Refactoring by Design Pattern in Software Development

<Object-Oriented Programming>

<Lee Zong Yang>

Abstract:

As time progress, the codes become bloated and harder to understand. This could lead to difficulty in spotting bugs, performing maintenance, and adding new feature. Although refactoring is not the "silver bullet", performing it correctly does drastically get rid of these problems. By reviewing literatures and carry out case study, the results of refactoring with design pattern are inspected in this research paper.

I. INTRODUCTION

In the developing of a game using C# and Swingame API, several difficulties faced in implementing features and maintaining high flexibility in the modification of existing code, modularity of class, and readability of the codes. The game is named as A Day in Candy Shop. The player will play as a waitress that need to serve the order of the customer. (Figure 13)

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" – Christopher Alexander. Although Christopher Alexander is an architect, what he says is useful for developing good object-oriented software. I found it very interesting since it is time-tested solution to known problem.

While I was developing function for the transition between main menu interface, instruction interface, game-playing interface, and end-game interface, I faced serious maintenance issue and bug-prone code by handling the logic that determines interface transition in only one class.

This indicates the technical debt had accumulated till a degree that made the code low-readability and thus increases the time needed to develop new features as more time needed to understand the system. At this point, I decided to perform refactoring to improve the quality of the code.

Refactoring is an alteration made to the internal structure of software result in understandable and easier to modify the code without altering its explicit behaviour. Refactoring brings following benefits: improves the design of software, make it easier to understand, higher chance to spot bugs, and increase the efficiency in developing software. (Fowler 1999)

II. RESEARCH METHODOLOGY

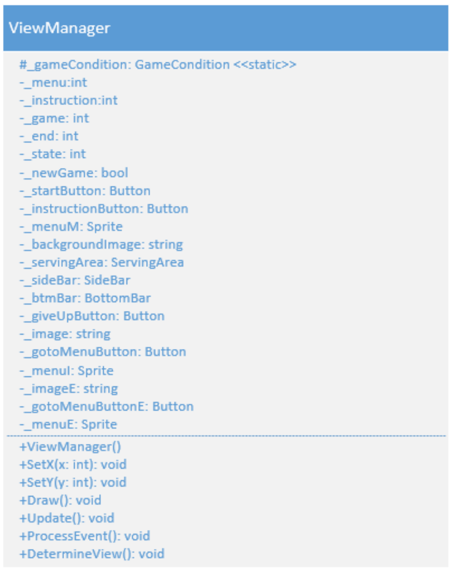
Applying appropriate design pattern could increase the modularity of software and improve the readability of the code. The research conduct by using case study approach and systematic literature review. The case studied is based on the software A Day in Candy Shop.

Systematic literature review act as a way to find solution to the problem encountered while developing A Day in Candy Shop. Comparison carries out between the structure of code before applying the solution and after. Outcome are inspected based on software engineering principle.

III. FINDINGS

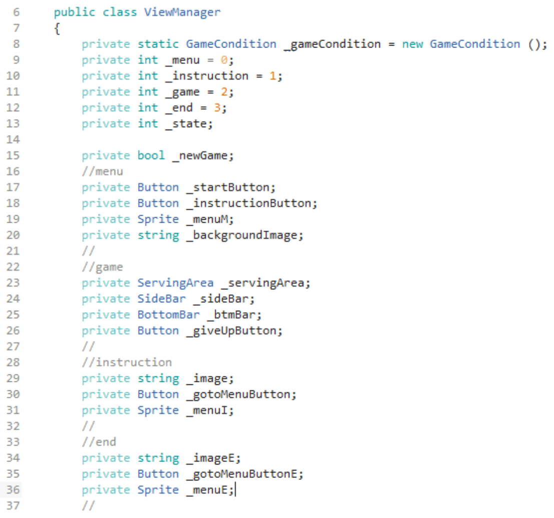
-Before Refactoring:

Class Diagram



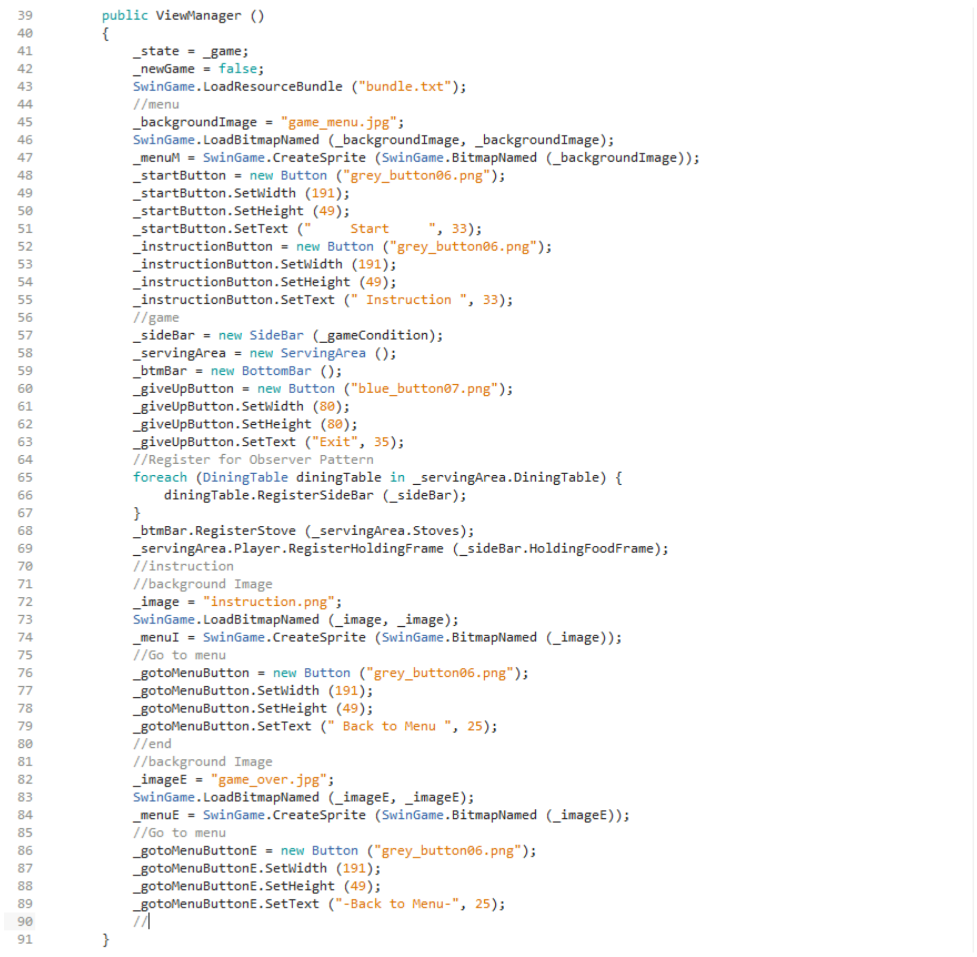
```
ViewManager
-------------------------------------------
#_gameCondition: GameCondition <<static>>
-_menu:int
-_instruction:int
-_game: int
-_end: int
-_state: int
-_newGame: bool
-_startButton: Button
-_instructionButton: Button
-_menuM: Sprite
-_backgroundImage: string
-_servingArea: ServingArea
-_sideBar: SideBar
-_btmBar: BottomBar
-_giveUpButton: Button
-_image: string
-_gotoMenuButton: Button
-_menuI: Sprite
-_imageE: string
-_gotoMenuButtonE: Button
-_menuE: Sprite
-------------------------------------------
+ViewManager()
+SetX(x: int): void
+SetY(y: int): void
+Draw(): void
+Update(): void
+ProcessEvent(): void
+DetermineView(): void
```

Code



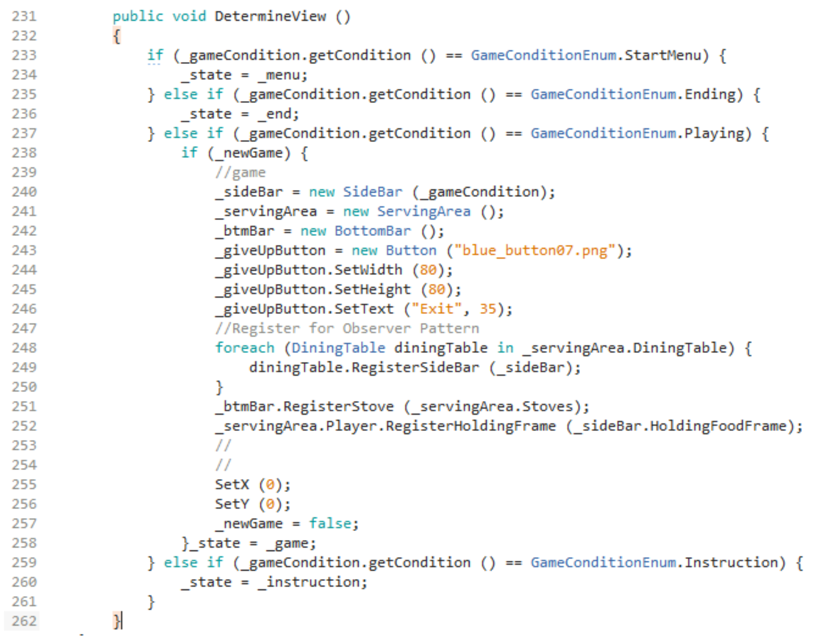
```
public class ViewManager
{
    private static GameCondition _gameCondition = new GameCondition ();
    private int _menu = 0;
    private int _instruction = 1;
    private int _game = 2;
    private int _end = 3;
    private int _state;

    private bool _newGame;
    //menu
    private Button _startButton;
    private Button _instructionButton;
    private Sprite _menuM;
    private string _backgroundImage;
    //
    //game
    private ServingArea _servingArea;
    private SideBar _sideBar;
    private BottomBar _btmBar;
    private Button _giveUpButton;
    //
    //instruction
    private string _image;
    private Button _gotoMenuButton;
    private Sprite _menuI;
    //
    //end
    private string _imageE;
    private Button _gotoMenuButtonE;
    private Sprite _menuE;
    //
```

Figure 1: ViewManager's field



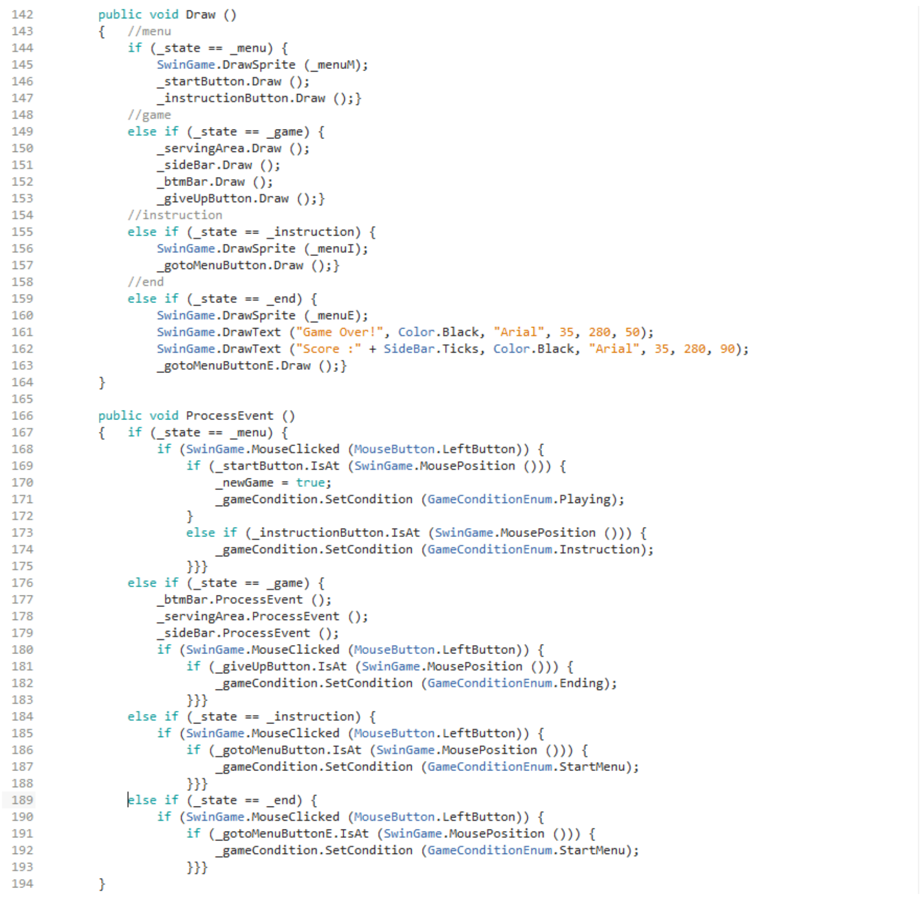
```
        public ViewManager ()
        {
            _state = _game;
            _newGame = false;
            SwinGame.LoadResourceBundle ("bundle.txt");
            //menu
            _backgroundImage = "game_menu.jpg";
            SwinGame.LoadBitmapNamed (_backgroundImage, _backgroundImage);
            _menuM = SwinGame.CreateSprite (SwinGame.BitmapNamed (_backgroundImage));
            _startButton = new Button ("grey_button06.png");
            _startButton.SetWidth (191);
            _startButton.SetHeight (49);
            _startButton.SetText ("     Start     ", 33);
            _instructionButton = new Button ("grey_button06.png");
            _instructionButton.SetWidth (191);
            _instructionButton.SetHeight (49);
            _instructionButton.SetText (" Instruction ", 33);
            //game
            _sideBar = new SideBar (_gameCondition);
            _servingArea = new ServingArea ();
            _btmBar = new BottomBar ();
            _giveUpButton = new Button ("blue_button07.png");
            _giveUpButton.SetWidth (80);
            _giveUpButton.SetHeight (80);
            _giveUpButton.SetText ("Exit", 35);
            //Register for Observer Pattern
            foreach (DiningTable diningTable in _servingArea.DiningTable) {
                diningTable.RegisterSideBar (_sideBar);
            }
            _btmBar.RegisterStove (_servingArea.Stoves);
            _servingArea.Player.RegisterHoldingFrame (_sideBar.HoldingFoodFrame);
            //instruction
            //background Image
            _image = "instruction.png";
            SwinGame.LoadBitmapNamed (_image, _image);
            _menuI = SwinGame.CreateSprite (SwinGame.BitmapNamed (_image));
            //Go to menu
            _gotoMenuButton = new Button ("grey_button06.png");
            _gotoMenuButton.SetWidth (191);
            _gotoMenuButton.SetHeight (49);
            _gotoMenuButton.SetText (" Back to Menu ", 25);
            //end
            //background Image
            _imageE = "game_over.jpg";
            SwinGame.LoadBitmapNamed (_imageE, _imageE);
            _menuE = SwinGame.CreateSprite (SwinGame.BitmapNamed (_imageE));
            //Go to menu
            _gotoMenuButtonE = new Button ("grey_button06.png");
            _gotoMenuButtonE.SetWidth (191);
            _gotoMenuButtonE.SetHeight (49);
            _gotoMenuButtonE.SetText ("-Back to Menu-", 25);
            //
        }
```

Figure 2: ViewManager's constructor



```
        public void DetermineView ()
        {
            if (_gameCondition.getCondition () == GameConditionEnum.StartMenu) {
                _state = _menu;
            } else if (_gameCondition.getCondition () == GameConditionEnum.Ending) {
                _state = _end;
            } else if (_gameCondition.getCondition () == GameConditionEnum.Playing) {
                if (_newGame) {
                    //game
                    _sideBar = new SideBar (_gameCondition);
                    _servingArea = new ServingArea ();
                    _btmBar = new BottomBar ();
                    _giveUpButton = new Button ("blue_button07.png");
                    _giveUpButton.SetWidth (80);
                    _giveUpButton.SetHeight (80);
                    _giveUpButton.SetText ("Exit", 35);
                    //Register for Observer Pattern
                    foreach (DiningTable diningTable in _servingArea.DiningTable) {
                        diningTable.RegisterSideBar (_sideBar);
                    }
                    _btmBar.RegisterStove (_servingArea.Stoves);
                    _servingArea.Player.RegisterHoldingFrame (_sideBar.HoldingFoodFrame);
                    //
                    //
                    SetX (0);
                    SetY (0);
                    _newGame = false;
                }_state = _game;
            } else if (_gameCondition.getCondition () == GameConditionEnum.Instruction) {
                _state = _instruction;
            }
        }
```

Figure 3: ViewManager's DetermineView method for interface transition



```
public void Draw ()
{   //menu
    if (_state == _menu) {
        SwinGame.DrawSprite (_menuM);
        _startButton.Draw ();
        _instructionButton.Draw ();}
    //game
    else if (_state == _game) {
        _servingArea.Draw ();
        _sideBar.Draw ();
        _btmBar.Draw ();
        _giveUpButton.Draw ();}
    //instruction
    else if (_state == _instruction) {
        SwinGame.DrawSprite (_menuI);
        _gotoMenuButton.Draw ();}
    //end
    else if (_state == _end) {
        SwinGame.DrawSprite (_menuE);
        SwinGame.DrawText ("Game Over!", Color.Black, "Arial", 35, 280, 50);
        SwinGame.DrawText ("Score :" + SideBar.Ticks, Color.Black, "Arial", 35, 280, 90);
        _gotoMenuButtonE.Draw ();}
}

public void ProcessEvent ()
{   if (_state == _menu) {
        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_startButton.IsAt (SwinGame.MousePosition ())) {
                _newGame = true;
                _gameCondition.SetCondition (GameConditionEnum.Playing);
            }
            else if (_instructionButton.IsAt (SwinGame.MousePosition ())) {
                _gameCondition.SetCondition (GameConditionEnum.Instruction);
            }}}
    else if (_state == _game) {
        _btmBar.ProcessEvent ();
        _servingArea.ProcessEvent ();
        _sideBar.ProcessEvent ();
        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_giveUpButton.IsAt (SwinGame.MousePosition ())) {
                _gameCondition.SetCondition (GameConditionEnum.Ending);
            }}}
    else if (_state == _instruction) {
        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_gotoMenuButton.IsAt (SwinGame.MousePosition ())) {
                _gameCondition.SetCondition (GameConditionEnum.StartMenu);
            }}}
    else if (_state == _end) {
        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_gotoMenuButtonE.IsAt (SwinGame.MousePosition ())) {
                _gameCondition.SetCondition (GameConditionEnum.StartMenu);
            }}}
}
```

Figure 4: ViewManager's Draw and ProcessEvent method

As shown in figure 1 and 2, each interface's fields need to declare and initialize in the ViewManager class leading to huge numbers of field which increase difficulty in tracing bug and maintaining code. Figure 3 show extra method with multiple conditional statements needs to determine interface transition. ViewManager class has grown into a monolith with long procedure and large conditional statements as shown in figure 4.

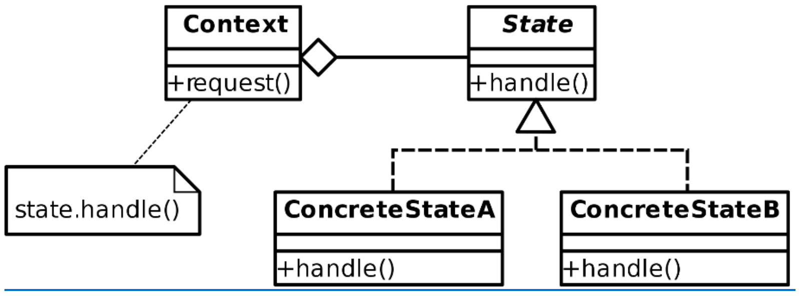
Special attention needs to be given to the if else statement that determines the interface transition which increases the time for the developer to modify and extend the software. The task to maintain or extend the functionality will grow more complicated when more interfaces are introduced due to modification need to be done to these long procedure and large conditional statements.

This violates the open-closed principle which state software needs to be open for extension but closed for modification. (Martin 1996)

Solution Discovery

By browsing through the design pattern catalog in Design Patterns Elements of Reusable Object-Oriented Software, I found that state pattern could solve the problem. (Gamma et al. 1995)

State Pattern



Applicable scenario:

1. An object’s behaviour depends on its present state/interface, and it needs to alter its behaviour at run-time based on its present state/interface.
2. Functions have large, nested conditional statements that result from multiple object’s states/interfaces.

Function:

1. When an object’s state/interface changes, its behaviours will change according to the current state/interface.
2. By referencing different state/interface objects, the context object will appear to be instantiated from another class.

Participants:

1. ViewManager – Context
   i. Interface to clients
   ii. Has an instance of View class that determines the present interface.
2. View – State/Interface Base Class
   i. Define a common base class for all concrete state/interface subclasses.
3. Menu, Instruction, Game, End – Concrete State/Interface Subclasses
   i. Inherited from the same class – View, so capability of transition between each other is provided.

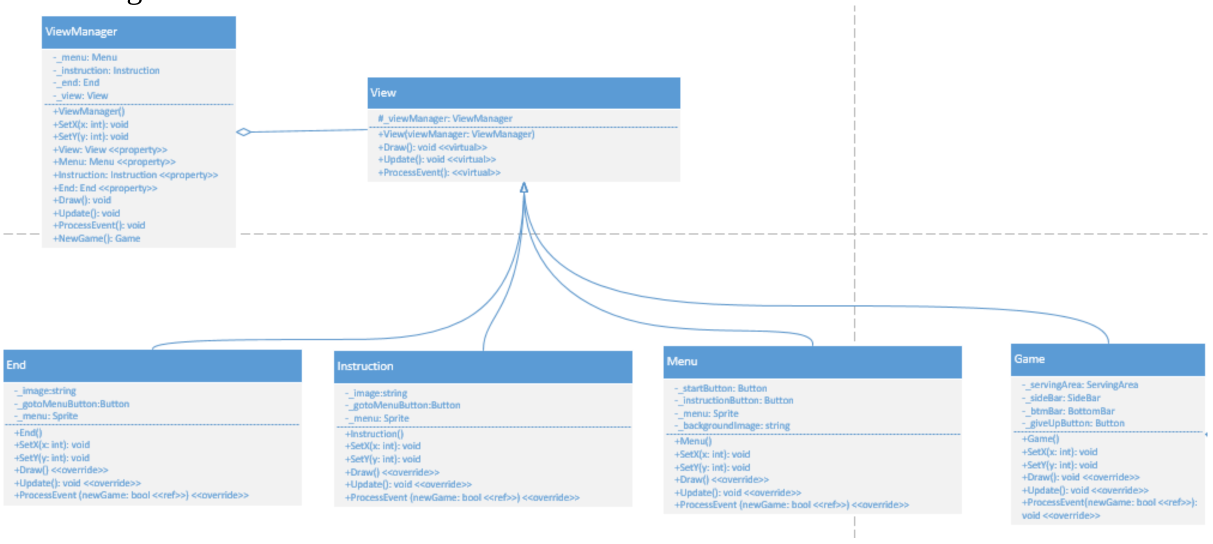
Process of Refactoring by using State Pattern:

First, I create a View class as an interface base class that contains the required virtual function for ViewManager class to delegate function to current concrete interface subclass. (Figure 8) Then I create four new classes (Menu, Instruction, Game, End) to act as concrete interface subclasses, inherit from View class and override the virtual function for distinct behaviour based on the subclass. (Figure 9, 10, 11, 12)

Next, each interface subclass will need to determine their successor interface and when to proceed with the transition. Thus, by changing the constructor of interface base class and interface subclasses to accept an object of ViewManager class as argument so the interface object could access ViewManager’s current interface and change it.

-After Refactoring:

Class Diagram



Code

```
    public class ViewManager
    {
        private Menu _menu;
        private Instruction _instruction;
        private End _end;
        private View _view;
```

Figure 5: ViewManager's field

```
        public ViewManager ()
        {
            SwinGame.LoadResourceBundle ("bundle.txt");
            _menu = new Menu (this);
            _instruction = new Instruction (this);
            _end = new End (this);
            _view = _menu;
        }
```

Figure 6: ViewManager's constructor

```
        public void Draw ()
        {
            _view.Draw ();
        }

        public void Update ()
        {
            _view.Update ();
        }

        public void ProcessEvent ()
        {
            _view.ProcessEvent ();
        }
```

Figure 7: ViewManager's Draw, Update, and ProcessEvent methods.

After refactoring by state pattern, the fields need to be declared and initialize in the ViewManager class reduce drastically as shown in figure 5 and 6. There is no need extra method (DetermineView method) for interface transition's determination in the ViewManager class. Moreover, the Draw, Update and ProcessEvent methods fulfill the principle of closed for modification.

Now, I could easily add new interfaces and transition by defining a new interface without the need to modify the ViewManager class. The logic that decides the interface transitions is divide between the interface subclasses. Furthermore, the interface of the ViewManager class currently managing will maintain integrity due to the interface transitions are atomic.

In this case, we had managed to create four interface subclasses which could be reused by another context. Whenever a context needs a game menu interface, a menu class could be reused. However, with the initial approach, reusability is very low as another interface has been bind together with the game menu interface inside the ViewManager class.

IV. DISCUSSION

In this case, I delegate the responsibility for deciding interface transition to the interface subclasses. Could ViewManager class be the one to decide the flow of interface transition? The answer depends on the scenario. When the transitions are more flexible, placing interface transition in the interface subclasses simplify the logic in ViewManager.

However, dependencies created between the interface subclasses and decrease modularity by this approach. In my implementation, I minimalize this problem by setting the interface transition in each interface subclass using public property in the ViewManager class instead of holding references to other interfaces. While placing interface transition in the ViewManager class is a possible solution. Huge conditional statement could branch out if the possibility of interface transition too many.

The classes in my design had increased by five as a result of refactoring. This could be one disadvantage of state pattern as the developer need to look through more classes. Nonetheless, increasing modularity by building additional classes will save time for future if new feature needs to be introduce as the possibility of refactor is reduced. Making interfaces explicit could also neutralize the effect needed to look through more classes as the code in interface subclasses will be easier to understand and maintain.

CONCLUSION

At this point, we can conclude that state pattern is a good solution in this scenario. By applying it, the software has fulfilled these object-oriented principles: encapsulate what varies, objects are loosely coupled, open for extension but closed for modification, and relied on abstraction. (Freeman et al. 2004) This produces high flexibility and modularity software.

REFERENCES

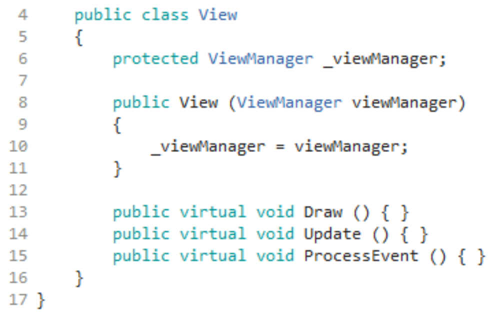
Fowler, M 1999, *Refactoring: Improving the Design of Existing Code (Object Technology Series)*, illustrated edition., Addison-Wesley Longman, Amsterdam.

Freeman, Elisabeth, Freeman, Eric, Bates, B & Sierra, K 2004, *Head First Design Patterns*, O' Reilly &amp; Associates, Inc.

Gamma, E, Helm, R, Johnson, R & Vlissides, J 1995, *Design Patterns: Elements of Reusable Object-oriented Software*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

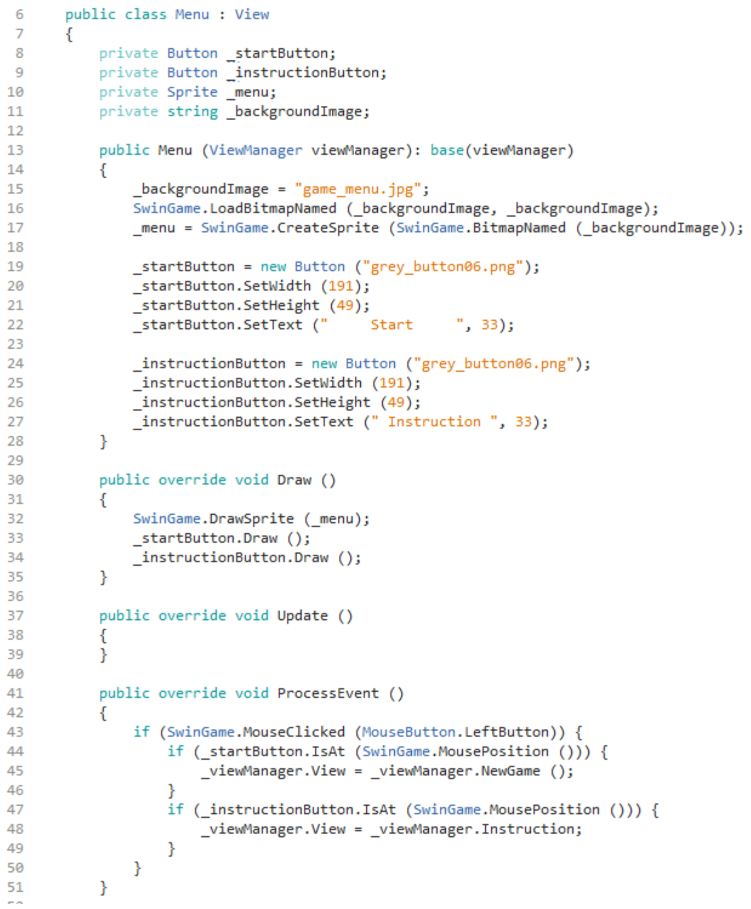
Martin, RC 1996, 'The open-closed principle', *More C++ gems*, vol. 19, no. 96, p. 9.

V. Appendix



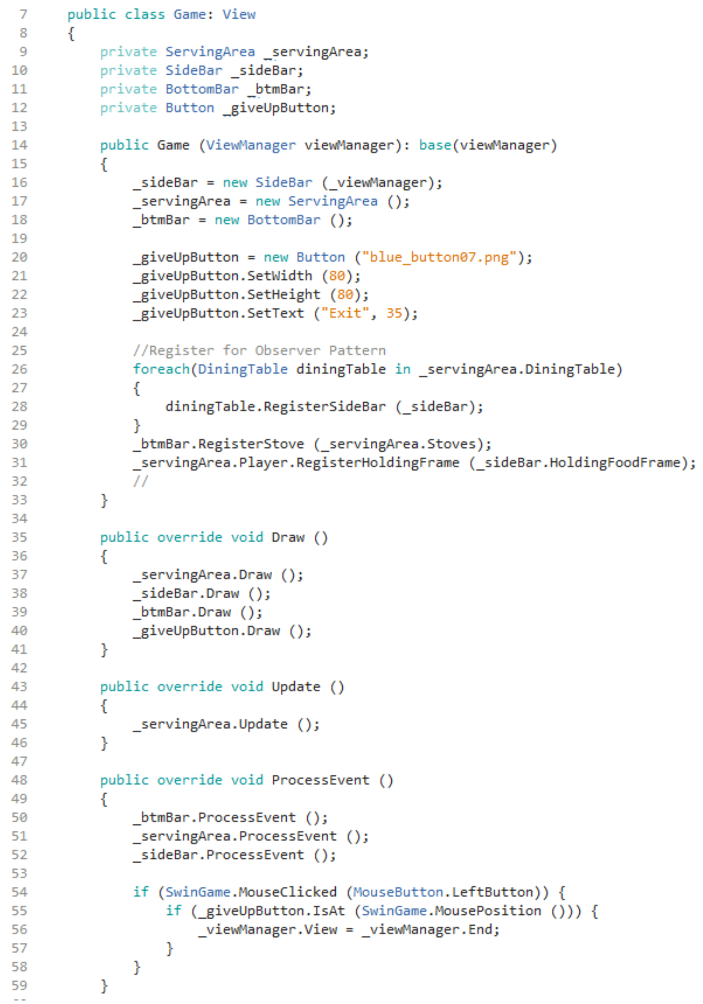
```
    public class View
    {
        protected ViewManager _viewManager;

        public View (ViewManager viewManager)
        {
            _viewManager = viewManager;
        }

        public virtual void Draw () { }
        public virtual void Update () { }
        public virtual void ProcessEvent () { }
    }
}
```

Figure 8: View Class



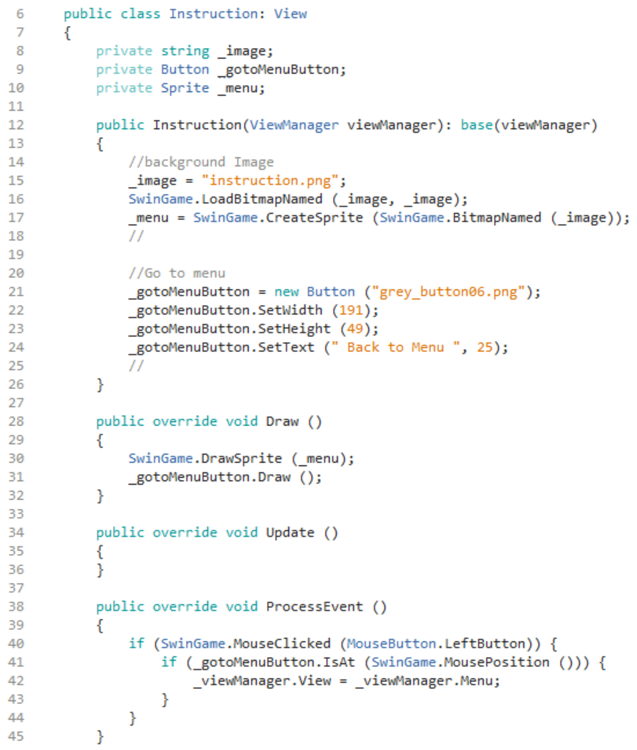
```
    public class Menu : View
    {
        private Button _startButton;
        private Button _instructionButton;
        private Sprite _menu;
        private string _backgroundImage;

        public Menu (ViewManager viewManager): base(viewManager)
        {
            _backgroundImage = "game_menu.jpg";
            SwinGame.LoadBitmapNamed (_backgroundImage, _backgroundImage);
            _menu = SwinGame.CreateSprite (SwinGame.BitmapNamed (_backgroundImage));

            _startButton = new Button ("grey_button06.png");
            _startButton.SetWidth (191);
            _startButton.SetHeight (49);
            _startButton.SetText ("     Start     ", 33);

            _instructionButton = new Button ("grey_button06.png");
            _instructionButton.SetWidth (191);
            _instructionButton.SetHeight (49);
            _instructionButton.SetText (" Instruction ", 33);
        }

        public override void Draw ()
        {
            SwinGame.DrawSprite (_menu);
            _startButton.Draw ();
            _instructionButton.Draw ();
        }

        public override void Update ()
        {
        }

        public override void ProcessEvent ()
        {
            if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
                if (_startButton.IsAt (SwinGame.MousePosition ())) {
                    _viewManager.View = _viewManager.NewGame ();
                }
                if (_instructionButton.IsAt (SwinGame.MousePosition ())) {
                    _viewManager.View = _viewManager.Instruction;
                }
            }
        }
```

Figure 9: Menu Class



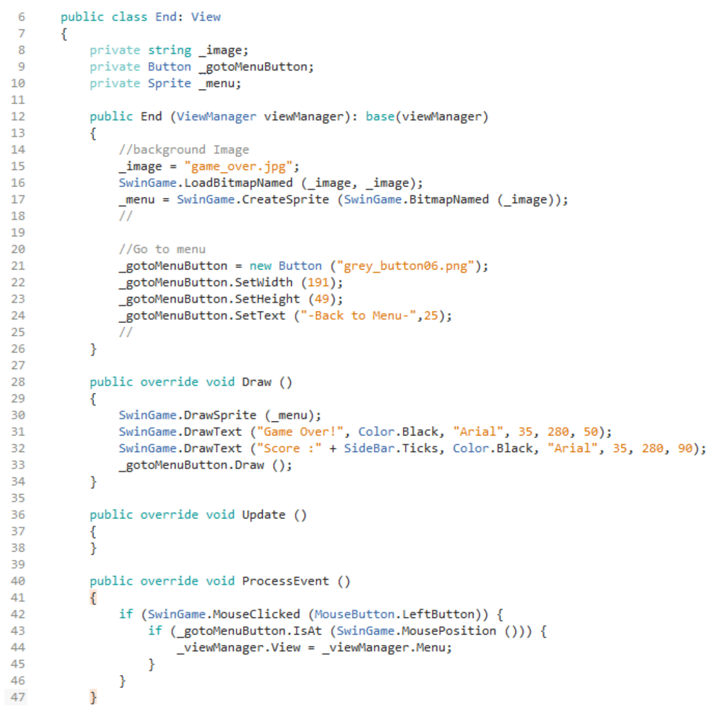
```csharp
public class Game: View
{
    private ServingArea _servingArea;
    private SideBar _sideBar;
    private BottomBar _btmBar;
    private Button _giveUpButton;

    public Game (ViewManager viewManager): base(viewManager)
    {
        _sideBar = new SideBar (_viewManager);
        _servingArea = new ServingArea ();
        _btmBar = new BottomBar ();

        _giveUpButton = new Button ("blue_button07.png");
        _giveUpButton.SetWidth (80);
        _giveUpButton.SetHeight (80);
        _giveUpButton.SetText ("Exit", 35);

        //Register for Observer Pattern
        foreach(DiningTable diningTable in _servingArea.DiningTable)
        {
            diningTable.RegisterSideBar (_sideBar);
        }
        _btmBar.RegisterStove (_servingArea.Stoves);
        _servingArea.Player.RegisterHoldingFrame (_sideBar.HoldingFoodFrame);
        //
    }

    public override void Draw ()
    {
        _servingArea.Draw ();
        _sideBar.Draw ();
        _btmBar.Draw ();
        _giveUpButton.Draw ();
    }

    public override void Update ()
    {
        _servingArea.Update ();
    }

    public override void ProcessEvent ()
    {
        _btmBar.ProcessEvent ();
        _servingArea.ProcessEvent ();
        _sideBar.ProcessEvent ();

        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_giveUpButton.IsAt (SwinGame.MousePosition ())) {
                _viewManager.View = _viewManager.End;
            }
        }
    }
```

Figure 10: Game Class



```csharp
public class Instruction: View
{
    private string _image;
    private Button _gotoMenuButton;
    private Sprite _menu;

    public Instruction(ViewManager viewManager): base(viewManager)
    {
        //background Image
        _image = "instruction.png";
        SwinGame.LoadBitmapNamed (_image, _image);
        _menu = SwinGame.CreateSprite (SwinGame.BitmapNamed (_image));
        //

        //Go to menu
        _gotoMenuButton = new Button ("grey_button06.png");
        _gotoMenuButton.SetWidth (191);
        _gotoMenuButton.SetHeight (49);
        _gotoMenuButton.SetText (" Back to Menu ", 25);
        //
    }

    public override void Draw ()
    {
        SwinGame.DrawSprite (_menu);
        _gotoMenuButton.Draw ();
    }

    public override void Update ()
    {
    }

    public override void ProcessEvent ()
    {
        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_gotoMenuButton.IsAt (SwinGame.MousePosition ())) {
                _viewManager.View = _viewManager.Menu;
            }
        }
    }
```

Figure 11: Instruction Class



```
public class End: View
{
    private string _image;
    private Button _gotoMenuButton;
    private Sprite _menu;

    public End (ViewManager viewManager): base(viewManager)
    {
        //background Image
        _image = "game_over.jpg";
        SwinGame.LoadBitmapNamed (_image, _image);
        _menu = SwinGame.CreateSprite (SwinGame.BitmapNamed (_image));
        //

        //Go to menu
        _gotoMenuButton = new Button ("grey_button06.png");
        _gotoMenuButton.SetWidth (191);
        _gotoMenuButton.SetHeight (49);
        _gotoMenuButton.SetText ("-Back to Menu-",25);
        //
    }

    public override void Draw ()
    {
        SwinGame.DrawSprite (_menu);
        SwinGame.DrawText ("Game Over!", Color.Black, "Arial", 35, 280, 50);
        SwinGame.DrawText ("Score :" + SideBar.Ticks, Color.Black, "Arial", 35, 280, 90);
        _gotoMenuButton.Draw ();
    }

    public override void Update ()
    {
    }

    public override void ProcessEvent ()
    {
        if (SwinGame.MouseClicked (MouseButton.LeftButton)) {
            if (_gotoMenuButton.IsAt (SwinGame.MousePosition ())) {
                _viewManager.View = _viewManager.Menu;
            }
        }
    }
```

Figure 12: End Class

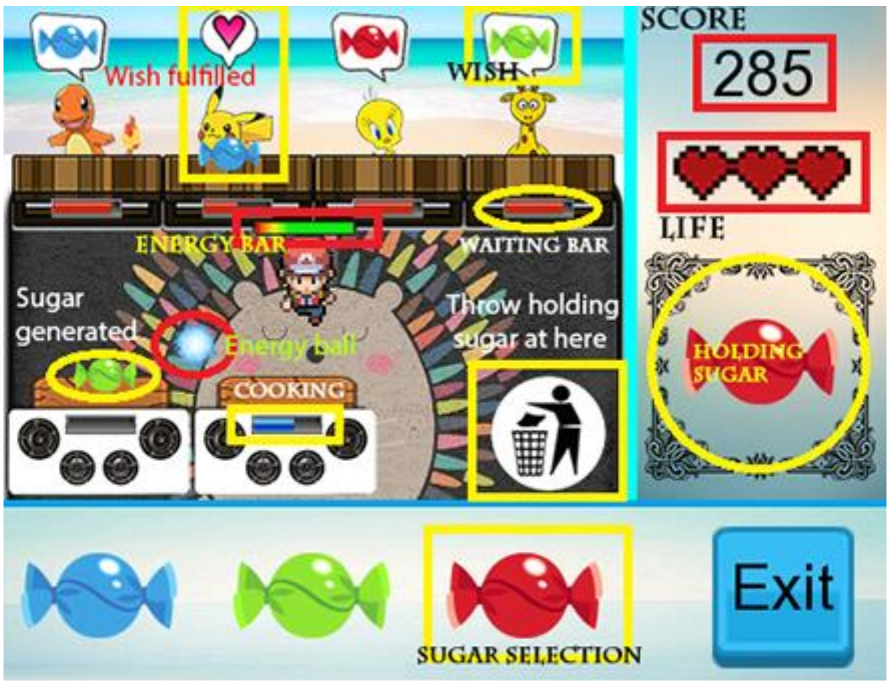


Figure 13: A Day in Candy Shop