Easy Viewing

Design Patterns in View

Using design patterns in development gives us a lot of advantages. It promotes loose-coupling, eases redevelopment, allows code reuse, and so on. However, improper use would give us ultra spaghetti code as they are scattered into multiple objects. This article demonstrates how design patterns can make our lives easier in common head aching cases.

By Man-ping Grace Chau & Ka-hung Hui

Background: MVC Pattern

In this article, we shall look at why Model-View-Controller (MVC) could be useful. Let us look at two examples: course registration and online XSL file generation (see Fig. 1 and 2).

In both cases, the applications repeat the same processes, namely:

- Obtaining data from disk
- Optimising the data and generating an input form for users
- Acquiring user input
- Processing user input and generating acknowledgement

- Saving the data back to disk

This process is applicable to web applications, like online shopping and attendance record systems.

Repetition of the processes is tantamount to reuse. It would be tedious to rewrite entire applications when only the presentation of the form and format of the dataset is changed. There should be a way to reuse the application's 'framework' only by altering parameters. And, this is what breeds the idea of MVCs.

In a web application, the Controller is responsible for accepting user inputs and performing error checking/correction. Based on the user request, it determines which Model and View should be used. While the Model is responsible for business logic and accessing data in disk, the View is solely for presentation purposes. In the example of course registration, the steps and the corresponding responsibilities of each object are as followed:

- The Model gets the data from the course database
- The View is updated and course registration form is generated for students
- Students choose the course according to the vacancy, and submit the form
- The Controller retrieves students' inputs and performs error detection
- The Model is informed of the students' inputs, and it updates the course database
- The View to display the acknowledgement is chosen, and registration result is displayed to the students

This relationship is detailed in Fig. 3.

While there may be similarities in the functionalities of web applications, an efficient user interface plays an integral role in its success. In the following section, we will look into how design patterns can be employed in constructing Views to deal with several common problems. For example, how to avoid writing repetitive codes by reusing presentation designs, enable redevelopment by promoting loose-coupling
between form components, and minimise classes spanned out by choosing the right design pattern to use.

Different Implementation for the Same Presentation

The need for different implementation of the same presentation originates from the popular use of mobile devices; while web browsers require HTML/XHTML, WAP phones need WML. Since the logic for building the form is the same, only the implementation for the form's components differs as per the various requirements of markup languages. In such situations, the Bridge pattern enables separation of the implementation and logic.

In this example, there are two kinds of form that we are interested in: the map quest form and a course registration form. The logic in constructing the forms is encapsulated in the objects MapQuestBuilder and RegisterBuilder. They require the same component, Table, to build the form. In Listing 1, we assume there is a certain way of learning about the implementation the user requires. Based on that information, different Table objects will be instantiated to build the form without changing the logic of the Builder, since all the Table objects share the same interface.

This example shows how to separate the implementation from the logic, to enable easy addition/reuse of components and switching the implementation of components without affecting the logic.

Fig. 4 shows that Builder can be added easily, and all objects can share components to construct different kinds of forms.

Maintaining States

Often, the user interface generated and the actions to be performed depend on the state of user; for example, whether a student has enough points to register for a course. One way to achieve this is to first perform state checking. The simplest way to check states is to use if-else statements. However, this type of checking introduces too many if-else statements. Additionally, if the number of possible states becomes too large, it will be hard for developers to keep track of the state transitions and the relationship between different states. Using the State pattern can solve this problem.

As shown in Listing 2, in State pattern, actions are performed according to the state of the user. The transformation of state can be

Listing 1:
```php
<?php
/**
 * Constructor
 */

function MapQuestBuilder(Slange)
{
    if (Slange === "XHTML") {
        $this->table = new XHTMLTable();
    } elseif (Slange === "WML") {
        $this->table = new WMLTable();
    } else {
        $this->table = new HTMLTable();
    }
    $this->statement = "";
}

/* Instance methods */

* * * Draw the input form *
*/

function DrawForm()
{
    $this->statement = "";
    $this->table->tableStart(1, 1);
    $this->statement = "";
    $this->table->rowStart();
    $this->statement = 
    $this->table->tableFooter();
    $this->statement = 
    $this->table->tableEnd();
    return $this->statement;
}

Listing 2:
```
Each visitor should identify the node type so that corresponding actions can be carried out. In PHP, this can be done by the new instanceof keyword, as shown in Listing 4.

Of course, the disadvantage of using this approach would be introducing too many if-else statements. This breaks the idea of encapsulation as the Visitor knows too much about the Node. The primary problem here is that only single dispatch is supported in PHP, meaning that the function to be run depends on the function name and type of object associated with the called function.

The Visitor pattern helps alleviate this problem by emulating "double dispatch". In double dispatch, the function to be run also depends on the type of object, which is passed to the function as an argument, in addition to the two criteria mentioned. See Listing 5 for an example.

In the example, when CourseNode:Accept() is called, the node type is identified and passed to PrintVisitor:Visit() via the "this" pointer. When PrintVisitor:Visit() is called, the type of node and visitor is identified before the correct implementation is chosen.

To increase the number of states, the Visitor pattern can be used together with the State pattern, by providing a visitor-type parameter to the checkState() method, as shown in Listing 6. The only inconvenience of this approach is that all checkState() methods in the state subclass must call the overridden checkState() method in the base state class.

Complex Components
There are several patterns that enable us to manipulate complex components. The simplest method is by delegation, which distributes the heavyweight processes to other objects. According to the Strategy pattern, each object may share the same interface such that the client using them is ignorant about the change of objects.

The next task is to draw menus in table cells in our first example with Table and Builder. The responsibility of drawing the menus can be encapsulated in different Menu objects, which share the same interface. In our case, more Menu objects can be added by subclassing, and the Builder can change the Menus without altering the code.

The Command pattern (see Fig. 6) is also employed in this example (see Listing 7). The Builder fixes the menu content by parameterising the Menu object. The Menu object is passed to the Table, and $menu->draw() is called to draw the different Menu. Only Builder, which controls the logic of the form, knows about the content of Menu. This effic-
tively ensures encapsulation, as the `Table` is ignorant about how the `Menu` changes.

Additional flexibility can be introduced with a string being used as a callback. As a result, even the callback function call in `Table` can be determined in `Builder`; such that `Table` remains ignorant about the action to manipulate both `Menu` and the `Menu` content.

This could be applied to an application for displaying library items, like books, CDs and magazines. The `Table` will have no idea of what it is displaying, since the `Builder` controls it.

To construct a form with even more complex structure, Chain of Responsibility can be employed (see Fig. 7). This is done by delegating the responsibilities in drawing a complex component to multiple objects. All these components should share the same interface, and the chain in per-

forming the action can be arranged by the Composite pattern. With Composite pattern, the same abstract class represents the primitives and the container, such that recursive composition can be used and any discrimination can be avoided.

As shown in Fig. 7, it is the Component, instead of the `Builder`, which decides what oth-

Listing 4:

```php
function Visit(Node $node) {
    // checking node type manually
    if ($node instanceof CourseNode) {
        $this->statement = $node->Name;
    } else if ($node instanceof ActivityNode) {
        $this->statement = $node->Time;
    } else {
        // same more if else block to check
        // conditions and change state if necessary
        $this->setState($this);
    }
}
```

Listing 5:

```php
class CourseNode {
    function Accept(Visitor $visitor) {
        // the type of node is identified
        $visitor->Visit($this);
        // concrete type of node is passed
        // to the visitor
    }
}
class PrintVisitor {
    function Visit(CourseNode $node) {
        $this->statement = $node->Name;
        // here, both the types of node and
        // visitor are known
    }
}
class ActivityNode {
    function Accept(Visitor $visitor) {
        $this->statement = $node->Time;
    }
}
```

Listing 6:

```php
class State {
    function checkState(Visitor $visitor) {
        $visitor->Visit($this);
    }
}
class ExceedState extends State {
    function checkState(Visitor $visitor) {
        if ($SESSION['limit'] === 0) {
            $SESSION['limit'] = new NormalState(); // change state
        } else {
            // more if else block to check
            // conditions and change state if necessary
            $this->setState($this);
        }
    }
}```
Listing 7:

class Table {
  // some code
  // Draw table content
  function draw(Sinfo) {
    // if (Sinfo->getTypet() == "menu") {
    Sthis->component = new Menu();
    // } // some more if-else block
    Sthis->statement = Sthis->component->draw(Sinfo);
    return Sthis->statement;
  }

  class Builder {
    // some code
    // Build the form
    function build(Sinfo) {
      Sthis->component = new Table();
      Sthis->statement = Sthis->component->draw(Sinfo);
      // some more if-else block
    }
  }
}

Listing 8:

class Table {
  // some code
  // Draw table content
  function draw(Sinfo) {
    // if (Sinfo->getTypet() == "menu") {
    Sthis->component = new Menu();
    // } // some more if-else block
    Sthis->statement = Sthis->component->draw(Sinfo);
    return Sthis->statement;
  }

  class Builder {
    // some code
    // Build the form
    function build(Sinfo) {
      Sthis->component = new Table();
      Sthis->statement = Sthis->component->draw(Sinfo);
      // some more if-else block
    }
  }
}

er components they would use to finish their job. This adds loose coupling, since the objects that request the action would not know how it is done. Besides, more flexibility is introduced as the object used can be changed at runtime and components can be added easily by subклассing. One disadvantage of this approach is that the interface of all components must be the same, so the Composite pattern treats them equally.

In Listing 8, Builder delegates the responsibility of drawing the entire table to Table. Table notices that it would not be able to handle the table content, and delegates the job to Menu. Menu checks if it can finish drawing by itself or it should further delegate the responsibility to others. In this case, each action caller can toss the unfinished job to the next object without caring whether the job would be thrown further. However, we have to ensure there is a recipient at the end of every chain.

Creating by Prototype

Sometimes, we need to have an array of objects to keep track of different components in the form. For example, one application may need to keep track of a group of Library components, while another keeps track of a group of Supermarket components. The simplest method is to have different factories for different forms (see Fig. 8). But it would be wasteful to have so many factories if the construction logic of initiating the component is the same.

The Prototype pattern's construction framework can be used repeatedly for creating different components (see Fig. 9). The Form is given the prototype object and it can use the same construction process with those objects cloned to build the entire form. The prototype object can be changed easily during runtime.

Listing 9:

// Get multiple DataTransfers from Model,
// register in session
foreach (Svalue as $valueObject) {
  $obj = &$valueObject->getDataType();
  $name = $obj->getNama();
  $s['Nama'] = $obj;
  $SESSION['Nama'] = $obj;
not update the View at once, because the web browser will not request new information unless the user refreshes it, which is referred to as the pull approach. However, the many advantages of the MVC model are sufficient to make it a valuable part of the development process.

**Conclusion**

In this article, we have introduced various ways to utilise different design patterns to ease the development of applications, like implementing different interfaces for the same presentation, maintaining states, building complex components, and creating objects by prototypes. These patterns, if used properly, help reduce development time, as well as make extensions of existing applications as easy as possible.

Man-ping Grace Chau and Ka-hung Hui are year-three undergraduate students at the Chinese University of Hong Kong, majoring in Information Engineering. They are also team mates of the school IT team, responsible for developing and maintaining over 10 systems in PHP for the entire school, including all kinds of attendance record systems, as well as school project databases.

**Links & Literature**

- Ootips.com’s introduction to MVC: [http://ootips.org/mvc-pattern.html](http://ootips.org/mvc-pattern.html)
- For more information about design patterns, please refer to Design Patterns – Element of Reusable Object Oriented Software by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, from Addison Wesley (ISBN 0201633612)