantiate an appropriate *DragSourceContext* and associate the *DragSourceContextPeer*.

If the Drag and Drop System is unable to initiate a Drag operation for some reason the *startDrag*() method shall throw a *java.awt.dnd.InvalidDnDOperationException* to signal such a condition. Typically this exception is thrown when the underlying platform system is either not in a state to initiate a Drag, or the parameters specified are invalid.

Note that during the Drag neither the set of operations the source, nor the set of *DataFlavors* exposed by the *Transferable* at the start of the Drag operation may change for the duration of the operation, in other words the operation(s) and data are constant for the duration of the operation with respect to the *DragSource*.

For security reasons the caller of the *startDrag*() method is required to have the AWTPermission "`startDrag`", invoking this method without such permission shall result in a *SecurityException* being thrown.

The *getFlavorMap*() method is used by the underlying system to obtain a *FlavorMap* object in order to map the *DataFlavors* exposed by the *Transferable* to data type names of the underlying DnD platform. [see later for details of the *FlavorMap*]

2.2.2 The *DragSourceContext* Definition

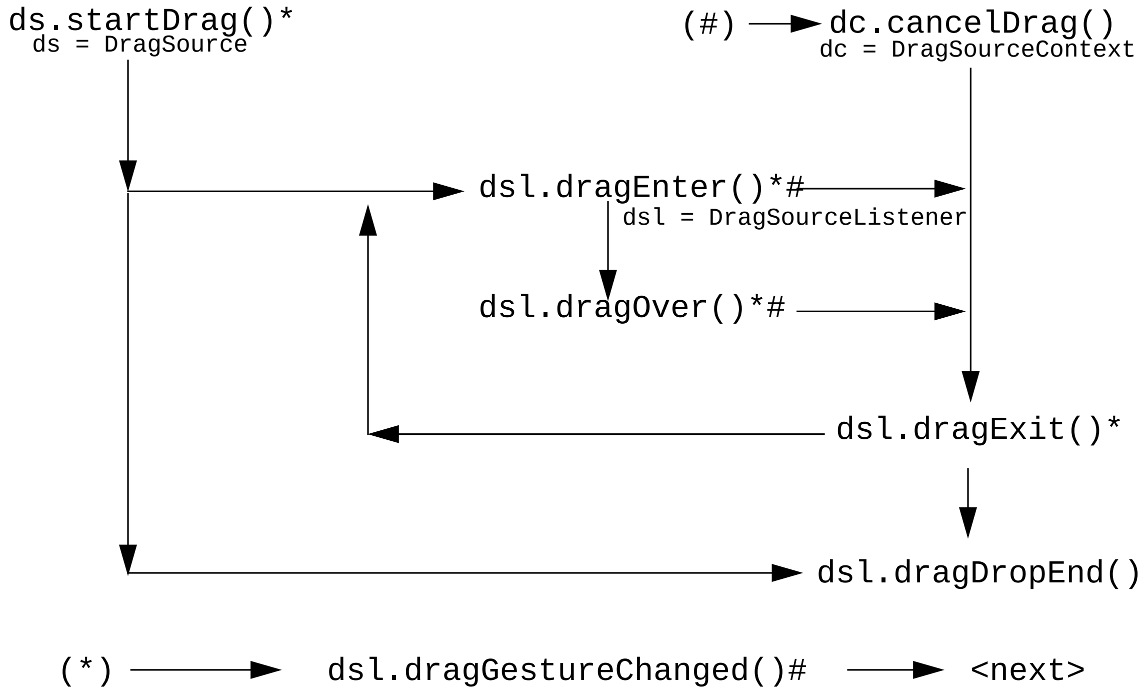
As a result of a *DragSource*'s *startDrag*() method being successfully invoked an instance of the *DragSourceContext* class is created. This instance is responsible for tracking the state of the operation on behalf of the *DragSource* and dispatching state changes to the *DragSourceListener.*

The *DragSourceContext* class is defined as follows:

```
public class DragSourceContext implements DragSourceListener
{

     protected DragSourceContext(
                    DragSource           ds,
                    DragSourceContextPeerdscp,
                    int                  actions,
                    Cursor               dragCursor,
                    Image                dragImage,
                    Point                dragOffset,
                    Transferable         transferable,
                    DragSourceListener   dsl
     );

     public DragSource   getDragSource();

     public Component    getComponent();
     public AWTEvent     getTrigger();

     public Image        getDragImage();

     public Point        getDragImageOffset();


     public int          getSourceActions();

     Cursor getCursor();
     void   setCursor(Cursor Cursor)
                 throws InvalidDnDOperationException;

     void cancelDrag() throws InvalidDnDOperationException;

     void addDragSourceListener(DragSourceListener dsl)
          throws TooManyListenersException;

     void removeDragSourceListener(DragSourceListener dsl);
}
```

Note that the *DragSourceContext* itself implements *DragSourceListener*, this is to allow the platform peer, the *DragSourceContextPeer* instance, created by the *DragSource*, to notify the *DragSourceContext* of changes in state in the ongoing operation, and thus allows the *DragSourceContext* to interpose itself between the platform and the *DragSourceListener* provided by the initiator of the operation.

The state machine the platform exposes, with respect to the source, or initiator of the Drag and Drop operation is detailed below:



Notifications of changes in state with respect to the initiator during a Drag and Drop operation, as illustrated above, are delivered from the *DragSourceContextPeer,* to the appropriate *DragSourceContext*, which delegates notifications, via a unicast JavaBeans compliant *EventListener* subinterface, to an arbitrary object that implements *DragSourceListener* registered with the *DragSource* via *startDrag*().

The primary responsibility of the *DragSourceListener* is to monitor the progress of the users navigation during the Drag and Drop operation and provide the “Drag-Over” effects feedback to the user. Typically this is accomplished via changes to the “Drag Cursor”.

Every *DragSource* object has 2 logical cursor states associated with it:

- The **Drop** *Cursor*, the cursor displayed when dragging over a valid *DropTarget*.
- The **NoDrop** *Cursor*, the cursor displayed when dragging over everything else (the initial state of the cursor at the start of a Drag).

The state of the *Cursor* may be modified by calling the *setCursor*() method of the *DragSourceContext*.

2.2.3 The *DragSourceListener* Definition

The *DragSourceListener* interface is defined as follows:

```
public interface java.awt.dnd.DragSourceListener
     extends java.util.EventListener {
     void dragEnter        (DragSourceDragEvent dsde);
     void dragOver         (DragSourceDragEvent dsde);
     void dragGestureChanged(DragSourceDragEvent dsde);
     void dragExit          (DragSourceEvent     dse);
     void dragDropEnd       (DragSourceDropEvent dsde);
}
```

As the drag operation progresses, the *DragSourceListener*'s *dragEnter(), dragOver(),* and *dragExit()* methods shall be invoked as a result of the users navigation of the logical "Drag" cursor's location intersecting the geometry of GUI *Component(s)* with associated *DropTarget(s)*. [See below for details of the *DropTarget's* protocol interactions].

The *DragSourceListener*'s *dragEnter*() method is invoked when the following conditions are true:

- The logical cursor's hotspot initially intersects a GUI *Component*'s visible geometry.
- That *Component* has an active *DropTarget* associated.
- The *DropTarget*'s registered *DropTargetListener dragEnter*() method is invoked and returns successfully.
- The registered *DropTargetListener* invokes the *DropTargetDragEvent*'s *acceptDrag*() method to accept the Drag based upon interrogation of the source's potential Drop actions and available data types (*DataFlavors*).

The *DragSourceListener's dragOver()* method is invoked when the following conditions are true:

- The cursor's logical hotspot has moved but still intersects the visible geometry of the *Component* associated with the previous *dragEnter*() invocation.
- That *Component* still has a *DropTarget* associated.
- That *DropTarget* is still active.
- The *DropTarget*'s registered *DropTargetListener dragOver*() method is invoked and returns successfully.
- The *DropTarget* does not reject the drag via *rejectDrag*().

The *DragSourceListener*'s *dragExit*() method is invoked when one of the following conditions is true:

- The cursor's logical hotspot no longer intersects the visible geometry of the *Component* associated with the previous *dragEnter*() invocation.

Or:

- The *Component* that the logical cursor's hotspot intersected that resulted in the previous *dragEnter*() invocation, no longer has an active *DropTarget* (or *DropTargetListener*) associated.

Or:

- The current *DropTarget*'s *DropTargetListener* has invoked *rejectDrag*() since the last *dragEnter*() or *dragOver*() invocation.

The *DragSourceListener's dragGestureChanged()* method is invoked when the state of the input device(s), typically the mouse buttons or keyboard modifiers, that the user is interacting with in order to preform the Drag operation, changes.

The *dragDropEnd()* method is invoked to signify that the operation is completed. The *isDropAborted()* and *isDropSuccessful()* methods of the *DragSourceDropEvent* can be used to determine the termination state. The *getDropAction*() method returns the operation that the *DropTarget* selected (via the *DropTargetDropEvent acceptDrop*() parameter) to apply to the Drop operation.[1]

Once this method is complete the *DragSourceContext* and the associated resources are invalid.

2.2.4 The *DragSourceEvent* Definition

The *DragSourceEvent* class is the root *Event* class for all events pertaining to the DragSource, and is defined as follows:

```
public class   java.awt.dnd.DragSourceEvent
       extends java.util.EventObject {
     public DragSourceEvent(DragSourceContext dsc);

     public DragSourceContext getDragSourceContext();
};
```

An instance of this event is passed to the *DragSourceListener dragExit*() method.

2.2.5 The *DragSourceDragEvent* Definition

The *DragSourceDragEvent* class is defined as follows:

```
public class java.awt.dnd.DragSourceDragEvent
       extends DragSourceEvent {
     public int getTargetActions();

     public int getGestureModifiers();

     public boolean isDropTargetLocal();
}
```

1. It would be nice to design an API that would allow the *DragSource* to be notified of the *DropTarget*'s selected operation before the DropTarget invokes the source *Transferable*'s *getTransferData*() method, sadly however, OLE's bass-ackwards DnD protocol forces the above design on us where the operation is reported after it has occurred, this makes life for the source implementor harder when supporting certain "Link" semantics.

An instance of the above class is passed to a *DragSourceListener's dragEnter(), dragOver(),* and *dragGestureChanged()* methods.

The *getDragSourceContext*() method returns the *DragSourceContext* associated with the current Drag and Drop operation.

The *getTargetActions()* method returns the drop actions, supported by, and returned from the current *DropTarget (*if any in the case of *dragGestureChanged()).*

The *getGestureModifiers()* returns the current state of the input device modifiers, usually the mouse buttons and keyboard modifiers, associated with the users gesture.

The *isDropTargetLocal*() method returns `true` if the current *DropTarget* is contained within the same JVM as the *DragSource,* and `false` otherwise. This information can be useful to the implementor of the *DragSource*'s *Transferable* in order to implement certain local optimizations.

2.2.6 The *DragSourceDropEvent* Definition

The *DragSourceDropEvent* class is defined as follows:

```
public public class java.awt.dnd.DragSourceDropEvent
       extends java.util.EventObject {

     public DragSourceDropEvent(DragSourceContext dsc);
     public DragSourceDropEvent(DragSourceContext dsc,
                                int               action,
                                boolean           success);

     public boolean isDropAborted();
     public boolean isDropSuccessful();

     public int getDropAction();
}
```

An instance of the above class is passed to a *DragSourceListener's dragDropEnd()* method. This event encapsulates the termination state of the Drag and Drop operation for the *DragSource*.

If the Drop occurs, then the participating *DropTarget* will signal the success or failure of the data transfer via the *DropTargetContext's dropComplete()* method, this status is made availlable to the initiator via the *isDropSuccessful()* method. The operation that the destination DropTarget selected to perform on the subject of the Drag (passed by the *DropTarget*'s *acceptDrop*() method) is returned via the *getDropAction*() method.

If the Drag operation was aborted for any reason prior to a Drop occurring, for example if the users ends the gesture outwith a *DropTarget*, or if the *DropTarget* invokes *rejectDrop*() , the *isDropAborted()* method will return `false`, otherwise `true.`

2.3 Drop Target

2.3.1 java.awt.Component additions for DropTarget (de)registration.

The *Java.awt.Component* class has two additional methods added to allow the (dis)association with a *DropTarget.* In particular:

```
public class java.awt.Component /* ... */ {
     // ...

     public synchronized
               void      setDropTarget(DropTarget dt);

     public synchronized
               DropTarget getDropTarget(DropTarget df);

     //
}
```

To associate a *DropTarget* with a *Component* one may invoke either; *DropTarget.setCompononent*() or *Component.setDropTarget*() methods. Thus conforming implementations of both methods are required to guard against mutual recursive invocations.

To disassociate a *DropTarget* with a *Component* one may invoke either; *DropTarget.setCompononent*(`null)` or *Component.setDropTarget*(`null`) methods.

Conformant implementations of both setter methods in *DropTarget* and *Component* should be implemented in terms of each other to ensure proper maintenance of each other’s state.

The *setDropTarget*() method throws *IllegalArgumentException* if the *DropTarget* actual parameter is not suitable for use with this class/instance of *Component*. It may also throw *UnsupportedOperationException* if, for instance, the *Component* does not support external setting of a *DropTarget*.

A caller of the *setDropTarget*() method requires the *AWTPermission:* “`setDropTarget`”, if the caller does not have this permission then *setDropTarget*() will throw a *SecurityException*.

2.3.2 The *DropTarget* Definition

A *DropTarget* encapsulates all of the platform-specific handling of the Drag and Drop protocol with respect to the role of the recipient or destination of the operation.

A single *DropTarget* instance may typically be associated with any arbitrary instance of *java.awt.Component*. Establishing such a relationship exports the associated *Component's* geometry to the client desktop as being receptive to Drag and Drop operations when the coordinates of the logical cursor intersects that visible geometry.

The *DropTarget* class is defined as follows:

```
public class java.awt.dnd.DropTarget
       implements DropTargetListener, Serializable {

     public DropTarget();

     public DropTarget(Component c);
     public DropTarget(Component c, DropTargetListener dsl);

     public Component getComponent();
     public void      setComponent(Component c);

     public DropTargetContext getDropTargetContext();


     public void
         addDropTargetListener(DropTargetListener dte)
            throws TooManyListenersException;

     public void
         removeDropTargetListener(DropTargetListener dte);

     public void    setActive(boolean active);
     public boolean isActive();

     public FlavorMap getFlavorMap();

     protected DropTargetContext createDropTargetContext();

     public void addNotify(ComponentPeer cp);
     public void removeNotify(ComponentPeer cp);
}
```

The *setComponent*() method throws *IllegalArgumentException* if the *Component* actual parameter is not appropriate for use with this class/instance of *DropTarget*, and may also throw *UnsupportedOperationException* if the Component specified disallows the external setting of a *DropTarget*.

For security reasons, callers of *setComponent*() require the *AWTPermission* "`setDropTarget`" otherwise a *SecurityException* shall be thrown.

The *addDropTargetListener*() and *removeDropTargetListener*() methods allow the unicast *DropTargetListener* to be changed.

The *setActive*() and *isActive*() methods allow the *DropTarget* to be made active or otherwise and for its current state to be determined.

The *getFlavorMap*() methods is used to obtain the *FlavorMap* associated with this *DropTarget* for the purposes of mapping any platform dependent type names to/from their corresponding platform independent *DataFlavors*.

The *createDropTargetContext*() method is typically only invoked to provide the underlying platform dependent peer with an instantiation of a new *DropTargetContext* as a Drag operation initially encounters the *Component* associated with the *DropTarget*. If no *DropTargetContext* is currently associated with a *DropTarget,* a permitted side-effect of an invocation of *getDropTargetContext*() is to instantiate a new *DropTargetContext*.

The *addNotify*() and *removeNotify*() methods are only called from *Component* to notify the *DropTarget* of the *Component*'s (dis)association with its *ComponentPeer.*

2.3.3 The *DropTargetContext* Definition

As the logical cursor associated with an ongoing Drag and Drop operation first intersects the visible geometry of a *Component* with an associated *DropTarget*, the *DropTargetContext* associated with the *DropTarget* is the interface, through which, access to control over state of the recipient protocol is achieved from the *DropTargetListener.*

The *DropTargetContext* interface is defined as follows:

```
public class DropTargetContext {
     public DropTarget getDropTarget();

     public Component getComponent();

     public DataFlavor[] getDataFlavors();

     public void getTransferable()
                    throws InvalidDnDOperationException;

     public void dropComplete(boolean success)
                    throws InvalidDnDOperationException;

     protected void acceptDrop(int action);
     protected void rejectDrop();

     public void addNotify(DropTargetContextPeer dtcp);
     public void removeNotify();

     protected Transferable
```

```
        createTransferableProxy(Transferable t,
                                boolean      isLocal
        );
}
```

The *getDataFlavors*() method returns an array of the *DataFlavors* available from the *DragSource*.

The *getTransferable*() method returns a *Transferable* (not necessarily the one the *Drag-Source* registered, it may be a proxy, and certainly shall be in the inter-JVM case) to enable data transfers to occur via its *getTransferData*() method. Note that it is illegal to invoke *getTransferData*() without first invoking an *acceptDrag*().

The *addNotify*() and *removeNotify*() methods are exclusively called by the underlying platform's *DropTargetContextPeer* in order to notify the *DropTargetContext* that a DnD operation is occurring/ceasing on the *DropTargetContext* and associated *DropTarget*.

The *createTransferableProxy*() method enables a *DropTargetContext* implementation to interpose a *Transferable* between the *DropTargetListener* and the *Transferable* provided by the caller, which is typically the underlying platform *DropTargetContextPeer.*

2.3.4 The *DropTargetListener* Definition

Providing the appropriate "Drag-under" feedback semantics, and processing of any subsequent Drop, is enabled through the *DropTargetListener* asssociated with a *DropTarget*.

The *DropTargetListener* determines the appropriate "Drag-under" feedback and its response to the *DragSource* regarding drop eligibility by inspecting the sources suggested actions and the data types available.

A particular *DropTargetListener* instance may be associated with a *DropTarget* via *addDropTargetListener()* and removed via *removeDropTargetListener()* methods.

```
public interface java.awt.dnd.DropTargetListener
       extends java.util.EventListener {
     void dragEnter            (DropTargetDragEvent dtde);
     void dragOver             (DropTargetDragEvent dtde);
     void dragExit             (DropTargetDragEvent dtde);
     void drop                 (DropTargetDropEvent dtde);
}
```

The *dragEnter()* method of the *DropTargetListener* is invoked when the hotspot of the logical "Drag" Cursor intersects a visible portion of the *DropTarget's* associated *Component's* geometry. The *DropTargetListener*, upon receipt of this notification, shall interrogate the operations or actions, and the types of the data (*DataFlavors*) as supplied by the *DragSource* to determine the appropriate actions and "Drag-under" feedback to respond with invocation of either *acceptDrag*() or *rejectDrag*().

The *dragOver()* method of the *DropTargetListener* is invoked while the hotspot of the logical “Drag” Cursor, in motion, continues to intersect a visible portion of the *DropTarget’s* associated *Component’s* geometry. The *DropTargetListener*, upon receipt of this notification, shall interrogate the operation “actions” and the types of the data as supplied by the *DragSource* to determine the appropriate “actions” and “Drag-under” feedback to respond with an invocation of either *acceptDrag*() or *rejectDrag*().

The get*CursorLocation()* method return the current co-ordinates, relative to the associated *Component’s* origin, of the hotspot of the logical “Drag” cursor.

The *getSourceActions()* method return the current “actions”, or operations (***ACTION_MOVE***, ***ACTION_COPY***, or ***ACTION_REFERENCE***) the *DragSource* associates with the current Drag and Drop gesture.

The *dragExit()* method of the *DropTargetListener* is invoked when the hotspot of the logical “Drag” Cursor ceases to intersect a visible portion of the *DropTarget’s* associated *Component’s* geometry. The *DropTargetListener*, upon receipt of this notification, shall undo any “Drag-under” feedback effects it has previously applied.

The *drop()* method of the *DropTargetListener* is invoked as a result of the *DragSource* invoking its *commitDrop()* method. The *DropTargetListener*, upon receipt of this notification, shall perform the operation specified by the return value of the *getSourceActions()* method on the *DropTargetDropEvent* object, upon the *Transferable* object returned from the *getTransferable()* method, and subsequently invoke the *dropComplete(*) method of the associated *DropTargetContext* to signal the success, or otherwise, of the operation.

2.3.5 The *DropTargetDragEvent* and *DropTargetDropEvent* Definitions

The *DropTargetEvent* and *DropTargetDragEvent* are defined as follows:

```
public abstract class java.awt.dnd.DropTargetEvent
       extends java.util.EventObject[1] {

     public DropTargetContext getDropTargetContext();

}
```

A *DropTargetEvent* is passed to the *DropTargetListener*’s *dragExit*() method.

```
public class java.awt.dnd.DropTargetDragEvent
       extends java.awt.dnd.DropTargetEvent {
     public DataFlavor[] getDataFlavors();

     Point  getCursorLocation();

     public int getSourceActions();
```

1. This could be a subclass of AWTEvent but there seems little motivation to make it so.

```
     public void acceptDrag(int operations);
     public void rejectDrag();

     public boolean isTransferableLocal();
}
```

A *DropTargetDragEvent* is passed to the *DropTargetListener's dragEnter()* and *dragOver()* methods.

The get*CursorLocation()* method return the current co-ordinates, relative to the associated *Component's* origin, of the hotspot of the logical "Drag" cursor.

The *getSourceActions()* method return the current "actions", or operations (***ACTION_MOVE***, ***ACTION_COPY***, or ***ACTION_REFERENCE***) the *DragSource* associates with the current Drag and Drop gesture.

The *getDataFlavors()* method returns the available type(s), in descending order of preference of the data that is the subject of the Drag and Drop operation.

The *DropTargetDropEvent* is defined as follows:

```
public class java.awt.dnd.DropTargetDropEvent
       extends java.awt.dnd.DropTargetEvent {

     Point  getCursorLocation();

     public int getSourceActions();

     public void acceptDrop(int dropAction);
     public void rejectDrop();

     public boolean isTransferableLocal();

     public Transferable getTransferable();

}
```

A *DropTargetDropEvent* is passed to the *DropTargetListener's drop()* method, as the Drop occurs (initiated by the *DragSource* via an invocation of *commitDrop()*). The *DropTarget-DropEvent* provides the *DropTargetListener* with access to the Data associated with the operation, via the *Transferable* returned from the *getTransferable*() method.

The return value of the *getSourceActions()* method is defined to be the action(s) defined by the source at the time at which the Drop occurred.

The return value of the *getCursorLocation()* method is defined to be the location at which the Drop occurred.

The *DropTargetListener.drop*() method shall invoke *acceptDrop*() with the selected operation as an actual parameter, prior to any invocation of *getTransferData*() on the Transferable associated with the Drop.

The *rejectDrop*() may be called to reject the Drop operation.

2.3.6 Autoscrolling support

Many GUI *Component*s present a scrollable "viewport" over a (potentially) large dataset. During a Drag and Drop operation it is desirable to be able to "autoscroll" such "viewports" to allow a user to navigate over such a dataset, scrolling to locate a particular member (initially not visible through the "viewport") that they wish to drop the subject of the operation upon.

Components that are scrollable provide Drag "autoscrolling" support to their *DropTarget* by implementing the following interface:

```
public interface DragAutoScrollingSupport {
     Insets getAutoscrollInsets();

     void autoScrollContent(Point cursorLocn);
}
```

An implementing *DropTarget* shall repeatedly call, at least every *Toolkit.getAutoscrollRepeatDelay*() milliseconds, the *autoScrollContent*() method of its associated *Component* (if present), passing the current logical cursor location in *Component* co-ordinates, when the following conditions are met:

- If the logical cursor's hotspot intersects with the associated *Component*'s visible geometry and the boundary region described by the *Insets* returned by the *getAutoscrollInsets*() method.
- If the logical cursor's hotspot has not moved (subject to the next condition below) for at least *Toolkit.getAutoscrollInitialDelay()* millseconds
- If any cursor movement subsequent to the initial triggering occurrence continues to intersect the *Rectangle* returned by *Toolkit.getAutoscrollCursorHysteresis*().

Should any of the above conditions cease to be valid, autoscrolling shall terminate until the next triggering condition occurs.

In order to support Autoscrolling the Toolkit class has been augmented as follows:

```
public class Toolkit {
     // ...

     public int getAutoscrollInitialDelay(); // ms
     public int getAutoscrollRepeatDelay();  // ms

     public Rectangle
               getAutoscrollCursorHysteresis(Point cc);
```

```
    // ...
}
```

The *getAutoscrollCursorHysteresis*() method returns a hysteresis rectangle surrounding the point specified (current cursor location) to be used for hit testing during hysteresis detection while autoscrolling.

2.4 Data Transfer Phase

In the case where a valid drop occurs, the *DropTargetListener's drop()* method is responsible for underatking the transfer of the data associated with the gesture. The *DropTargetDropEvent* provides a means to obtain a *Transferable* object that represent that data object(s) to be transferred.

From the *drop*() method, the *DropTargetListener* shall initially either *rejectDrop*() (immediately returning thereafter) or *acceptDrop*() specifying the selected operation from those returned by *getSourceActions*().

Subsequent to an *acceptDrop*(), *getTransferable*() may be invoked, and any data transfers performed via the returned *Transferable*'s *getTransferData*() method. Finally, prior to returning the *drop*() method shall signal the success of any transfers via an invocation of *dropComplete*().

Upon returning from the *drop*() method the *Transferable* and *DragSourceContext* instances are no longer guaranteed to be valid and all references to them shall be discarded by the recipient to allow them to be subsequently garbage collected.

When using the ***ACTION_REFERENCE*** operation the source and destination should take care to agree upon the object and the associated semantics of the transfer. Typically in intra-JVM transfers a live object reference would be passed between source and destination, but in the case of inter-JVM transfers, or transfers between native and Java applications, live object references do not make sense, so some other 'reference' type should be exchanged such as a URI for example. Both the *DragSource* and *DropTarget* can detect if the transfer is intra-JVM or not.

2.4.1 FlavorMap and SystemFlavorMap

All the target DnD platforms represent their transfer data types using a similar mechanism, however the representations do differ. The Java platform uses MIME types encapsulated within a *DataFlavor* to represent its data types. Unfortunately in order to permit the tranfer of data between Java and platform native applications the existence of these platform names need to be exposed, thus a mechanism is required in order to create an extensible (platform independent) mapping between these platform dependent type names, their representations, and the Java MIME based *DataFlavors*.

The implementation will provide a mechanism to externally specify a mapping between platform native data types (strings) and MIME types (strings) used to construct *DataFlavors*. This external mapping will be used by the underlying platform specific implementation code in order to expose the appropriate *DataFlavors* (MIME types), exported by the source, to the destination, via the underlying platform DnD mechanisms.

Both the *DragSource* and *DropTarget* classes provide access for the underlying system to map platform dependent names to and from *DataFlavors*.

```
public interface FlavorMap {
     java.util.Map getNativesForFlavors(DataFlavor[] dfs);
     java.util.Map getFlavorsForNatives(String[] natives);
}
```

The *getNativesForFlavors*() method takes an array of *DataFlavor*s and returns a *Map* object containing zero or more keys of type *DataFlavor*, from the actual parameter *dfs*, with associated values of type *String*, which correspond to the platform dependent type name for that MIME type.

The *getFlavorsForNatives*() method takes an array of *String* types and returns a Map object containing zero or more keys of type *String*, from the actual parameter natives, with associated values of type *DataFlavor*, which correspond to the platform independent type for that platform dependent type name.

The *Map* object returned by both methods may be mutable but is not required to be.

For example on Win32 the Clipboard Format Name for simple text is "CF_TEXT" (actually it is the integer 1) and on Motif it is the X11 Atom named "STRING", the MIME type one may use to represent this would be "text/plain charset=us-ascii". Therefore a platform portable *FlavorMap* would map between these names; CF_TEXT on win32 and STRING on Motif/X11.

Typically, as implemented in the *SystemFlavorMap* these mappings are held in an external persistent configuration format (a properties file or URL) and are loaded from the platform to configure the *FlavorMap* appropriately for a given platform.

The *SystemFlavorMap* class is provided to implement a simple, platform configurable mechanism for specifying a system-wide set of common mappings, and is defined as follows:

```
public class SystemFlavorMap implements FlavorMap {
     public static FlavorMap getSystemFlavorMap();

     public synchronized Map
          getNativesForFlavors(DataFlavor[] dfs);

     public synchronized Map
          getFlavorsForNatives(String[] natives);

     public static String
```

```
          encodeJavaMIMEType(DataFlavor df);

     public static String
          encodeJavaMimeType(java.util.mime.MimeType mime);

     public static boolean
          isEncodedJavaMimeType(String mimeStr);

     public static DataFlavor
          createFlavorFromEncodedJavaMimeType(String ejmts);

     public static java.util.mime.MimeType
          createMimeTypeFromEncodedJavaMimeType(
                         String ejmts
          );
}
```

The *SystemFlavorMap* class provides a simple implementation, using a properties file (see *java.awt.Properties*), of a persistent platform *FlavorMap*. Using the value of the AWT property "`AWT.flavorMapFileURL`" (see *Toolkit.getProperty*()) or the default file location of *System.getProperty("*`java.home`*") + File.separator + "lib" + File.separator + "*`flavormap.properties`*"*, this class creates the appropriate *Map*s from the properties found therein.

In addition the class provides several static convenience functions used to encode and decode Java *MimeType*s to and from a platform dependent namespace. The syntax of the properties file is:

{ <platform_type_name> '=' <IETF_MIME_RFC_conformant_specification> <nl> } *

The default implementations of *DragSouce* and *DropTarget* return the *SystemFlavorMap* from their *getFlavorMap*() method.

3.0 Issues

3.0.1 What are the implications of the various platform protocol engines?

Due to limitations of particular underlying platform Drag and Drop and Window System implementations, the interaction of a Drag operation, and the event delivery semantics to AWT *Components* is platform dependent. Therefore during a drag operation a *DragSource* may process platform Window System Events pertaining to that drag to the exclusion of normal event processing.

Due to interactions between the single-threaded design center of the platform native DnD systems, and the native window system event dispatching implementations in AWT, "call-backs" into *DropTargetListener* and *DragSourceListener* will occur either on, or synchronized with the AWT system event dispatch thread. This behavior is highly undesirable for security reasons but is an implementation, not architectural, feature, and is unavoidable.

3.0.2 Inter/Intra VM transfers?

To enable intra-JVM Drag and Drop Transfers the existing *DataFlavor* class will be extended to enable it to represent the type of a "live" object reference, as opposed to a Serialized (persistent) representation of one. Such objects may be transferred between source and destination within the same JVM and *ClassLoader* context.

3.0.3 Lifetime of the Transferable(s)?

Transferable objects, their associated *DataFlavor*s, and the objects that encapsulate the underlying data specified as the operand(s) of a drag and drop operation shall remain valid until the *DragSourceListener*, associated with the *DragSource* controlling the operation, receives a *dragDropEnd*() event.

3.0.4 Implications of ACTION_MOVE semantics on source objects exposed via *Transferable*?

The "source" of a successful Drag and Drop (***ACTION_MOVE***) operation is required to delete/relinquish all references to the object(s) that are the subject of the *Transferable* immediately after transfer has been successfully completed.

3.0.5 Semantics of *ACTION_REFERENCE* operation.

As a result of significant input from developers to an earlier version of the specification an additional operation/action tag; ***ACTION_REFERENCE*** was added to include existing platform Drag and Drop"Link" semantics.

It is believed that Reference, or Link, semantics are already sufficiently poorly specified for the platform native Drag and Drop to render it essentially useless even between native applications, thus between native and platform independent Java applications it is not recommended.

For Java to Java usage the required semantic; within the same JVM/*ClassLoader*, is defined such that the destination shall obtain a Java object reference to the subject(s) of the transfer. Between Java JVM's or *ClassLoader*s, the semantic is implementation defined, but could be implemented through transferring a URL from the source to the destination.

Appendix A : DropTargetPeer definition

Although not a normative part of this specification this definition is included for clarity:

```
public interface DropTargetPeer {

    void addDropTarget(DropTarget dt);

     void removeDropTarget(DropTarget dt);
}
```

Appendix B : DragSourceContextPeer definition

Although not a normative part of this specification this definition is included for clarity:

```
public interface DragSourceContextPeer {

    void startDrag(DragSourceContext ds,
                   AWTEvent          trigger,
                   Cursor            c,
                   int               actions
    ) throws InvalidDnDOperationException;


    Component getComponent();


    void cancelDrag() throws InvalidDnDOperationException;


    Cursor getCursor();


    void setCursor(Cursor c)
          throws InvalidDnDOperationException;


    AWTEvent getTrigger();
}
```

Appendix C : DropTargetContextPeer definition

Although not a normative part of this specification this definition is included for clarity:

```
public interface DropTargetContextPeer {

    DropTarget getDropTarget();

    DataFlavor[] getTransferDataFlavors();

    Transferable getTransferable()
                 throws InvalidDnDOperationException;

    boolean isTransferableJVMLocal();

    void acceptDrag(int dragAction);

    void rejectDrag();

    void acceptDrop(int dropAction);

    void rejectDrop();

    void dropComplete(boolean success);
}
```