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

(Draft: 0.6).

Laurence P. G. Cable.

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

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 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.

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 sensory feedback to that user during navigation over the presentation elements in the GUI, originating from the source of the drag, and usually terminating in a drop over a target, between which, any subsequent data transfer occurs.

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

- A Drag Source comes into existence, associated with some presentation element in the GUI, and some potentially transferable data.
- 1 or more Drop Targets come into/go out of existence, associated with presentation elements in the GUI, potentially capable of consuming transferable data.
- A human user gestures to initiate a Drag and Drop operation on a presentation element in the GUI associated with a Drag Source.

  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.
- As the user gestures navigate over presentation elements in the GUI associated with Drop Target(s), the Drag Source receives notifications in order to provide “Drag Over” feedback effects, and the Drop Target(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 Components, possibly resulting in the logical “Drag” cursor entering, crossing, and subsequently leaving associated Drop Targets.

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

  Drop Target objects manifest “Drag Under” feedback to the user, in the typical case, by rendering animations into their associated GUI Components 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 Drag Source and the set of data types comprehensible by the Drop Target.

- When the user terminates the drag operation, normally resulting in a successful Drop, both the Drag Source and Drop Target receive notifications that include, and result in the transfer of, the information associated with the Drag Source.

2.2 Drag Source

The *DragSource* is the entity responsible for the co-ordination of the Drag and Drop operation for the initiating client.

2.2.1 The *DragSource* definition

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

```
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 DragSource getDragSource(Component c);
                      Point imageOffset,

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

     public void   setDefaultDragCursor(Cursor c);
     public Cursor getDefaultDragCursor();

     public void   setDefaultDropCursor(Cursor c);
     public Cursor getDefaultDropCursor();
```

```
    public void   setDefaultNoDropCursor(Cursor c);
    public Cursor getDefaultNoDropCursor();

    public boolean isDragTrigger(AWTEvent trigger);

    public Rectangle getDragThresholdBBox(Component c,
                                          Point hotspot);
}
```

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, will obtain a *DragSource* instance either prior to, or at the time a users gesture, effecting an associated *Component,* in order to both determine if the gesture does in fact trigger a Drag and Drop operation, and to subsequently process the resulting operation itself.

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

Window system platforms usually define a set of gestures (typically mouse or keyboard related) that are associated with a Drag and Drop operation. The *DragSource* provides a mechanism to allow testing of an *AWTEvent* stream to determine if any of the events contained within are gestures that should initiate a Drag operation on the particular platform, in particular the *isDragTrigger()* method will inspect an *AWTEvent* to determine if it is a Drag triggering event. In addition, the *getDragThresholdBBox()* method returns a *Rectangle*, centered at the *Point* specified by the *hotspot* parameter. This “threshold” defines the platform default minimum amount by which the hotspot of the “logical” cursor must move, in either *x* or *y* coordinates in order to complete the triggering gesture.

When such 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. The *DragSource*, if successful in starting a Drag operation, returns a *DragSourceContext* instance to the caller of s*tartDrag().* The *startDrag()* method’s parameters include the *AWTEvent* that triggered the operation, the data associated with the operation, the *Component* the gesture logically occurred in, and the possible operation(s) themselves (ACTION_COPY, ACTION_MOVE, ACTION_REFERENCE).

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 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.

The *DragSourceContext* returned provides status and control to the originator of the Drag operation for its duration via the associated *DragSourceListener* interface passed in the *startDrag()* call.

The *Transferable* instance associated with the *DragSourceContext* 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*..

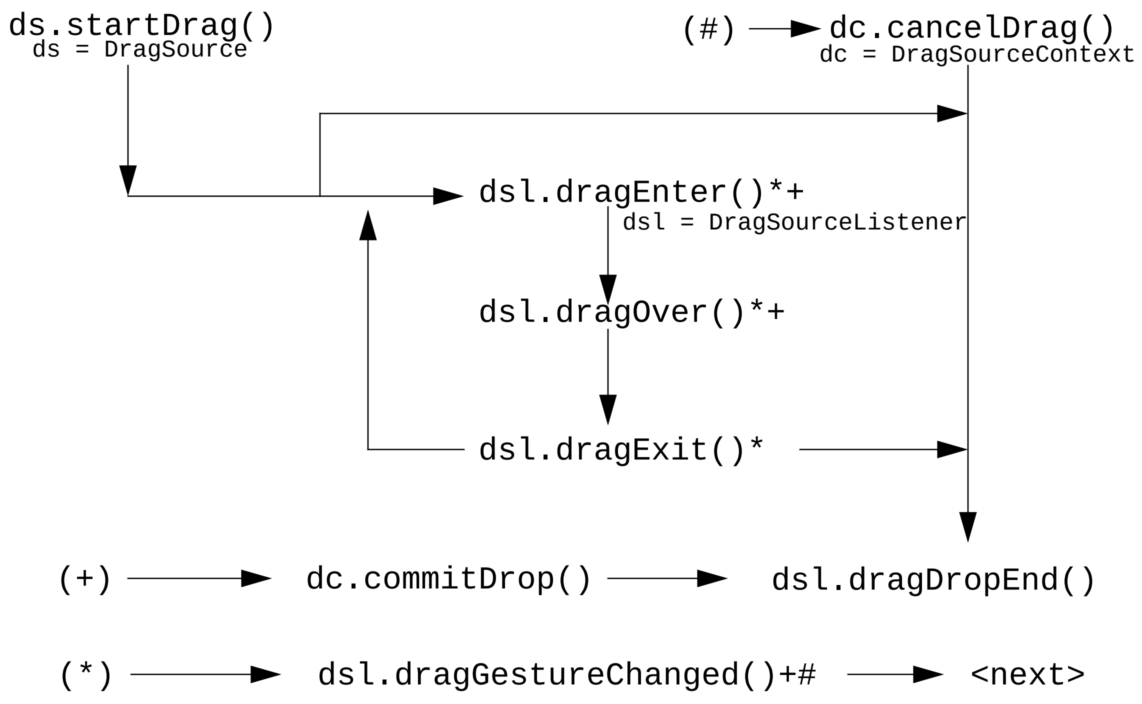
Note that multiple (collections) of either homogeneous, or heterogeneous, objects may be subject of a Drag and Drop operation, by creating a container object, that is the subject of the transfer, and implements *Transferable*.

2.2.2 The *DragSourceContext* Definition

The *DragSourceContext* interface is defined as follows:

```
public interface DragSourceContext {
     DragSource   getDragSource();

     Component    getComponent();

     AWTEvent     getTrigger();

     public Transferable getTransferable();

     void cancelDrag() throws InvalidDnDOperationException;

     void commitDrop() throws InvalidDnDOperationException;

     int  getSourceActions();
     void setSourceActions(int actions)
                throws InvalidDnDOperationException;

     void   setCursor(Cursor Cursor)
                   throws InvalidDnDOperationException;
     Cursor getCursor();
     void addDragSourceListener(DragSourceListener dsl)
          throws TooManyListenersException;

     void removeDragSourceListener(DragSourceListener dsl);
}
```

The state machine the *DragSource* implements, 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 *DragSource*, to the appropriate *DragSourceContext*, which delegates notifications, via a unicast JavaBeans compliant *EventListener* subinterface, to an arbitrary object that implements *DragSourceListener.*

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

Every *DragSource* object has 3 default *Cursor*s associated with it:

- The **Drag** *Cursor*, the cursor displayed when dragging occurs over no *DropTarget*. This cursors value may be set/get using the *setDefaultDragCursor()/getDefaultDragCursor()* methods of *DragSource*.
- The **NoDrop** *Cursor,* the cursor displayed when dragging over an invalid *DropTarget*. This cursors value may be set/get using the *setDefaultNoDropCursor()/getDefaultNoDropCursor()* methods of *DragSource*.

- The **Drop** *Cursor*, the cursor displayed when dragging over a valid *DropTarget*. This cursors value may be set/get using the *setDefaultDropCursor()/getDefaultDropCursor()* methods of *DragSource*.

Theses default values are used by the *DragSource* during the drag operation, but can be overridden by the *DragSourceListener* by calling the *setCursor*() method of the *DragSourceContext* interface obtained via the *DragEvent*.

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         (DragSourceDragEvent dsde);
     void drop              (DragSourceDragEvent dsde);
     void dragDropEnd      (DragSourceDropEvent dsde);
}
```

The *DragSourceListener's dragBegin()* method is called as a result of *startDrag()* being invoked on the associated *DragSource* object and is intended as a simple notification of drag commencement.

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 *Components* with associated *DropTargets*. [See below for details of the *DropTarget's* protocol interactions].

Note that during the Drag the set of operations the source can provide may change, however the *DataFlavors* exposed by the *Transferable* at the start of the Drag operation are constant for the duration of the operation.

On some platforms (e.g Win32) the underlying Drag and Drop protocol associates the platform Drop Target abstraction with platform Window System Windows, in other systems (e.g Motif) these abstractions can be arbitrary regions within a Window System Window. Such differences have subtle effects upon the semantics of the *DropSourceListener* methods, since *DropTargets* are associated at the *java.awt.Component* granularity which necessarily implies that typically *DropTargets* geometries will be subregions of Window System Windows.

However the implications of this platform dependent distinction on this protocol is somewhat transparent with respect to either a *DropSource* or *DropTarget* as follows.

The *DragSourceListener's dragEnter()* method is invoked when the users gesture results in the logical cursor's hotspot location initially intersecting with the screen geometry of a distinct *Component* with a *DropTarget* associated with it.

On platforms where potential *DropTargets* are exposed at the platform Window System Window granularity, the *DragSource's dragEnter()* method will be called when the cursor intersects the Window System Window that contains one or more *Components* that have *DropTargets* associated with them. Although the cursor may not in fact have actually intersected the geometry of a particular *Component* with a *DropTarget* associated with it in the destination, the expected platform "Drag Over" feedback will occur since the destination will respond with a "no drop" semantic. On platforms where potential *DropTargets* are exposed below the granularity of a platform Window System Window, the feedback will occur upon intersection with the geometry of the associated *Component*.

The *DragSourceListener's dragOver()* method is invoked as the logical cursor's hotspot moves, contained within the geometry of a *Component* with an associated *DropTarget*.

The *DragSourceListener* may, during invocations of its *dragEnter(), DragOver(), dragExit() or dragGetsureChanged()* methods, change either, the current *Cursor*, via a call to *setCursor()*, the permitted actions, via a call to *setSourceActions*(), or cancel the drag operation, via a call to *cancelDrag()*.

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.

Should the change in gesture be interpreted as a drop, the method signals this back through the *DragSource* via a call to the *commitDrop()* method. The *DragSource* will notify the *DropTarget* of the Drop and transfer the *Transferable*[] data at this time .

Subsequently the *dragDropEnd()* method is invoked to signify that the operation is complete. The *isDragAborted()* and *isDropSuccessful()* methods of the *DragSourceDropEvent* can be used to determine the termination state. Once this method is complete the *DragSourceContext* and the associated resources are invalid.

2.2.4 The *DragSourceDragEvent* Definition

The *DragSourceDragEvent* class is defined as follows:

```
public class java.awt.dnd.DragSourceDragEvent
       extends java.util.EventObject[1] {
     public DragSourceContext getDragSourceContext();

     public int getTargetActions();

     public int getGestureModifiers();
}
```

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

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

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

2.2.5 The *DragSourceDropEvent* Definition

The *DragSourceDropEvent* class is defined as follows:

```
public public class java.awt.dnd.DragSourceDropEvent
       extends java.util.EventObject {
     boolean isDragCancelled();
     boolean isDropSuccessful();
}
```

An instance of the above class is passed to a *DragSourceListener's dragDropEnd()* method.

If the Drag operation was aborted for any reason, by the initiator invoking the *Drag-SourceContext's cancelDrag()* method for instance, the *isDropAborted()* method will return `true`, else `false.`

If the Drop occurs, then the parcticipating *DropTarget* will signal the success or failure of the data transfer via the *DropTargetContext's dropComplete()* method this status is made avialable to the initiator via the *isDropSuccessful()* method.

2.3 Drop Target

2.3.1 java.awt.Component Additions

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)
                           throws IllegalArgumentException;

     public synchronized
               DropTarget getDropTarget(DropTarget df);
     //
}
```

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

To associate a *DropTarget* with a *Component* onr may invoke either; *DropTarget.setCom pononent*(`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*() mehtod throws if the *DropTarget* actual parameter is not suitable for use with this class/instance of *Component*.

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 receipient or destination of the operation.

A single *DropTarget* instance may 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 receiptive to Drag and Drop operations when the coordinates of the logical cursor intersects that geometry.

The *DropTarget* class is defined as follows:

```
public class java.awt.dnd.DropTarget {

     public DropTarget();

     public DropTarget(Component c);

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

     public DropTargetContext getDropTargetContext();

     public boolean supportsAutoScrolling();

     public void   setAutoScrollInsets(Insets scrollInsets);
     public Insets getAutoScrollInsets();

     public void setDefaultTargetActions(int actions);
     public int  getDefaultTargetActions();

     public void
         addDropTargetListener(DropTargetListener dte)
            throws TooManyListenersException;

     public void
         removeDropTargetListener(DropTargetListener dte);

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

}
```

If the *Component* associated with a particular *DropTarget* supports scrolling, the *DropTarget* can expose this facility to the Drag and Drop operation, thus allowing scrollable *Components* to “autoscroll” their contents under the logical cursor while it is quiescent within the region defined by the intersection of the *Insets* set via the *setAutoScrollInsets()* method and the bounding box of the *Component*.

The *setComponent*() method throws if the *Component* actual parameter is not appropriate for use with this class/instance of *DropTarget*.

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 void setTargetActions(int actions)
               throws InvalidDnDOperationException;

     public int getTargetActions();

     public DataFlavor[] getDataFlavors();

     public void getTransferable()
               throws InvalidDnDOperationException;
     public void dropComplete(boolean success)
                    throws InvalidDnDOperationException;

     //

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

The *DropTargetContext* provides the *DropTargetListener* with the ability to determine the location of the logical cursor within the geometry of the associated *Component*, via the *getCursorLocation()* method, manipulate the operations that it is capable of supporting (ACTION_COPY, ACTION_MOVE, or ACTION_REFERENCE), via the *setTargetActions()* and *getTargetActions()* methods, and also to signal the end (successful or otherwise) of any data transfer that may result from a Drop operation on the associated *DropTarget*, via the *dropComplete()* method.

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 dragScroll           (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 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.

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.

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 *dragScroll()* method of the *DropTargetListener* is invoked when the hotspot of the logical "Drag" Cursor has been quiescent (stationary) for a short (TBD) period of time, and both intersects a visible portion of the *DropTarget's* associated *Component's* geometry and the Scroll *Insets*. The *DropTargetListener*, upon receipt of this notification, shall scroll, if it supports scrolling, the contents of the associated *Component* based upon the location of the logical "Drag" cursor.

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();

     Point  getCursorLocation();

     public int getSourceActions();
}
```

The get*CursorLocation()* method return the 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.

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

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

```
     public DataFlavor[] getDataFlavors();
}
```

A *DropTargetDropEvent* is passed to the *DropTargetListener's dragEnter(), dragOver(), dragExit()* and *dragScroll()* methods.

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 {

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

     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 *DropTargetDropEvent* 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*() prior to any invocation of *getTransferData*() on the Transferable associated with the Drop.

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

2.4 Data Transfer Phase

If the initiator of the Drag operations commits the Drop by invoking the *DragSourceContext's commitDrop()* method the DropTarget is notified via an invocation of the *DropTargetListener's drop()* method. If the initiator cancels the operation by invoking the *DragSourceContext's cancelDrop()* method the *DropTarget* is notified via an invocation of the *DropTargetListener's dragExit()* method.

In the case where a drop occurs, the *DropTargetListener's drop()* method is responsible for initiating 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.

The *java.awt.datatransfer.Transferable* API provides the mechanisms to perform that data transfer itself.

Once the *DropTarget* has processed the transfer(s) of data provided, or if it is unable to complete the transfer for any reason, it signals its completion, along with an succcess or failure of the transfer itself, by invoking the *DropTargetContext's dropCompleted()* method. At this point the *Transferable* and *DragSourceContext* instances are no longer valid and all references to them should be discarded to allow them to be subsequently garbage collected.

2.4.1 Mapping platform dependent data types to MIME types

All the target DnD platforms represent their transfer data types using a similar mechanism, however the representations do differ. A mechanism is required in order to create an extensible (platform dependent) mapping between these type names, their representations, and MIME based *DataFlavors*.

The implementation will provide a mechanism to externally specify a mapping between platform native data types (strings) and MIME types (strings). 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 mechanisms.

3.0 Open 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 exlcusion of normal event processing. This will be elaborated upon in future revisions of the specification.

Initially, 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, it is likely that “callbacks” into *DropTarget*, *DropTargetContext*, and *DropTargetListener* will occur on the AWT system event dispatch thread. This behavior is highly undesirable but is an implementation, not architectural, feature, and will be addressed in a future release.

3.0.2 Security?

TBC

3.0.3 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.

To be Specified in a later draft.

3.0.4 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.

3.0.5 Impact of InfoBus?

Since the Infobus specifiation defeines that *DataItem*s implement *Transferable* they are potentially capable of being dragged and dropped between InfoBus members.

Further investigation is underway to determine if any additional

3.0.6 Implications of “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.

3.0.7 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.