Technical Solution

Requirements Overview

Technical Skill	Requirement contributed to	Purpose	Methods used in

Requirements omitted below are those likely unable to be displayed via code, more suited to being displayed in the interface section of design (e.g. 3), or that are implied during testing (e.g. 12).

Requirement 1

Requirement	A mechanic which limits the player from making careless movements within a dungeon.
Explanation/Purpose	The purpose of the light mechanic is to prevent careless movements throughout a dungeon, and to make players think about where they are going. This is aiming to add some sense of pressure to being in the dungeon, and the idea that the player should not be comfortable while in there, and should be trying to get out.
Techniques used	2d Arrays

```
if (Light == 1)
{
    Utils.ShowForm(this.dungeon.game.MainEstateMap, false);
    this.dungeon.DungeonForm.Invalidate();
    this.dungeon.DungeonForm.Hide();
    if (!this.dungeon.IsCustom)
    {
        this.dungeon.game.MainEstateMap.DisableDungeonButton(this.dungeon.ID);
        this.dungeon.Attempted = true;
    }
    this.dungeon.MapForm.Hide();
}
else
{
    for (int i = 0; i < pbArr.GetLength(0); i++)
    {
        for (int j = 0; j < pbArr.GetLength(1); j++)
```

```
        {
            pbArr[i, j].Enabled = false;
        }
    }

    Utils.RemoveTempControls(this.dungeon.DungeonForm);
    Coordinate currentCoord = new Coordinate(x, y);
    this.dungeon.CurrentPosition = currentCoord;
    this.dungeon.ActiveRoom = this.dungeon.RoomMap[x, y];

    this.Invalidate();
    this.Hide();
    this.dungeon.RoomMap[x, y].Initialise();
    UpdateVisibility(currentCoord);

    List<Coordinate> adjacentPoints = this.dungeon.MapAdjMatrix[currentCoord];

    for (int i = 0; i < pbArr.GetLength(0); i++)
    {
        for (int j = 0; j < pbArr.GetLength(1); j++)
        {
            Coordinate buttonCoord = new Coordinate(i, j);

            if (!adjacentPoints.Any(adj => adj.X == buttonCoord.X && adj.Y == buttonCoord.Y))
            {
                pbArr[i, j].Enabled = false;
            }
            else
            {
                pbArr[i, j].Enabled = true;
            }
        }
    }
    pbArr[x, y].Enabled = false;
    Light--;
    UpdateLightLabel();
}
```

Requirement 2

Requirement	Navigation of a dungeon via a suitably displayed user interface. With clearly signified contents of rooms.
Explanation/Purpose	Relationship between dungeon room icons & room contents on the mapform shows the user the contents of rooms Rooms the player has not viewed (been adjacent to) have their contents hidden. A unique icon which overrides the room's content icon is displayed over the room the player is currently in, in order to prevent player confusion.
Techniques used	- 2d array

```
private void UpdateVisibility(Dungeon.Coordinate coordinate)
{
    int x = coordinate.X;
    int y = coordinate.Y;

    this.dungeon.RoomMap[x,y].IsExplored = true;
    this.dungeon.RoomMap[x,y].UpdateVisiblity();
    foreach (Dungeon.Coordinate coord in this.dungeon.MapAdjMatrix[new Dungeon.Coordinate(x, y)])
    {
        this.dungeon.RoomMap[coord.X, coord.Y].IsExplored = true;
        this.dungeon.RoomMap[coord.X, coord.Y].UpdateVisiblity();
    }
    this.dungeon.UpdateMap(pbArr);

}

for (int y = 0; y < 5; y++)
{
    for (int x = 0; x < 5; x++)
    {
        if (RoomMap[y, x].IsVisible)
        {
            RoomContent content = RoomMap[y, x].RoomType;
            switch (content)
            {
                case RoomContent.nulled:
```

```
                    pbArr[y, x].Hide();
                    break;
                case RoomContent.Empty:
                    pbArr[y, x].BackgroundImage = Image.FromFile("roomicons/empty.png");
                    break;
                case RoomContent.EasyEnemy:
                    pbArr[y, x].BackgroundImage = Image.FromFile("roomicons/Easy.png");
                    break;
                case RoomContent.HardEnemy:
                    pbArr[y, x].BackgroundImage = Image.FromFile("roomicons/Hard.png");
                    break;
                case RoomContent.Reward:
                    pbArr[y, x].BackgroundImage = Image.FromFile("roomicons/Reward.png");
                    break;
                case RoomContent.Exit:
                    pbArr[y, x].BackgroundImage = Image.FromFile("roomicons/Exit.png");
                    break;
            }
        }
        else
        {
            if (RoomMap[y, x].RoomType == RoomContent.nulled)
            {
                this.MapForm.pbArr[y, x].Hide();
            }
            else
            {
                pbArr[y, x].BackgroundImage = Image.FromFile("roomicons/Locked.png");
            }
        }
    }
}
if (CurrentPosition != null)
{
    int newx = this.CurrentPosition.X;
    int newy = this.CurrentPosition.Y;

    pbArr[newx, newy].BackgroundImage = Image.FromFile("roomicons/CurrentRoom.png");
}
```

Requirement 4

Requirement	A hamlet equivalent, a form of “base” for the player to consolidate and setup their party for the next dungeon. - The hamlet must contain different facilities (buildings) for the player to manage a variety of mechanics: - Purchasing new heroes - Treat conditions & Restore health. - Upgrade heroes - Create custom dungeons.
Explanation/Purpose	Wagon – Allows the user to swap out members of their current party with their other owned characters Stagecoach – Allows the user to purchase new characters, they must first source the character for a small scaling fee, and then purchase the characters themselves. The player can source characters repeatedly until they find one they want to buy. Hospital – Allows the user to heal and remove any negative status effects from their characters. Barracks – Allows the user to upgrade a character’s stats, such as health and armour. This allows for players to ensure their preferred characters with specific abilities can be scaled along with their level, and do not fall into a level of weakness that makes them redundant.
Techniques used	

```
public partial class HamletForm : Form
    {
        Game game;

        public BarracksForm BlacksmithForm { get; set; }
        public HospitalForm HospitalForm { get; set; }
        public StagecoachForm StagecoachForm { get; set; }
        public WagonForm WagonForm { get; set; }

        public HamletForm(Game game)
        {
            InitializeComponent();
```

```
            this.BlacksmithForm = new BarracksForm(game);
            this.HospitalForm = new HospitalForm(game);
            this.StagecoachForm = new StagecoachForm(game);
            this.WagonForm = new WagonForm(game);
            this.game = game;
        }


        private void BTN_hamlet_stagecoach_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(this.StagecoachForm);
        }

        private void BTN_hamlet_hospital_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(this.HospitalForm);
        }

        private void BTN_hamlet_blacksmith_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(this.BlacksmithForm);
        }

        private void BTN_hamlet_tradermarket_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(this.WagonForm);
        }

        private void btn_hamlet_ready_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(this.game.MainEstateMap, false);
            this.Hide();
        }

        private void HamletForm_Load(object sender, EventArgs e)
        {

        }

        private void CustomDungeon_BTN(object sender, EventArgs e)
```

```
        {
            Utils.ShowForm(new CustomDungeonSelect(game));
        }

        private void BTN_SaveGame_Click(object sender, EventArgs e)
        {
            if (this.game.SavePath == null)
            {
                Utils.ShowForm(new SaveFileNameInput(game, null));
            }
            else
            {
                string path = this.game.SavePath;
                using (FileStream fs = new FileStream(path, FileMode.Create))
                {
                    using (StreamWriter sw = new StreamWriter(fs))
                    {
                        sw.WriteLine(this.game.Save());
                    }
                }
            }
        }
    }

public partial class HospitalForm : Form
    {
        private Game game;
        private int Cost;
        public HospitalForm(Game game)
        {
            InitializeComponent();
            this.game = game;
        }

        private void HospitalForm_Load(object sender, EventArgs e)
        {
            HealButton.Enabled = false;

            CurrentGoldLBL.Text = "Current gold: " + Convert.ToString(game.Gold);
```

```
            for (int i = 0; i < this.game.Party.Length; i++)
            {
                Character hero = this.game.Party[i];
                if (hero.Health != hero.MaxHealth)
                {
                    SelectHero_CMB.Items.Add(hero);
                }
            }

            SelectHero_CMB.ValueMember = "Character";
            SelectHero_CMB.DisplayMember = "FullTitle";

            this.Controls.Add(SelectHero_CMB);

            SelectHero_CMB.SelectedItem = null;
        }

        private void HeroCMBBox_SelectChange(object sender, EventArgs e)
        {
            if (SelectHero_CMB.SelectedItem != null)
            {
                Character c = (Character)SelectHero_CMB.SelectedItem;
                HealButton.Text = "Heal";
                HealButton.Enabled = true;

                this.Cost = (int)(150 + ((c.MaxHealth - c.Health) * 0.5) + (c.Level * 50));

                HealButton.Text = $"Heal ({this.Cost} gold)";

                SelectedHeroHealthLBL.Text = $"Health: {c.Health}/{c.MaxHealth}";
                SelectedHeroHealthLBL.Location = new Point((SelectHero_CMB.Location.X) + (SelectHero_CMB.Width / 2) -
(SelectedHeroHealthLBL.Width / 2), SelectedHeroHealthLBL.Location.Y);
                SelectedHeroHealthLBL.Refresh();
            }
        }

        private void OnClose(object sender, FormClosedEventArgs e)
        {
            SelectHero_CMB.SelectedItem = null;
            SelectHero_CMB.Items.Clear();
```

```
        }

        private void HealButton_Click(object sender, EventArgs e)
        {
            Character SelectedHero = null;
            if (this.game.Gold > this.Cost)
            {
                game.Gold -= this.Cost;
                SelectedHero = (Character)SelectHero_CMB.SelectedItem;
                SelectedHero.SetHealth(SelectedHero.MaxHealth);
            }
            SelectedHeroHealthLBL.Text = $"Health: {SelectedHero.Health}/{SelectedHero.MaxHealth}";
            CurrentGoldLBL.Text = "Current gold: " + Convert.ToString(game.Gold);
            SelectedHeroHealthLBL.Refresh();
            CurrentGoldLBL.Refresh();
        }
    }
public partial class BarracksForm : Form
    {
        private Game game;
        private int CurrentCost = -1;
        public BarracksForm(Game game)
        {
            InitializeComponent();
            this.game = game;
        }

        private void ConfirmButton_Click(object sender, EventArgs e)
        {
            Character SelectedHero = (Character)SelectHero_CMB.SelectedItem;

            if (CurrentCost != -1)
            {
                if (game.Gold > CurrentCost)
                {
                    game.Gold -= CurrentCost;
                    SelectedHero.MaxHealth = CandidateCharacter.MaxHealth;
                    SelectedHero.MaxMana = CandidateCharacter.MaxMana;
                    SelectedHero.CritChance = CandidateCharacter.CritChance;
                    SelectedHero.DodgeChance = CandidateCharacter.DodgeChance;
```

```
                SelectedHero.Armour = CandidateCharacter.Armour;
                SelectedHero.Speed = CandidateCharacter.Speed;
                SelectedHero.Level++;
                SelectedHero.ValidateStats();
                this.Close();
            }
        }

    }
    private void HideArrows()
    {
        foreach (Label l in ArrowLabels)
        {
            l.Hide();
        }
    }
    private void ShowArrows()
    {
        foreach (Label l in ArrowLabels)
        {
            l.Show();
        }
    }

    private List<Label> ArrowLabels = new List<Label>();

    private void BarracksForm_Load(object sender, EventArgs e)
    {
        HPLabel.Text = "";
        ManaLabel.Text = "";
        CritLabel.Text = "";
        DodgeLabel.Text = "";
        SpeedLabel.Text = "";
        ArmourLabel.Text = "";

        ArrowLabels.Add(ArrowLabel1);
        ArrowLabels.Add(ArrowLabel2);
        ArrowLabels.Add(ArrowLabel3);
        ArrowLabels.Add(ArrowLabel4);
        ArrowLabels.Add(ArrowLabel5);
```

```
            ArrowLabels.Add(ArrowLabel6);

            HideArrows();

            HPLabelAfter.Text = "";
            ManaLabelAfter.Text = "";
            CritLabelAfter.Text = "";
            DodgeLabelAfter.Text = "";
            SpeedLabelAfter.Text = "";
            ArmourLabelAfter.Text = "";

            ConfirmButton.Enabled = false;

            CurrentGoldLBL.Text = "Current gold: " + Convert.ToString(game.Gold);

            for (int i = 0; i < this.game.Party.Length; i++)
            {
                SelectHero_CMB.Items.Add(this.game.Party[i]);
            }

            SelectHero_CMB.ValueMember = "Character";
            SelectHero_CMB.DisplayMember = "FullTitle";

            this.Controls.Add(SelectHero_CMB);

            SelectHero_CMB.SelectedItem = null;
        }
        private Character prevSelection;
        private Character CandidateCharacter;
        private void HeroCMBBox_SelectChange(object sender, EventArgs e)
        {
            if (SelectHero_CMB.SelectedItem != null && SelectHero_CMB.SelectedItem != prevSelection)
            {
                Character c = (Character)SelectHero_CMB.SelectedItem;
                prevSelection = c;

                int BaseX;
                int BaseY;
                CharacterControls cc = c.Controls;
                cc.RefreshStatLabels();
```

```
                this.Invalidate();

                string CritText = cc.LBL_Crit.Text;
                string ArmourText = cc.LBL_Armour.Text;
                string DodgeText = cc.LBL_Dodge.Text;
                string SpeedText = cc.LBL_Speed.Text;
                string HPText = cc.LBL_HPFraction.Text;
                string ManaText = cc.LBL_ManaFraction.Text;

                HPLabel.Text = "Health: " + HPText;
                ManaLabel.Text = "Mana: " + ManaText;
                CritLabel.Text = CritText;
                DodgeLabel.Text = DodgeText;
                SpeedLabel.Text = SpeedText;
                ArmourLabel.Text = ArmourText;

                BaseX = HPLabel.Location.X; BaseY = HPLabel.Location.Y;

                CurrentCost = 2000;

                ConfirmButton.Text = $"Level up | Cost: {CurrentCost}";

                var className = c.GetType();

                Character tempChar = (Character)Activator.CreateInstance(className, new object[] { CharacterType.Hero,
c.Level + 1 });
                Character SelectedHero = (Character)SelectHero_CMB.SelectedItem;


                tempChar.MaxHealth = SelectedHero.MaxHealth < tempChar.MaxHealth ? tempChar.MaxHealth :
(int)Math.Ceiling(SelectedHero.MaxHealth * 1.15f);
                tempChar.MaxMana = SelectedHero.MaxMana < tempChar.MaxMana ? tempChar.MaxMana :
(int)Math.Ceiling(SelectedHero.MaxMana * 1.15f);
                tempChar.CritChance = SelectedHero.CritChance < tempChar.CritChance ? tempChar.CritChance :
(float)Math.Round(SelectedHero.CritChance * 1.15f, 2);
                tempChar.DodgeChance = SelectedHero.DodgeChance < tempChar.DodgeChance ? tempChar.DodgeChance :
(float)Math.Round(SelectedHero.DodgeChance * 1.15f, 2);
                tempChar.Armour = SelectedHero.Armour < tempChar.Armour ? tempChar.Armour :
(int)Math.Ceiling(SelectedHero.Armour * 1.15f);
```

```
                tempChar.Speed = SelectedHero.Speed < tempChar.Speed ? tempChar.Speed : (int)Math.Round(SelectedHero.Speed
* 1.15f, 2);

                tempChar.Controls = new CharacterControls(tempChar, tempChar.Abilities, tempChar.MaxHealth,
tempChar.MaxMana);
                var tempCC = tempChar.Controls;

                CritText = tempCC.LBL_Crit.Text;
                ArmourText = tempCC.LBL_Armour.Text;
                DodgeText = tempCC.LBL_Dodge.Text;
                SpeedText = tempCC.LBL_Speed.Text;
                HPText = tempCC.LBL_HPFraction.Text;
                ManaText = tempCC.LBL_ManaFraction.Text;

                HPLabelAfter.Text = "Health: " + HPText;
                ManaLabelAfter.Text = "Mana: " + ManaText;
                CritLabelAfter.Text = CritText;
                DodgeLabelAfter.Text = DodgeText;
                SpeedLabelAfter.Text = SpeedText;
                ArmourLabelAfter.Text = ArmourText;

                CandidateCharacter = tempChar;

                ShowArrows();

                if (CurrentCost < game.Gold)
                {
                    ConfirmButton.Enabled = true;
                }

                this.Invalidate();

            }
        }

        private void OnClose(object sender, FormClosingEventArgs e)
        {
            SelectHero_CMB.SelectedItem = null;
            SelectHero_CMB.Items.Clear();
            prevSelection = null;
```

```
            HideArrows();
        }
    }

public partial class StagecoachForm : Form
    {
        private int CalculateCost(Character character)
        {
            Character c = character;
            int cost = 0;

            cost += c.MaxHealth * 10;
            cost += c.MaxMana * 10;
            cost += (int)c.CritChance * 100 * 5;
            cost += (int)c.DodgeChance * 100 * 5;
            cost += c.Armour * 5;
            cost += (int)c.Speed * 5;
            cost += c.Level * 500;
            cost += c.Abilities.Length * 100;

            return cost;

        }
        private Game game;
        private Random rndm = new Random();
        private List<Character> SaleList = new List<Character>();
        public StagecoachForm(Game game)
        {
            InitializeComponent();
            this.game = game;
        }

        private void label1_Click(object sender, EventArgs e)
        {

        }
        private void StagecoachForm_Load(object sender, EventArgs e)
        {
            this.Size = new Size(1000, 900);
        }
```

```
        private void UpdateSaleList()
        {
            List<Control> controlstoremove = new List<Control>();
            foreach (Control c in this.Controls)
            {
                try
                {
                    if ((string)c.Tag != "protected")
                    {
                        controlstoremove.Add(c);
                    }
                }
                catch
                {
                    controlstoremove.Add(c);
                }
            }
            foreach (Control c in controlstoremove)
            {
                this.Controls.Remove(c);
            }
            for (int i = 0; i < SaleList.Count; i++)
            {
                Character c = SaleList[i];

                PictureBox pb = c.Controls.Picture;
                Form form = this;
                CharacterControls cc = c.Controls;


                int x = 10;
                int y = 10 + (i * 205);


                pb.Location = new Point(x, y);
                pb.Size = new Size(100, 200);
                pb.BackColor = Color.Black;


                cc.ClassLabel.Location = new Point(pb.Location.X + pb.Width + 5, pb.Location.Y + 10);
```

```
                cc.ClassLabel.ForeColor = Color.Green;


                cc.NameLabel.Location = new Point(cc.ClassLabel.Location.X, cc.ClassLabel.Location.Y + cc.ClassLabel.Height
+ 5);
                cc.NameLabel.Text = "Name";
                cc.NameLabel.ForeColor = Color.Green;

                cc.LBL_HPFraction.Location = new Point(cc.ClassLabel.Location.X + cc.ClassLabel.Width + 5, pb.Location.Y +
20);
                cc.LBL_ManaFraction.Location = new Point(cc.LBL_HPFraction.Location.X, cc.LBL_HPFraction.Location.Y +
cc.LBL_HPFraction.Height + 5);

                if (!cc.LBL_HPFraction.Text.Contains("Health: "))
                {
                    cc.LBL_HPFraction.Text = $"Health: {cc.LBL_HPFraction.Text}";
                    cc.LBL_ManaFraction.Text = $"Mana: {cc.LBL_ManaFraction.Text}";
                }
                cc.LBL_Crit.Location = new Point(cc.LBL_HPFraction.Location.X + cc.LBL_HPFraction.Width + 5, pb.Location.Y
+ 20);
                cc.LBL_Armour.Location = new Point(cc.LBL_Crit.Location.X, cc.LBL_Crit.Location.Y + cc.LBL_Crit.Height +
5);
                cc.LBL_Dodge.Location = new Point(cc.LBL_Crit.Location.X + cc.LBL_Crit.Width + 5, pb.Location.Y + 20);
                cc.LBL_Speed.Location = new Point(cc.LBL_Crit.Location.X + cc.LBL_Crit.Width + 5, cc.LBL_Crit.Location.Y +
cc.LBL_Crit.Height + 5);

                cc.ClassLabel.BackColor = Color.Transparent;
                cc.NameLabel.BackColor = Color.Transparent;
                cc.LBL_Crit.BackColor = Color.Transparent;
                cc.LBL_Armour.BackColor = Color.Transparent;
                cc.LBL_Dodge.BackColor = Color.Transparent;
                cc.LBL_Speed.BackColor = Color.Transparent;

                cc.ClassLabel.Parent = form;
                cc.NameLabel.Parent = form;
                cc.LBL_Crit.Parent = form;
                cc.LBL_Armour.Parent = form;
                cc.LBL_Dodge.Parent = form;
                cc.LBL_Speed.Parent = form;
                cc.LBL_HPFraction.Parent = form;
```

```
            cc.LBL_ManaFraction.Parent = form;

            Button purchaseBTN = new Button
            {
                Size = new Size(120,120),
                Location = new Point(cc.LBL_Dodge.Location.X + cc.LBL_Dodge.Width + 5, cc.LBL_Dodge.Location.Y),
                Text = $"Purchase ({CalculateCost(c)} gold)",
                Tag = c,
            };
            purchaseBTN.Click += (sender, e) => PurchaseChar(sender, e, (Character)purchaseBTN.Tag);

            form.Controls.Add(pb);
            form.Controls.Add(purchaseBTN);
            form.Controls.Add(cc.ClassLabel);
            form.Controls.Add(cc.NameLabel);
            form.Controls.Add(cc.LBL_Crit);
            form.Controls.Add(cc.LBL_Armour);
            form.Controls.Add(cc.LBL_Dodge);
            form.Controls.Add(cc.LBL_Speed);
            form.Controls.Add(cc.LBL_HPFraction);
            form.Controls.Add(cc.LBL_ManaFraction);
            cc.LBL_HPFraction.BringToFront();
            cc.LBL_ManaFraction.BringToFront();

        }
        this.Invalidate();
    }

    private void PurchaseChar(object sender, EventArgs e, Character character)
    {
        if (this.game.Gold > CalculateCost(character))
        {
            this.game.Gold -= CalculateCost(character);
            this.game.OwnedCharacters.Add(character);
            character.SetName();
            this.SaleList.Remove(character);
            UpdateSaleList();
        }
    }
```

```
        private void BuyNewChar_Click(object sender, EventArgs e)
        {
            if (this.game.Gold > Convert.ToInt32(BuyNewChar.Text.Split('(')[1].Split('g')[0]))
            {
                AddChar(1);
                this.game.Gold -= Convert.ToInt32(BuyNewChar.Text.Split('(')[1].Split('g')[0]);
                BuyNewChar.Text = "Buy New Character (" +
Convert.ToString(Convert.ToInt32(BuyNewChar.Text.Split('(')[1].Split('g')[0]) + (30 * this.game.Level)) + "g)";
            }
        }
        private void AddChar(int count)
        {
            for (int i = 0; i < count; i++)
            {
                if (SaleList.Count == 4)
                {
                    SaleList[rndm.Next(SaleList.Count)] = CharacterFactory.CreateRandomCharacter(CharacterType.Hero,
rndm.Next(game.Level - 1, game.Level + 2), false);
                }
                else
                {
                    SaleList.Add(CharacterFactory.CreateRandomCharacter(CharacterType.Hero, rndm.Next(game.Level - 1,
game.Level + 2), false));
                }
            }
            UpdateSaleList();
        }

    }

public partial class WagonForm : Form
    {
        private Game game;
        public WagonForm(Game game)
        {
            InitializeComponent();
            this.game = game;
        }

        private List<Character> ownedchars = new List<Character>(), currentparty = new List<Character>();
```

```
        private List<ComboBox> CMBoxes = new List<ComboBox>();
        private List<Label> PartyLabels = new List<Label>();
        private void TraderForm_Load(object sender, EventArgs e)
        {
            ownedchars = this.game.OwnedCharacters;
            currentparty = this.game.Party.ToList();
            CMBoxes = new List<ComboBox> { CMB_AllySlot1, CMB_AllySlot2, CMB_AllySlot3, CMB_AllySlot4 };
            PartyLabels = new List<Label> { Party1, Party2, Party3, Party4 };
            foreach (Character character in ownedchars.Concat(game.Party))
            {
                foreach (ComboBox cb in CMBoxes)
                {
                    cb.DisplayMember = "FullTitle";
                    cb.Items.Add(character);
                    cb.Refresh();
                }
            }
            for (int i = 0; i < currentparty.Count; i++)
            {
                PartyLabels[i].Text = currentparty[i].FullTitle;
                CMBoxes[i].SelectedItem = this.currentparty[i];
            }

        }

        private void TraderForm_FormClosed(object sender, FormClosingEventArgs e)
        {
            bool validParty = true;
            for (int i = 0; i < CMBoxes.Count; i++)
            {
                if (CMBoxes[i].SelectedItem == null)
                {
                    validParty = false;
                }
            }
            if (validParty)
            {
                for (int i = 0; i < CMBoxes.Count; i++)
                {
                    this.game.Party[i] = (Character)CMBoxes[i].SelectedItem;
```

```
                }
            }
            foreach (ComboBox cb in CMBoxes)
            {
                cb.Items.Clear();
            }
            this.ownedchars.Clear();
            this.currentparty.Clear();
        }

        private void comboBox_SelectedIndexChanged(object sender, EventArgs e)
        {
            ComboBox changedComboBox = (ComboBox)sender;

            foreach (ComboBox cb in CMBoxes)
            {
                if (cb != changedComboBox)
                {
                    if (changedComboBox.SelectedItem == cb.SelectedItem)
                    {
                        cb.SelectedIndex = -1;
                    }
                }
                cb.Refresh();
            }
        }
    }
```

Requirement 5

Requirement	Randomly generated dungeons - Layout must be random - Contents of rooms must be random - All dungeons MUST have 1 exit room, as far away from the center as possible
Explanation/Purpose	Sub-algorithm to find a candidate exterior room to place the exit in ensure that as long as the room count is 2, there will be an exit room. Selection at the start of the method also ensures that the room count being generated for is atleast 2.
Techniques used	2d arrays

```
public void Generate(int roomcount)
        {

            if (roomcount < 2) { roomcount = 2; }
            roomcount -= 1;
            int[,] Map = new int[size, size];
            Map[size / 2, size / 2] = 1;

            Random rndm = new Random();
            // loop generates a temporary list of "candidates"
            // (adjacent points in the array) for a new room to be created on.
            for (int i = roomcount; i > 0; i--)
            {
                List<Coordinate> candidateCoords = new List<Coordinate>();
                for (int y = 0; y < size; y++)
                {
                    for (int x = 0; x < size; x++)
                    {
                        Coordinate coord = new Coordinate(x, y);
                        if (Map[x, y] == 0 && IsAdjacentToValue(coord, Map))
                        {
                            candidateCoords.Add(coord);
                        }
                    }
                }
```

```
                Coordinate chosenCoord = candidateCoords[rndm.Next(0, candidateCoords.Count)];
                Map[chosenCoord.X, chosenCoord.Y] = rndm.Next(1, 5);
            }

            // following chunk of loops/ifs sets the "exit" room to a random existing room that is the furthest from the
center as possible.
            List<List<int>> OutsideRooms = new List<List<int>>();
            for (int i = 0; i < size; i++)
            {
                if (Map[i, 0] > 0)
                {
                    OutsideRooms.Add(new List<int>() { i, 0 });
                }
                if (Map[0, i] > 0)
                {
                    OutsideRooms.Add(new List<int>() { 0, i });
                }
            }
            for (int i = 0; i < size; i++)
            {
                if (Map[i, 4] > 0)
                {
                    OutsideRooms.Add(new List<int>() { i, 4 });
                }
                if (Map[4, i] > 0)
                {
                    OutsideRooms.Add(new List<int>() { 4, i });
                }
            }
            if (OutsideRooms.Count < 1)
            {
                for (int i = 1; i < size - 1; i++)
                {
                    if (Map[i, 1] > 0)
                    {
                        OutsideRooms.Add(new List<int>() { i, 1 });
                    }
                    if (Map[1, i] > 0)
                    {
                        OutsideRooms.Add(new List<int>() { 1, i });
```

```
                    }
                }
                for (int i = 1; i < size - 1; i++)
                {
                    if (Map[i, 3] > 0)
                    {
                        OutsideRooms.Add(new List<int>() { i, 3 });
                    }
                    if (Map[3, i] > 0)
                    {
                        OutsideRooms.Add(new List<int>() { 3, i });
                    }
                }
            }
            int OutsideChoice = rndm.Next(0, OutsideRooms.Count);
            Map[OutsideRooms[OutsideChoice][0], OutsideRooms[OutsideChoice][1]] = 5; // 5 means the room is the exit

            ProcessDungeonGrid(Map);
        }
```

Requirement 6

Requirement	The player being able to retreat from the dungeon, which can only be done while outside of combat. This would result in “failing” the dungeon.
Explanation/Purpose	A player should not be able to retreat during combat, this ensures that they are punished for entering into a dungeon / combat instance that they are not at a sufficient level to handle. At the end of the CombatStart() method, the retreat button is disabled, and when combat is concluded in the ConcludeRoom() method, it is re-enabled.
Techniques used	

```
private void BTN_GoBack_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(this.dungeon.game.MainEstateMap, false);
            this.Hide();
        }
```

```
public void CombatStart()
        {
            GenSpeedOrderChars();

            Form form = this.room.Dungeon.DungeonForm;

            NextTurnBTN = new Button();
            NextTurnBTN.Size = new Size(90, 35);
            NextTurnBTN.Location = new Point((form.Width / 2) - (NextTurnBTN.Width / 2), (form.Height / 10) * 2 + 35);
            NextTurnBTN.Text = "Increment turn";
            NextTurnBTN.TextAlign = System.Drawing.ContentAlignment.MiddleCenter;
            NextTurnBTN.BackColor = Color.White;

            form.Controls.Add(NextTurnBTN);
            NextTurnBTN.Show();
            NextTurnBTN.Click += new EventHandler(ExecuteNextTurn);

            TurnInfoLBL = new Label();
            TurnInfoLBL.AutoSize = true;
            TurnInfoLBL.Location = new Point(500,65);
            TurnInfoLBL.Font = new Font(FontFamily.GenericSansSerif, 12);
            TurnInfoLBL.Text = "";
            TurnInfoLBL.Tag = "Dynamic";
            TurnInfoLBL.TextAlign = System.Drawing.ContentAlignment.TopLeft;
            TurnInfoLBL.BackColor = Color.Transparent;

            form.Controls.Add(TurnInfoLBL);
            TurnInfoLBL.Show();

            this.room.Dungeon.DungeonForm.ViewMap.Enabled = false;
            this.room.Dungeon.DungeonForm.BTN_GoBack.Enabled = false;

        }

public void ConcludeRoom()
        {
            this.Dungeon.DungeonForm.ViewMap.Enabled = true;
            this.Dungeon.DungeonForm.BTN_GoBack.Enabled = true;
        }
```

Requirement 8 – Not done

Requirement	The ability for the player to save the state of their current game, and load it back up.
Explanation/Purpose	
Techniques used	
Code not yet in a finished state	

Requirement 9 – Not done

Requirement	Turn based combat. - Order in which characters take turns dependant on their speed stat (higher = quicker = go sooner)
Explanation/Purpose	
Techniques used	
Would like to make some tweaks to code	

Requirement 10

Requirement	A system ensuring enemies making informed decisions, e.g. when to heal/attack/defend/buff allies. - Heal when themselves or allies (enemies to the player) are low on health - Attack the lowest health enemy (To ensure damage is not spread out so much that it becomes ineffective) - Attacks randomly if none of the player's characters are on distinctly lower health than the others & attempts to deal splash damage
Explanation/Purpose	Sufficiently competent "AI" enemies are required in order to ensure that battles actually feel challenging, and that there is some level of competent decision making happening for the enemies.
Techniques used	
`Character LowestEN, HighestEN, LowestAL, HighestAL, ChosenTarget = null;`	

```
Ability ChosenAbility = null;

LowestEN = Allies.OrderBy(x => x.Health).First();
HighestEN = Allies.OrderBy(x => x.Health).Last();
LowestAL = Enemies.OrderBy(x => x.Health).First();
HighestAL = Enemies.OrderBy(x => x.Health).Last();


if (LowestAL.Health < LowestAL.MaxHealth * 0.33)
{ // tries to do a support move if there is a weak ally
    List<Ability> intersect = new List<Ability>();
    foreach (Ability ability in c.Abilities)
    {
        if (ability.GetType() == typeof(SmokeBomb) && LowestAL == c)
        {
            intersect.Add(ability);
        }
        if (ability.GetType() == typeof(Heal) ||
            ability.GetType() == typeof(DivineIntervention) ||
            ability.GetType() == typeof(DivineShield))
        {
            intersect.Add(ability);
        }
    }
    if (intersect.Count > 0)
    {
        ChosenAbility = intersect[rndm.Next(0, intersect.Count)];
        ChosenTarget = LowestAL;
    }
}
else if (LowestEN.Health < LowestEN.MaxHealth * 0.6)
{
    // tries to damage a finish off weak enemies
    ChosenTarget = LowestEN;

    ChosenAbility = Utils.RemoveSupportAbilities(c.Abilities).OrderBy(a => a.Damage).Last();
}
else
{
    // if nobody is weak, try to deal splash damage, alternatively kill the weakest enemy
```

```
    bool found = false;
    foreach (Ability ability in c.Abilities)
    {
        if (ability.effect == Ability.SpecialEffect.SplashAll ||
            ability.effect == Ability.SpecialEffect.SplashL ||
            ability.effect == Ability.SpecialEffect.SplashR ||
            ability.effect == Ability.SpecialEffect.SplashLR)
        {
            ChosenAbility = ability;
            ChosenTarget = Allies[rndm.Next(0,Allies.Length)];
            found = true;
        }
    }
    if (!found)
    {
        ChosenAbility = c.Abilities.OrderBy(a => a.Damage).Last();
        ChosenTarget = LowestEN;
    }
}

CombatTurnHandler.HandleCombatInteraction(c, ChosenTarget, ChosenAbility, this);
```

Requirement 11

Requirement	Progressive & scaling difficulty & loot/rewards - All enemy stats should scale with the level of the player - Gold granted from chests in dungeons should scale with the level of the player
Explanation/Purpose	This requirement ensures that the player never feels as though they are exceeding the level of strength of their enemies to a point where combat/playing becomes unenjoyably easy. Additionally, it actively encourages the player to either upgrade their existing characters or to buy new ones. Characters they player sources from the stagecoach will also be within a certain range around the player's level. The generate enemy array method passes in the player's level to the character factory, which then creates a character of that level, meaning their stats will reflect that level and match the player.

Techniques used	

```
private Character[] GenerateEnemyArr()
        {
            Character[] characterArr = new Character[4];
            for (int i = 0; i < 4; i++)
            {
                characterArr[i] = CharacterFactory.CreateRandomCharacter(CharacterType.Enemy, this.Dungeon.game.Level);
                characterArr[i].MultiplyStats(this.Dungeon.DifficultyMult);
            }
            return characterArr;
        }
public RewardRoom(RoomContent roomCont, Dungeon dungeon) : base(roomCont, dungeon)
        {
            RoomType = roomCont;
            this.Dungeon = dungeon;
            this.backdropFilePath = $"roombackdrops/room{rndm.Next(0, 5)}.png";
            this.GoldReward = (int)(rndm.Next(800, 1501) * this.Dungeon.game.Level * this.Dungeon.DifficultyMult);
        }
```

Requirement 13 & 15

Requirement	A “Dungeon keeper” mode, where the player can create dungeons & decide the layout of the dungeon. These dungeons can then be stored and shared with other players. - Creates an easily shareable key for each dungeon generated. i. Ensure the input for a dungeon key is appropriately validated - Stored in a file. - Intuitive interface for dungeon creation
Explanation/Purpose	Requirements 13 an 15, for players to be able to create custom dungeons, is crucial to the overall solution requirement and problem statement. A recursive depth first search algorithm is used to ensure that every room is connected in some way, which prevents islands. Additionally the dungeon is checked for the presence of 1 exit room.

	The keyparser class is used to turn the saved dungeon into a shareable single-line key. Regular expressions are used to validate a dungeon's pattern input which is how dungeons are shared and loaded by players. All of these validation checks are don't every time the player updates their current map, if these and other validation checks return true (such as the player having entered a name for the dungeon), the save button is enabled for the player to export their dungeon into sharable key form. A label is also displayed to let the player know whether or not the current layout of their dungeon is valid or not.
Techniques used	Recursion, Depth first search, 2d arrays, Dictionary, External image file loading, Regular expressions, External file saving to (.txt)

```
public partial class CustomDungeonMapForm : Form
    {
        public Button[,] pbArr;
        private int[,] Grid = new int[5, 5];
        private int roomCount { get
            {
                var flatpbArr = pbArr.Cast<Button>();
                return flatpbArr.Count(x => x.Tag != null && int.TryParse(x.Tag.ToString(), out int tagValue) && tagValue >
0);
            }
        }
        private int exitCount
        {
            get
            {
                var flatpbArr = pbArr.Cast<Button>();
                return flatpbArr.Count(x => x.Tag != null && int.TryParse(x.Tag.ToString(), out int tagValue) && tagValue
== 5);
            }
        }
        private Dictionary<int, Image> BackImgDict = new Dictionary<int, Image>
            {
                { 0, Image.FromFile("roomicons/Nulled.png") },
                { 1, Image.FromFile("roomicons/empty.png") },
                { 2, Image.FromFile("roomicons/Easy.png") },
                { 3, Image.FromFile("roomicons/Hard.png") },
```

```
            { 4, Image.FromFile("roomicons/Reward.png") },
            { 5, Image.FromFile("roomicons/Exit.png") }
        };
    public CustomDungeonMapForm()
    {
        InitializeComponent();
        pbArr = new Button[5, 5]
        {
            { MapBox1, MapBox2,MapBox3,MapBox4,MapBox5 },
            { MapBox6, MapBox7,MapBox8,MapBox9,MapBox10 },
            { MapBox11,MapBox12,MapBox13,MapBox14,MapBox15 },
            { MapBox16,MapBox17,MapBox18,MapBox19,MapBox20 },
            { MapBox21,MapBox22,MapBox23,MapBox24,MapBox25 },
        };
        for (int x = 0; x < 5; x++)
        {
            for (int y = 0; y < 5; y++)
            {
                int tempX = x;
                int tempY = y;
                pbArr[x, y].Tag = 0;
                pbArr[x, y].Click += (sender, e) => PictureBox_Click(sender, e, tempX, tempY);
            }
        }
        pbArr[2, 2].BackColor = Color.White;
        pbArr[2, 2].Tag = 1;
        pbArr[2, 2].Enabled = false;
        BTN_Save.Enabled = false;
    }
    private void PictureBox_Click(object sender, EventArgs e, int x, int y)
    {
        pbArr[x, y].Tag = Convert.ToInt16(pbArr[x, y].Tag) + 1;

        if (Convert.ToInt16(pbArr[x, y].Tag) > 5)
        {
            pbArr[x, y].Tag = 0;
        }


        isValidLBL.Text = CheckValidGrid() && exitCount == 1 ? "Valid Format" : "Invalid Format";
```

```
            UpdatePBImgs();
            CheckForValidSaveState();
        }
        private void CheckForValidSaveState()
        {
            if (CheckValidGrid() && DNGN_Name_TXT.Text.Length > 0 && DNGN_Name_TXT.Text.Length < 24 && roomCount > 1 &&
exitCount == 1)
            {
                BTN_Save.Enabled = true;
                return;
            }
            BTN_Save.Enabled = false;
        }
        private void NameTextChanged(object sender, EventArgs e)
        {
            CheckForValidSaveState();
        }
        private void UpdatePBImgs()
        {
            for (int x = 0; x < 5; x++)
            {
                for (int y = 0; y < 5; y++)
                {
                    pbArr[x, y].BackgroundImage = BackImgDict[Convert.ToInt16(pbArr[x, y].Tag)];
                    pbArr[x, y].Refresh();
                }
            }

        }

        private bool CheckValidGrid()
        {
            bool[,] visited = new bool[5, 5];

            DepthFirstSearch(2,2, visited);

            for (int x = 0; x < 5; x++)
            {
                for (int y = 0; y < 5; y++)
                {
```

```
                if (Convert.ToInt16(pbArr[x, y].Tag) >= 1 && !visited[x, y])
                {
                    return false;
                }
            }
        }

        for (int x = 0; x < 5; x++)
        {
            for (int y = 0; y < 5; y++)
            {
                Grid[x, y] = Convert.ToInt16(pbArr[x, y].Tag);
            }
        }

        return true;
    }

    private void DepthFirstSearch(int x, int y, bool[,] visited)
    {
        if (x < 0 || x >= 5 || y < 0 || y >= 5) return;
        if (visited[x, y]) return;
        if (Convert.ToInt16(pbArr[x, y].Tag) < 1) return;

        visited[x, y] = true;

        DepthFirstSearch(x - 1, y, visited);
        DepthFirstSearch(x + 1, y, visited);
        DepthFirstSearch(x, y - 1, visited);
        DepthFirstSearch(x, y + 1, visited);
    }

    private void BTN_Save_Click(object sender, EventArgs e)
    {
        KeyParser.ToFile(Grid, (float)Decimal.Divide(DifficultyTrackBar.Value, 10), DNGN_Name_TXT.Text);
        this.Close();
    }

    private void DIffBarChange(object sender, EventArgs e)
    {
```

```
            DifficultyLBL.Text = $"Difficulty: x{Decimal.Divide(DifficultyTrackBar.Value, 10)}";
        }


    }

public static class KeyParser
    {
        // format example:
        // testname|0.8|4:1,0:2,2:1,0:1,2:1,1:1,0:1,2:1,0:1,3:2,1:1,2:1,1:1,0:1,4:1,2:1,0:4,5:1,2:2

        public static void ToFile(int[,] grid, float difficulty, string name)
        {
            string output = $"{name}|{difficulty}|";

            string flatGrid = "";

            for (int y = 0; y < 5; y++)
            {
                for (int x = 0; x < 5; x++)
                {
                    flatGrid += grid[y, x];
                }
            }


            int counter = 1;

            List<string> RLEList = new List<string>();

            for (int i = 0; i < flatGrid.Length - 1; i++)
            {
                if (flatGrid[i] == flatGrid[i + 1])
                {
                    counter++;
                }
                else
                {
                    RLEList.Add($"{flatGrid[i]}:{counter}");
                    counter = 1;
```

```
                }
            }
            RLEList.Add($"{flatGrid[flatGrid.Length - 1]}:{counter}");

            Console.Write("[");
            foreach (string s in RLEList)
            {
                output += $"{s},";
                Console.Write($"{s},");
            }
            Console.Write("]");

            output = output.Remove(output.Length - 1, 1);

            Console.WriteLine("\n\n");
            Console.WriteLine(output);

            StreamWriter sw = new StreamWriter($"savedDungeons/{name}.txt");

            Clipboard.SetText(output);

            sw.WriteLine(output); sw.Close();

        }
        public static string To_Key(int[,] grid, float difficulty, string name)
        {
            string output = $"{name}|{difficulty}|";

            string flatGrid = "";

            for (int y = 0; y < 5; y++)
            {
                for (int x = 0; x < 5; x++)
                {
                    flatGrid += grid[y, x];
                }
            }

            int counter = 1;
```

```
            List<string> RLEList = new List<string>();

            for (int i = 0; i < flatGrid.Length - 1; i++)
            {
                if (flatGrid[i] == flatGrid[i + 1])
                {
                    counter++;
                }
                else
                {
                    RLEList.Add($"{flatGrid[i]}:{counter}");
                    counter = 1;
                }
            }
            RLEList.Add($"{flatGrid[flatGrid.Length - 1]}:{counter}");

            foreach (string s in RLEList)
            {
                output += $"{s},";
            }

            output = output.Remove(output.Length - 1, 1);

            return output;
        }
        public static Dungeon DGN_ParseFromKey(string key, Game game)
        {
            string data = key;

            string output = "";

            float diff = (float)Convert.ToDouble(data.Split('|')[1]);
            Console.WriteLine(diff);
            foreach (string pair in data.Split('|')[2].Split(','))
            {
                string[] splitpair = pair.Split(':');
                string character = splitpair[0];
                int freq = Convert.ToInt16(splitpair[1].ToString());

                for (int i = 0; i < freq; i++)
```

```
                {
                    output += character;
                }

            }

            int[,] map = new int[5, 5];

            int k = 0;

            for (int y = 0; y < 5; y++)
            {
                for (int x = 0; x < 5; x++)
                {
                    map[y, x] = Convert.ToInt16(Convert.ToString(output[k]));
                    k++;
                }
            }

            for (int y = 0; y < 5; y++)
            {
                for (int x = 0; x < 5; x++)
                {
                    Console.Write(map[y, x] + " ");
                }
                Console.WriteLine();
            }
            Console.WriteLine("----------");
            Console.WriteLine($"Name: {data.Split('|')[0]}");
            Dungeon test = new Dungeon(game, diff, map, data.Split('|')[0], true);
            Console.WriteLine(test);
            return new Dungeon(game, diff, map, data.Split('|')[0], true);
        }


    }
public partial class CustomDungeonSelect : Form
    {
        Game game;
        public CustomDungeonSelect(Game game)
```

```
        {
            InitializeComponent();
            this.game = game;
            LoadDGN_Button.Enabled = false;
        }

        private void CreateDGN_Button_Click(object sender, EventArgs e)
        {
            Utils.ShowForm(new CustomDungeonMapForm());
        }

        private void LoadDGN_Button_Click(object sender, EventArgs e)
        {
            Dungeon d = KeyParser.DGN_ParseFromKey(DGN_Key_TXT.Text, game);
            Utils.ShowForm(d.DungeonForm, false);
            d.UpdateMap(this.game.GetDungeon(1).MapForm.pbArr);
            Utils.ShowForm(d.MapForm, true, true);
            this.Hide();
            this.game.MainHamlet.Hide();
        }

        private void KeyTXT_Changed(object sender, EventArgs e)
        {
            if (Regex.Match($"{DGN_Key_TXT.Text}", @"^[a-zA-Z ]+\|[0,1,2](\.[0-9])?\|([0-5]{1}:([0-9]){1,2},)+([0-
5]{1}:([0-9]){1,2}){1}$").Success)
            {
                LoadDGN_Button.Enabled = true;
            }
            else
            {
                LoadDGN_Button.Enabled = false;
            }
        }
    }
```

Requirement 16

Requirement	Multiple unique character classes, to allow for variety and strategy. - Different classes have different strengths and weaknesses (Stat points such as health & armour represent what the class would logically have more & less of) - Different classes have different sets of abilities which they can have. Some abilities are exclusive, whereas others can be had by any class.
Explanation/Purpose	A sufficient range of classes is required to ensure that there is some variety to the gameplay. It also allows for different sets of characters to synergise better or worse together and for the player to consider carefully what types of characters they have within their party. To ensure that there is variety between more than just the name of the character's class, I have ensured that there are abilities that only certain classes can have, as well as making different classes of characters' stats scale differently. E.g. Wizards have lower health, but higher mana. These proportions are maintained when a character has a higher level.
Techniques used	Inheritance, Overriding

```
public class Wizard : Character
    {
        public Wizard(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }
        public override void InitializeStats()
        {
            base.InitializeStats();
            this.CritChance += 0.1f;
            this.MaxHealth = (int)(this.MaxHealth * 0.7);
            this.Armour = (int)(this.Armour * 0.8);
            this.Speed = (int)(this.Speed * 1.2);
            this.MaxMana = (int)(this.MaxMana * 3);
            base.ValidateStats();
        }
    }

    public class Rogue : Character
    {
```

```
        public Rogue(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.CritChance += 0.15f;
            this.MaxHealth = (int)(this.MaxHealth * 0.7);
            this.DodgeChance += 0.15f;
            this.Armour = 0;
            this.Speed *= 2;
            base.ValidateStats();
        }
    }

    public class Paladin : Character
    {
        public Paladin(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 2);
            this.Armour = (int)(this.Armour * 1.7);
            this.Speed = (int)(this.Speed * 0.8);
            this.CritChance = 0;
            this.MaxMana = (int)(this.MaxMana * 1.35);
            this.DodgeChance = (int)(this.DodgeChance * 0.5);
            base.ValidateStats();
        }
    }

    public class Barbarian : Character
    {
        public Barbarian(CharacterType type, int level) : base(type, level)
```

```
        {
            InitializeStats();
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 1.3);
            this.Armour = (int)(this.Armour * 0.85);
            this.Speed = (int)(this.Speed * 0.95);
            this.CritChance = 0.1f;
            this.DodgeChance = (int)(this.DodgeChance * 0.5);
            base.ValidateStats();
        }
    }

    public class Monk : Character
    {
        public Monk(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 0.9);
            this.Armour = (int)(this.Armour * 0.9);
            this.Speed = (int)(this.Speed * 3);
            this.DodgeChance = (int)(this.DodgeChance * 2.5);
            base.ValidateStats();
        }
    }

    public class Druid : Character
    {
        public Druid(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }
```

```
        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 1.3);
            this.Armour = (int)(this.Armour * 1.1);
            this.Speed = 20;
            this.CritChance = 0;
            this.MaxMana = (int)(this.MaxMana * 2.5);
            this.DodgeChance = (int)(this.DodgeChance * 0.3);
            base.ValidateStats();
            // completely healing focused
        }
    }

    public class Bard : Character
    {
        public Bard(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 0.8);
            this.Armour = (int)(this.Armour * 0.4);
            this.Speed = (int)(this.Speed * 2);
            this.CritChance = 0;
            this.MaxMana = (int)(this.MaxMana * 2.5);
            this.DodgeChance = (int)(this.DodgeChance * 0.3);
            base.ValidateStats();
        }
    }

    public class Bloodhunter : Character
    {
        public Bloodhunter(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
```

```
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 1.5);
            this.Armour = (int)(this.Armour * 1.15);
            this.Speed = (int)(this.Speed * 1.15);
            this.CritChance = 0;
            base.ValidateStats();
        }
    }

    public class HighwayMan : Character
    {
        public HighwayMan(CharacterType type, int level) : base(type, level)
        {
            InitializeStats();
        }

        public override void InitializeStats()
        {
            base.InitializeStats();
            this.MaxHealth = (int)(this.MaxHealth * 0.8);
            this.Armour = (int)(this.Armour * 0.7);
            this.Speed = (int)(this.Speed * 2.3);
            this.CritChance *= 1.5f;
            base.ValidateStats();
        }
    }
```

Requirement 17

Requirement	A sufficient range of abilities - Each class must have the ability to have at least 4 abilities (Since characters have 4 abilities)

	- There must be enough abilities to ensure that most classes have unique combinations of abilities that are sought after by the player, to differentiate characters of one class from each other. - Some abilities should have special modifiers about them. For example this may result in them inflicting a status effect, or not being able to miss.
Explanation/Purpose	It is important that every class of character has atleast 4 abilities, as that was an aspect I was trying to replicate from the existing system Darkest Dungeon 1. Due to the fact that abilities were class-restricted, and one ability may only be used by a few certain classes, in order to solve the issue of having less than 4 abilities for a certain class I added 2 abilities: Kick and Punch. These abilities are universal, and have no class based restrictions whatsoever. This addition also presented the accidental benefit of meaning that some characters that had the potential to wield strong abilities, such as bloodhunter having blood nova and wizard having earthquake, may not actually end up having them. This then means not every bloodhunter is as good as every other, meaning a player is far less likely to find 1 bloodhunter and stick with it during their entire playthrough, providing variety to the gameplay.
Techniques used	Inheritance, Overriding, Abstract class

```
public abstract class Ability

public class Fireball : Ability
    {
        public Fireball()
        {
            effect = SpecialEffect.Burn;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 50;
            MissChance = 0.1f;
            CritChanceMult = 2;
            Costs = new AbilityCost(10, 0);
            ValidCharacterTypes = new Type[] { typeof(Wizard), typeof(Druid), typeof(Bard) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
```

```
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class Backstab : Ability
    {
        public Backstab()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 4 };
            Damage = 100;
            MissChance = 0.2f;
            CritChanceMult = 3;
            Costs = new AbilityCost(0, 0);
            ValidCharacterTypes = new Type[] { typeof(Rogue), typeof(HighwayMan) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class Smite : Ability
    {
        public Smite()
        {
            effect = SpecialEffect.Strengthen;
            TargetablePositions = new List<int> { 1, 2 };
            Damage = 70;
            MissChance = 0.05f;
            CritChanceMult = 1;
            Costs = new AbilityCost(20, 0);
            ValidCharacterTypes = new Type[] { typeof(Paladin), typeof(Monk) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
```

```
    }
    public class Berserk : Ability
    {
        public Berserk()
        {
            effect = SpecialEffect.Strengthen;
            TargetablePositions = new List<int> { -1 };
            Damage = 0;
            MissChance = 0.0f;
            CritChanceMult = 1;
            Costs = new AbilityCost(0, 50);
            ValidCharacterTypes = new Type[] { typeof(Barbarian) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class Heal : Ability
    {
        public Heal()
        {
            effect = SpecialEffect.HPRegen;
            TargetablePositions = new List<int> { -1 }; // Targets allies
            Damage = -30; // Negative for healing
            MissChance = 0.0f;
            CritChanceMult = 1;
            Costs = new AbilityCost(20, 0);
            ValidCharacterTypes = new Type[] { typeof(Paladin), typeof(Druid), typeof(Monk) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class PoisonArrow : Ability
```

```
    {
        public PoisonArrow()
        {
            effect = SpecialEffect.Poison;
            TargetablePositions = new List<int> { 3, 4 };
            Damage = 20;
            MissChance = 0.1f;
            CritChanceMult = 1;
            Costs = new AbilityCost(10, 0);
            ValidCharacterTypes = new Type[] { typeof(Rogue), typeof(HighwayMan) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class Frostbolt : Ability
    {
        public Frostbolt()
        {
            effect = SpecialEffect.Frost;
            TargetablePositions = new List<int> { 3, 4 };
            Damage = 30;
            MissChance = 0.2f;
            CritChanceMult = 2;
            Costs = new AbilityCost(15, 0);
            ValidCharacterTypes = new Type[] { typeof(Wizard), typeof(Druid) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class DivineIntervention : Ability
    {
        public DivineIntervention()
```

```
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { -1 };
            Damage = -100;
            MissChance = 0.0f;
            CritChanceMult = 1;
            Costs = new AbilityCost(50, 0);
            ValidCharacterTypes = new Type[] { typeof(Paladin), typeof(Monk) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.ApplyStatusEffect(source, target, StatusEffects.Invincible, 1);
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class DrainLife : Ability
    {
        public DrainLife()
        {
            effect = SpecialEffect.DrainMana;
            TargetablePositions = new List<int> { 2, 3 };
            Damage = 20;
            MissChance = 0.2f;
            CritChanceMult = 1;
            Costs = new AbilityCost(20, 0);
            ValidCharacterTypes = new Type[] { typeof(Bard), typeof(Bloodhunter) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class Teleport : Ability
    {
        public Teleport()
        {
```

```
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { -1 };
            Damage = 0;
            MissChance = 0.0f;
            CritChanceMult = 1;
            Costs = new AbilityCost(20, 0);
            ValidCharacterTypes = new Type[] { typeof(Wizard) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }

    public class ShurikenThrow : Ability
    {
        public ShurikenThrow()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 3, 4 };
            Damage = 30;
            MissChance = 0.1f;
            CritChanceMult = 2;
            Costs = new AbilityCost(10, 0);
            ValidCharacterTypes = new Type[] { typeof(Rogue), typeof(Monk) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
public class Shoot : Ability
    {
        public Shoot()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 2, 3, 4 };
            Damage = 25;
```

```
            MissChance = 0.05f;
            CritChanceMult = 1;
            Costs = new AbilityCost(5, 0);
            ValidCharacterTypes = new Type[] { typeof(HighwayMan), typeof(Rogue) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class SmokeBomb : Ability
    {
        public SmokeBomb()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { -1 };
            Damage = 0;
            MissChance = 0;
            CritChanceMult = 1;
            Costs = new AbilityCost(5, 0);
            ValidCharacterTypes = new Type[] { typeof(Rogue) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.ApplyStatusEffect(source, target, StatusEffects.Invincible, 1);
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class Punch : Ability
    {
        public Punch()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 1,2,3,4 };
            Damage = 35;
            MissChance = 0.3f;
            CritChanceMult = 1;
            Costs = new AbilityCost(0, 0);
```

```
            ValidCharacterTypes = Utils.CharacterTypes;
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class Kick : Ability
    {
        public Kick()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 35;
            MissChance = 0.3f;
            CritChanceMult = 1;
            Costs = new AbilityCost(0, 0);
            ValidCharacterTypes = Utils.CharacterTypes;
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class SwordSlash : Ability
    {
        public SwordSlash()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 1, 2, 3 };
            Damage = 70;
            MissChance = 0.1f;
            CritChanceMult = 1;
            Costs = new AbilityCost(5, 0);
            ValidCharacterTypes = new Type[] { typeof(Paladin), typeof(Barbarian) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
```

```
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class BloodSuck : Ability
    {
        public BloodSuck()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 65;
            MissChance = 0.1f;
            CritChanceMult = 1;
            Costs = new AbilityCost(5, 20);
            ValidCharacterTypes = new Type[] { typeof(Bloodhunter) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class BloodNova : Ability
    {
        public BloodNova()
        {
            effect = SpecialEffect.SplashAll;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 90;
            MissChance = 0f;
            CritChanceMult = 1;
            Costs = new AbilityCost(15, 80);
            ValidCharacterTypes = new Type[] { typeof(Bloodhunter) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
```

```
    public class Earthquake : Ability
    {
        public Earthquake()
        {
            effect = SpecialEffect.SplashAll;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 40;
            MissChance = 0.15f;
            CritChanceMult = 1;
            Costs = new AbilityCost(12, 0);
            ValidCharacterTypes = new Type[] { typeof(Wizard), typeof(Druid) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class Parry : Ability
    {
        public Parry()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { -1 };
            Damage = 0;
            MissChance = 0;
            CritChanceMult = 1;
            Costs = new AbilityCost(0, 0);
            ValidCharacterTypes = new Type[] { typeof(Rogue), typeof(Monk), typeof(HighwayMan) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.ApplyStatusEffect(source, target, StatusEffects.Invincible, 1);
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class ThunderStrike : Ability
    {
        public ThunderStrike()
```

```
        {
            effect = SpecialEffect.ConfirmCrit;
            TargetablePositions = new List<int> { 1, 2 };
            Damage = 35;
            MissChance = 0.1f;
            CritChanceMult = 2;
            Costs = new AbilityCost(8, 0);
            ValidCharacterTypes = new Type[] { typeof(Paladin), typeof(Barbarian), typeof(Wizard) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class Weaken : Ability
    {
        public Weaken()
        {
            effect = SpecialEffect.Weaken;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 0;
            MissChance = 0.1f;
            CritChanceMult = 1;
            Costs = new AbilityCost(5, 0);
            ValidCharacterTypes = new Type[] { typeof(Bard) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.ApplyStatusEffect(source, target, StatusEffects.Weak, 1);
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class DivineShield : Ability
    {
        public DivineShield()
        {
            effect = SpecialEffect.None;
            TargetablePositions = new List<int> { -1 };
```

```
            Damage = 0;
            MissChance = 0;
            CritChanceMult = 1;
            Costs = new AbilityCost(10, 0);
            ValidCharacterTypes = new Type[] { typeof(Paladin) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.ApplyStatusEffect(source, target, StatusEffects.Invincible, 1);
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class MindControl : Ability
    {
        public MindControl()
        {
            effect = SpecialEffect.Weaken;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 0;
            MissChance = 0.2f;
            CritChanceMult = 1;
            Costs = new AbilityCost(15, 0);
            ValidCharacterTypes = new Type[] { typeof(Wizard), typeof(Bard) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.ApplyStatusEffect(source, target, StatusEffects.Weak, 2);
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class PerfectShot : Ability
    {
        public PerfectShot()
        {
            effect = SpecialEffect.Bullseye;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 40;
            MissChance = 0f;
```

```
            CritChanceMult = 2;
            Costs = new AbilityCost(15, 0);
            ValidCharacterTypes = new Type[] { typeof(HighwayMan) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
    public class Whirlwind : Ability
    {
        public Whirlwind()
        {
            effect = SpecialEffect.SplashAll;
            TargetablePositions = new List<int> { 1, 2, 3, 4 };
            Damage = 30;
            MissChance = 0.25f;
            CritChanceMult = 1;
            Costs = new AbilityCost(10, 0);
            ValidCharacterTypes = new Type[] { typeof(Barbarian) };
        }

        public override void Execute(Character source, Character target, CombatInstance CI)
        {
            CombatTurnHandler.HandleCombatInteraction(source, target, this, CI);
        }
    }
```

Requirement 18

Requirement	Some attacks can inflict status effects onto characters - Status effects persist across turns. They can have a positive or negative impact on the character they are placed on. - Positive example: Strength, Negative example: Poison

Explanation/Purpose	Status effects were another aspect of the existing system(s) Darkest dungeon 1 (and 2). Each status effect lasts a certain number of turns. Some “tick” each turn, such as poison which deals damage on every tick, and some just give a certain effect to the character whom is afflicted, such as strength making the character deal more damage. The way status effects work are, abilities are given special effects, some of which then map to status effects. E.g. The ability fireball has the special effect burn, which then inflicts the target with the status effect burn. The reason for the seemingly redundant difference is some special effects do not map to status effects, e.g. the bullseye special effect means that an ability cannot miss, but inflicts no status effect.
Techniques used	Dictionaries

```
public static void TickSpecialFX(Character c)
{
    if (c != null)
    {
        List<StatusEffects> keys = new List<StatusEffects>(c.ActiveStatusEffects.Keys);

        foreach (StatusEffects key in keys)
        {
            if (c.ActiveStatusEffects[key] > 0)
            {
                switch (key)
                {
                    case StatusEffects.Burn:
                        c.DrainHealth(Math.Max((int)(c.MaxHealth * 0.09), 10));
                        break;
                    case StatusEffects.Poison:
                        c.DrainHealth(Math.Max((int)(c.MaxHealth * 0.11), 12));
                        break;
                    case StatusEffects.HPRegeneration:
                        c.AddHealth(Math.Max((int)(c.MaxHealth * 0.08), 10));
                        break;
                    case StatusEffects.ManaRegenetation:
                        c.AddMana(Math.Max((int)(c.MaxMana * 0.25), 5));
                        break;
                }
                c.ActiveStatusEffects[key] -= 1;
            }
```

```
            else
            {
                c.ActiveStatusEffects.Remove(key);
            }
        }
        c.ValidateStatsCombat();
        c.UpdateControlValues();
    }
}

public static void ApplyStatusEffect(Character source, Character target, StatusEffects statuseffect, int turnDuration)
{
    if (source == target)
    {
        turnDuration++;
    }
    if (target.ActiveStatusEffects.Keys.Contains(statuseffect))
    {
        target.ActiveStatusEffects[statuseffect] += turnDuration;
    }
    else
    {
        target.ActiveStatusEffects.Add(statuseffect, turnDuration);
    }
    source.Controls.RefreshStatListBox();
    target.Controls.RefreshStatListBox();
}

public enum StatusEffects
{
    Poison,
    Burn,
    Weak,
    Strength,
    HPRegeneration,
    ManaRegenetation,
    Invisible,
    Invincible
}
```

```
public enum SpecialEffect
{
    None,
    DrainMana, // removes mana from target
    Bullseye, // 0% miss chance
    ConfirmCrit, // 100% crit chance
    Piercing, // ignores armour
    Frost, // (Slow)
    SplashR, SplashL, SplashLR, SplashAll,

    // all below apply status effect
    Burn,
    Poison,
    Weaken,
    Strengthen,
    HPRegen,
    ManaRegen,
}
```