

Assignment

by Hywel Kenealy Jones

FILE	15888_HYWEL_KENEALY_JONES_ASSIGNMENT_339836_1047896658.ODT (330.22K)		
TIME SUBMITTED	16-APR-2021 10:40PM (UTC+0100)	WORD COUNT	3290
SUBMISSION ID	149582856	CHARACTER COUNT	19451

Software Engineering Principles and Testing
ACCA5030

Hywel Jones 1903723
07/03/2021

Table of Contents

Software design.....	3
User Stories	3
CRC Cards	5
Testing	7
Software Testing Tables	8
Code Tests	9

Software design

One of the first important steps in producing software is the design phase. Often designers used to need a requirement or specification document before undertaking development of a product, this was often used to cement what was to be delivered. Unfortunately due to miscommunication or the inherent difficulty of writing down someone else idea as a list of bullet points this led to problems and often customers were not satisfied and developers had proceeded under false assumptions which could create technical debt.

A new method was needed to bridge this gap, here is where user stories come in. User stories allow the customer to express a process or action that the software should complete which simultaneously allows them to create a deliverable for developers and explains the expectation of how the requirement should work. These user stories can be expanded upon with diagrams and other discussions and decisions so that both sides are fully on board and content with how to proceed with the user story.

User tasks are used as a grouping mechanism for user stories. These allow user stories to be grouped and to maintain their granularity to describe the different facets of one single task that a user undertakes. For example for the user task of “checkout”, there might be several user stories such as “select delivery type”, “different payment option” and “donate to charity in checkout”. Additionally these user stories can be categorized using the MoSCoW methodology, which is a way of categorizing items according to their necessity for the end product. MoSCoW stands for Must have, Should have, Could have and Won’t have, using this sort of prioritization method helps the software team to deliver what the business deems critical first whilst helping the customer realize some of their requirements might need to be worked on later or in future releases.

These user stories can then be prioritized as some of these user stories might be required for a minimally viable product whilst the other user stories are nice to have functionalities that can be part of a future release. Using this approach along with an agile methodology, which is an iterative and dynamic approach to software development, this allows customers to have better control and allows developers some flexibility throughout the software development process. For example if the customer expectations for certain functionalities change or time factors become a bigger issue, user stories can be re-prioritized much easier than otherwise.

Some of the disadvantages of this approach is that there can be a lack of a concrete endpoint and some large or different user stories can be made a priority half way through the project that can disrupt or require different approaches to resolve had they been known at the start. Obviously if software design is to follow this methodology testing and having decoupled code is very important as the code base needs to be as modular as possible to deal with the potential for change.

User Stories

Here we start by creating very general user activities followed by creating our user tasks and breaking those down into user stories.



The user stories in release 1 should give us a minimal viable product, that is the user would be able to join a game, play moves and find out who won or lost. We then have release 2 below this which provides more nice to have features that improves the user experience significantly and would be more in line with what other online games feature these days. There are of course even more improvements than could be added, such as online matchmaking, player vs player scores and player profile images and text for example.

Now for the development team these user stories can be given story points, this is a method for developers to assign an arbitrary estimation of the difficulty of completing each user story. It is relatively easy at a glance to give vague estimations of difficulty, for example creating a help page to display the rules would be considered easy compared to implementing a real time chat feature where both players could chat to each other as they joined lobbies. But getting developers to agree on a time estimation is much more difficult and depends on the skill and experience of the developer making the estimation.

Story points can be combined with other methodologies such as sprint/scrum methodologies and with additional analysis and reflection can provide powerful management and time estimation tools. For example if the story point estimations and actuals are plotted for each team member then a coefficient can be determined if that team member usually is on average correct, underestimating or overestimating the difficulty of the user stories. This combined with calculating the sprint velocity, that is how many story points on average a team is able to complete in a given sprint, management can turn relatively abstract guesses into somewhat good guesstimates that can be used to manage the project.

CRC Cards

CRC cards stand for class-responsibility-collaboration card, these cards are often used at the start of the design phase for developers to figure out what should be a class, what their responsibilities should be and how they relate to other classes. Undertaking this step early on can help stop some common mistakes such as having a superclass, that is one class that takes on almost all responsibilities for the application, or duplicated responsibilities across several disjoint classes. Mainly it helps developers think about the practical problem in an object oriented way and helps minimize the need for refactoring early on in the project due to lack of planning.

Here our Game class is the entry point for the game/application and serves as the initialization point for our player and board classes.

Game.cs

Class: Game			
Responsibilities		Collaborators	
Get Two players		Player	
Create empty board		Board	
Declare Winner			

Player.cs

Class: Player			
Responsibilities		Collaborators	
Make moves		Board	
User information			

Board.cs

Class: Board			
Responsibilities		Collaborators	
Record Moves		WinConditions	
Check rules		Rules	
Check Win Conditions			

Due to the additional rules involved in this variant of the game some of the win conditions can become rather complicated and might need to be checked several times a turn if other counters are captured or deleted. This leads us to separate the win conditions from the implementation of the rules, that is how the game calculates if counters should be removed or captured etc.

WinConditions.cs

Class: WinConditions			
Responsibilities		Collaborators	
Check for win conditions • Horizontal • Vertical • Diagonally			

Rules.cs

Class: Rules			
Responsibilities		Collaborators	
Capture counter rule			
Destroy counter rule			

During development it was decided that the Rules class should orchestrate the running all of the individual rules classes. So in addition to the above CRC classes there is also a TrapHorizontal, TrapVertical, TrapDiagonal, DestroyHorizontal, DestroyVertical and DestroyDiagonal classes. These classes implement the IC4Rules interface that has only one method RunRule(). This is so that all these rules can be added as classes to the Rules class and then run through the interface method call. The WinConditions class would be similarly broken down to three win condition classes to verify that all the vertical, horizontal and diagonal win conditions are also met.

TC "TC "

Testing

As noted previously testing is very important in agile software development to ensure a viable product is produced without incurring large refactoring or technical debt. Technical debt is where code that should be refactored or improved is left as is, or bugs that aren't critical to fix or are intermittent aren't fixed immediately at the expense of project progress but rather ignored for more project progress. As more and more time passes this technical debt tends to increase and the complexity of resolving the problem also tends to increase as additional code calls or makes use of the affected code. Therefore it is beneficial in the long term for teams to spend more time mitigating potential issues upfront rather than accumulating debt that needs to be repaid at possibly exponential interest in the future.

Testing itself can be split into to two main categories, which is white box testing and black box testing. White box testing is when the tester knows about the inner workings of the code, this might be thought of as writing a unit test to systematically test every path through a piece of code. Black box testing is the opposite, where the tester doesn't know the inner workings of the code, so each test is mainly focused on if expected input gives the expected output without necessarily working through each path in the code.

Whitebox testing can be more time consuming to complete if the code has many logical paths through and requires extensive testing and knowledge of the code to reach 100% test coverage but it can logically prove a function is correct. Blackbox testing can be more fuzzy in that it doesn't necessarily prove a function is correct and bugs might still exist, but it can very quickly prove that a function provides expected outcomes for some limited range of values that we can expect as input.

Software Testing Tables

Here we create a table of tests where we test to make sure that the application is working correctly.

Test Name	Preconditions	Inputs	Outputs	Description
BoardInitialized	None	Board class	Board initialized with non null board array.	Check Board initialization is correct.
BoardInitializedEmptyBoard	None	Board class	Board array initialized to all empty values.	Check that Board array is actually full of empty values first.
PlayerInitialization	None	Player class String name Move	Initialized player object with a string value name set.	We need to initialize players and test that the name is set as part of the initialization.
GameInitialization	None	Player classes with different move counters and configuration object	Game initialized	Test initialization
GameDifferentCountersThrowsException	None	Player classes with same move counters and configuration object	Exception thrown	Two players must have different counters, exception should be thrown.
RuleTrapHorizontalPositive	Prepopulated board	Board with rule just before execution	Board changed as per rule	Test that rule works as expected on a prepopulated board.
RuleTrapHorizontalNegative	Prepopulated board	Board where rule shouldn't be executed	Board not changed	Test that rule doesn't execute when condition hasn't merited it.
RuleTrapVerticalPositive	Prepopulated board	Board with rule just before execution	Board changed as per rule	Test that rule works as expected on a prepopulated board.
RuleTrapVerticalNegative	Prepopulated board	Board where rule shouldn't be executed	Board not changed	Test that rule doesn't execute when condition hasn't merited it.
RuleTrapDiagonalPositive	Prepopulated board	Board with rule just before execution	Board changed as per rule	Test that rule works as expected on a prepopulated board.
RuleTrapDiagonalNegative	Prepopulated board	Board where rule shouldn't be executed	Board not changed	Test that rule doesn't execute when condition hasn't merited it.
RuleDestroyVerticalPositive	Prepopulated board	Board with rule just before execution	Board changed as per rule	Test that rule works as expected on a prepopulated board.

RuleDestroyVerticalPrepopulated board Negative	Board where rule shouldn't be executed	Board not changed	Test that rule doesn't execute when condition hasn't merited it.
RuleDestroyHorizontalPositive	Board with rule just before execution	Board changed as per rule	Test that rule works as expected on a prepopulated board.
RuleDestroyHorizontalNegative	Board where rule shouldn't be executed	Board not changed	Test that rule doesn't execute when condition hasn't merited it.
RuleDestroyDiagonalPositive	Board with rule just before execution	Board changed as per rule	Test that rule works as expected on a prepopulated board.
RuleDestroyDiagonalNegative	Board where rule shouldn't be executed	Board not changed	Test that rule doesn't execute when condition hasn't merited it.
WinConditionHorizontalPositive	Board with win condition just before execution	Win declared	Test that a win is declared when it occurs.
WinConditionHorizontalNegative	Board where win shouldn't be declared	Win not declared	Test win not declared when no win condition present.
WinConditionVerticalPositive	Board with win condition just before execution	Win declared	Test that a win is declared when it occurs.
WinConditionVerticalNegative	Board where win shouldn't be declared	Win not declared	Test win not declared when no win condition present.
WinConditionDiagonalPositive	Board with win condition just before execution	Win declared	Test that a win is declared when it occurs.
WinConditionDiagonalNegative	Board where win shouldn't be declared	Win not declared	Test win not declared when no win condition present.

Code Tests

UnitTests1.cs

```
using System;
using Connect4.Classes;
using Connect4.Classes.RunRules;
using Microsoft.VisualStudio.TestTools.UnitTesting;
```

```

using static Connect4.Classes.Board;

namespace Connect4UnitTests
{
    [TestClass]
    public class UnitTest1
    {
        [TestCategory("Board")]
        [TestMethod]
        public void BoardInitialized()
        {
            var board = new Board(new C4Configuration());

            Assert.IsNotNull(board);
            Assert.IsNotNull(board.BoardArray);
        }

        [TestCategory("Board")]
        [TestMethod]
        public void BoardInitializedEmptyBoard()
        {
            var board = new Board(new C4Configuration());

            bool isEmpty = true;

            for (int i = 0; i < board.BoardArray.GetLength(0); i++)
            {
                for (int j = 0; j < board.BoardArray.GetLength(1); j++)
                {
                    if (board.BoardArray[i, j] != Board.Move.Empty)
                    {
                        isEmpty = false;
                    }
                }
            }

            Assert.IsTrue(isEmpty);
        }

        [TestCategory("Player")]
        [TestMethod]
        public void PlayerInitialization()
        {
            var player = new Player("Dave", Board.Move.X);

            Assert.IsNotNull(player);
            Assert.AreEqual("Dave", player.Name);
            Assert.AreEqual(Board.Move.X, player.CounterType);
        }
    }
}

```

```

    }

    [TestCategory("Game")]
    [TestMethod]
    public void GameInitialization()
    {
        var game = new Game(new Player("Dave",Board.Move.O),new Player("Bob",
Board.Move.X),new C4Configuration());

        Assert.IsNotNull(game);
    }

    [TestCategory("Game")]
    [ExpectedException(typeof(ApplicationException))]
    [TestMethod]
    //White box testing example.
    public void GameDifferentCountersThrowsException()
    {
        var game = new Game(new Player("Dave", Board.Move.X), new Player("Bob",
Board.Move.X), new C4Configuration());
    }

    [TestCategory("Rules")]
    [TestMethod]
    //Black box testing example.
    public void RuleTrapHorizontal()
    {
        var config = new C4Configuration();
        var board = new Move[config.x, config.y];

        for (int i = 0; i < board.GetLength(0); i++)
        {
            for (int j = 0; j < board.GetLength(1); j++)
            {
                board[i, j] = Move.Empty;
            }
        }

        board[0, 6] = Move.O;
        board[1, 6] = Move.X;
        board[2, 6] = Move.O;

        var trapHorizontalRule = new TrapHorizontal();
        trapHorizontalRule.RunRule(board,3);

        Assert.AreEqual(Move.O,board[1,6]);
    }

    [TestCategory("Rules")]

```

```

[TestMethod]
//Black box testing example.
public void RuleDestroyVertical()
{
    var config = new C4Configuration();
    var board = new Move[config.x, config.y];

    for (int i = 0; i < board.GetLength(0); i++)
    {
        for (int j = 0; j < board.GetLength(1); j++)
        {
            board[i, j] = Move.Empty;
        }
    }

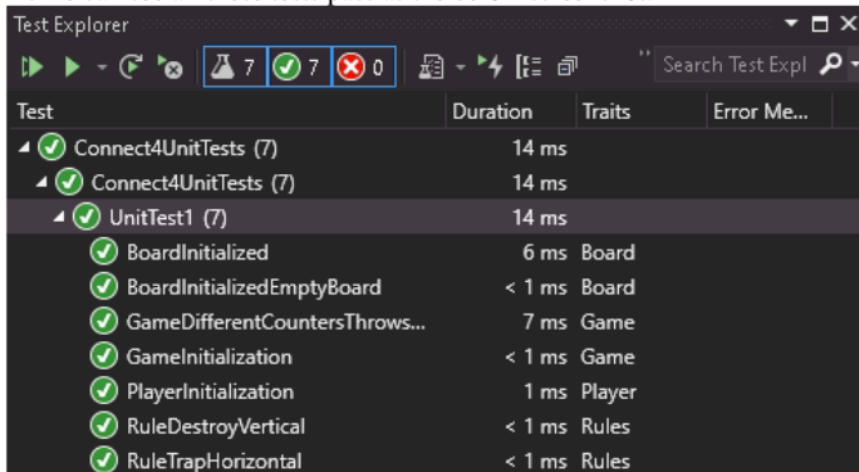
    board[0, 6] = Move.O;
    board[0, 5] = Move.O;
    board[0, 4] = Move.O;
    board[0, 3] = Move.X;

    var destroyVerticalRule = new DestroyVertical();
    destroyVerticalRule.RunRule(board, 1);

    Assert.AreEqual(Move.X, board[0, 6]);
}
}
}

```

As we can see all these tests pass as the below screenshot:



Program.cs

```
using Connect4.Classes;
```

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Connect4
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Enter name for player1:");
            var player1Name = Console.ReadLine();

            Console.WriteLine("Enter name for player2:");
            var player2Name = Console.ReadLine();

            var player1 = new Player(player1Name, Board.Move.X);
            var player2 = new Player(player2Name, Board.Move.O);

            var game = new Game(player1, player2, new C4Configuration());

            game.GameWelcome();
            while(!game.IsGameWon())
            {
                game.PlayGame();
            }

            Console.WriteLine("Thanks for playing!");
        }
    }
}

```

Game.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Connect4.Classes
{
    public class Game
    {
        public Game(Player player1, Player player2, C4Configuration configuration)
        {
            if(player1.CounterType == player2.CounterType)

```

```

        {
            throw new ApplicationException("Counters of both players cannot be the same!");
        }

        GameBoard = new Board(configuration);
        Player1 = player1;
        Player2 = player2;
        RuleSet = new Rules();
    }

    private Board GameBoard { get; set; }

    private Player Player1 { get; set; }
    private Player Player2 { get; set; }

    private Rules RuleSet { get; set; }

    public bool IsGameWon()
    {
        //Run win rules
        return false;
    }

    public void GameWelcome()
    {
        Console.WriteLine("Welcome to the game both players. When prompted " +
            "please type in the integer column number you wish to drop your counter in.");
        GameBoard.DrawBoard();
    }

    public void PlayGame()
    {
        if (!IsGameWon())
        {
            var lastMoveCol = Player1.MakeMove(GameBoard);

            RuleSet.RunRuleSet(GameBoard, lastMoveCol);

            GameBoard.DrawBoard();
        }
        else
        {
            //Win condition
            //Declare winner
        }

        if (!IsGameWon())
        {
            var lastMoveCol = Player2.MakeMove(GameBoard);

```

```

        RuleSet.RunRuleSet(GameBoard, lastMoveCol);

        GameBoard.DrawBoard();
    }
    else
    {
        //Win condition
        //Declare winner
    }
}
}
}
}

```

Board.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Connect4.Classes
{
    public class Board
    {
        public Move[,] BoardArray { get; set; }

        public Board(C4Configuration configuration)
        {
            BoardArray = new Move[configuration.x, configuration.y];
        }

        public void MakeMove(int column, Move counter)
        {
            int innerCol = column - 1;

            int y = 0;

            for(int i = BoardArray.GetLength(0); i >=0 ; i--)
            {
                if (BoardArray[innerCol, i] == Move.Empty)
                {
                    y = i;
                    break;
                }
            }
        }
    }
}

```

```

        BoardArray[innerCol.y] = counter;
    }

    public void DrawBoard()
    {
        Console.WriteLine("-----");
        for (int i = 0; i < BoardArray.GetLength(1); i++)
        {
            if(i==0)
            {
                Console.Write(" ");
                for (int j = 0; j < BoardArray.GetLength(0); j++)
                {
                    Console.Write("_" + (j+1) + " ");
                }
                Console.WriteLine();
            }

            for (int j = 0; j < BoardArray.GetLength(0); j++)
            {
                if (j == 0)
                {
                    Console.Write(i+1 + " | ");
                }

                if (BoardArray[j, i] == Move.Empty)
                {
                    Console.Write(" _ ");
                }
                else
                {
                    Console.Write(" " + BoardArray[j, i].ToString() + " ");
                }
            }
            Console.WriteLine();
        }
        Console.WriteLine("-----");
    }

    public enum Move
    {
        Empty = 0,
        X = 1,
        O = 2
    }
}

```

Player.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static Connect4.Classes.Board;

namespace Connect4.Classes
{
    public class Player
    {
        public Player(string name, Move counterType)
        {
            Name = name;
            CounterType = counterType;
        }

        public Board Board { get; set; }
        public string Name { get; set; }
        public Move CounterType { get; set; }

        public int MakeMove(Board board)
        {
            Console.WriteLine(Name + " please input the column number for your next token: ");
            var input = Console.ReadLine();
            var x = int.Parse(input);

            board.MakeMove(x, CounterType);

            return x;
        }
    }
}

```

Rules.cs

```

using Connect4.Classes.RunRules;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Connect4.Classes
{
    public class Rules
    {
        private List<IC4Rule> RuleList { get; set; }
    }
}

```

```

public Rules()
{
    RuleList = new List<IC4Rule>();

    //Trap counter rule block
    RuleList.Add(new TrapHorizontal());
    //RuleList.Add(new TrapVertical());
    //RuleList.Add(new TrapDiagonal());

    //Destroy counter rule block
    //RuleList.Add(new DestroyHorizontal());
    RuleList.Add(new DestroyVertical());
    //RuleList.Add(new DestroyDiagonal());
}

public void RunRuleSet(Board board, int lastMoveCol)
{
    RuleList.ForEach(rule =>
    {
        if (rule.RunRule(board.BoardArray, lastMoveCol))
            return;
    });
}
}
}

```

IC4Rules.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using static Connect4.Classes.Board;

namespace Connect4.Classes
{
    public interface IC4Rule
    {
        bool RunRule(Move[,] board, int lastMoveCol);
    }
}

```

TrapHorizontal.cs

```

using System;
using System.Collections.Generic;
using System.Linq;

```

```

using System.Text;
using System.Threading.Tasks;
using static Connect4.Classes.Board;

namespace Connect4.Classes.RunRules
{
    public class TrapHorizontal : IC4Rule
    {
        public bool RunRule(Move[,] board, int lastMoveCol)
        {
            bool IsRuleHasRun = false;
            int x = lastMoveCol-1;
            int y = 0;
            int xMax = board.GetLength(0)-1;
            int yMax = board.GetLength(1);

            for (int i = 0; i < board.GetLength(1); i++)
            {
                if (board[x, i] != Board.Move.Empty)
                {
                    y = i;
                    break;
                }
            }

            var thisMoveCounter = board[x, y];

            bool IsCheckLeft = true;
            bool IsCheckRight = true;
            //If too close left don't check left rule
            if (x-2<0)
            {
                IsCheckLeft = false;
            }
            //If too far right don't check right rule
            if (x + 2 > xMax)
            {
                IsCheckRight = false;
            }

            if(IsCheckLeft)
            {
                if (board[x - 2, y] == thisMoveCounter && board[x - 1, y] != thisMoveCounter && board[x
- 1, y] != Board.Move.Empty)
                {
                    //We have a left trap.
                    //Convert counter.
                    board[x - 1, y] = thisMoveCounter;
                    IsRuleHasRun = true;
                }
            }
        }
    }
}

```

```

    }
}

if(IsCheckRight)
{
    if (board[x + 2, y] == thisMoveCounter && board[x + 1, y] != thisMoveCounter && board[x
+ 1, y] != Board.Move.Empty)
    {
        //We have a right trap.
        //Convert counter.
        board[x + 1, y] = thisMoveCounter;
        IsRuleHasRun = true;
    }
}

return IsRuleHasRun;
}
}
}

```

DestroyVertical.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Connect4.Classes.RunRules
{
    public class DestroyVertical : IC4Rule
    {
        public bool RunRule(Board.Move[,] board, int lastMoveCol)
        {
            bool IsHasRuleRun = false;
            int x = lastMoveCol-1;
            int y = 0;
            int maxY = board.GetLength(1);

            for (int i = 0; i < board.GetLength(1); i++)
            {
                if (board[x, i] != Board.Move.Empty)
                {
                    y = i;
                    break;
                }
            }

            //If there is a possibility for 3 counters below

```

```

        if(y+3 < maxY)
        {
            var thisCounterType = board[x, y];
            if(board[x, y+1] != thisCounterType && board[x, y + 2] != thisCounterType && board[x, y +
3] != thisCounterType)
            {
                IsHasRuleRun = true;
                board[x, y] = Board.Move.Empty;
                board[x, y+1] = Board.Move.Empty;
                board[x, y+2] = Board.Move.Empty;
                board[x, y + 3] = thisCounterType;

            }
        }

        return IsHasRuleRun;
    }
}

```

WinConditions.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Connect4.Classes
{
    public class WinConditions
    {
        {
            public static bool IsGameWon(Board board, int lastMoveCol)
            {
                //check win conditions

                return false;
            }
        }
    }
}

```

The solution has been zipped and added as a file submission.

Assignment

GRADEMARK REPORT

FINAL GRADE

70/100

GENERAL COMMENTS

Instructor

Eval design (10): 6

User Stories (20): 13

CRC (20): 14

SW Testing (30): 22

Coding (20): 15

Feedback:

=====

The evaluation of the design approach is limited, I wanted a little more detail on the benefits and drawbacks of the user stories and story mapping approach.

For the CRC cards a more detail should be added to the responsibilities.

The evaluation of SW testing was a bit brief but there are a number of tests identified.

Code provides a significant amount of the features required by the assignment.

PAGE 2

PAGE 3

PAGE 4



Comment 1

Do not forget the "...so that...." element of the user story, this is very important because this reflects the benefit of the user story without which we have no justification for implementing it.

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21
