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 21 日

目次

1 はじめに・・・ 3
1.1 レポートの目的・・・ 3
1.2 開発したソフトウェアの概要・・・ 3
1.2.1 Web アプリケーション・・・ 3
1.2.2 ゲーム・・・ 3
2 Ruby on Rails・・・ 3
2.1 MVC アーキテクチャ・・・ 3
2.2 データベース管理とマイグレーション・・・ 4
2.3 Active Record・・・ 5
3 Web アプリケーション n 年日記・・・ 5
3.1 設計・・・ 5
3.2 実装・・・ 5
4 ゲーム・・・ 10
4.1 設計・・・ 10
4.1.1 Strategy パターン・・・ 10
4.1.2 Observer パターン・・・ 12
4.2 実装・・・ 14
4.2.1 Strategy パターンの実装・・・ 14
4.2.2 Observer パターンの実装・・・ 18
5 考察・・・ 23
6 まとめ・・・ 23
7 参考文献・・・ 24

1　はじめに

1.1　レポートの目的

このレポートの目的は、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 [1] というアプリケーションを参考にした。10 年日記は通常の日記アプリとは異なり、ある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その日付の過去の日記が表示されないことが不便だと感じた。むしろ、その日付の過去の日記を見返すことができる方が便利だと考えたため、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する。

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。Ruby on Rails（以下、Rails）は、MVC アーキテクチャに基づく Web アプリケーションフレームワークであり、開発の効率性を向上させるための機能が充実している。以下に、Rails の特徴をいくつか紹介する。

2.1　MVC アーキテクチャ

Rails は Model-View-Controller（MVC）アーキテクチャ を採用しており、Web アプリケーションの構造を明確に分離することで、拡張性と保守性を向上させている。

Model（モデル） はアプリケーションのデータやビジネスロジックを管理する。View（ビュー）はユーザーとのインターフェースを担当する。Controller（コントローラ） は Model と View の橋渡しを行い、リクエストの処理とデータの制御を行う。この分割により、各コンポーネントが独立

して開発・変更できるため、スケーラビリティが向上する。例えば、データベースのスキーマを変更する場合でも、View のコードには影響を与えずに Model 層のみを修正すればよい。

MVC アーキテクチャにおけるデータの流れを図 1 に示す。

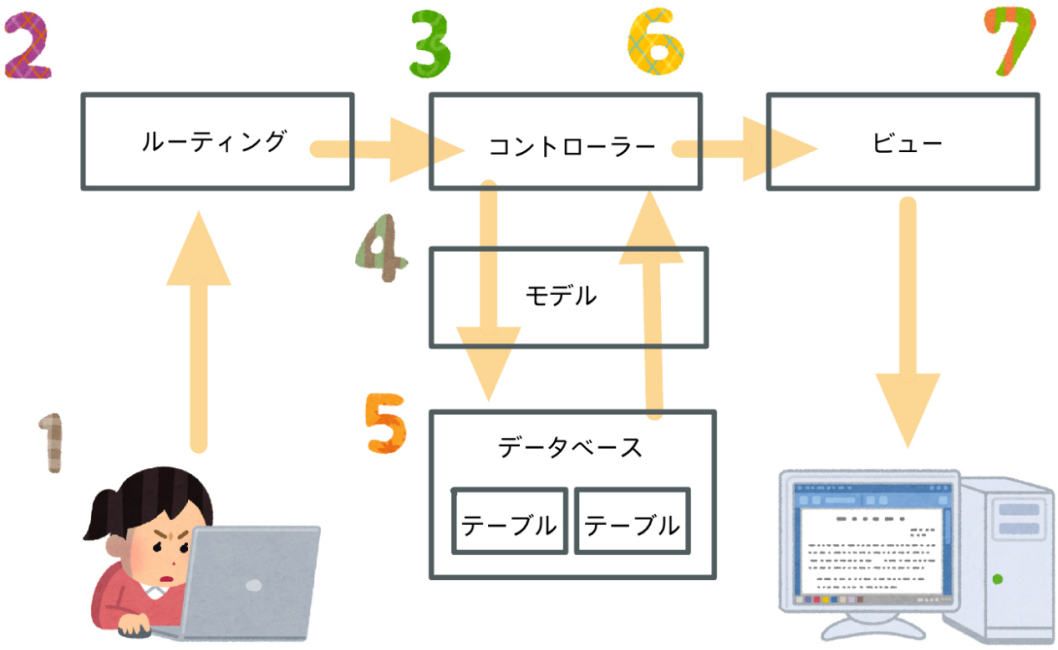

図 1　MVC アーキテクチャ

アプリケーションの処理は、ユーザーの操作によるリクエストの送信から始まる。まず、ルーティングによってリクエストが解析され、適切なコントローラーに振り分けられる。コントローラーはアプリケーションの処理を制御し、必要に応じてモデルにデータの取得を依頼する。モデルはデータの管理とデータベースとのやり取りを担当し、コントローラーからの要求に応じてデータを取得または加工し、それを返す。コントローラーは受け取ったデータをビューに渡し、ビューはそのデータをもとに画面を生成し、ユーザーに表示する。MVC を採用することで、役割が明確に分かれ、コードの管理が容易になり、開発の拡張性やメンテナンス性が向上する。

2.2　データベース管理とマイグレーション

Rails は ActiveRecord を用いた ORM（オブジェクト・リレーショナル・マッピング） により、データベースとのやりとりを抽象化している。これにより、開発者は SQL を直接書かずにデータを操作できる。 また、マイグレーション機能を活用することで、データベースの構造変更が簡単に行える。データベースの変更履歴をバージョン管理できるため、どのような変更が行われたのかを記録し、必要に応じてロールバックも可能である。またデータベースの知識がなくても構造変更ができるため、Rails 側で rails db:migrate を実行するだけで、自動的にテーブルを作成・更新でき

る。このように Rails のデータベース管理機能は開発のスピードを向上させつつ、システムの一貫性を保つ設計になっている。

2.3 Active Record

Active Record とは、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 SQLite を用いた。

3.1 設計

DB 設計を表 1 に示す。

表 1　diary_entries テーブルのカラム定義

カラム名	データ型	説明
id	整数型 (主キー)	各日記エントリの一意の識別子
content	テキスト型	日記の内容
date	日付型	日記の記録日
created_at	日時型	レコードが作成された日時
updated_at	日時型	レコードが最後に更新された日時

上記のテーブルは、日記エントリの情報を管理するために使用される。id は各エントリを一意に識別するための主キーであり、date カラムには日記の記録日を格納する。created_at および updated_at は、データの作成・更新時刻を管理する。

3.2 実装

実装の一部を示す。モデルとして DiaryEntry クラスをリスト 1 に示す。

```
class DiaryEntry < ApplicationRecord
  def self.fetch_entries(selected_date, n_years)
    (0..n_years-1).map do |years_ago|
      target_date = selected_date - years_ago.years
      find_or_initialize_by(date: target_date)
    end.reverse
  end
end
```

リスト 1　DiaryEntry クラス

コントローラにあたる DiaryEntriesController は、日記エントリの表示、作成、編集、更新を担当している。以下に、主要なアクションの説明を示す。

- **index**: 指定した日付に基づいて過去 n 年分の日記を表示
- **new**: 新規日記作成画面を表示
- **create**: 新しい日記エントリを作成
- **edit**: 日記エントリの編集画面を表示
- **update**: 日記エントリを更新

DiaryEntriesController クラスの実装をリスト 2 に示す。

```
class DiaryEntriesController < ApplicationController
# メイン画面（指定した日付に基づいて過去年分の日記を表示）n
def index
  # が指定されていない場合、今日の日付を使用selected_date
  @selected_date = params[:selected_date] ? Date.parse(params[:selected_date]) :
      Date.today

  # が指定されていない場合、の月の開始日を使用start_dateselected_date
  @start_date = params[:start_date] ? Date.parse(params[:start_date]) :
      @selected_date.beginning_of_month

  @n_years = 5 # 過去何年分表示するか

  # 過去年分の日記データを取得n
  @diary_entries = DiaryEntry.fetch_entries(@selected_date, @n_years)
end

# 日記の新規作成画面
def new
  @diary_entry = DiaryEntry.new(date: params[:date])
end

# 日記の作成
def create
  @diary_entry = DiaryEntry.new(diary_entry_params)
  if @diary_entry.save
    redirect_to diary_entries_path(selected_date: @diary_entry.date), notice: '日記が
        作成されました。'
  else
    flash.now[:alert] = '日記の作成に失敗しました。'
    flash.now[:alert] += @diary_entry.errors.full_messages.join(", ")
    render :new
  end
end

# 日記の編集画面
def edit
  @diary_entry = DiaryEntry.find(params[:id])
end

# 日記の更新
```

```
39   def update
40     @diary_entry = DiaryEntry.find(params[:id])
41     if @diary_entry.update(diary_entry_params)
42       redirect_to diary_entries_path(selected_date: @diary_entry.date), notice: '日記が
           更新されました。'
43     else
44       flash.now[:alert] = '日記の更新に失敗しました。'
45       flash.now[:alert] += @diary_entry.errors.full_messages.join(", ")
46       render :edit
47     end
48   end
49
50   private
51
52   def diary_entry_params
53     params.require(:diary_entry).permit(:content, :date)
54   end
55 end
```

リスト 2　DiaryEntryController クラス

ビューとして index.html.erb をリスト 3 に示す。

```
1  <h1>年日記n</h1>
2  <div class="container">
3    <div class="left-column">
4      <div class="diary-entries">
5        <h2><%= @selected_date.strftime('%年Y %月m %日d') %> の過去<%= @n_years 年分%></h2>
6        <% @diary_entries.reverse.each do |entry| %>
7          <div id="diary-entry-<%= entry.id %>" class="diary-entry">
8            <p><%= entry.date.strftime('%年Y %月m %日d (%A)') %></p>
9            <% if entry.persisted? && entry.content.present? %>
10             <!-- 日記が存在し、内容がある場合 -->
11             <p><%= entry.content %></p>
12             <%= link_to '編集', edit_diary_entry_path(entry), class: 'edit-link', data:
                   { id: entry.id } %>
13           <% elsif entry.persisted? %>
14             <!-- 日記が存在するが、内容がない場合 -->
15             <%= link_to '編集', edit_diary_entry_path(entry), class: 'edit-link', data:
                   { id: entry.id } %>
16           <% else %>
17             <!-- 日記が存在しない場合 -->
18             <%= link_to '新規作成', new_diary_entry_path(date: entry.date), class: '
                   edit-link', data: { id: entry.id } %>
19           <% end %>
20         </div>
21       <% end %>
22     </div>
23   </div>
24
25   <div class="right-column">
26     <div class="calendar">
27       <%= month_calendar(start_date: @start_date, previous_label: nil, next_label: nil
           ) do |date| %>
28         <div class="day">
```

```
29            <% if date == @selected_date %>
30              <!-- 選択中の日付 -->
31              <strong class="selected-date"><%= link_to date.day, diary_entries_path(
                    selected_date: date) %></strong>
32            <% elsif date == Date.today %>
33              <!-- 今日の日付 -->
34              <span class="today-date"><%= link_to date.day, diary_entries_path(
                    selected_date: date) %></span>
35            <% else %>
36              <!-- 通常の日付 -->
37              <%= link_to date.day, diary_entries_path(selected_date: date) %>
38            <% end %>
39          </div>
40        <% end %>
41      </div>
42
43      <div class="navigation">
44        <%= link_to 'Previous Year', diary_entries_path(start_date: @start_date.
              beginning_of_year - 1.year, selected_date: @selected_date) %> |
45        <%= link_to 'Previous Month', diary_entries_path(start_date: @start_date.
              beginning_of_month - 1.month, selected_date: @selected_date) %> |
46        <%= link_to 'Today', diary_entries_path(start_date: Date.today.
              beginning_of_month, selected_date: Date.today) %> |
47        <%= link_to 'Next Month', diary_entries_path(start_date: @start_date.
              beginning_of_month + 1.month, selected_date: @selected_date) %> |
48        <%= link_to 'Next Year', diary_entries_path(start_date: @start_date.
              beginning_of_year + 1.year, selected_date: @selected_date) %>
49      </div>
50    </div>
51  </div>
```

リスト 3　index.html.erb

図 2 に示すのは、n 年日記アプリケーションのホーム画面である。このアプリケーションは、ユーザーが日記を記録し、過去の同じ日付の日記を一覧表示することができる Web アプリケーションである。

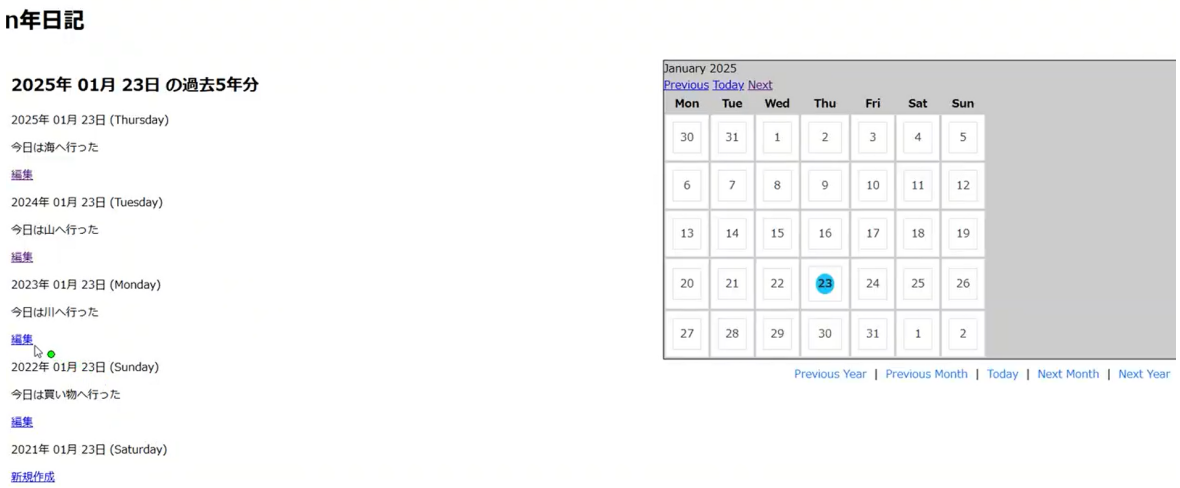

図 2　ホーム画面

ホーム画面は大きく分けて左カラムと右カラムの 2 つのセクションに分かれている。

左カラムには、選択された日付の過去 n 年分の日記エントリが表示される。各日記エントリは以下の情報を含む。

- 日記の記録日
- 日記の内容（存在する場合）
- 編集リンク（内容が存在する場合）
- 新規作成リンク（内容が存在しない場合）

ユーザーは、日記エントリの内容を確認し、必要に応じて編集することができる。また、日記が存在しない場合は、新規作成リンクをクリックして新しい日記を作成することができる。

右カラムには、カレンダーとナビゲーションリンクが表示される。

カレンダーは、選択された月の日付を表示し、ユーザーが特定の日付を選択できるようにする。カレンダーの日付は以下のように表示される。

- 選択中の日付は太字で強調表示される。
- 今日の日付は特別なスタイルで表示される。
- 通常の日付はリンクとして表示され、クリックするとその日付の日記エントリが表示される。

ナビゲーションリンクは、ユーザーが異なる期間の日記エントリを簡単に表示できるようにする。以下のリンクが含まれる。

- 前年の表示

- 前月の表示
- 今日の日付の表示
- 次月の表示
- 次年の表示

これらのリンクを使用することで、ユーザーは簡単に異なる期間の日記エントリを表示し、過去の日記を振り返ることができる。

このように、n 年日記アプリケーションは、ユーザーが日記を記録し、過去の日記を簡単に参照できるように設計されている。ユーザーは、日記の内容を編集したり、新しい日記を作成したりすることができ、日々の出来事を記録し続けることができる。

4 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。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、ゲームが抱えている課題を解決するためにデザインパターンを適用する。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。デザインパターンについては [3] を参考にした。

4.1.1 Strategy パターン

Strategy パターンは、一連のアルゴリズムを定義してカプセル化し交換できるようにするデザインパターンである。Strategy パターンを使うと、アルゴリズムを利用するクライアントとは独立して、アルゴリズムを変更することができる。このゲームでは、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、敵に体当たりした時にダメージを与えるだけのキャラクターや、攻撃時に自爆して敵を道連れにするキャラクターがいる。Strategy パターンを用いることで、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。

今回作成した Strategy パターンに関連するクラス図を図 3 に示す。

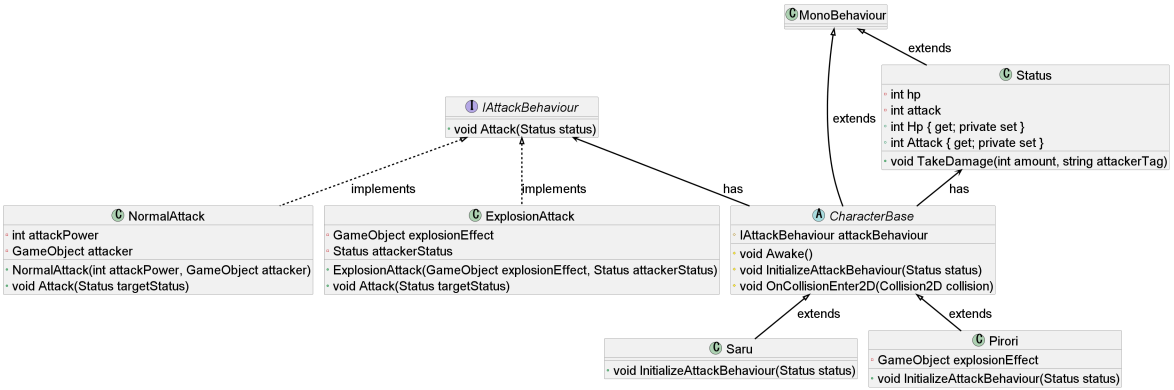

図 3　Strategy パターンのクラス図

各クラスの説明を表 2 に示す。

クラス名	役割
IAttackBehaviour	攻撃の振る舞いを定義するインターフェース
NormalAttack	通常攻撃の振る舞いを実装するクラス
ExplosionAttack	爆発攻撃の振る舞いを実装するクラス
CharacterBase	キャラクターの基底クラス (抽象クラス)
Saru	通常攻撃を行うキャラクターのクラス
Pirori	爆発攻撃を行うキャラクターのクラス
Status	キャラクターのステータスを管理するクラス
MonoBehaviour	Unity のすべてのスクリプトの基本クラス。 このクラスを継承することで Start(), Awake(), OnCollisionEnter2D() などの Unity が提供するメソッドを使用できるようになる。

表 2　Strategy パターンに関連する各クラスの役割

まず攻撃の振る舞いを定義する IAtttackBehavior インターフェースを定義する。IAttackBehavior を実装したクラスとして、通常攻撃の振る舞いを実装する NormalAttack クラス, 爆発攻撃の振る舞いを実装する ExplosionAttack クラスを定義する。続いて、キャラクターの基底クラスとなる CharacterBase クラスを定義し、AttackBehavior 型のインスタンス変数 attackBehaviour を保持させる。CharacterBase クラスを継承したキャラクタークラスとして、Saru クラスと Pirori クラスを定義する。これらのクラスの初期化の際に任意の攻撃パターンを設定する。今回は Saru クラスは NormalAttack クラス、Pirori クラスは ExplosionAttack クラスを攻撃パターンとして設定するものとする。攻撃実行時は attackBehaviour の Attack メソッドを呼び出すことで、キャラクターに設定された攻撃パターンが実行される。Attack メソッドの具体的な処理は NormalAttack クラスと ExplosionAttack クラスに委譲されている。もし攻撃パターンを変更したい場合は、SetAttackBehavior メソッドを呼び出すことで、attackBehaviour の中身を変更することができる。

また、Status クラスはキャラクターのステータスを管理するクラスである。ここではインスタンス変数 hp, attack や TakeDamage メソッドといったダメージ関連処理のみを記述している。加えて、MonoBehaviour クラスは Unity のすべてのスクリプトの基本クラスであり、キャラクターがゲーム内オブジェクトとして動作するために継承する必要がある。

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。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。

今回実装する Observer パターンのクラス図を図 4 に示す。

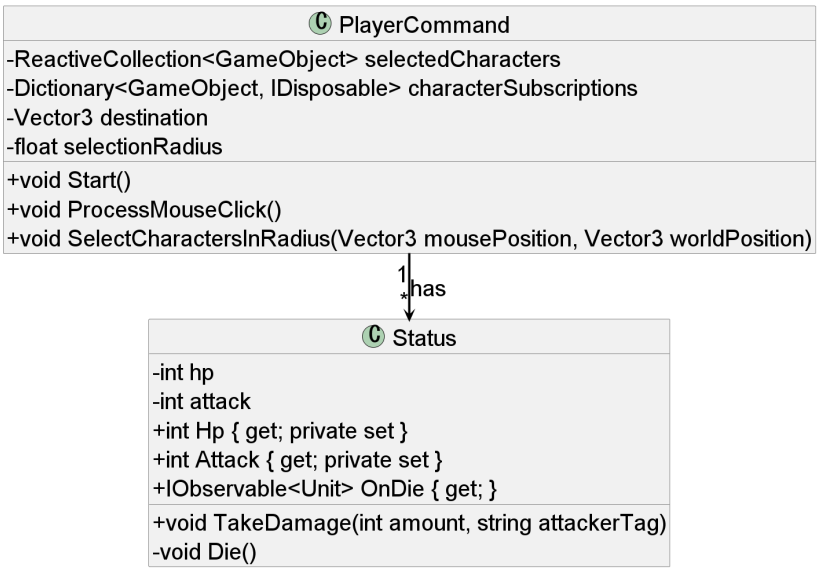

図 4　Observer パターンのクラス図

PlayerCommand クラスは、プレイヤーが選択したキャラクターを操作するためのクラスである。レイヤーがクリックした位置にあるキャラクターを選択し、その次にクリックした位置に移動させることができる。Status クラスはキャラクターのステータスを管理するクラスである。ここでは両クラスともに、死亡イベントに関連する処理のみを記述し、それ以外の処理は省略している。

また、今回実装する Observer パターンのシーケンス図を図 5 に示す。

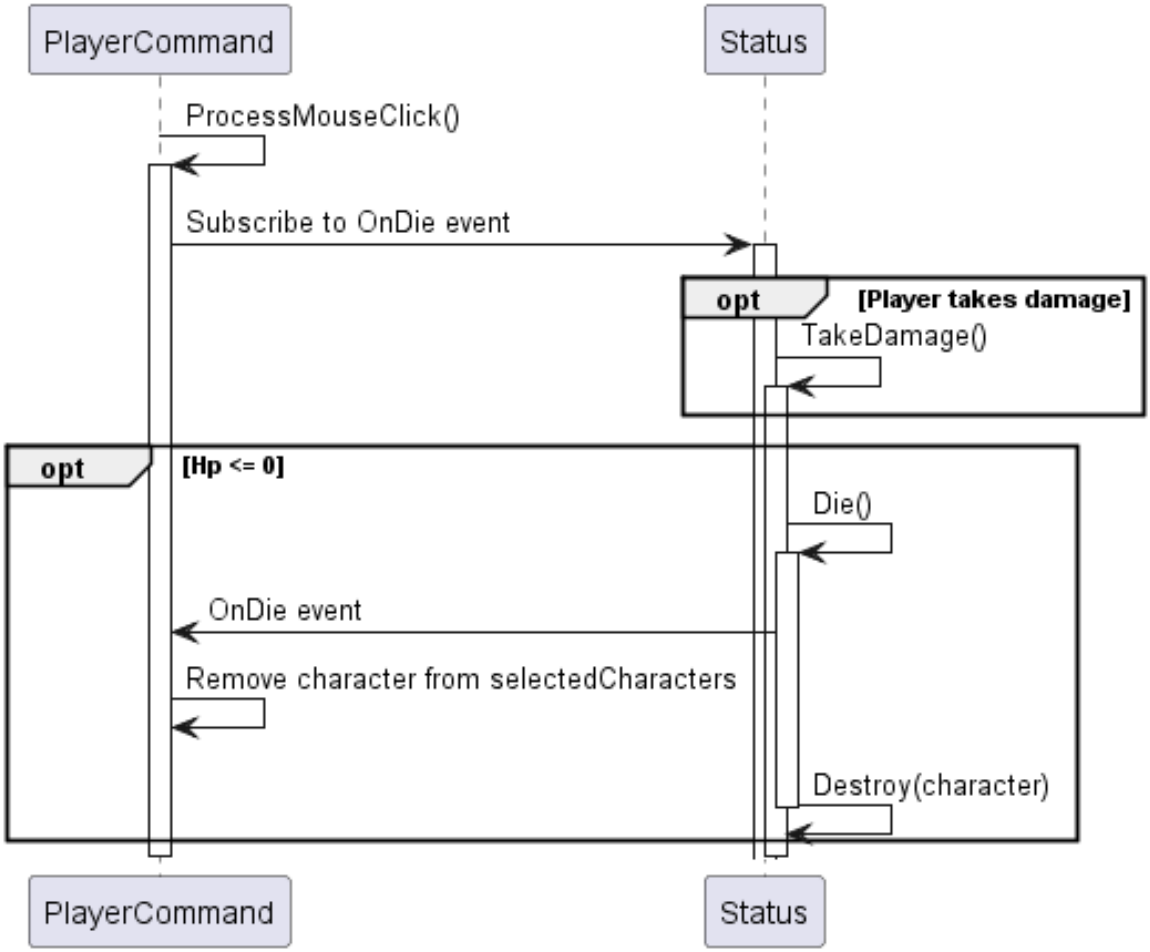

図 5　Observer パターンのシーケンス図

図 5 に示すシーケンス図は、Observer パターンを用いたキャラクターの死亡イベントの管理を表している。本シーケンス図では、PlayerCommand クラスがキャラクターの死亡イベントを購読し、キャラクターが死亡した際に selectedCharacters コレクションから削除する処理を実装している。

1. **マウスクリックの処理** PlayerCommand クラスの ProcessMouseClick メソッドが呼び出され、プレイヤーがクリックした位置にあるキャラクターを選択する。
2. **死亡イベントの購読** PlayerCommand クラスが Status クラスの OnDie イベントを購読することで、キャラクターが死亡した際に通知を受け取る準備をする。
3. **ダメージの受け取り** キャラクターが攻撃を受けた場合、Status クラスの TakeDamage メソッドが呼び出され、キャラクターの HP が減少する。
4. **HP のチェックと死亡処理** TakeDamage メソッド実行後、キャラクターの HP が 0 以下になった場合、Status クラスの Die メソッドが実行され、OnDie イベントが発行される。
5. **死亡イベントの発行** Die メソッド内で OnDie イベントが発行され、PlayerCommand クラスに通知される。

6. **キャラクターの削除** PlayerCommand クラスは OnDie イベントを受け取ると、死亡したキャラクターを selectedCharacters コレクションから削除し、さらに Status クラスで Destroy メソッドを呼び出してキャラクターを削除する。

このように Observer パターンを利用することで、PlayerCommand クラスはキャラクターの死亡を監視し、死亡時に適切な処理を実行できる仕組みを構築している。

4.2 実装

4.2.1 Strategy パターンの実装

ここでは Strategy パターンに関連するクラスの実装を示す。まず攻撃の振る舞いを定義する IAtttackBehavior インターフェースをリスト 4 に示す。

```
public interface IAttackBehaviour
{
    void Attack(Status status);
}
```

リスト 4　IAttackBehaviour インターフェース

IAttackBehavior を実装したクラスとして、通常攻撃の振る舞いを実装する NormalAttack クラス, 爆発攻撃の振る舞いを実装する ExplosionAttack クラスをそれぞれリスト 5 とリスト 6 に示す。

```
using UnityEngine;

/// <summary>
/// 通常攻撃
/// </summary>
public class NormalAttack : IAttackBehaviour
{
    private int attackPower;
    private GameObject attacker;

    public NormalAttack(int attackPower, GameObject gameObject)
    {
        this.attackPower = attackPower;
        this.attacker = gameObject;
    }

    public void Attack(Status targetStatus)
    {
        if (targetStatus != null && attacker.tag != targetStatus.gameObject.tag)
        {
            targetStatus.TakeDamage(attackPower, attacker.tag);
        }
    }
}
```

リスト 5　NormalAttack クラス

```
using UnityEngine;
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;

/// <summary>
/// ぶつかった敵とともに爆発
/// </summary>
public class ExplosionAttack : IAttackBehaviour
{
    private GameObject explosionEffect;
    private Status attackerStatus;
    private const string explosionEffectAddress = "省略";

    public ExplosionAttack(Status attackerStatus)
    {
        this.attackerStatus = attackerStatus;
        LoadExplosionEffect();
    }

    private void LoadExplosionEffect()
    {
        // 省略
    }

    private void OnExplosionEffectLoaded(AsyncOperationHandle<GameObject> obj)
    {
        // 省略
    }

    public void Attack(Status targetStatus)
    {
        if (targetStatus != null && explosionEffect != null)
        {
            GameObject.Instantiate(explosionEffect, targetStatus.transform.position,
                Quaternion.identity);
            GameObject.Destroy(targetStatus.gameObject); // 敵を破壊
            GameObject.Destroy(attackerStatus.gameObject); // 自分自身も破壊
        }
    }
}
```

リスト 6　ExplosionAttack クラス

次に、キャラクターの基底クラスとなる CharacterBase クラスをリスト 7 に示す。

```
using UnityEngine;

/// <summary>
/// キャラクターの基底クラス
/// </summary>
public abstract class CharacterBase : MonoBehaviour
{
    protected IAttackBehaviour attackBehaviour;
    private Status status;
```

```
10
11      /// <summary>
12      /// 開始時に呼び出されるメソッド
13      /// </summary>
14      protected void Awake()
15      {
16          status = GetComponent<Status>();
17          if (status != null)
18          {
19              InitializeAttackBehaviour(status);
20          }
21          else
22          {
23              Debug.LogError("Status component not found on " + gameObject.name);
24          }
25      }
26
27      /// <summary>
28      /// 攻撃パターンを初期化するメソッド
29      /// </summary>
30      protected abstract void InitializeAttackBehaviour(Status status);
31
32      protected virtual void OnCollisionEnter2D(Collision2D collision)
33      {
34          if (collision.gameObject.CompareTag("Enemy"))
35          {
36              var targetStatus = collision.gameObject.GetComponent<Status>();
37              if (targetStatus != null)
38              {
39                  attackBehaviour.Attack(targetStatus);
40              }
41          }
42      }
43
44      /// <summary>
45      /// 攻撃パターンを設定するメソッド
46      /// </summary>
47      public void SetAttackBehaviour(IAttackBehaviour attackBehaviour)
48      {
49          this.attackBehaviour = attackBehaviour;
50      }
51  }
```

リスト 7　CharacterBase クラス

Saru クラスと Pirori クラスの初期化の際に、Saru クラスは NormalAttack クラス、Pirori クラスは ExplosionAttack クラスを攻撃パターンとして設定する。実装をリスト 8 とリスト 9 に示す。Strategy パターンを適用しているため、コメントアウトしている部分を有効にすることで攻撃パターンを変更することができる。

```
1   public class Saru : CharacterBase
2   {
3       protected override void InitializeAttackBehaviour(Status status)
```

```
    {
        SetAttackBehaviour(new NormalAttack(status.Attack, gameObject));
        // 例えば以下のように書き換えると攻撃パターンを爆発攻撃に変更することができる
        // SetAttackBehaviour(new ExplosionAttack(status));
    }
}
```

リスト 8　Saru クラス

```
public class Pirori : CharacterBase
{
    protected override void InitializeAttackBehaviour(Status status)
    {
        SetAttackBehaviour(new ExplosionAttack(status));
        // 例えば以下のように書き換えると攻撃パターンを通常攻撃に変更することができる
        // SetAttackBehaviour(new NormalAttack(status.Attack, gameObject));
    }
}
```

リスト 9　Pirori クラス

キャラクターのステータスを管理する Status クラスの実装をリスト 10 に示す。なおダメージ関連処理以外の変数、メソッドは省略している。

```
using System;
using System.Collections;
using UnityEngine;
using UniRx;

/// <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
{

    private int hp;
    public int Hp
    {
        get => hp;
        private set
        {
            hp = value;
            if (hp <= 0)
            {
                Die();
            }
        }
    }

    private int attack;
    public int Attack
    {
        get => attack;
        private set => attack = value;
```

```
31      }
32
33      /// <summary>
34      /// ダメージを受けるメソッド
35      /// </summary>
36      public virtual void TakeDamage(int amount, string attackerTag)
37      {
38          Hp -= amount;
39      }
40
41      // 省略
42  }
```

リスト 10　Status クラス

以上のようにして、Strategy パターンを用いてキャラクターの攻撃パターンを実装した。Saru による通常攻撃と Pirori による爆発攻撃の様子を図 6 と図 7 に示す。

図 6　Saru による通常攻撃

図 7　Pirori による爆発攻撃

また攻撃パターンを入れ替えてみても、正常に動作することが確認できた。

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を示す。まず、PlayerCommand クラスの実装をリスト 11 に示す。赤文字の部分が今回 Observer パターンを適用するにあたって注目すべき部分である。

```
1  using System;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using UnityEngine.EventSystems;
5  using UniRx;
6  using UniRx.Triggers;
7
```

```
8  /// <summary>
9  /// キャラクターの選択とコマンドの発行を管理するスクリプト
10 /// </summary>
11 public class PlayerCommand : MonoBehaviour
12 {
13     [SerializeField] private GameObject uiManager;
14     private ButtonManagerScript buttonManagerScript;
15     private ReactiveCollection<GameObject> selectedCharacters = new ReactiveCollection
           <GameObject>();
16     private Dictionary<GameObject, IDisposable> characterSubscriptions = new
           Dictionary<GameObject, IDisposable>();
17     private Vector3 destination;
18
19     [SerializeField] private float selectionRadius = 2f;
20
21     private void Start()
22     {
23         buttonManagerScript = uiManager.GetComponent<ButtonManagerScript>();
24
25         // 左クリックでキャラクター選択または移動先の設定
26         this.UpdateAsObservable()
27             .Where(_ => Input.GetMouseButtonDown(0) && !EventSystem.current.
                   IsPointerOverGameObject())
28             .Where(_ => buttonManagerScript.SelectedButtonType.Value == ButtonType.
                   None)
29             .Subscribe(_ => ProcessMouseClick())
30             .AddTo(this);
31
32         // 右クリックで他のキャラクターの選択を解除
33         this.UpdateAsObservable()
34             .Where(_ => Input.GetMouseButtonDown(1))
35             .Subscribe(_ => buttonManagerScript.ResetOther())
36             .AddTo(this);
37     }
38
39     /// <summary>
40     /// マウスクリック時の処理
41     /// キャラクター選択または移動先の設定を行うメソッド
42     /// </summary>
43     private void ProcessMouseClick()
44     {
45         Vector3 mousePosition = Input.mousePosition;
46         mousePosition.z = Camera.main.nearClipPlane;
47         Vector3 worldPosition = Camera.main.ScreenToWorldPoint(mousePosition);
48         worldPosition.z = 0;
49
50         // 選択されたキャラクターがいない場合、選択範囲内のキャラクターを選択
51         if (selectedCharacters.Count == 0)
52         {
53             SelectCharactersInRadius(mousePosition, worldPosition);
54             return;
55         }
56
57         // 選択されたキャラクターがいる場合、移動先を設定
```

```
58          destination = worldPosition;
59          foreach (GameObject character in selectedCharacters)
60          {
61              if (character == null || !character) continue;
62
63              PlayerMovement mover = character.GetComponent<PlayerMovement>();
64              if (mover != null)
65              {
66                  mover.SetDestination(destination);
67              }
68
69              // 選択解除時にスプライトを非表示にする
70              var indicator = character.GetComponent<SelectionIndicator>();
71              if (indicator != null)
72              {
73                  indicator.Hide();
74              }
75          }
76
77          selectedCharacters.Clear();
78      }
79
80      /// <summary>
81      /// 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
82      /// </summary>
83      private void SelectCharactersInRadius(Vector3 mousePosition, Vector3 worldPosition
            )
84      {
85          //selectionIndicator.Show(mousePosition);
86
87          GameObject[] characters = GameObject.FindGameObjectsWithTag("Player");
88          foreach (GameObject character in characters)
89          {
90              float distance = Vector3.Distance(character.transform.position,
                    worldPosition);
91              if (distance > selectionRadius) continue;
92
93              Status status = character.GetComponent<Status>();
94              if (status == null) continue;
95
96              selectedCharacters.Add(character);
97
98              // 選択されたキャラクターのスプライトを表示
99              var indicator = character.GetComponent<SelectionIndicator>();
100             if (indicator != null)
101             {
102                 indicator.Show();
103             }
104
105             // 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
106             if (!characterSubscriptions.ContainsKey(character))
107             {
108                 var subscription = status.OnDie
109                     .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);
        }
    }
}
```

リスト 11　PlayerCommand クラス

このコードは、Observer パターンを用いてキャラクターの死亡イベントを監視し、適切な処理を実行する仕組みを実装している。まず、status.OnDie を購読することで、キャラクターが死亡したときに通知を受け取る準備をする。OnDie はキャラクターの死亡時に発火するイベントであり、これを購読することで、死亡時に実行する処理を事前に登録することができる。

次に、Subscribe を使用して、OnDie が発生した際の処理を指定する。まず、死亡したキャラクターを選択中のキャラクターリストから削除する。これは、プレイヤーが操作可能なキャラクターのリストから、死亡したキャラクターを適切に除外するために行われる。その後、デバッグ用のログを出力し、キャラクターが削除されたことを記録する。さらに、キャラクターごとに保持されていた購読情報を削除し、不要なイベントの購読を解除する。これにより、死亡したキャラクターに関する処理が不要になったときに、適切にクリーンアップを行うことができる。最後に、AddTo(this) を呼び出し、購読を現在のオブジェクトに紐づける。これによって、オブジェクトが破棄された際に購読も自動的に解除されるため、不要なイベント購読が残ることによるメモリリークを防ぐことができる。

次に、Status クラスの実装をリスト 12 に示す。赤文字の部分が今回 Observer パターンを適用するにあたって注目すべき部分である。

```

using System;
using System.Collections;
using UnityEngine;
using UniRx;

/// <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
{
```

```
12
13      private int hp;
14      public int Hp
15      {
16          get => hp;
17          private set
18          {
19              hp = value;
20              if (hp <= 0)
21              {
22                  Die();
23              }
24          }
25      }
26
27      private int attack;
28      public int Attack
29      {
30          get => attack;
31          private set => attack = value;
32      }
33
34      // キャラクターが死んだことを通知するイベント
35      public IObservable<Unit> OnDie => onDie;
36      private Subject<Unit> onDie = new Subject<Unit>();
37
38      /// <summary>
39      /// ダメージを受けるメソッド
40      /// </summary>
41      public virtual void TakeDamage(int amount, string attackerTag)
42      {
43          Hp -= amount;
44      }
45
46      protected virtual void Die()
47      {
48          Instantiate(deadEffect, transform.position, Quaternion.identity);
49          onDie.OnNext(Unit.Default); // 死亡イベントを発行
50          onDie.OnCompleted(); // イベントの完了を通知
51          Destroy(gameObject);
52      }
53
54  }
```

リスト 12　Observer パターンにおける Status クラス

このコードは、キャラクターの死亡イベントを Observer パターンを用いて管理する仕組みを実装している。OnDie プロパティは IObservable¡Unit¿型として定義され外部から購読可能だが、Subject¡Unit¿の onDie は private であり、外部から直接イベントを発行できない。これにより、死亡イベントの発火は Status クラス内でのみ行われる。

キャラクターがダメージを受けて HP が 0 以下になると Die メソッドが呼ばれる。このメソッドでは、Instantiate により死亡エフェクトを表示し、onDie.OnNext(Unit.Default) によって購読者に

死亡イベントを通知する。続いて onDie.OnCompleted を実行し、イベントの購読が完了したことを示す。最後に Destroy(gameObject) を呼び出してキャラクターをゲーム内から削除する。

この設計により、死亡イベントを購読している PlayerCommand クラスなどが通知を受け取り、適切な処理を実行できる。

以上のように、Observer パターンを用いることで、キャラクターの死亡に伴う処理を他のクラスと疎結合に管理できるようになる。

5 考察

授業内で挙がった抽象クラス不要論について考える。Java や C#などの言語では、インターフェースにデフォルト実装を持たせることができる。共通のメソッドなどをインターフェースに定義しておくことで、同じ処理を複数のクラスで実装する必要がなくなり、コードの重複を防ぐことができる。コードの重複を防ぐことによって、保守性を高めることができる。他にも、Java や C#などの言語では単一クラスしか実装継承できないが、インターフェースを使うことで複数のインターフェースを実装することができる。またインターフェースのデフォルト実装では、メソッドをオーバーライドすることができるため、抽象クラスよりもすべての点においてインターフェースのほうが優れているように思える。

しかし抽象クラスとインターフェースの異なる点として、まず Unity においては、ゲーム内のオブジェクトはすべて MonoBehaviour を継承している必要がある。そのため、もし抽象クラスを使わずにインターフェースだけで実装しようとすると、インターフェースは MonoBehaviour を継承することができないため、ゲーム内でプログラムを動作させることができない。そのため、Unity においては仕様継承と実装継承の両方を使うことができる抽象クラスが有用である。

また、抽象クラスはインターフェースと異なり、フィールドやコンストラクタを持つことができる。そのため、インターフェースだけでは実装できない処理を抽象クラスで実装することができる。今回の例では、抽象クラス CharacterBase は IAttackBehavior 型のインスタンス変数を持ち、その中身を変更することで攻撃パターンを変更することができる。

このように、抽象クラスはインターフェースにない特性を持っているため、状況によっては抽象クラスを使うことが有用であると考える。

6 まとめ

本レポートでは、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た事例を整理した。具体的には、Ruby on Rails を用いた n 年日記の開発と、Unity を用いた 2D アクションゲームの設計と実装について述べた。n 年日記の開発では、Rails の MVC アーキテクチャを活用し、データベース管理やマイグレーション機能を駆使して効率的な開発を行った。ゲーム開発においては、Strategy パターンと Observer パターンを適用し、キャラクターの攻撃パターンや死亡イベントの管理を実装した。これにより、コードの再

利用性と拡張性を向上させ、柔軟な設計を実現した。今後の課題としては、さらなる設計パターンの学習と適用を通じて、より高度なソフトウェアアーキテクチャの理解を深めることである。この授業で得た知識と経験を基に、今後も継続的に学習を続け、技術力を向上させていきたい。

7　参考文献

参考文献

[1] 10 年日記, https://web.10nikki.com/welcome, 2025/02/21 閲覧.

[2] Rails MVC について復習, https://zenn.dev/goldsaya/articles/2e7b501538007a, 2025/02/21 閲覧.

[3] Eric Freeman, Elisabeth Robson 著, 佐藤 直生 監訳, 木下 哲也 訳, ”Head First デザインパターン 第 2 版 —頭とからだで覚えるデザインパターンの基本”, オライリージャパン, 2022.