OPENDPI: A TOOLKIT FOR DEVELOPING DOCUMENT-CENTERED ENVIRONMENTS

Olivier Beaudoux
*ESEO, Computer Science Department & Laboratoire de Recherche en Informatique / INRIA Futurs**
4 rue Merlet de la Boulaye, 49009 Angers, France

Michel Beaudouin-Lafon
Laboratoire de Recherche en Informatique / INRIA Futurs
Université Paris-Sud, LRI - Bt 490, 91405 Orsay - France

Keywords: Document centered systems, interaction model, active components, GUI toolkits.

Abstract: Documents are ubiquitous in modern desktop environments, yet these environments are based on the notion of application rather than document. As a result, editing a document often requires juggling with several applications to edit its different parts. This paper presents OpenDPI, an experimental user-interface toolkit designed to create document-centered environments, therefore getting rid of the concept of application. OpenDPI relies on the DPI (Document, Presentation, Instrument) model: documents are visualized through one or more presentations, and manipulated with interaction instruments. The implementation is based on a component model that cleanly separates documents from their presentations and from the instruments that edit them. OpenDPI supports advanced visualization and interaction techniques such as magic lenses and bimanual interaction. Document sharing is also supported with single display groupware as well as remote shared editing. The paper describes the component model and illustrates the use of the toolkit through concrete examples, including multiple views and concurrent interaction.

1 INTRODUCTION

1.1 Documents versus Applications

The fact that the document is the main object of interest within interactive workspaces had been identified 25 years ago : “The document is the heart of the world, and unifies it” (Johnson et al., 1989). However, today’s environments are mainly based on the application concept where every application is dedicated to a specific kind of data. Users are thus forced to juggle between applications in order to edit a single document. Software publishers react to this fact by proposing three different and complementary strategies:

1. Building small applications within larger ones – For example, both Word and PowerPoint (which are provided within a single software package) include two different small-applications dedicated to vectorial drawings.
2. Open architecture based on plug-ins – For example, many plug-ins are available for the PhotoShop application, thus extending its initial functionalities by following users’ needs.
3. Software suite with common interfaces – For example, Photoshop, Illustrator, GoLive and InDesign applications have a common look and feel to their graphic interfaces and some common tools, thus resulting in a more natural interaction since swapping from one application to another is less visible.

These three approaches aim at positioning the document in the heart of the interaction rather than the application. However, they don’t reach their goals since users still have to juggle between more than one application and document in order to achieve a single task. They try to make applications less visible by reducing their gap, but they remain centered on the application concept.

1.2 Interactions and Documents

The document concept often suggests the data that documents can contain. However, it could be relevant to specify the semantic of actions they can handle in addition to the semantic of their data, such as Web services which aim at specifying “interfaces” of offered services (W3C, 2001). We have built our model in this direction : it defines how documents can be perceived by users and, above all, how users can (in-

Beaudoux O. and Beaudouin-Lafon M. (2005).
OPENDPI: A TOOLKIT FOR DEVELOPING DOCUMENT-CENTERED ENVIRONMENTS.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 39-47
DOI: 10.5220/0002511900390047

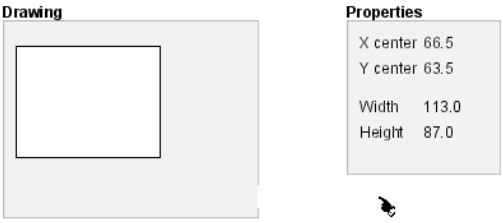


Figure 1: Two presentations of a shape (opendpi.scenarios.Presentations)

ter)act on documents. We provide a model that combines a document model and an interaction model in a unifying way. In so doing, we offer a generic mechanism which makes data independent from actions that can be done on this data, and actions independent from interactions that can induce these actions.

1.3 Structure of this Paper

The second section describes the core component model of DPI. It explains how this component model is used to define the three main components D, P, and I, and how such components communicate with each other. In the third section, we focus on the benefits of using a single and generic model for all interactive components by illustrating the resulting simplicity of their implementation. In the fourth section, we present the replication point concept as a general concept that we use to provide basic groupware capabilities and a full alternate rendering engine. We compare the DPI model with other complementary approaches and works in the fifth section. Finally, the implementation of the model in the OpenDPI toolkit and the perspective of our work is discussed in the conclusion.

2 THE CORE COMPONENT MODEL

In the DPI (Document, Presentation, Instrument) conceptual model, documents are visualized through one or more presentations, and manipulated with interaction instruments (Beaudoux and Beaudouin-Lafon, 2001). The conceptual model defines the structures of these three components and the way they communicate. In this paper, we focus on the generic component model that was used to build the OpenDPI toolkit that implements the DPI model. By instantiating this component model, we allow the implementation of the three D, P and I components.

In order to explain the core component model and its instantiation for the D, P and I components, we use a simple example that tackles the main aspects of our model (figure 1)[1]. It consists of two presentations of a single shape (such as a rectangle): the first presentation displays the shape as a graphical object, and the second presentation displays its properties within textual fields.

2.1 Observable State

The state of a component is defined by both its proper state and its structural state. The *proper state* characterizes the property values of the component. For example, the shape defines the *x, y, width* and *height* properties. Moreover, components can be themselves composed in a hierarchical way, and they thus define a *structural state*.

Depending on the context, the changes of component states can interest other components. DPI components are thus considered as *observable* instances in the sense of the “observable / observer” design pattern (Gamma et al., 1994). Each component class defines a property *observer interface* as the means of observing its proper state. For example, the shape of our example defines the following interface:

```
interface ShapeObserver
    extends PropertyObserver {[2]
    void newWidth(double width);
    void newHeight(double height);
    void newX(double x);
    void newY(double y);
     // etc.
}
```

In a complementary way, the structure observer interface, common to all component classes, is the means of observing the structural state of components. In order to simplify representation of DPI components, we extend the UML notation in a way that clearly displays observable and observer interfaces, and their relation (see figures 2 and 3): a black point depicts the observable interface of components, and a white point (respectively a white point combined with the aggregation symbol) depicts a property observer interface (respectively the structure observer interface) implemented by the observers.

2.2 Applying the Model to Documents

Documents are structured as trees where each node is a DPI component. The domain data part of a doc-

[1] All given examples have been implemented in classes mentioned in corresponding figures. They can be tested by downloading OpenDPI at http://www.eseo.fr/~obeaudoux/opendpi

[2] In all this paper, we omit Java keywords public and protected for compactness.

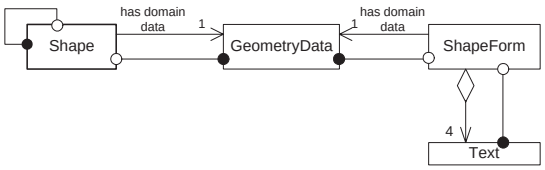


Figure 2: Components involved in the two presentations of the shape

ument contains components that are observed by associated presentations. In turn, embedded presentations also contain node components which are typically graphic in nature.

The synchronism between presentations and domain data within a single document is based on the observation mechanism. Figure 2 shows how the rectangle example is built:

- Domain data of the shape is defined in the *GeometryData* component class through the four properties *xCenter*, *yCenter*, *width* and *height*.
- The "Drawing" presentation contains one component instance of the *Shape* class. The shape component observes every change in the domain data through the *GeometryDataObserver* interface in order to update its own state consequently. Moreover, it observers its proper state that can vary among user's interactions through the above defined *ShapeObserver* interface in order to update the domain data.
- The "Properties" presentation defines in the *ShapeForm* component that allows the edition of the domain data through four *Text* components. The form observes the domain data in order to refresh text fields consequently, and the proper state of its text fields in order to update the domain data.

2.3 Action Producer and Consumer

2.3.1 Specifying Actions

The observation concerns the changes of component states and thus does not allow communication between components out of such changes. An action allows the transmission of a state independent of any component: it is handled by the system in order to link actions, from the users to the documents.

The state of an action is defined through a set of properties. For example, the translate action simply defines the *dX* and *dY* properties that describe a movement on the x and y axis:

```
class Translate extends Action {
    double dX;
    double dY;
    double getDX() {return dX;}
    void setDX(double dX) { this.dX = dX; }
    // same code for dY property...
}
```

An action class only defines such a state and does not characterize any behavior related to the action. This a consequence of the polymorphic nature of actions (Beaudouin-Lafon and Mackay, 2000): a polymorphic action has an imprecise semantic defined by the action itself, and precise semantics defined by objects that can consume the action. As a consequence, an action class does not define in what manner the action is to be executed or cancelled: this manner is defined by consumers of actions.

2.3.2 Producer and Consumer Interfaces

The production of an action from a producer component to a consumer component follows a cycle defined through the definition of both producer and consumer interfaces.

The *consumer interface* of an action class *A* is implemented by all classes of components that define their ability to consume instances of *A*. It consists of a set of four methods invoked by producers in the following order[3]:

1. The *can*-method carries out the feasibility test of the action related to the current context.
2. Whenever the can-method has returned true, the *begin*-method is invoked and starts the action.
3. The *do*-method is then invoked and represents the main loop of the action consumption.
4. The *end*-method is finally invoked when the action have to stop.

For example, the translate action defines the following consumer interface:

```
interface TranslateConsumer{
  extends Consumer {
    boolean canTranslate(Producer p);
    void beginTranslate(Producer prodpucer,);
    void doTranslate(double dx, double dy);
    void endTranslate(Producer p);
}
```

The *producer interface* of an action class A is implemented by all classes of components that define their ability to produce instances of A. It specifies a *minimal contract* that producers must satisfy in order to be able to produce the action. For example, the *pick-color* action defines a producer interface so that the consumer provides its picked-color to the producer:

```
interface PickColorProducer extends Producer {
        void colorPicked(Color c);
}
```

[3]The consumer interface also defined *undo*, *redo* and *echo* methods that we do not describe in this paper.

Note that, however, most actions does not need to define an associated producer interface since their minimal contract is empty. For example, the *translate* action does not defined a dedicated producer interface since the translation does not need any specific contract in order to be consumed.

In order to clarify the UML representation of DPI components, we extend our component notation in order to display the producer and consumer interfaces (see figure 3): a black square depicts the producer interface that qualifies the component which can produce the action, and a white square depicts the consumer interface that qualifies the component which can consume the action.

2.3.3 Concurrency

When multiple actions are produced on a component at the same time, the state of the component may be modified concurrently. In order to ensure data integrity, we have introduced the *marking* of components.

The rule of marking is defined as follows: an action that modifies a state of the consumer component can be produced *only if* no mark has been set on its state. As soon as consumers perform such a marking while they consume actions, the previous rule forbids the production of concurrent actions. This is carried out by *can*-methods that return false whenever concerned properties have been already marked. In the following example, consuming a translation induces the marking of both the *x* and *y* properties of the shape thus forbidding any concurrent action (such as another translation).

Moreover, the marking mechanism, shortly explained in this paper, is quite similar to fine grain *locking* techniques such as in DistEdit application (Knister and Prakash, 1990). However, they have significant differences: locking aims at *ensuring* consistency of *data* while marking aims at *avoiding* concurrent *actions*. As a consequence, a mark does not need to explicitly and strictly forbid subsequent modification of the marked element (*i.e.* a set method can be invoked on a marked property): it only forbits concurrent actions. In addition, a mark does not need to be associated to a particular owner. These two points make marking quite easy for programmers.

2.4 Applying the Model to Instruments

Action chaining from users to documents is based on the instrumental interaction (Beaudouin-Lafon, 2000). The physical part of instruments detects gestural actions (or gestures) made by users on human input devices, then the logical part transforms the gestures in intentional actions (or actions) that are produced by instruments and consumed by documents.

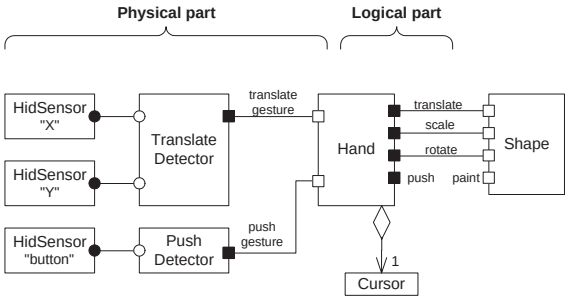


Figure 3: The “hand” instrument

Figure 3 shows how the producer - consumer model is used in combination with the observable - observer model in order to build such a chaining. It illustrates the production of the translate action by the “hand” instrument and its consumption by the shape.

In order to be aware of the easiness of our approach, we explain the figure by analyzing the pieces of code for both the *Hand* and *Shape* components. The *Hand* component is defined as follows:

```
class Hand  extends Tool implements
    TranslateProducer,
    TranslateGestureConsumer,
    PushGestureConsumer
{
    Translate translate;
    Consumer shape;

    void beginPushGesture(Producer p) {
      shape = getPickedConsumer(translate);
       if (shape != null)
         translate.beginAction(shape);
    }
    void doTranslateGesture(
      double dx, double dy) {
      super.doTranslateGesture(dx, dy);
      if (shape != null) {
         translate.setDX(dx);
         translate.setDY(dy);
         translate.doAction();
      }
    }
    void endPushGesture(Producer p) {
     if (shape != null) {
         translate.endAction();
         shape = null;
      }
    }
}
```

The hand instrument is linked to the mouse and observes the proper states of the mouse *x*, *y* and *button* sensors during its construction (not shown in the code). This observation is delegated to two gesture

detector components: the translation detector produces a *translate gesture* whenever the *x* or *y* sensor state changes, and the push gesture detector produces a *push gesture* whenever the button sensor state changes. When the push gesture starts (*beginPushGesture* method), the hand does a *picking* that consists of finding which component located under the cursor *can* consume the translate action. In the example, the picking returns the shape. When the hand consumes the translate gesture (*doTranslateGesture* method), it updates the location of its cursor and invokes the execution of the action on the picked shape. Finally, when the push gesture stops (*endPushGesture* method), the hand terminates the action.

In turn, the *Shape* class involved in the consumption phase is defined as follows:

```
class Shape extends Component
    implements TranslateConsumer
{
    boolean canTranslate(Producer p)  {
      return !isMarked("x") && !isMarked("y");
    }
    void beginTranslate(Producer p) {
      mark("x"); mark("y");
    }
    void doTranslate(double dx, double dy) {
      setLocation(getX() + dx, getY() + dy);
    }
    void endTranslate(Producer p) {
       unmark("x"); unmark("y");
    }
}
```

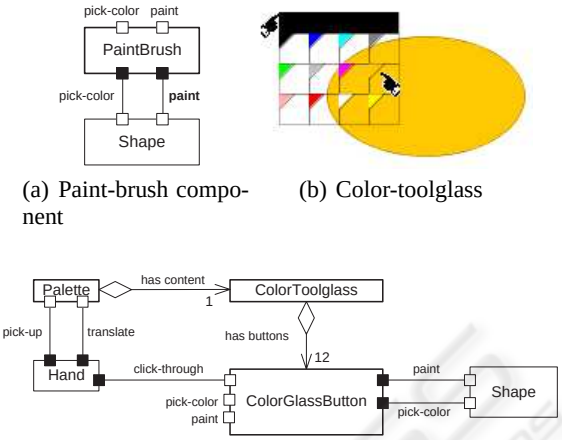
When the shape consumes a translation, it first mark the modified properties: such a marking is checked for by the producer before producing the translate action. Then subsequent calls to the do-method are done by the producer and the shape modifies its *x* and *y* properties accordingly. Finally, when the translate action stops, the shape unmarks the previously marked properties.

3 BENEFITS OF A GENERIC MODEL

3.1 Direct Manipulation

The DPI component model allows the creation of both direct and non-direct manipulation components. This is a significant difference between standard GUI toolkits and OpenDPI: traditional GUI toolkits are based on the widget model which does not allow the definition of direct manipulation components such as paint-brush, or magnetic guide (opendpi.scenarios.Magnetism).

Figure 4 displays two painting tools: a paint-brush (a) and and color-toolglass (b, c) (Bier et al., 1993).



(a) Paint-brush component (b) Color-toolglass

(c) Color-toolglass component

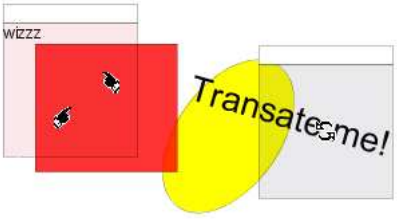
Figure 4: Painting tools (opendpi.scenarios.Toolglass)

Both these tools can produce the pick-color and paint actions (on any shape component for example). Their implementations follow strictly the same guidelines as the ones previously explained for the translate action example. The environment of the color-toolglass is a little more complex but the code remains quite simple:

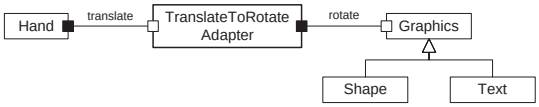
1. The palette that contains the toolglass can be translated around the workspace. It can also be picked-up by an instrument such as the hand instrument in order to be used in a bimanual way (in the opposite case, the toolglass behaves like a usual palette of colors).

2. The toolglass contains 12 colored buttons that can consume the click-through action. When the click-through action is consumed, the button produces in turn the paint action on the picked graphical component located above the hand cursor (and thus above the button).

As we can observe, each of the components involved in this painting process are interactive components. This allows the chaining of actions in many different ways. For example, since the color glass-button and the brush are graphics components, they can be "painted": by consuming the paint action, they change their associated color. In the same way, their color can be picked. Such combinations of painting and color-picking can be done in many manners by using any color toolglass or paint-brush. We therefore claim that using a common model for *all* interactive objects will help to discover such mixing and enriching interaction capabilities.



(a) Two adapters (opendpi.scenarios.Adapters)



(b) Translate-to-rotate adapter component

Figure 5: Adapters (opendpi.scenarios.Adapters)

3.2 Genericity of Actions

Tool instruments naturally inherit from the genericity of actions: they can operate in many contexts. We have illustrated this point through the translate action example: the translation can be produced in a common way on many components and is thus generic. Moreover, it may be interesting to override its default behavior defined in the *Shape* class. For example, when a UML component is translated, it may be useful to translate a clone of the component rather than the original component so that the time-consuming computation of its linking is done when the interaction ends. We have also experiment the genericity of the paint action regarding image painting: when an *Image* component consumes the paint action, it applies a filtering effect so that the painting color is reinforced.

3.3 Adapting Actions

In some circumstances, a user may want to apply an action to a component but does not have any instrument that could produce such an action. Rather than purchase a new instrument, the user may prefer to add an *adapter* to an existing instrument. Figure 5 illustrates how a translation →rotation adapter works:

1. The adapter is added to an instrument that can produce the translate action, e.g. the hand instrument.
2. When the hand instrument produces the translate action, the adapter transforms the consumed translation into a produced rotation. The transformation is based on a simple mathematical operation that sets the *angle* property value of the rotate action proportionally to the *dX* property of the translate action.

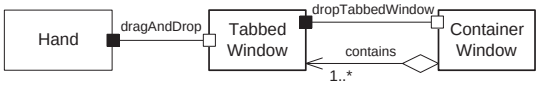


Figure 6: Drag & drop of tabbed windows (opendpi.scenarios.DnDPage)

The following code well illustrates the easiness of its implementation:

```
class TranslateToRotateAdapter
    extends Component implements
    TranslateConsumer, RotateProducer
{
    Rotate rotate;
    Consumer shape;

    boolean canTranslate(Producer p) {
      shape =
        getPickedConsumer(rotate, getX(), getY());
      if (shape != null) return shape.canRotate(p)
      else return false;
    }
    void beginTranslate(Producer p) {
      rotate.beginAction(shape);
    }
    void doTranslate(double dx, double dy) {
       rotate.setAngle(dx);
      rotate.doAction();
    }
    void endTranslate(Producer p) {
      rotate.endAction();
    }
}
```

After checking for the rotation feasibility, the adapter just replicates the production cycle from the translate action to the rotate action. The implementation of adapters remains so simple that it may be automated so that users can specify their own adapters (for example by setting the mathematical operation such as the one underlined in the previous code).

Moreover, adapters can be used in order to relax the marking of components. Figure 5-a shows the *SlidingTranslateAdapter* (labelled “wizzz”) that transform any concurrent translate actions into sliding-translate actions by filtering the feasibility method on the consumer.

3.4 Interoperability through Actions

The DPI component model allows the *interoperability of actions* among applications. For example, the pick-color action could be done on an application by an instrument provided by another application. As a result, a color can be picked from any DPI application and used to paint an object in any other DPI application.

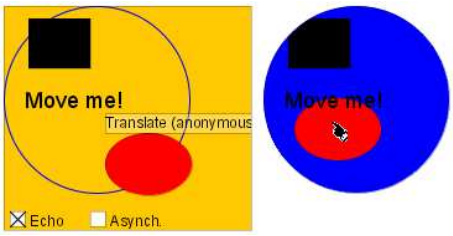


Figure 7: A local replication point (opendpi.scenarios.Sharing)

Another well-known interaction that allows the interoperability among applications is the drag'n drop. However, such an interaction is not the easier one that GUI toolkits implement. Figure 6 shows that, from the DPI perspective, this interaction is not more complex than others. It illustrates how a tabbed window may be dragged from a container window then dropped into another container window:

1. The instrument produces the *drag-and-drop* action on the selected tabbed window. This action consists in taking the window, translating it on the workspace, and finally dropping it on a targeted container window.

2. When the drag & drop ends, the tabbed window produces in turn the *dropTabbedWindow* action to the targeted container window. Note that, throughout the dragging stage, the picking is used in conjunction with the feasibility method in order to check if the drop action can be produced.

4 REPLICATION POINT

A *replication point* is an abstract object that, when attached to two or more (mostly graphical) containers $C_{1..n}$, cross-replicates the initial contents of $C_{1..n}$ and subsequently dispatches all user actions within a container C_i to all the other containers $C_{j\neq i}$. Figure 7 displays a replication point associated to two containers, a clipping rectangle (on the left) and a clipping circle (on the right), that share their content (a rectangle, an ellipse ant a text). While the user translates the ellipse within the right container, the replication point replicates the translation to the left container. Since the left container has enabled the echoing mode (see next section), the translate action is played through an echo that consists in tagging the ellipse with the name of the action when the action begins, then playing a translation animation when the actions ends.

4.1 Application to Groupware

The remote replication point is a replication point which is identified by its unique IP group address. It defines the way of synchronizing shared components among sharing containers. This synchronization induces a strong spatial coupling of sharing containers since they have exactly the same contents, and a strong temporal coupling since these contents remain identical at any time.

In order to extend this synchronization behavior, we introduce the concept of a *behavior point* that can be attached to a component contained in or equal to a container attached to a replication point. Such behavior points define extended behaviors by relaxing the temporal and/or spatial coupling, thus resulting in a flexible coupling as defined in (Dewan and Choudhary, 1992). For example, we have defined an *asynchronous point* that relaxes the temporal coupling by allowing users to work asynchronously on the component from which the asynchronous point is attached. In the same way, we provide awareness behavior points that allow the insertion of specific information in sharing containers though this information is not necessarily the same for each sharing container. Such behavior points thus relax the spatial coupling. For example, the *echoing point* allows the perception of a remote action produced inside the component from which it is attached to through an echo of the action (Beaudouin-Lafon and Karsenty, 1992) (see figure 7), rather than through the original execution the action. Such an echo often consists of an animation that "summarizes" the result of the action, thus avoiding the need to display all the disturbing details of remote actions.

It is important to note that the implementation of remote sharing point does not provide any concurrency control yet. At this time, the remote replication point works locally between two or more separated application instances (opendpi.scenarios.RemoteSharing).

4.2 Application to Alternate Rendering

Figure 8 illustrates the use of replication points for magnifying glasses, magic lenses (Bier et al., 1993), and radar views. The radar view implementation is trivial: a local replication point is both attached to the radar rectangle and to the "window" layer of the scene. The magnifying glass also uses a local replication both attached to the glass content and to the main layer of the scene. The magic lens uses the same technique but the replicated main layer is attached to an outline renderer. The use of replication points for both the magnifying glass and the magic lens may be found unusual. However, it is motivated by the in-

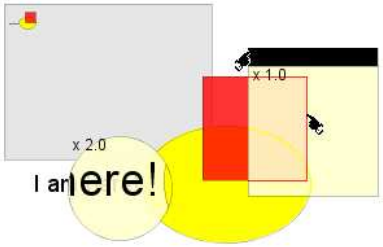


Figure 8: Radar view and (magic) lenses (opendpi.scenarios.Rendering)

teraction consistency that OpenDPI guarantees: when users *interact above* lenses, the interaction remains *consistent*. In figure 8, the user moves the outlined rectangle in a consistent way: he must point the rectangle's outline in order to translate it, which allows the translation of the ellipse when pointing its masked portion.

5 RELATED WORK

5.1 Document Centered Systems

OOE system extends the NextStep operating system by allowing the edition of composite documents (Backlund, 1997). It is based on the display PostScript capabilities of the NextStep: OOE applications share a common display language so that areas of documents edited through an application can be displayed without alteration by any another application. However, OOE only simplifies the way of users swap between applications. OLE framework allows the edition of composite documents by defining a communication protocol between applications (Brockschmidt, 1995). Consequently, it promotes the interoperability between applications. However, it is based on the application concept, forces to use a complex mechanism of interoperability, and does not allow action interoperability such as needed in the pick-color example. OpenDoc framework (Apple, 1994) is the closer approach to our model. It defines documents as a set of hierarchically structured and typed *parts*. Each part is associated with its own content model and interaction model, can be viewed through a dedicated *part viewer*, and can be edited within a dedicated *part editor*. However, the granularity of part editors and viewers are still high, there is no interoperability between editors, and the interface changes from an editor to another.

5.2 User Interface Models

The main common point between the MVC model (Krasner and Pope, 1988) and our DPI model is the separation and the synchronization of domain data from their presentation. Despite this fact, domain data, presentation and instruments of DPI radically differ respectively from the model, view and controller of MVC. Firstly, the domain data is defined through a detailed model organized as a tree structure. The model of MVC rather focuses on defining primitives (such as our property does) that can be viewed and edited through widgets (view + controller pairs). Secondly, the presentation is also defined through a detailed model based on scene-graphs while the view of MVC does not provide any display model. Finally, the instrument should definitively not be compared with the controller of MVC. The instrument concept defines an overall interaction model while MVC does not address any since it only deals with widgets.

5.3 Document Model

DOM specification (W3C, 2004) defines the document as a tree where each node (called element) has a state defined by a set of attributes and/or child nodes. This definition is thus compatible with the proper and structural states of DPI components: each DPI component is a node and the DPI properties might be considered as DOM attributes. However, the DOM model does not address any aspects of action that can be performed on documents and their elements and, as a consequence, does not define any interaction model. Moreover, the concurrent modification of a document is not taken into account.

5.4 Component Model

Using software components within workspaces is an idea that appeared less than ten years ago. For example, we can cite the COM architecture (Microsoft, 1995) and the JavaBeans component platform (Sun, 1997). The implementation of the DPI model has some similarities with the JavaBeans approach (e.g. the definition of properties and the intensive use of introspection). However, the JavaBeans model, like the COM architecture, is truly generic and aims at *building* applications by assembling components. In our approach, we focus on components that can *substitute* applications: our goals and motivation thus radically differ. Moreover, the DPI model defines a generic component model that aims at being instantiated in a document and an interaction model, while component software architectures do no address such a problem.

6 CONCLUSION AND PERSPECTIVE

We have proposed in this article a component model based on documents and instruments. It aims to substitute the application concept to the software component concept at the workspace level in order to overcome the problem induced by the intensive use of widgets in today's workspaces. The proposed model goes into the opposite direction from widgets: DPI components are open-boxes that respect an unifying contract, while widgets are black-boxes that aim at masking the internal complexity.

The overall DPI model has been implemented in the OpenDPI java toolkit[4]. It uses Piccolo for displaying graphics (Bederson et al., 2000). It provides its own high level management of multiple human input devices under Linux and a standard one under other operating systems. The toolkit is made of about 250 classes and 18000 lines of code. We have implemented a set of interactive components through scenario, such as the ones presented in this paper, that are often considered too complex to develop. We have noted that, by experimenting student projects, the design and programming of new DPI components is quite easy as soon as the DPI concepts are captured (which was mainly done by analyzing sample codes).

Students have underlined the elegance of the approach and its unifying purpose. DPI components can be implemented without many programming efforts in varied contexts such as bimanual interaction or single display groupware, with the ability to guarantee the interaction consistency. By defining the replication point concept, the OpenDPI toolkit provides an alternate rendering engine that can be used to build interaction-consistent magic lenses, and gives the foundation of the groupware facet of the model.

The next step of our work consists of validating the DPI model by implementing a software suite dedicated to specific tasks. Such a tool will allow our approach to be more precisely qualified, based on the intensive use of interactive components without any application (except the kernel of OpenDPI), and to discover potential new problems that this approach may induce. We are also about to refine the DPI model by using well known standards. The document model will be based on DOM and the presentation model will be based on SVG (W3C, 2003). We will thus extend DOM by specifying how the (inter)action and collaboration aspects of DPI can be added on top of it.

[4]http://www.eseo.fr/~obeaudoux/opendpi

REFERENCES

Apple (1994). Opendoc technical summary. Technical documentation, Apple Computer Inc.

Backlund, B. E. (1997). OOE: A compound document framework. *ACM SIGCHI Bulletin*, 29(1):68–75.

Beaudouin-Lafon, M. (2000). Instrumental interaction: An interaction model for designing post-wimp interfaces. In *Proc. CHI'00*, pages 446–453. ACM Press.

Beaudouin-Lafon, M. and Karsenty, A. (1992). Transparency and awareness in a real-time groupware system. In *Proc. UIST'92*, pages 171–180. ACM Press.

Beaudouin-Lafon, M. and Mackay, W. (2000). Reification, polymorphism and reuse: Three principles for designing visual interfaces. In *Proc. AVI'00*, pages 102–109. ACM Press.

Beaudoux, O. and Beaudouin-Lafon, M. (2001). DPI: A conceptual model based on documents and interaction instruments. In *Proc. IHM-HCI'01*, pages 247–263. Springer Verlag.

Bederson, B. B., Meyer, J., and Good, L. (2000). Jazz: An extensible zoomable user interface graphics toolkit in java. In *Proc. UIST'00*, pages 171–180. ACM Press.

Bier, E. A., Stone, M. C., Pier, K., Buxton, W., and DeRose, T. D. (1993). Toolglass and magic lenses: the see-through interface. In *Proc. of SIGGRAPH'93*, pages 73–80. ACM Press.

Brockschmidt, K. (1995). *Inside OLE, Second Edition*. Microsoft Press.

Dewan, P. and Choudhary, R. (1992). A high-level and flexible framework for implementing multiuser user interfaces. *ACM Trans. on Information Systems*, 10(4):345–380.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Johnson, J., Roberts, T. L., Verplank, W., Smith, D. C., Irby, C., Beard, M., and Mackey, K. (1989). The Xerox Star: A retrospective. *IEEE Computer*, 22(9):11–29.

Knister, M. J. and Prakash, A. (1990). DistEdit: a distributed toolkit for supporting multiple group editors. In *Proc. CSCW'90*, pages 343–355. ACM Press.

Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80. *Journal of Objet Oriented Programming*, pages 26–49.

Microsoft (1995). The component object model specification. Specification Document.

Sun (1997). JavaBeans API specification. Specification document.

W3C (2001). Web services description language (WSDL) 1.1. Technical report, Consortium W3C.

W3C (2003). Scalable vector graphics (SVG) 1.1 specification. Technical report, Consortium W3C.

W3C (2004). Document object model (DOM) level 3 core specification. Technical report, Consortium W3C.