ソフトウェアアーキテクチャ 特論 期末レポート

M24J4045A 中邑熙正

2025 年 2 月 20 日

1　はじめに

1.1　レポートの目的

このレポートでは、ソフトウェアアーキテクチャ特論の授業で学んだ知識をもとに、Web アプリケーションとゲームの両方に適用し、その適用事例を整理することである。

1.2　開発したソフトウェアの概要

1.2.1　Web アプリケーション

今回作成したのは n 年日記という Web アプリケーションである。n 年日記は、ユーザが日記を書くことができる Web アプリケーションである。n 年日記の開発に当たって、10 年日記というアプリケーションを参考にした。10 年日記とは〜〜アプリケーションである。通常の日記アプリケーションと異なり、n 年日記はその日付の未来の日記を一覧表示することができる。例えば、2021 年 1 月 1 日の日記を書くと、2022 年 1 月 1 日、2023 年 1 月 1 日、・・・、2030 年 1 月 1 日の日記も一覧表示される。しかし 10 年日記を利用していて感じた不便な点として、その日付の過去の日記ではなく、未来の日記が一覧表示される点があった。そのため n 年日記では、その日付の過去の日記が一覧表示されるように改良したアプリケーションを作成した。

1.2.2　ゲーム

今回扱うゲームは、増殖する細菌・微生物たちを操作して、敵の微生物を倒す 2D アクションゲームである。以前から個人開発していたものであり、今回はこのゲームをソフトウェアアーキテクチャの観点から再設計する。

2 Ruby on Rails

n 年日記の開発では、フレームワークに Ruby on Rails を用いている。以下に、Ruby on Rails の概要と特徴について説明する。Ruby on Rails は、Web アプリケーションフレームワークの一つである。Ruby on Rails は、Ruby というプログラミング言語で書かれており、MVC（Model-View-Controller）アーキテクチャを採用している。Ruby on Rails は、Convention over Configuration（規約より設定）という設計方針を持っており、開発者が設定を行う必要がある部分を減らすことで、開発効率を向上させている。また、DRY（Don't Repeat Yourself）の原則に基づいており、同じコードを繰り返さないようにすることで、保守性を高めている。以下に、Ruby on Rails をソフトウェアアーキテクチャの観点から解説する。

2.1 MVC アーキテクチャ

Ruby on Rails の MVC 構造について。データの流れを図示する。デザインパターンの集合体。

2.2 Convention over Configuration（規約より設定）

Ruby on Rails の「規約に従うことで設定を減らす」設計方針について。

2.3 DRY（Don't Repeat Yourself）の原則

同じコードを繰り返さないようにする原則について。

2.4 マイグレーション

マイグレーションとは、データベースのスキーマを変更するための仕組みである。マイグレーションを用いることで、データベースのバージョン管理を行うことができる。カプセル化の概念に基づいている。

2.5 Active Record

Active Record とは、Ruby on Rails の MVC アーキテクチャの中の Model に相当する部分である。Active Record は、データベースとのやり取りを行うためのクラスであり、データベースのテーブルと 1 対 1 で対応している。Active Record を用いることで、データベースの操作を簡単に行うことができる。DB とのやり取りを抽象化することで、データベースの変更に強くなる。

3 Web アプリケーション n 年日記

n 年日記の開発では、フレームワークに Ruby on Rails、データベースに MySQL を用いた。デザインには css を用いている。

3.1 設計

システム構成図シーケンス図、クラス図などモデルとして DiaryEntry クラス、コントローラーとして DiaryEntryController クラス。その他 View など。

3.2 実装

実際のコードを示す。日記画面のスクショなど

4 ゲーム

ゲームエンジンである Unity と C#を用いて開発した。

4.1 設計

ソフトウェアアーキテクチャの設計原則に基づいて、ゲームの設計を行うにあたって、今回は GoF の 23 種類のデザインパターンの中から、Strategy パターンと Observer パターンを適用した。

4.1.1 Strategy パターン

Strategy パターンとは、アルゴリズムを切り替えることができるようにするためのデザインパターンである。Strategy パターンを用いることで、アルゴリズムの切り替えを容易に行うことができる。このゲームでは、敵キャラクターごとに異なる攻撃パターンを持っている。例えば、プレイヤーに体当たりした時にダメージを与えるだけの敵キャラクターと、爆発を起こしてプレイヤーにダメージを与える敵キャラクターがいる。Strategy パターンを用いることで、敵キャラクターごとに異なる攻撃パターンを実装することができる。ここでは、Strategy パターンを用いて敵キャラクターの攻撃パターンを実装する方法を説明する。

インターフェース AtttackBehavior を定義それを実装したクラス NormalAttack, ExplosionAttack を作成敵キャラクタークラスに AttackBehavior attackBehaviour を持たせるそして attackBehaviour の Attack メソッドを呼び出すこれにより、敵キャラクターごとに異なる攻撃パターンを実装することができる。attackBehaviour の中身が変わることで、敵キャラクターの攻撃パターンを変更することができる。これによりポリモーフィズムを実現する。

クラス図を示す。

4.1.2 Observer パターン

Observer パターンとは、オブジェクトの状態変化を他のオブジェクトに通知するためのデザインパターンである。Observer パターンを用いることで、オブジェクト間の依存関係を減らすことができる。

Observer パターンの例を図として示す。

このゲームでは、プレイヤー側のキャラクターは選択して動かせる。しかし、選択されている

キャラクターの中で、あるキャラクターが死んでしまった場合、その集団を移動させる時に null を参照してしまい、エラーが発生してしまう。そのため、そのキャラクターを即座に配列から削除する処理を実装しなければならない。そこで、Observer パターンを用いて、キャラクターの死亡イベントを購読することで、死亡したキャラクターを配列から削除する処理を実装する。

ここでシーケンス図を示す。

コードを用いて説明する。まずキャラクターの死亡イベントを購読する。C#では、Subject クラスが定義されている。Subject クラスは、Observer パターンの Subject に相当するクラスであり、キャラクターの死亡イベントを購読するためのメソッドを持っている。キャラクターの HP が 0 になると、死亡イベントが発行される。するとオブザーバに死亡イベントが通知され、オブザーバはそのキャラクターを配列から削除する処理を行う。

4.1.3 State パターン

4.2 実装

実際のコードを示す。ゲーム画面のスクショなど

4.2.1 Strategy パターンの実装

以下に、Strategy パターンを用いた敵キャラクターの攻撃パターンの実装例を示す。

```
// インターフェースの定義
public interface IAttackBehavior {
    void Attack();
}

// クラスの実装NormalAttack
public class NormalAttack : IAttackBehavior {
    public void Attack() {
        // 通常攻撃の実装
        Console.WriteLine("Normal Attack!");
    }
}

// クラスの実装ExplosionAttack
public class ExplosionAttack : IAttackBehavior {
    public void Attack() {
        // 爆発攻撃の実装
        Console.WriteLine("Explosion Attack!");
    }
}

// 敵キャラクタークラスの実装
public class Enemy {
    private IAttackBehavior attackBehavior;

    public Enemy(IAttackBehavior attackBehavior) {
        this.attackBehavior = attackBehavior;
```

```
    }

    public void PerformAttack() {
        attackBehavior.Attack();
    }

    public void SetAttackBehavior(IAttackBehavior attackBehavior) {
        this.attackBehavior = attackBehavior;
    }
}
```

Listing 1　Strategy パターンの実装例

4.2.2　Observer パターンの実装

以下に、Observer パターンを用いたキャラクターの死亡イベントの実装例を示す。

```

    /// <summary>
    /// キャラクターのステータスを管理するクラス
    /// </summary>
    public class Status : MonoBehaviour, IStatus
    {
        [SerializeField] private CharacterData characterData;
        [SerializeField] private ParticleSystem deadEffect;

        public ReactiveProperty<int> Hp { get; private set; }

        // キャラクターが死んだことを通知するイベント
        public IObservable<Unit> OnDie => onDie;
        private Subject<Unit> onDie = new Subject<Unit>();

        //ステータスの初期化
        void Awake()
        {
            Hp = new ReactiveProperty<int>(characterData.Hp);//を使う必要
                性ReactiveProperty
        }

        public virtual void TakeDamage(int amount, string attackerTag)
        {
            Hp.Value -= amount;
            if (Hp.Value <= 0)
            {
                Die();
            }
        }

        protected virtual void Die()
        {
            Instantiate(deadEffect, transform.position, Quaternion.identity);
            onDie.OnNext(Unit.Default); // 死亡イベントを発行
            onDie.OnCompleted(); // イベントの完了を通知
            Destroy(gameObject);
```

```
        }
    }

/// <summary>
/// プレイヤーが選択したキャラクターを操作するクラス
/// </summary>
public class PlayerCommand : MonoBehaviour {
    private ReactiveCollection<GameObject> selectedCharacters
        = new ReactiveCollection<GameObject>();
    private Dictionary<GameObject, IDisposable> characterSubscriptions
        = new Dictionary<GameObject, IDisposable>();
    [SerializeField] private float selectionRadius = 2f;


    /// 中略


    /// <summary>
    /// クリック位置付近のキャラクターを選択して停止させるメソッド
    /// </summary>
    private void SelectCharactersInRadius(Vector3 mousePosition, Vector3 worldPosition
        ){

        GameObject[] characters = GameObject.FindGameObjectsWithTag("Player");
        foreach (GameObject character in characters)
        {
            float distance = Vector3.Distance(character.transform.position,
                worldPosition);
            if (distance > selectionRadius) continue;

            Status status = character.GetComponent<Status>();
            if (status == null) continue;

            selectedCharacters.Add(character);

            // 選択されたキャラクターのスプライトを表示
            var indicator = character.GetComponent<SelectionIndicator>();
            if (indicator != null)
            {
                indicator.Show();
            }







            // 選択範囲内のキャラクター数の変化を把握するために死亡イベントを購読
            if (!characterSubscriptions.ContainsKey(character))
            {
                var subscription = status.OnDie
                    .Subscribe(_ =>{
                        selectedCharacters.Remove(character);
                        Debug.Log("キャラクターが死んだため、selectedCharacters から削除しました。");
```

```
                        characterSubscriptions.Remove(character);
                    })
                    .AddTo(this);


                characterSubscriptions[character] = subscription;
            }

            //キャラクターを停止
            Rigidbody2D rb = character.GetComponent<Rigidbody2D>();
            if (rb == null) continue;
            rb.velocity = Vector2.zero;

            status.SetState(PlayerState.Selected);
        }
    }
}

\subsubsection{パターンの実装State}
```

Listing 2　Observer パターンの実装例

5　まとめ

レポートの総括今後の課題や展望

6　感想

この授業で学んだことを活かしつつ、新たな設計手法を学ぶことで技術を磨き、今後の開発に生かしていきたい。きれいなプログラムを書けるようになりたい。

7　参考文献

参考文献

[1]　著者名, Head First デザインパターン, 出版社, 出版年.

[2]　著者名, 「論文タイトル」, 雑誌名, 巻 (号), ページ, 出版年.

[3]　著者名, 「記事タイトル」, URL, アクセス日.