Document-View-Presentation Pattern

Ku-Yaw Chang, Lih-Shyang Chen and Chi-Kong Lai

No. 1, University Road,
Department of Electrical Engineering,
National Cheng-Kung University, Tainan 701, Taiwan, R.O.C.
E-mail: canseco@mirac.ee.ncku.edu.tw

Abstract

When designing the architecture of an interactive system, the challenge is to keep the data model and the functional core that manipulates the data independent of the user interface. However, we usually categorize the rendering function, which translates the data into the form suitable for visual purposes, as the output part of the user interface. In other words, the rendering process of the data and the way to output the rendering result are tightly coupled. This limitation results in the need to render the data once for each user interface whenever the data changes, even in the case that multiple user interfaces share the same rendering result.

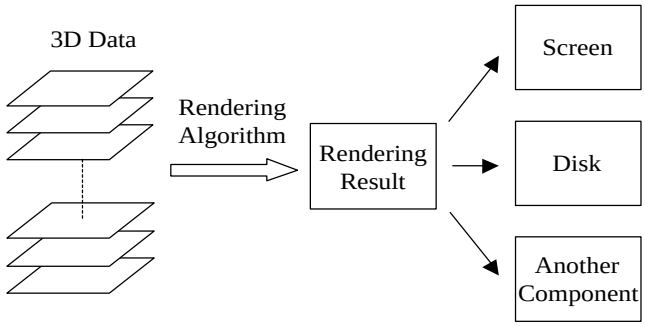
In this paper, we describe the Document-View-Presentation (DVP) pattern for interactive software systems. This pattern builds on the Document-View pattern and strongly decouples the rendering function from the output of the rendering result. This separation makes it possible to render the data once and output the result in different methods repeatedly. This architecture is especially suitable to those interactive systems with computationally expensive rendering algorithms, such as graphics or image processing systems. Moreover, the DVP pattern also introduces the 3-tier concept into the design of an interactive system to have “thin” clients, i.e. user interfaces. Such a thin user interface could be distributed to another process or even another machine without a major impact on the system. This capability possibly allows an interactive system to become a Web-based application without much effort.

Document-View-Presentation

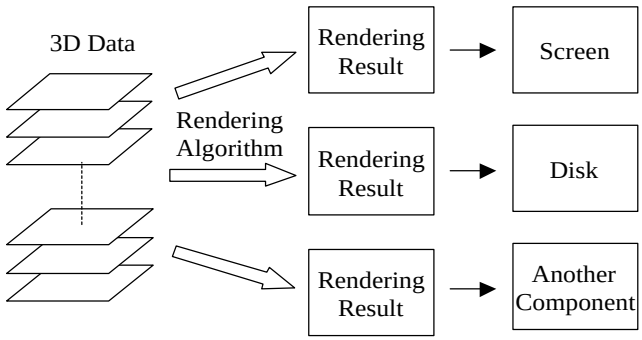
The *Document-View-Presentation* architectural pattern (DVP) divides an interactive application into three components. The document component contains the core functionality and data. Views perform service requests and render the data of the document. Presentation components receive events or service requests and output the rendering result of the view. The change-propagation mechanism helps to keep the state of (1) the document and its views, (2) the view and its presentation components synchronized.

Example

Consider a medical application system such as Discover[1] for 3D imaging generation and diagnostic analysis. The system usually offers several windows for representing and manipulating a 3D data interactively. These windows usually contain different algorithms, such as volume rendering or surface rendering, to render the data. Usually, the amount of computation for these rendering algorithms is so high that sometimes we would speed it up by distributed computing. For the same rendering algorithm, this system supports three different methods to output the rendering result, including displaying it on the screen (i.e. windows), saving it to the disk, and sending it to another component for further processing. When the 3D data changes, all the output methods must reflect changes immediately by executing the rendering algorithm once, not three times, as illustrated in Fig. 1. In particular, for those interactive applications with computationally expensive rendering algorithms, there is a significant difference (in terms of system performance) between executing the algorithm once and three times.



(a)



(b)

Fig. 1 Render the 3D data (a) once and (b) three times to reflect the data change.

In those cases, there is a need to provide different output methods without a major impact on the performance of the system. The system should be able to "plug-and-play" different output methods or even input methods at run time.

Context

Interactive application systems such as graphics or image processing systems with computationally expensive rendering algorithms.

Problem

When designing the architecture of an interactive system, one issue that needs our attention is to keep the data and its functional core independent of the user interface, which comprises input and output. Very often, we would design the system to allow the data model to have many different user interfaces associated with it. And when the data changes, all its user interfaces must reflect changes to the data immediately. Moreover, the rendering function, which translates the data into the form suitable for visual purposes, is regarded as part of the user interface. This limitation makes it unavoidable to execute the rendering function once whenever a user interface needs to be updated.

Suppose there are three different user interfaces with the same rendering function being associated with the same data. All these user interfaces need to be updated to reflect data changes. With the aforementioned limitation, when the data changes, the rendering function would be executed three times, one for each user interface. However, since these user interfaces are with the same rendering function, they should be able to share the rendering result. That is, the rendering function could be executed once in total and the result could be reused by different user interfaces.

Briefly speaking, regarding the rendering function as part of the user interface may multiply the required computational time for updating all the user interfaces to reflect the data changes. The situation becomes worse especially when the rendering algorithm is computationally expensive.

The following *forces* influence the solution:

- The output of the results and behavior of the application must reflect data changes immediately.
- The rendering result could be reused by different output methods.
- Attaching or detaching different output methods to the same rendering result should be easy, and even possible at run time.
- Supporting different output methods should not affect the core of the application, including the rendering function to represent the data.

Solution

The *Document-View-Presentation* pattern (DVP), which builds on the *Document-View* pattern[2,3], divides an interactive system into three components: *document*, *view* and *presentation*.

The *document* component contains data and core functionality to manipulate the data. It is independent of any specific input and output methods.

View components contain the rendering algorithms of the application. They are hidden behind the presentation components and do not interact with end users directly. They accept requests and fulfill the corresponding services by calling the core functionality provided by the document or its own rendering algorithm. They also obtain data from the document and use different algorithms to render the data. A document can have more than one view component associated with it.

Presentation components are the representatives of their views for input and output. They receive user events or messages from other components, and turn them into service requests. They do not implement these services directly. Instead, they forward these requests to their associated views. In addition, they also obtain the rendering result from their views and output the rendering result to a destination based on their own characteristic, such as displaying them on the window, or saving them to the disk. Each presentation component could be for input only, for output only, or for both input and output at the same time. They can be visible or invisible to users. The user can directly interact with the system through those visible presentation components.

Each presentation component should be associated with exactly one view. However, a view component can have several presentation components depending on its applications. In fact, the view component of DVP plays a role similar to the *controller* component of Command Processor design pattern[2]. It allows a system to support different modes of input and output, i.e. different presentation components.

The separation of the document from the view components allows different rendering approaches of the same data. The separation of the view from its presentation components allows different output methods to share the same rendering result of the view. When the presentation component receives the events triggered by the user, it first translates these events into service requests and then forwards the requests to its view. The view fulfills the service request to change the data of the document. Once the data of the document is changed, the document notifies all its dependent views to render the data. When the rendering result is changed, the view notifies all its presentation components. Each presentation retrieves the current rendering result from the view and outputs the result to different devices, such as windows, disks or the network. The change-propagation mechanism could be the Publisher-Subscriber pattern[2] or the Observer pattern[4].

Structure

The *document* component maintains the data model and contains the functional core to manipulate the data. It exports procedures that perform application-specific processing so that the dependent views can call these procedures to provide the service of requests and access the data. Of course, a change-propagation mechanism is provided within the document. It maintains a registry of the dependent view components. Changes to the data of the document trigger the change-propagation mechanism to notify all the dependent views.

View components render the data of the document and keep the rendering results. Each view defines an update procedure that is invoked by the change-propagation mechanism. When the update procedure is called, each view retrieves the data from the document and renders the data. In fact, another change-propagation mechanism is also provided within each view. It maintains a registry of the dependent presentation components. Changes to rendering result trigger the change-propagation

mechanism to notify all its dependent presentation components. View components also accept service requests from the presentation components and fulfill these services by calling the core functionality provided by the document or its own rendering algorithm.

The *presentation* components receive input messages and translate them into different service requests. Each presentation component also defines an update procedure that is invoked by the change-propagation mechanism. When the update procedure is called, each presentation component retrieves the current rendering result from its view and outputs the result according to its own characteristics such as putting the result on the screen.

The Class-Responsibility-Collaborator (CRC) cards[5] for the *document*, *view* and *presentation* components are illustrated in Fig. 2.

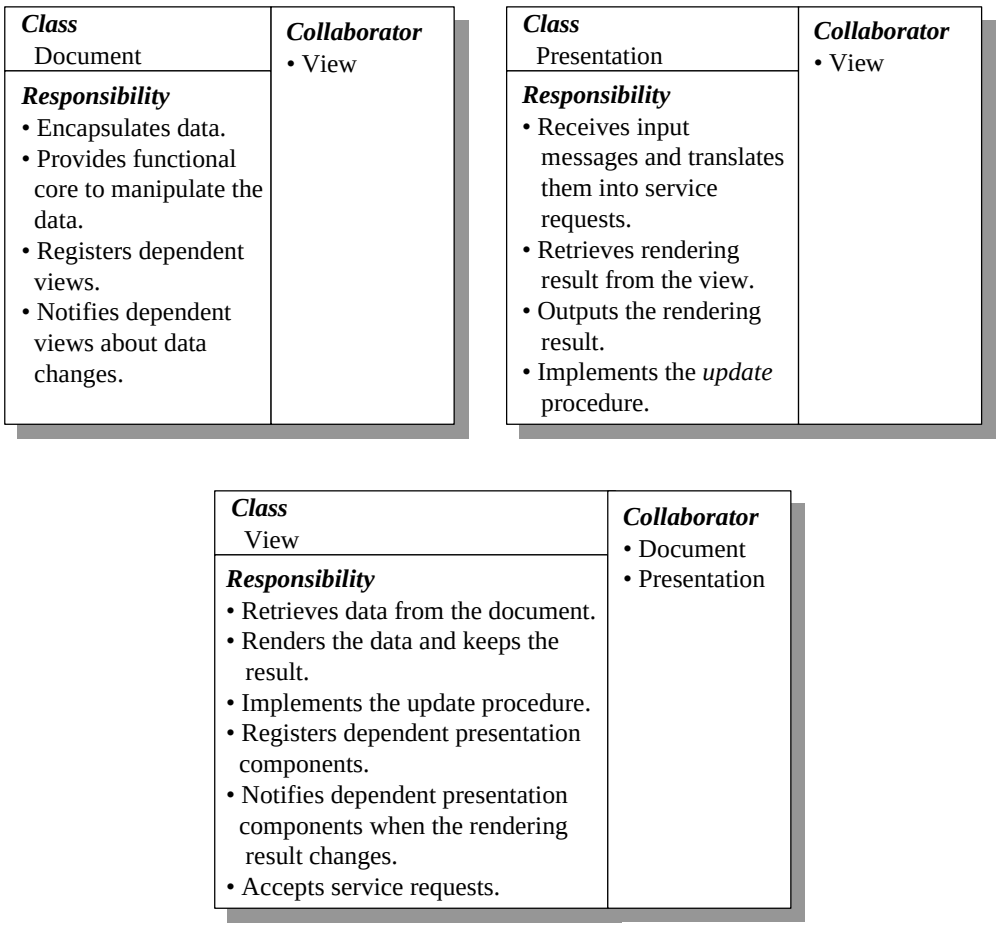


Fig. 2 The CRC cards for the *document*, *view* and *presentation* components.

Fig. 3 shows the principal relationships between the components of the DVP pattern. View and presentation components are subscribers to document and view components respectively. In other words, a document can have more than one view associated with it, while a view can have more than one presentation component. In a C++ implementation, each component is defined as a separate class. View and presentation classes share a common parent - *Subscriber* that defines the *update* interface. Similarly, document and view classes share another common parent – *Publisher* that defines the *attach*, *detach* and *notify* interfaces.

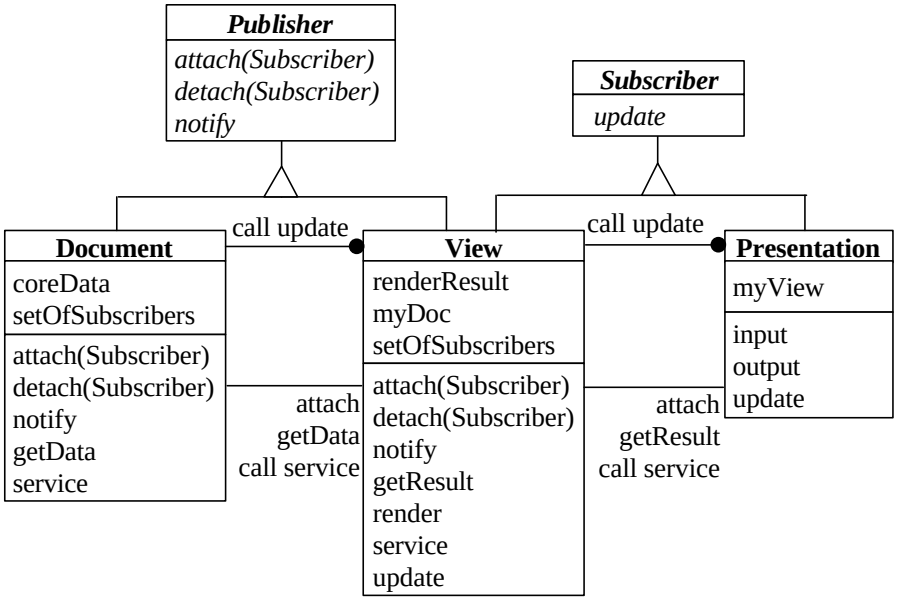


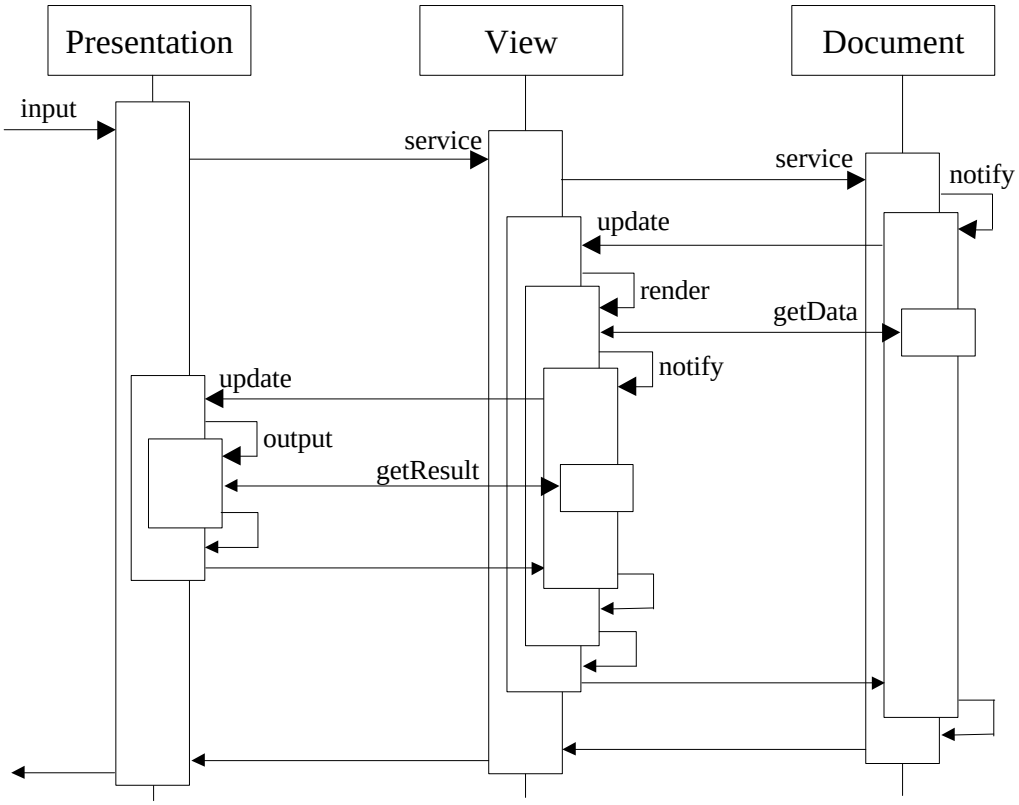
Fig. 3 The object model of DVP with Object Modeling Technique (OMT)[6] notation.

Dynamics

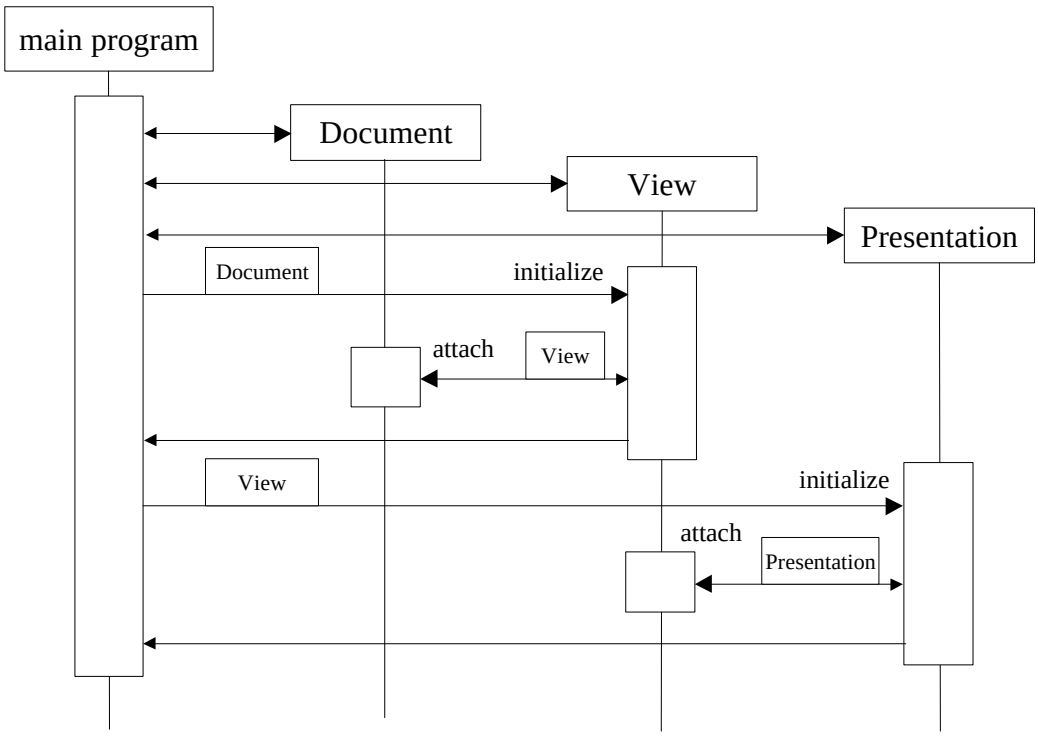
We will illustrate the behavior of a DVP architecture with the following two scenarios.

Scenario I shows how an input message changes the document and how this change triggers the change-propagation mechanism. For simplicity, only one view-presentation pair is shown in the diagram.

- The presentation receives an input message and translates the message into a service request for the view.
- The view invokes a service procedure provided by the document.
- The document fulfills this requested service, which usually changes some data in the document.
- The document notifies all its dependent views by calling the *update* procedures.
- Each view retrieves data from the document and calls the *render* procedure to render the data.
- The view notifies all its dependent presentation components by calling the *update* procedures.
- Each presentation component retrieves the rendering result from the view and outputs the result.



Scenario I



Scenario II

Scenario II shows how the DVP triad is initialized. This initialization is usually conducted by the code inside the main program with the following steps:

- The document instance is created with initialized data structures.
- A view object is created. This takes a reference to the document as a parameter for its initialization.
- The view subscribes to the change-propagation mechanism of the document by calling the *attach* procedure.
- A presentation object is created. This takes a reference to the view as a parameter for its initialization.
- The presentation subscribes to the change-propagation mechanism of the view by calling the *attach* procedure.

Implementation

To implement the DVP pattern, carry out the following fundamental steps:

1. *Define the data model and core functionality.* Analyze the application domain and map it onto an appropriate data model. Separate the core functionality from the desired input and output behavior. Design the document component to encapsulate the data and core functionality. Provide functions for accessing the data.

   ☞ In our Discover example, a BS object (a collection of 2D **B**inary **S**lices) is used to represent 3D objects. Each document contains a BS object and provides methods to access and manipulate the data, i.e. the BS object. Since the document plays the role of the publisher, it inherits from the `Publisher` base class, which will be described in step 2.

```
class Document: public Publisher {
private:
// BSOBJ is the data structure for BS objects
    BSOBJ coreData;

public:
    Document();

// methods for views to access the core data
    BSOBJ * getData();

// methods for views to manipulate the data
    void doThreshold(int low, int high);
    void doCut(CONTOUR * con2r, int alpha, int beta);
}
```

2. *Implement the first change-propagation mechanism.* Assign the role of the publisher to the document based on the Publisher-Subscriber pattern. Extend the document with a registry that holds references to the observing objects – views. Provide a procedure to allow views to subscribe and unsubscribe for the change notification of the data. The document's notify procedure calls the update procedure of all observing objects.

☞ In our C++ example, two abstract classes `Publisher` and `Subscriber` are defined. The `Publisher` class holds a set of references to current subscribers, and provides `attach()` and `detach()` methods to allow observing objects to subscribe and unsubscribe. The `Subscriber` class holds the update interface. Any method like `doThreshold()`that modifies the sate of the document will call the `notify()` method to notify current subscribers. The method `notify()` iterates over all `Subscriber` objects in the registry and calls their `update()` method.

```
class Publisher {
private:
    Set<Subscriber *> registry;

public:
    virtual void attach(Subscriber * s) {
        registry.add(s); }
    virtual void detach(Subscriber * s) {
        registry.remove(s); }

protected:
    virtual void notify() {
    // call update for all subscribers
        Iterator<Subscriber *> iter(registry);
        While (iter.next()) {
            Iter.curr()->update();
        }
    }
};

class Subscriber {
public:
// default is to do nothing
    virtual void update() { }
};
```

3. *Define the contents of the rendering result.* Analyze the required information for different output behavior and define the contents of the rendering result.

   ☞ In 3D medical imaging, we often need different information of a voxel, such as its color, depth (z-value) and transparency, for different purposes. Here, we use the data structure `RENDER_DATA` to collect all required information.

4. *Design and implement the views.* Design the rendering algorithm of each view. Specify and implement a rendering procedure to render the data of the document. Implement the update procedure to reflect the change of the document. The update procedure calls the rendering procedure to render the data. Provide a procedure for each service request. Fulfill the service by calling the core functions of the document or its own algorithm. The initialization of a view needs to subscribe itself to the document.

   ☞ All the concrete view classes share an abstract base class `View`, which defines the common behavior of different views in Discover. The `View` class inherits from both `Publisher` and `Subscriber` base classes. It contains two member variables to represent the rendering result with

corresponding access methods and its relationship with document. The constructor of `View` subscribes to the document to establish this relationship. The destructor removes it from the registry by unsubscribing. `View` also provides methods to receive the service requests from its presentation components. Whenever the `update()` method is called, it generates the new rendering result and then notifies its registered observing objects about this change.

```
class View: public Publisher, public Subscriber {
protected:
    RENDER_DATA renderResult;
    Document * myDoc;

public:
    View (Document * doc) : myDoc(doc)
        { myDoc->attach(this); }
    virtual ~View() { myDoc->detach(this); }

// methods for presentation components to access
// the render data
    RENDER_DATA  * getResult() {
        return (&renderResult);
    }

// methods for presentation components to
// manipulate the data
    void doThreshold(int low, int high) {
        myDoc->doThreshold(low, high);
    }
    void doCut(CONTOUR * con2r, int alpha, int beta) {
        myDoc->doCut(con2r, alpha, beta);
    }

// define the default behavior to reflect changes of
// Document's data
    virtual void update() {
        this->render();
        this->notify();
    }

// abstract method to be redefined:
// default is to do nothing
    virtual void render() {}
};
```

The class definition of `SurfaceRenderingView` demonstrates a concrete view of the system. It redefines the `render()` method to render the BS object in the surface-based approach. In addition, it also provides methods such as `doRotate()` and `doSetColor()` to change the rendering result without actually modifying the data of the document.

```
class SurfaceRenderingView : public View {
public:
    SurfaceRenderingView(Document * doc) : View(doc) {}

// method for surface rendering algorithm
    virtual void render() {
    // get core data from Document
        BSOBJ * bs = myDoc->getData();
    // do the surface rendering algorithm
        SurfaceRendering(&renderData, bs);
    }

// methods to change the rendering attributes
    void doRotate(int alpha, int beta);
    void doSetColor(int r, int g, int b);
};
```

5. *Implement the second change-propagation mechanism.* Assign the role of the publisher to the view. Extend the view with a registry that holds references to the observing objects – presentation components. Provide a procedure to allow presentation components to subscribe and unsubscribe for the change notification of the rendering result. The view's notify procedure calls the update procedure of all observing objects. Note that to the document the view appears as subscriber, while to the presentation components it appears as a publisher.

6. *Design and implement the presentation components.* Specify the input and output behavior for each presentation component. The initialization of a presentation component needs to subscribe itself to the view.

   ☞ All the concrete presentation classes share an abstract base class `Presentation`. The `Presentation` class inherits from the `Subscriber` base class. It contains a member variable to represent its relationship with `View`. The constructor of `Presentation` subscribes to the view to establish this relationship. The destructor removes it from the registry by unsubscribing.

   The class definition of `WndPresentation` demonstrates a presentation component that can receive user inputs and display the rendering result on the screen. It redefines the `update()` method to obtain the current rendering result and display it on the screen. In addition, it also provides call-back functions to receive user inputs.

```
class Presentation: public Subscriber {
protected:
    View * myView;
public:
    Presentation(View * view) : myView(view) {
        myView->attach(this);    }
    virtual ~Presentation() { myView->detach(this); }

// define the default behavior to reflect changes of
// View's rendering result
    virtual void update() { this->output ();}
// abstract method to be redefined
    virtual void output() {}
};
```

```
class WndPresentation : public Presentation {
public:
    WndPresentation (View * view) : Presentation(view)
        {}

// method to output the rendering result
    virtual void output () {
    RENDER_DATA  * result;
    result = myView->getResult();
    // display the image on the screen: …..
    }

// call-back functions to receive user inputs
    void onThreshold() {
        int low, high;  // required parameters
        // open a dialog box to obtain parameters: ..…
        myView->doThreshold(low, high);
    }

    void onRotate() {
        int alpha, beta; // required parameters
        // open a dialog box to obtain parameters: …..
        myView->doRotate(alpha, beta);
    }
};
```

Known Uses

Discover[1,7] is a distributed interactive system for scientific visualization and has been on trial in National Cheng-Kung University hospital in Taiwan since 1993. The system is built on the DVP architectural pattern. In addition to the general imaging analysis and generation functions, Discover also provides the following features:

1. Image Integration

Different but related 3D objects may be generated individually by different rendering algorithms or from different medical imaging devices. Sometimes physicians need to find out their relations in space for diagnostic analysis. Discover allows physicians to integrate several existing 3D objects, i.e. *source objects*, into a new one, i.e. the *integrated object*, for such visualization purposes.

As illustrated in Fig. 4, before the integration process begins, each view is initialized with a visible *default presentation* component, which can accept user events and display the rendering result in a window. When a 3D object is chosen to be a source object, an *output presentation* is created and attached to the view at run time. This *output presentation* component is invisible to users and does not receive input messages at all. At the same time, the *output presentation* is also a publisher and outputs the view's rendering result to its subscribers. In addition, a new document, view and default presentation triad for the integrated object is also created. This new document registers with the *output presentation* of each source object for obtaining their rendering results respectively.

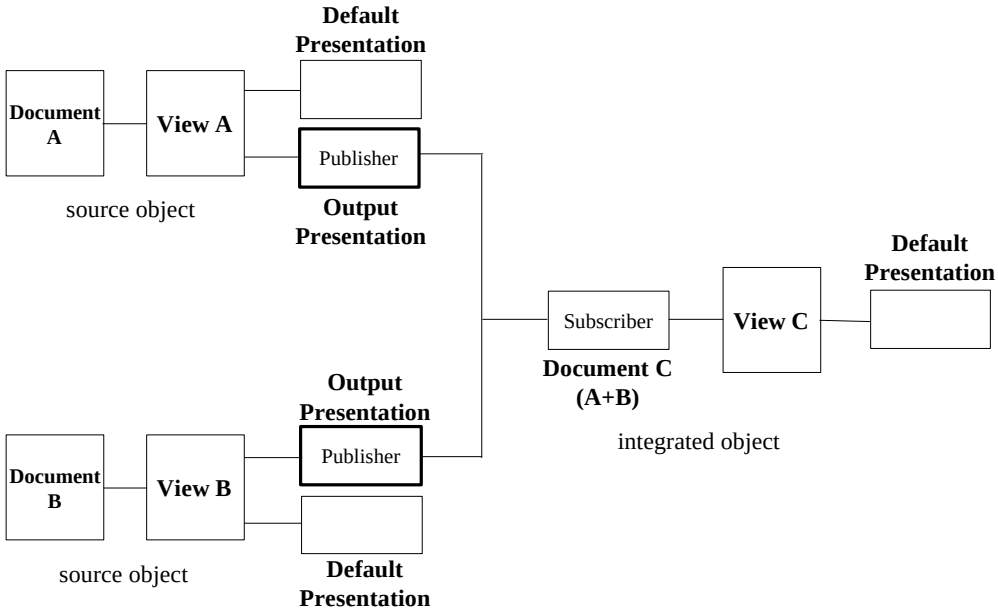


Fig. 4 An integrated object could be easily obtained by attaching an additional *output presentation* to each source object.

Thus when the physician modifies any one of the source objects through its *default presentation*, the rendering result of the view will be updated by executing the rendering algorithm once. Both *default presentation* and *output presentation* components must reflect the changes immediately: the former displays the rendering result to the physician, while the latter notifies the integrated document. The integrated document updates its data by retrieving the new rendering result from the *output presentation*. On the heels of this, the view of the integrated object renders the integrated data and notifies its *default presentation* to display the result to the physician.

2. Web-based application

Usually the 3D imaging generation and diagnostic analysis provided by Discover are very time-consuming functions. Although it is possible to install the Discover system in every physician's desktop of the hospital, not every desktop has enough computation power to run such heavy computation jobs within an acceptable response time. Therefore, Discover also allows physicians to use Web browsers to control it. Thus, we can install Discover on a server machine with high computation power. Physicians can use their desktops with different computation ability to control Discover through the network. The response time for each desktop is almost the same without considering the differences between the network traffic and the server's load.

As illustrated in Fig. 5, when the Web server receives a service request coming from a Web browser to control Discover, it launches a **Common Gateway Interface** (CGI) application to fulfill this service. This CGI application first creates an *input presentation* and an *output presentation* for the view component through the automation mechanism[8], and then forwards the service request to the *input presentation*. When the rendering result kept by the view changes, the *output presentation* delivers the rendering result to CGI, which forwards the result to the Web server. At last, the rendering result will be sent to the Web browser and the physician can see the image on the browser.

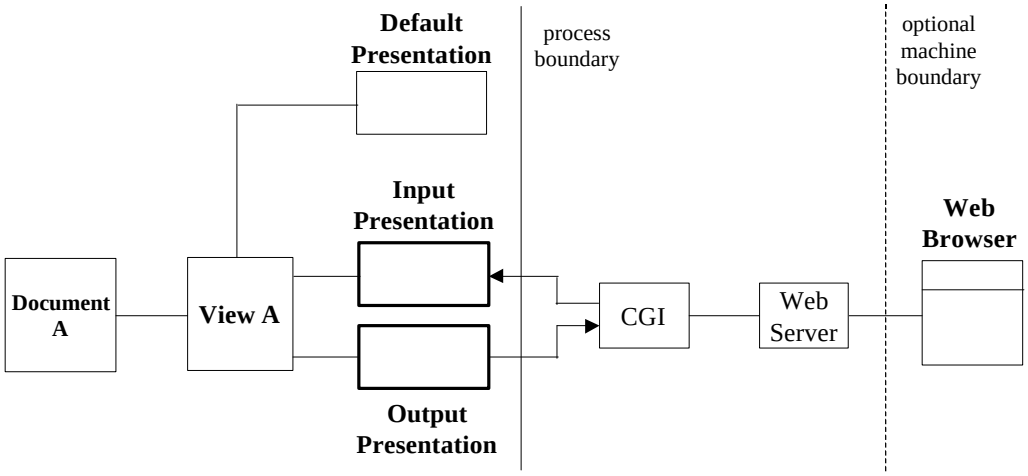


Fig. 5. Discover could be controlled by a Web browser through the cooperation between the *input presentation* and the *output presentation*.

Consequences

The Document-View-Presentation architectural pattern has several **benefits**:

1. *Reuse the rendering result.* When multiple user interfaces, i.e. presentation components, share the same rendering result, the rendering function needs to be executed only once. This can avoid multiplying the required computational time and acquire a more quick response, especially when the rendering function is heavy in computation.
2. *Separation of concerns.* This separation of the rendering process and the input/output behavior allows the software developer to concentrate on the development of the rendering algorithms without worrying with any input/ output processes. Of course, they can also focus on providing different input and output functions without involving the development of rendering algorithms.
3. *"Pluggable" presentation components.* Multiple presentation components can be implemented and used with a view. These presentation components can be added or even substituted at run time, without any influence on the view.
4. *Thin user interface.* To remove the rendering process from the input and output process makes the user interface "thin". The thin user interface can be distributed to another process or another machine easily. For example, an interactive system can become a Web-based application if we distribute the user interface through the HTTP protocol.
5. *Easy to implement DVP using modern frameworks for window applications.* Several frameworks for developing window applications, such as MFC of Visual C++ or OWL of Borland C++, adopt the Document-View pattern as their default system architecture[3]. The DVP, in essence, is an extension of the Document-View pattern. Therefore, we can implement the DVP pattern easily by making use of all the classes related to the document-view architecture and their communication mechanism provided by the framework. We can suppress the input/output functions of

the views in the framework and provide additional presentation components. Of course, we still need to implement the communication mechanism between the view and its presentation components.

The **liability** of this pattern is as follows:

1. *Increased complexity*. The division of each user interface into the view and presentation components may result in a complex system structure. If the algorithm to render the data of the document is simple, this separation increases complexity without gaining much performance improvement.

See Also

- The *Model-View-Controller pattern* $(MVC)^2$ divides an interactive application into three components. The *model* contains the core functionality and data. *Views* display information to users. *Controllers* handle user input. The *Document-View pattern* (DV), a variant of MVC, combines the view and the controller from MVC into a single component. In these two patterns, to display information to users, in fact, includes rendering (or interpreting) the data and putting the result on the screen.

  The components of the DVP pattern and the above-mentioned two patterns, together with their major responsibility are summarized in Table 1. The common place of these three patterns is to put the core functionality and data together into a single component, i.e. the *model* or the *document*. The major feature that makes the DVP pattern different from the MVC/DV patterns is to separate the "raw" input and output process from other components. This includes (1) dividing the display of data in the document to users into two parts: the data rendering portion and its output portion, (2) decoupling the close relations between user inputs and service requests.

MVC	DV	DVP
Model • core functionality • data	**Document** • core functionality • data	**Document** • core functionality • data
View • display (render + output)	**View** • display (render + output) • user input	**View** • rendering functionality • accept service requests
Controller • user input		**Presentation** • output • user input

Table 1 A summary of three patterns for interactive systems.

In addition, the architectural partition of an interactive system like MVC/DV patterns could be regarded as a 2-tier client/server architecture inside an application, as shown in Fig. 6. The data and functional core runs on the server side while the clients run the rendering function and graphical user interfaces. For

those systems with heavy rendering algorithms, the client side becomes very ”fat”. Correspondingly, the DVP pattern could be regarded as a 3-tier architecture, as illustrated in Fig. 7. In such a 3-tier system architecture, the middle tier consists of the business-logic units that are usually independent of the end-user applications and can be reused by various applications. In Discover, the system always keeps the client side “thin” by removing the data rendering process (the business logic unit for a computer graphics application) from the client side to the middle tier on the server side. This separation allows different clients (presentation components) to reuse the rendering result of the middle tier (view).

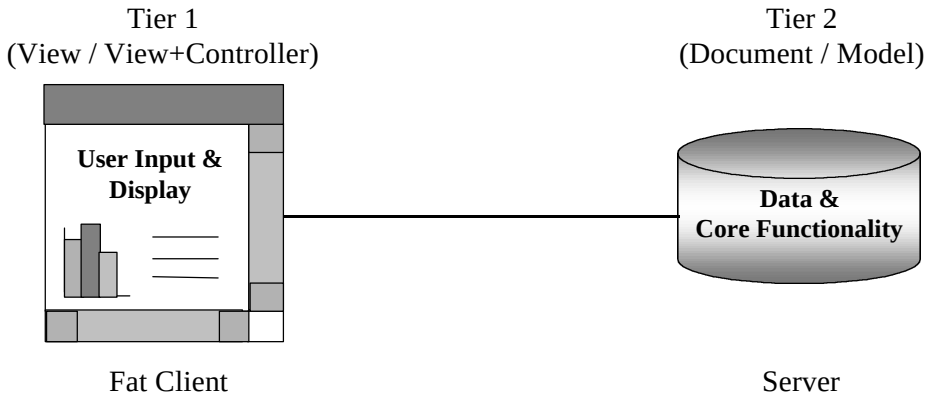


Fig. 6. Both MVC and DV patterns divide an interactive application into two tiers.

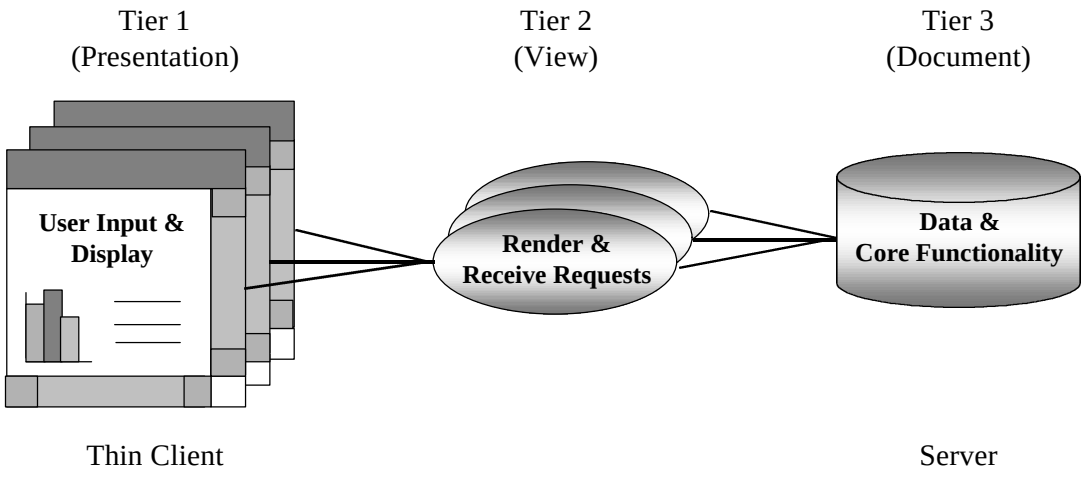


Fig. 7 The DVP pattern divides an interactive application into three tiers.

References

1. P. W. Liu et al., "Distributed Computing: New Power for Scientific Visualization," *IEEE Computer Graphics and Applications*, Vol. 16, No. 3, May 1996, pp.42-51.
2. F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal, *A System of Patterns - Pattern-Oriented Software Architecture*, John Wiley & Sons, New York, 1996.
3. D. Kruglinski: *Inside Visual C++*, Microsoft Press, 1996.
4. E. Gamma, E. Helm, R. Johnson and J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
5. K. Beck and W. Cunningham, "A Laboratory For Teaching Object-Oriented Thinking," *Proceedings of OOPSLA'89*, N. Meyrowitz(Ed), Special Issue of SIGPLAN Notices, Vol. 24, No. 10, October 1989, pp. 1-6.
6. J. Rumbaugh et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
7. K. Y. Chang and L. S. Chen, "Using Design Patterns to Develop a Hyper-controllable Medical Image Application," *Proceedings of PLoP'98*, Monticello, Illinois, Aug 11-14, 1998.
8. R. M. Adler, "Emerging Standards for Computing Software," *Computer*, March 1995, pp. 68-77.

Acknowledgements

Special thanks to Rosana T. Vaccare Braga for her valuable comments and suggestions for the improvement of this pattern.