Assignment 1 - Design Pattern

Application 1

Introduction

Definition

Strategy Design Pattern

- Strategy pattern deals with the change in class behavior at runtime. The objects consist of strategies and the context object judges the behavior at runtime of each strategy
- Place each messaging type into its own class to achieve single responsibility principle.
- These classes can easily interchangeable without modification.

Example

```
public class MessageService {
    public void send(String type, String msg){
        if(type.equals(anObject: "WeChat")){
            System.out.println(x: "Sending message using WeChat");
            // WechatClient client = getClient();
            // .........
            //   System.out.println("Failed ending message");
            // }
        } else if(type.equals(anObject: "WhatsApp")){
            System.out.println(x: "Sending message using WhatsApp");
            // Twilio.init(ACCOUNT_SID, AUTH_TOKEN);
            // .........
            //     .create();
        }
```

Change component when processing the items

Essay 1 - Source

Game Development Application

Algorithm Group Application

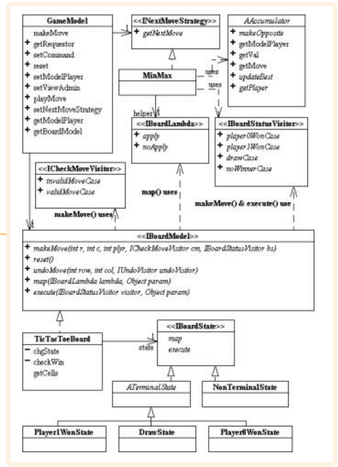
The Strategy pattern identifies an algorithm group, encapsulates each group, and allows switching between them. This Pattern helps the algorithm change at a rate different from the clients who require its operation. It is commonly used in sorting algorithms, in which one of the sorting types could be implemented according to the context of the situation. The GoF calls this behavior the Chain of Responsibility pattern, through which a chain of handlers processes a request, and a handler might process the request or pass it to the next one. This Pattern can be used in applications where several request handlers, such as web servers or middleware applications, must handle requests.

Essay 7

The Strategy pattern is originally and mostly used in game development, where various algorithms or behaviors must be swapped at runtime. The strategy pattern shows what algorithms should be used, encapsulates these types, and allows for changes. It lets developers alter an object's behavior by substituting the algorithm or strategy that the object employs instead of modifying it. For example, the strategy pattern is applied rather actively to control characters' behavior, make decisions in AI, and calculate physics in games created with Unity3D and Unreal Engine [28]. For instance, a a game character's behavior in an action game may differ based on state – idle, attack, or escape state. In the Strategy pattern, these behaviors may be defined in a set of strategies so that characters may easily change their behavior during the game. This approach does more than just slimming down the number of files. It also makes adding or changing behaviors much easier without impacting the game mechanics.

Simple Implementation

UML Design

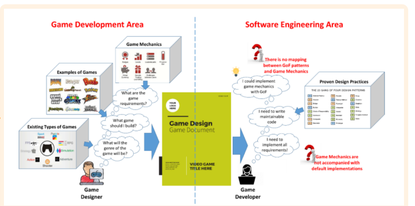


Coding Practice

```
public IRequestor getRequestor() {
    return new IRequestor() { //user makes move using requestor
        public void setTokenAt(final int row, final int col, final
                int player, final IRejectCommand rejectCommand) {
            boardModel.makeMove(row, col, player,
                new ICheckMoveVisitor() {
                    public void invalidMoveCase() {
                        rejectCommand.execute(); // Tell view
                    }
                    public void validMoveCase() { // Tell view
                        iCommand.setTokenAt(row, col, player);
                    }
                },
                new IBoardStatusVisitor() {
                    public Object player0WonCase(IBoardModel host,
                                                 Object param) {
                        viewAdmin.win(0);  return null; // Tell view
                    }
                    public Object player1WonCase(IBoardModel host,
                                                 Object param) {
                        viewAdmin.win(1);  return null;  // Tell view
                    }
                    public Object drawCase(IBoardModel host,
                                           Object param)  {
                        viewAdmin.draw();  return null;  // Tell view
                    }
                    public Object noWinnerCase(IBoardModel host,
                                               Object param)  {
                        makeMove(player);  return null;  //Computer's turn
                    }
                });
        }
    };
}
```

Complex system design

Background



Background of using strategy pattern in Game

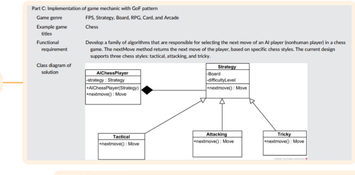
Problem Description

Task Description: Each player in BloodBowl can move inside the field, block his enemy, throw (i.e., pass) the ball, and attempt to steal the ball.

We ask you to implement a new possible move (namely: `jumpToBlock`) in which a player attempts to block the opponent from a further distance: not by standing in his way, but by jumping. The logic of the move is the same as block, but: (a) it can be executed when the defender is 2 or 3 blocks away from the attacker; and (b) has 50% and 33% chances of being successful and injuring the blocked player (stunned). Upon an unsuccessful `jumpToBlock` attempt, the blocker gets injured (KO). The starting class for this task is: `MoveActionState`. The correction guidelines (i.e., how we graded the correctness of the solutions) are presented below:

- The developer successfully spots the parts that need to be updated (`if` statements in G1, classes and methods in G2 and G3) (3 points)
- The developer successfully implements the `execute` functionality (3 points)
- The developer successfully implements the `isLegal` functionality (3 points)
- The developer successfully implements the `endsTeamTurn` functionality (1 points)

UML Diagram



Code Implementation

```
public class Main {
    public static void main(String[] args) {
    }
}
class Player {
    HashMap<String, Command> commands;
    boolean active;

    public Player() {
        commands = new HashMap();
        //add code here
    }
    public void executeCommand(String commandKey) {
        //add code here
    }
    //void changeTurn(): changes the turn
    public void changeTurn() {
        System.out.println("Turn changed");
    }
    //returns true if it's player's turn
    public boolean isPlaying() {
        return true;
    }
    //returns true if the amount is smaller than the highest player with cash
    public boolean canRaise(double amount) {
        return true;
    }
    //returns true if the amount is below the holding cash of the Player.
    public boolean cashInHand(double amount) {
        return true;
    }
    public void setActive(boolean b) { //sets the Players state in current round
        active = b;
    }
interface Command {}
```

Experimental Result

While this is an experiment, the result of comparing this patter with the other patterns. But no need to include this in my part.

Essay 3 - Simulated Application (2)

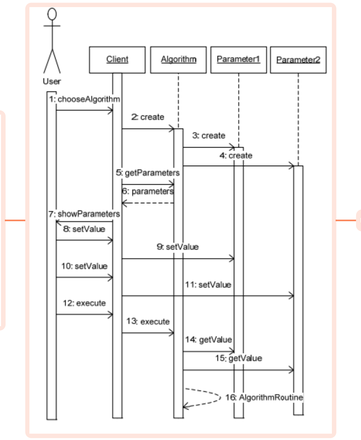
Application 2

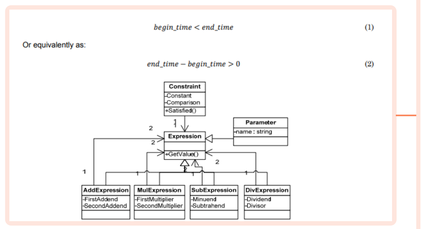
Essay 4 - Parameterized Strategy Pattern

An optimization of strategy pattern, which did the extension to the strategy pattern. With practical Examples

Sample

UML Diagram





Code Implementation

```
abstract class Algorithm
{
    public Algorithm()
    { }

    protected Parameter[] parameters;

    public Parameter[] getParameters()
    { return parameters.copy(); }

    public abstract void execute();
}

class GeneticAlgorithm : Algorithm
{
    public GeneticAlgorithm()
    {
        parameters = new Parameter[3];
        parameters[0] = new IntParameter("Popul size", 50, 1000, 100);
        parameters[1] = new IntParameter("Max iterations", 2, 10000, 20);
        parameters[2] = new DoubleParameter("Mutation Probability", 0.0, 0.9, 0.05);
    }

    public override void execute()
    {
        int populationsize  = ((IntParameter)parameters[0]).GetValue();
        int maxiterations   = ((IntParameter)parameters[1]).GetValue();
        double mutationprob = ((DoubleParameter)parameters[2]).GetValue();
        genetic_algorithm(populationsize, maxiterations, mutationprob);
    }

    private void genetic_algorithm(int populationsize, int maxiterations, double mutationprob)
    {
        //Genetic Algorithm itself
    }
}
```

Essay 5 - Implementation in mobile application

Design UML Diagram

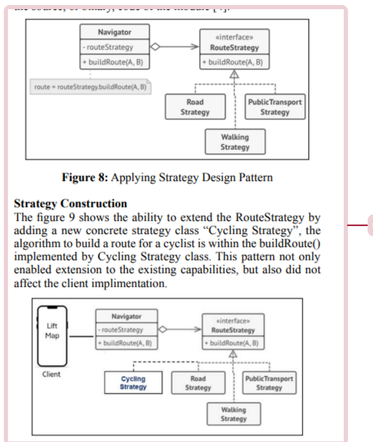


Figure 8: Applying Strategy Design Pattern

Strategy Construction

The figure 9 shows the ability to extend the RouteStrategy by adding a new concrete strategy class "Cycling Strategy", the algorithm to build a route for a cyclist is within the buildRoute() implemented by Cycling Strategy class. This pattern not only enabled extension to the existing capabilities, but also did not affect the client implementation.

Coding

```
public class Program
{
    public static void Main(string[] args)
    {
using StrategyPatternDemo;
StrategyPatternDemo.Navigator context;

            Console.WriteLine("Please select your travel strategy:");
            Console.WriteLine("A - RoadMap");
            Console.WriteLine("B - WalkMap");
            Console.WriteLine("C - PublicTransportMap");
            ConsoleKeyInfo cki;

            do {
                cki = Console.ReadKey(true);
                if (cki.Key == ConsoleKey.A)
                {
                    context = new Navigator(new RoadMap());
                }
                else if (cki.Key == ConsoleKey.B)
                {
                    context = new Navigator(new WalkMap());
                }
                else if (cki.Key == ConsoleKey.C)
                {
                context = new Navigator(new PublicTransportMap());
                }
```