Implementasi Enam Modul *Mini Games* dengan Menerapkan *Builder Pattern* untuk Sistem Permainan Heart Meister pada Unity 2D

Nama Mahasiswa	:	Afrizal
NRP	:	5110 100 214
Jurusan	:	Teknik Informatika FTIF ITS
Dosen Pembimbing 1	:	Imam Kuswardayan, S.Kom., M.T.
Dosen Pembimbing 2	:	Ridho Rahman H., S.Kom., M.Sc.

ABSTRAK

Heart Meister merupakan permainan sosial pada perangkat smartphone. Permainan Heart Meister bergenre Role Play Game (RPG) dengan tema fantasi. Permainan ini berfokus pada pertarungan antar pet yang dimiliki pemain. Setiap pet memiliki atribut sebagai indikator kekuatannya. Level dari masing-masing atribut bisa dinaikkan dengan menyelesaikan dungeon yang ada. Akan tetapi terkadang pemain ingin menaikkan level dari atribut tersebut secepat mungkin untuk menjadikan pet-nya semakin kuat. Masalah lain yaitu pet yang bukan merupakan pet utama dari pemain akan susah dinaikkan level atributnya karena jarang atau tidak pernah dipakai. Dengan demikian dibutuhkan suatu sistem untuk menaikkan level atribut dari pet secara cepat dan menarik.

Modul mini game ini dibuat dengan tujuan untuk memenuhi kebutuhan penaikan level atribut dari pet pada permainan Heart Meister. Masing-masing atribut dari pet dapat dinaikkan dengan jenis mini game yang berbeda. Hasil penaikan level atribut yang didapat berdasarkan berhasil atau tidaknya pemain dalam menyelesaikan mini game.

Modul mini games dibangun dengan menerapkan konsep Builder Pattern. Builder pattern digunakan untuk mengatur pembuatan level pada mini game. Dengan penerapan builder pattern ini pembuatan level dari tiap mini game ditangani oleh kelas yang sama. Dengan demikian penerapan builder pattern ini akan memudahkan jika terjadi penambahan level dari mini game.

Kata kunci: Builder Pattern, Heart Meister, Mini Games, Unity 2D

Implementation of Six Mini Games Module for Heart Meister Game System on Unity 2D by Applying Builder Pattern

Student's Name	:	Afrizal
Student's ID	:	5110 100 214
Department	:	Informatics Engineering, FTIF-ITS
First Advisor	:	Imam Kuswardayan, S.Kom., M.T.
Second Advisor	:	Ridho Rahman H., S.Kom., M.Sc.

ABSTRACT

Heart Meister is a social game wich run in smartphone device. It has Role Play Game (RPG) genre with fantasy theme. The main gameplay of this game is battle between player's pet. Each pet has some attributes as indicator of it strength. Each attribute can be enhanced by completing any dungeon. But sometimes there are player who want to enhance their pet as fast as possible in order to make it stronger. Thus this game requires a system that can handle this problem.

This mini games module is proposed to meet the need of pet's attributes enhancement in Heart Meister game. Each pet's attribute can be enhanced with different mini game. The result of enhancement depends on capability of player to completing the mini game.

This mini games module is developed with applying the concept of Builder Pattern. Builder pattern is used to organize the process of constructing level on each mini game. Thus the implementation of builder pattern will simplify the process of level construction if the new level is needed.

Keywords: Builder Pattern, Heart Meister, Mini Games, Unity 2D

DAFTAR KODE SUMBER

Kode Sumber 4.1. Kelas GameLevel 52
Kode Sumber 4.2. Kelas LevelBuilder 52
Kode Sumber 4.3. Kelas LevelManufacturer 53
Kode Sumber 4.4. Fungsi Update dan GameOver pada Mini Game Attack...54
Kode Sumber 4.5. Fungsi Update pada kelas DefenseEnemy..57
Kode Sumber 4.6. Fungsi Pengecek Masukan Pengguna pada Kelas Mini Game Speed..59
Kode Sumber 4.7. Fungsi UpdateGrid pada Kelas MatchThree 61
Kode Sumber 4.8. Fungsi UpdatePresentTile pada Kelas MatchThree 62
Kode Sumber 4.9. Fungsi SolveTheGrid pada Kelas TileSolver 63
Kode Sumber 4.10. Fungsi CheckMatched pada Kelas TileSolver 63
Kode Sumber 4.11. Fungsi Move pada Kelas SpecialDefenseEnemy 66
Kode Sumber 4.12. Fungsi OnCollisionEnter pada Kelas SpecialDefenseWall..67
Kode Sumber 4.13. Fungsi SpawnGround dan SpawnHealth pada Kelas HealthObjectSpawner.....................................70
Kode Sumber 8.1. Kelas AttackGameManager 95
Kode Sumber 8.2. Kelas AttackPlayer 96
Kode Sumber 8.3. Kelas GameLevel 96
Kode Sumber 8.4. Kelas DefenseGameManager..................98
Kode Sumber 8.5. Kelas DefensePlayer..............................99
Kode Sumber 8.6. Kelas SpeedGameManager 101
Kode Sumber 8.7. Kelas MatchThree..................................105
Kode Sumber 8.8. Kelas TileData.......................................107
Kode Sumber 8.9. Kelas TileSolver 109

Kode Sumber 8.10. Kelas `SpecialDefenseGameManager` 111
Kode Sumber 8.11. Kelas `HealthGameManager` 113

BAB II
TINJAUAN PUSTAKA

Pada bagian ini dipaparkan teori-teori serta pustaka yang dipakai pada waktu penelitian.

2.1. *Social Game*

Social Game adalah sebuah tipe dari permainan *online* yang dimainkan melalui jaringan sosial. Pada umumnya *social game* menyajikan fitur *multiplayer* dan mekanika permainan *asynchronous*. *Social game* biasanya diimplementasikan menjadi *browser game*. Walaupun demikian *social game* juga dapat diimplementasikan pada *platform* lain seperti perangkat bergerak.

2.2. Unity

Unity adalah sebuah ekosistem pengembangan permainan dengan mesin *rendering* yang kuat. Unity juga secara penuh terintegrasi dengan kumpulan perangkat intuitif yang lengkap dan alur kerja yang cepat untuk membuat konten 3D dan 2D. Unity mendukung publikasi *multiplatform* yang mudah, serta ribuan aset berkualitas siap pakai dengan komunitas yang saling berbagi pengetahuan [3].

2.2.1. *Monobehaviour*

Monobehaviour adalah kelas dasar dari semua *script* yang digunakan pada *game object* dalam Unity [4]. Untuk mengatur perilaku dari objek pada permainan, kelas yang diterapkan pada objek tersebut harus diturunkan dari kelas *Monobehaviour.* Kelas yang diturunkan dari kelas *Monobehaviour* secara otomatis akan mempunyai fungsi *Start* dan *Update.*

2.2.2. *Game Object*

Game Object adalah objek dasar pada Unity yang mewakili karakter, peraga, dan pemandangan [5]. *Game Object* berperan sebagai wadah untuk komponen-komponen yang mengimplementasikan fungsionalitas secara nyata pada permainan. *Game Object* selalu memiliki komponen *Transform* yang merepresentasikan posisi dan orientasi objek pada permainan.

2.2.3. *RaycastHit2D*

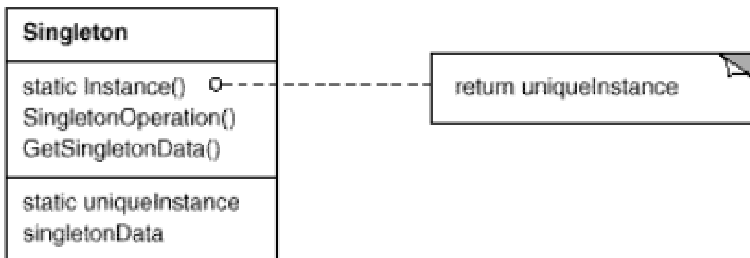
Konsep dari *Raycast* seperti sinar laser yang ditembakkan dari atas menuju suatu arah tertentu. Objek yang terkena kontak dapat tertdeteksi dan dilaporkan. Fungsi ini mengembalikan objek dengan referensi *collider* dari objek yang terkena sinar [6].

2.3. *Design Pattern*

Design pattern adalah solusi berulang umum yang diterapkan untuk masalah yang sering terjadi dalam desain perangkat lunak. Sebuah *design pattern* bukanlah desain jadi yang dapat dijalankan secara langsung. *Design pattern* merupakan deskripsi atau *template* untuk pemecahan masalah yang dapat digunakan dalam berbagai situasi [7].

2.4.1. *Singleton Pattern*

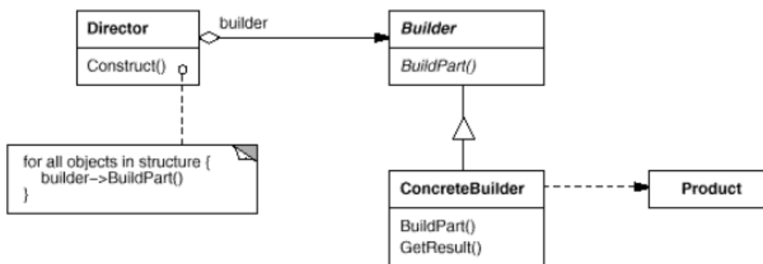
Singleton pattern merupakan pola yang digunakan untuk memastikan bahwa sebuah kelas hanya memiliki satu *instance* dan menghasilkan akses global kepada kelas ini [8]. Pola ini merupakan pola yang paling simpel dalam *design pattern.* Pola ini hanya melibatkan satu kelas yang bertanggung jawab untuk menginstansiasi dirinya sendiri. Tujuan dari pola ini yaitu untuk menghasilkan sebuah kelas yang dapat diakses secara langsung dari mana saja. Struktur dari *singleton pattern* dapat dilihat pada Gambar 2.1.



Gambar 9.1. Struktur *Singleton Pattern*

2.4.2. *Builder Pattern*

Builder pattern digunakan untuk memisahkan konstruksi dari sebuah objek yang kompleks dari representasinya. Dengan demikian proses konstruksi yang sama dapat menghasilkan representasi yang berbeda [8]. Pola ini mengijinkan sebuah kelas klien untuk membangun sebuah objek kompleks hanya dengan menentukan tipe dan isinya. Pada pengembangan permainan, pola ini dapat digunakan untuk mengatur pembuatan level pada permainan tersebut. Struktur dari *builder pattern* dapat dilihat pada Gambar 2.2.



Gambar 9.2. Struktur *Builder Pattern*

BAB III
ANALISIS DAN PERANCANGAN

Pada bab ini dibahasmengenai analisis sistem, perancangan perangkat lunak, serta implementasi modul *mini game* dari Heart Meister.

3.1. Tahap Analisis

Tahap ini membahas deskripsi umum sistem, analisis permasalahan, analisis aktor, kasus penggunaan, dan spesifikasi kebutuhan perangkat lunak.

3.1.1. Deskripsi Umum Sistem

Pada permainan *Heart Meister* sebagai permainanutama memiliki fitur untuk melatih karakternya. Latihan tersebut bertujuan untuk menaikkan atribut tertentu pada karakter yang dipakai. Untuk membuat fitur *training* ini menjadi lebih menarik maka disusunlah *mini game* yang merepresentasikan proses latihan dari karakter. Terdapat 6 *mini game* yang akan dibuat. Keenam *mini game* tersebut masing-masingmerepresentasikan latihan pada 6 atribut karakter, yaitu *attack, defense, speed, special attack, special defense,* dan *health.*

Pada *mini game* untuk penaikan atribut *attack* pemain diminta untuk menembak target yang bergerak. Pemain nantinya akan diminta untuk menembak target dengan jumlah tertentu. Pada *mini game* ini pemain diberi batas waktu serta jumlah kesempatan tertentu untuk mengenai objek yang bukan target. Jika waktu atau kesempatan tersebut telah habis sedangkan pemain belum menembak semua targetnya, maka permainan akan selesai dan hasil latihan dinyatakan gagal. Untuk setiap level dari karakter yang dilatih, tingkat kesulitan serta targetnya akan berbeda.

Pada *mini game* untuk penaikan atribut *defense* pemain diminta untuk menggerakkan karakter maju menuju musuh. Karakter akan bergerak maju jika pemain menge-*tap* layar. Di sisi

lain musuh akan menembak karakter dengan jeda waktu tertentu. Jika karakter dari pemain terkena tembakan dari musuh, karaktertersebut akan bergeser ke belakang dengan jarak tertentu. Dalam hal ini jika karakter dari pemain mundur hingga melewati batas, maka permainan akan selesai dan hasil latihan dinyatakan gagal. Dengan demikian pemain harus menge-*tap* layar dengan cepat untuk mencapai tujuan dan memenangkan *game* ini. Untuk setiap level dari karakter yang dilatih, jeda waktu dari tembakan musuh akan berbeda.

Pada *mini game* untuk penaikan atribut *speed* terdapat sekumpulan objek yang bergerak ke suatu arah. Pemain harus menge-*tap* target dari sekumpulan objek tersebut. Untuk memenangkan permainan pemain harus menge-*tap* target dengan jumlah tertentu. Pemain juga harus menghindari menge-*tap* objek selain target. Jika pemain salah menge-*tap* objek selain target permainan akan selesai dan hasil latihan dinyatakan gagal.

Pada *mini game* untuk penaikan atribut *special attack* terdapat sekumpulan objek yang tersusun dalam bentuk *tile.* Terdapat beberapa objek yang berbeda pada susunan tersebut. Pemain diminta untuk menghilangkan beberapa target dengan cara memasangkan 3 buah objek dari target yang sama. Pemain harus menyelesaikan permintaan tersebut dengan batas waktu tertentu. *Mini game* ini termasuk dalam kategori *match-three games.*

Pada *mini game* untuk penaikan atribut *special defense* pemain diminta untuk memasukkan bola ke gawang lawan. Pemain hanya bisa bergerak ke kiri dan ke kananuntuk memantulkan bola sekaligus melindungi gawangnya, begitu juga dengan lawan yang digerakkan oleh *CPU.* Pemain dinyatakan menang jika berhasil memasukkan bola ke gawang lawan sebanyak 3 kali. Sebaliknya pemain dinyatakan kalah jika bola masuk ke gawangnya sebanyak 3 kali.

Pada *mini game* untuk penaikan atribut *health* pemain diminta mengambil objek hati dengan jumlah tertentu. Dalam *game* ini pemain akan secara otomatis berlari ke depan secara terus menerus. Pemain hanya dapat mengendalikan karakter untuk

menghindari rintangan di depannya. Karakter pada *game* akan melompati rintangan jika pemain menge-*tap* layar. Pemain dinyatakan menang jika telah mendapatkan objek hati dengan jumlah yang ditentukan. Sebaliknya pemain dinyatakan kalah apabila menabrak rintangan di depannya.

Modul dari *mini game* ini nantinya akan berintegrasi dengan modul dari permainan utama. Pada saat modul *mini game* ini diakses, modul dari permainan utama akan memberikan data yang dibutuhkan untuk menjalankan proses *training* pada *mini game.*

3.1.2. Analisis Permasalahan

Permasalahan yang diangkat pada tugas akhir ini adalah bagaimana mengimplementasikan fitur *mini game* pada permainan Heart Meister. Fitur *mini game* ini merupakan representasi dari latihan untuk meningkatkan status atribut dari *pet.* Macam dari *mini game* yang dibangun merepresentasikan latihan untuk atribut *attack, defense, speed, special attack, special defense,* dan *health.*

Untuk setiap atribut dari *pet* dibuat *mini game* yang merepresentasikan latihan dari masing-masing atribut tersebut. Masing-masing *mini game* akan memiliki beberapa level. Pada masing-masing level akan terdapat rintangan yang berbeda. Akan tetapi proses pembuatan rintangan tersebut tentu saja sama untuk tiap level. Dengan demikian diperlukan sebuah metode untuk menangani proses pembuatan level serta memudahkan saat terjadi penambahan atau pengurangan jumlah level pada *mini game.* Dalam hal ini konsep *builder pattern* dapat digunakan untuk menangani masalah tersebut.

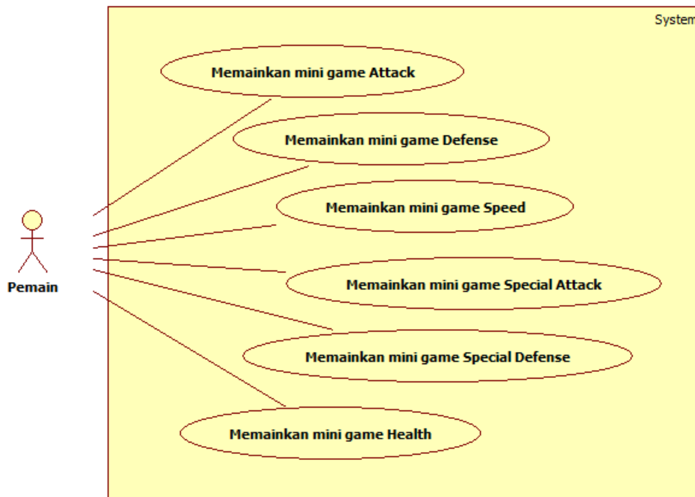
3.1.3. Aktor

Aktor mendefinisikan entitas-entitas yang terlibat dan berinteraksi langsung dengan sistem. Entitas ini bisa berupa manusia maupun sistem atau perangkat lunak lain. Aktor yang terdapat pada sistem ini hanya memiliki sebuah peran, yaitu

sebagai pengguna. Pengguna dari perangkat lunak ini yaitu pemain.

3.1.4. Kasus Penggunaan

Berdasarkan analisis spesifikasi kebutuhan fungsional dan analisis aktor dari sistem dibuat kasus penggunaan sistem. Kasus-kasus penggunaan dalam sistem ini akan dijelaskan secara rinci pada subbab ini. Kasus penggunaan digambarkan dalam sebuah diagram kasus penggunaan. Diagram kasus penggunaan dapat dilihat pada Gambar 3.1. Sedangkan penjelasan dari setiap kasus penggunaan dapat dilihat pada Tabel 3.1.



Gambar 10.1. Diagram Kasus Penggunaan

Tabel 10.1. Daftar Kode Diagram Kasus Penggunaan

Kode Kasus Penggunaan	Nama
UC-0001	Memainkan *mini game Attack*
UC-0002	Memainkan *mini game Defense*

UC-0003	Memainkan *mini game Speed*
UC-0004	Memainkan *mini game Special Attack*
UC-0005	Memainkan *mini game Special Defense*
UC-0006	Memainkan *mini game Health*

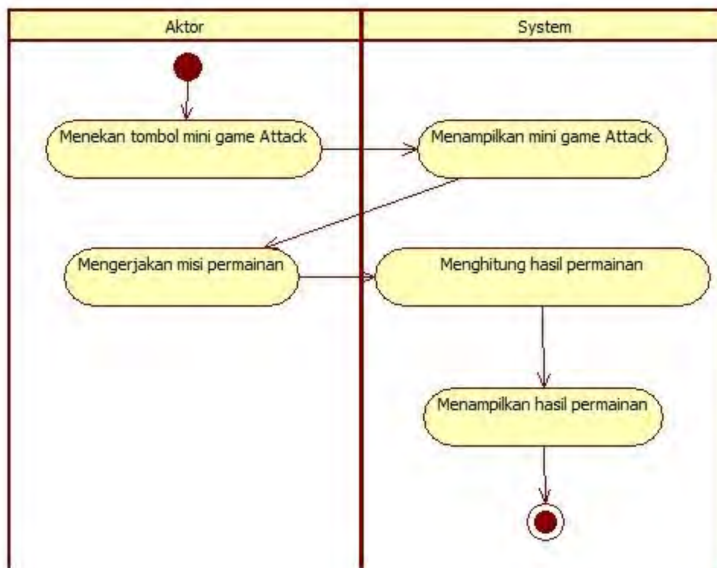
3.1.4.1. Memainkan Mini Game Attack

Pada kasus penggunaan ini, sistem menerima input berupa perintah untuk menuju*mini game Attack*. Setelah itu sistem akan menampilkan permainan untuk menaikkan atribut *attack*. Spesifikasi dari kasus penggunaan ini dapat dilihat pada Tabel 3.2. Pada kasus penggunaan ini, pemain memainkan *mini game Attack* sedangkan sistem akan menghitung waktu, jumlah target yang berhasil ditembak, serta sisa kesempatan dari pemain. Setelah itu sistem akan menghitung tingkat keberhasilan dari pemain serta penambahan atribut *attack* yang didapatkan. Diagram aktivitas dari kasus penggunaan ini bisa dilihat pada Gambar 3.2.

Tabel 10.2. Spesifikasi Kasus Penggunaan Memainkan *Mini Game Attack*

Nama	Memainkan *mini game Attack*
Kode	UC-0001
Deskripsi	Menampilkan permainan untuk menaikkan atribut *attack.* Hasil dari permainan ini adalah tingkat keberhasilan pemain serta nilai penambahan atribut *attack.*
Tipe	Fungsional
Pemicu	Pengguna menekan tombol menuju *mini game Attack.*
Aktor	Pemain
Kondisi Awal	-
Aliran: **Kejadian Normal**	1. Pemain menekan tombol menuju *mini game Attack.* 2. Sistem menampilkan *mini game Attack*. 3. Pemain mengerjakan misi dari permainan.

	4. Sistem menghitung sisa waktu, jumlah target, serta kesempatan dari pemain. 5. Sistem menampilkan hasil permainan dan nilai penambahan atribut *attack.*
Kejadian Alternatif	-
Kondisi Akhir	Tampilan hasil permainan berupa nilai penambahan atribut.



Gambar 10.2. Diagram Aktivitas Memainkan Mini Game Attack

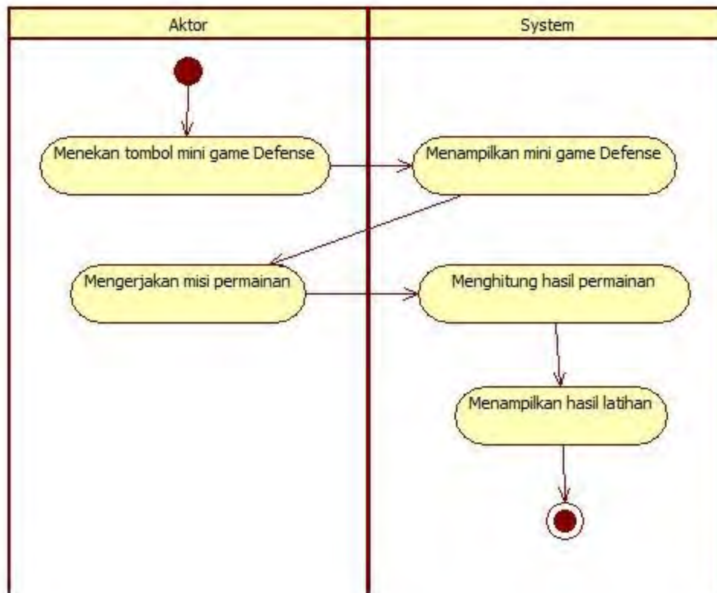
3.1.4.2. Memainkan *Mini Game Defense*

Pada kasus penggunaan ini sistem menerima input berupa perintah untuk menuju *mini game Defense.* Kemudian sistem akan menampilkan permainan untuk menaikkan atribut *defense*. Spesifikasi dari kasus penggunaan ini dapat dilihat pada Tabel 3.3. Pada kasus penggunaan ini, pemain memainkan *mini game Defense* sedangkan sistem akan menghitung jarak yang berhasil ditempuh pemain. Setelah itu sistem akan menghitung tingkat

keberhasilan pemain serta nilai penambahan atribut *defense* yang didapatkan. Diagram aktivitas dari kasus penggunaan ini bisa dilihat pada Gambar 3.3.

Tabel 10.3. Spesifikasi Kasus Penggunaan Memainkan *Mini Game Defense*

Nama	Memainkan *mini game Defense*
Kode	UC-0002
Deskripsi	Menampilkan permainan untuk menaikkan atribut *defense*. Hasil dari permainan ini adalah tingkat keberhasilan pemain serta nilai penambahan atribut *defense*.
Tipe	Fungsional
Pemicu	Pengguna menekan tombol menuju *mini game Defense*.
Aktor	Pemain
Kondisi Awal	-
Aliran: **Kejadian Normal**	1. Pemain menekan tombol menuju *mini game Defense*. 2. Sistem menampilkan *mini game Defense*. 3. Pemain mengerjakan misi dari permainan. 4. Sistem menghitung jarak yang ditempuh pemain dalam permainan. 5. Sistem menampilkan hasil permainan dan nilai penambahan atribut *defense*.
Kejadian Alternatif	-
Kondisi Akhir	Tampilan hasil permainan berupa nilai penambahan atribut.



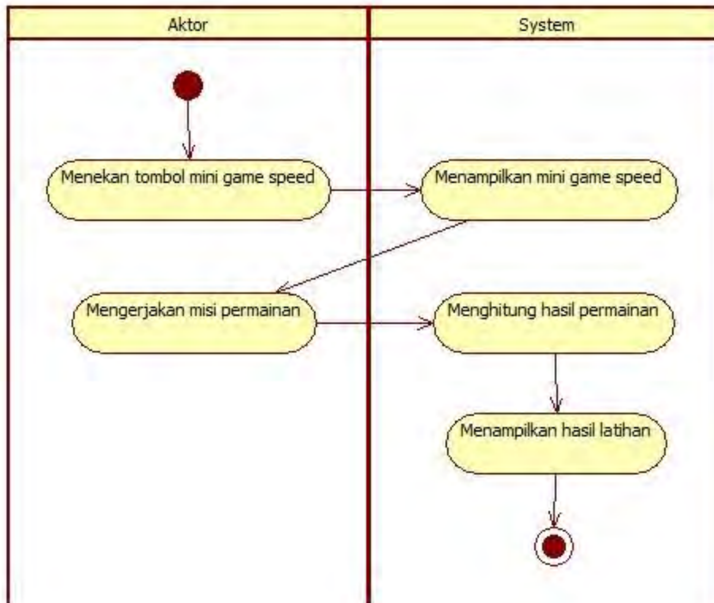
Gambar 10.3. Diagram Aktivitas Memainkan Mini Game Defense

3.1.4.3. Memainkan *Mini Game Speed*

Pada kasus penggunaan ini sistem menerima input berupa perintah untuk menuju *mini game Speed.* Kemudian sistem akan menampilkan permainan untuk menaikkan atribut *speed*. Spesifikasi dari kasus penggunaan ini dapat dilihat pada Tabel 3.4. Pada kasus penggunaan ini, pemain memainkan *mini game Speed* sedangkan sistem akan menghitung musuh yang berhasil dihilangkan oleh pemain. Setelah itu sistem akan menghitung tingkat keberhasilan pemain serta nilai penambahan atribut *speed* yang didapatkan. Diagram aktivitas dari kasus penggunaan ini bisa dilihat pada Gambar 3.4.

Tabel 10.4. Spesifikasi Kasus Penggunaan Memainkan *Mini Game Speed*

Nama	Memainkan *mini game Speed*
Kode	UC-0003
Deskripsi	Menampilkan permainan untuk menaikkan atribut *speed.* Hasil dari permainan ini adalah tingkat keberhasilan pemain serta nilai penambahan atribut *speed.*
Tipe	Fungsional
Pemicu	Pengguna menekan tombol menuju *mini game Speed.*
Aktor	Pemain
Kondisi Awal	-
Aliran: **Kejadian Normal**	1. Pemain menekan tombol menuju *mini game Speed.* 2. Sistem menampilkan *mini game Speed*. 3. Pemain mengerjakan misi dari permainan. 4. Sistem menghitung jumlah musuh yang berhasil dihilangkan oleh pemain. 5. Sistem menampilkan hasil permainan dan nilai penambahan atribut *speed.*
Kejadian Alternatif	-
Kondisi Akhir	Tampilan hasil permainan berupa nilai penambahan atribut.



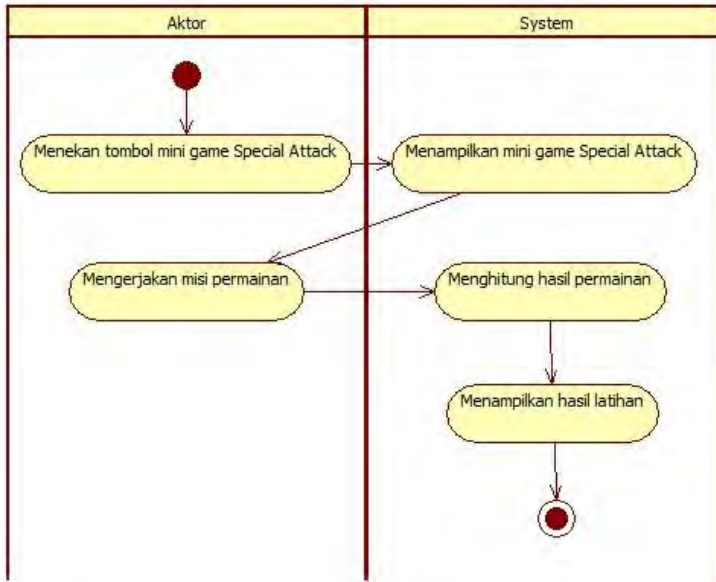
Gambar 10.4. Diagram Aktivitas Memainkan Mini Game Speed

3.1.4.4. Memainkan Mini Game Special Attack

Pada kasus penggunaan ini sistem menerima input berupa perintah untuk menuju *mini game Special Attack.* Kemudian sistem akan menampilkan permainan untuk menaikkan atribut *special attack*. Spesifikasi dari kasus penggunaan ini dapat dilihat pada Tabel 3.5. Pada kasus penggunaan ini, pemain memainkan *mini game Special Attack* sedangkan sistem akan menghitung jumlah objek yang berhasil dipasangkan oleh pemain. Setelah itu sistem akan menghitung tingkat keberhasilan pemain serta nilai penambahan atribut *special attack* yang didapatkan. Diagram aktivitas dari kasus penggunaan ini bisa dilihat pada Gambar 3.5.

Tabel 10.5. Spesifikasi Kasus Penggunaan Memainkan *Mini Game Special Attack*

Nama	Memainkan *mini game Special Attack*
Kode	UC-0004
Deskripsi	Menampilkan permainan untuk menaikkan atribut *special attack.* Hasil dari permainan ini adalah tingkat keberhasilan pemain serta nilai penambahan atribut *special attack.*
Tipe	Fungsional
Pemicu	Pengguna menekan tombol menuju *mini game Special Attack.*
Aktor	Pemain
Kondisi Awal	-
Aliran: **Kejadian Normal**	1. Pemain menekan tombol menuju *mini game Special Attack.* 2. Sistem menampilkan *mini game Special Attack.* 3. Pemain mengerjakan misi dari permainan. 4. Sistem menghitung jumlah objek yang berhasil dipasangkan oleh pemain. 5. Sistem menampilkan hasil permainan dan nilai penambahan atribut *special attack.*
Kejadian Alternatif	-
Kondisi Akhir	Tampilan hasil permainan berupa nilai penambahan atribut.



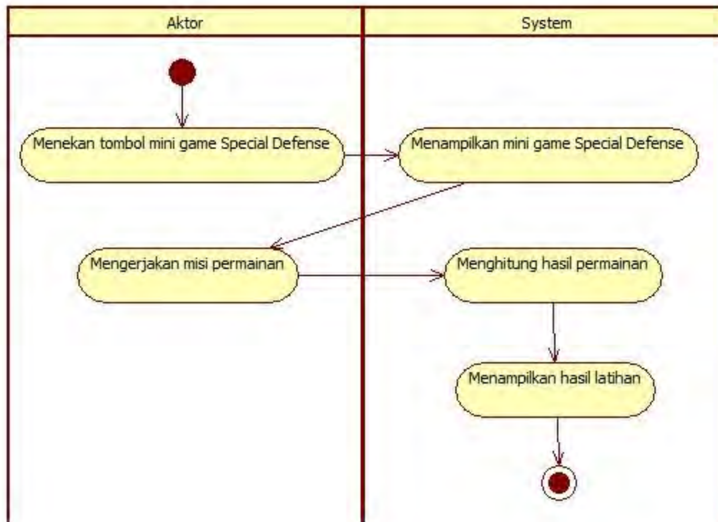
Gambar 10.5. Diagram Aktivitas Memainkan Mini Game Special Attack

3.1.4.5. Memainkan Mini Game Special Defense

Pada kasus penggunaan ini sistem menerima input berupa perintah untuk menuju *mini game Special Defense.* Kemudian sistem akan menampilkan permainan untuk menaikkan atribut *special defense*. Spesifikasi dari kasus penggunaan ini dapat dilihat pada Tabel 3.6. Pada kasus penggunaan ini, pemain memainkan *mini game Special Defense* sedangkan sistem akan menghitung skor yang didapatkan pemain dan musuh. Setelah itu sistem akan menghitung tingkat keberhasilan pemain serta nilai penambahan atribut *special defense* yang didapatkan. Diagram aktivitas dari kasus penggunaan ini bisa dilihat pada Gambar 3.6.

Tabel 10.6. Spesifikasi Kasus Penggunaan Memainkan *Mini Game Special Defense*

Nama	Memainkan *mini game Special Defense*
Kode	UC-0005
Deskripsi	Menampilkan permainan untuk menaikkan atribut *special attack.* Hasil dari permainan ini adalah tingkat keberhasilan pemain serta nilai penambahan atribut *special attack.*
Tipe	Fungsional
Pemicu	Pengguna menekan tombol menuju *mini game Special Attack.*
Aktor	Pemain
Kondisi Awal	-
Aliran: **Kejadian Normal**	1. Pemain menekan tombol menuju *mini game Special Attack.* 2. Sistem menampilkan *mini game Special Attack.* 3. Pemain mengerjakan misi dari permainan. 4. Sistem menghitung skor yang didapatkan pemain dan musuh. 5. Sistem menampilkan hasil permainan dan nilai penambahan atribut *special attack.*
Kejadian Alternatif	-
Kondisi Akhir	Tampilan hasil permainan berupa nilai penambahan atribut.



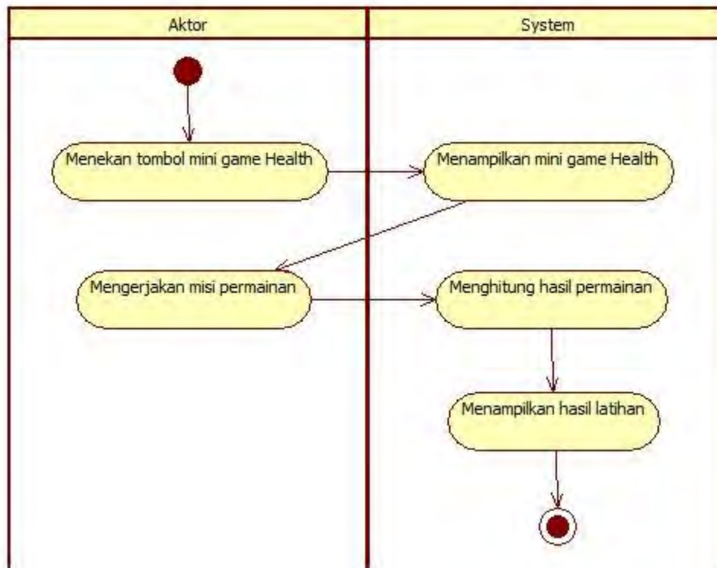
Gambar 10.6. Diagram Aktivitas Memainkan Mini Game Special Defense

3.1.4.6. Memainkan *Mini Game Health*

Pada kasus penggunaan ini sistem menerima input berupa perintah untuk menuju *mini game Health.* Kemudian sistem akan menampilkan permainan untuk menaikkan atribut *health*. Spesifikasi dari kasus penggunaan ini dapat dilihat pada Tabel 3.7. Pada kasus penggunaan ini, pemain memainkan *mini game Health* sedangkan sistem akan menghitung jumlah objek hati yang berhasil didapatkan oleh pemain. Setelah itu sistem akan menghitung tingkat keberhasilan pemain serta nilai penambahan atribut *health* yang didapatkan. Diagram aktivitas dari kasus penggunaan ini bisa dilihat pada Gambar 3.7.

Tabel 10.7. Spesifikasi Kasus Penggunaan Memainkan *Mini Game Health*

Nama	Memainkan *mini game Health*
Kode	UC-0006
Deskripsi	Menampilkan permainan untuk menaikkan atribut *health.* Hasil dari permainan ini adalah tingkat keberhasilan pemain serta nilai penambahan atribut *health.*
Tipe	Fungsional
Pemicu	Pengguna menekan tombol menuju *mini game Health.*
Aktor	Pemain
Kondisi Awal	-
Aliran: **Kejadian Normal**	1. Pemain menekan tombol menuju *mini game Health.* 2. Sistem menampilkan *mini game Health*. 3. Pemain mengerjakan misi dari permainan. 4. Sistem menghitung jumlah objek hati yang didapatkan pemain. 5. Sistem menampilkan hasil permainan dan nilai penambahan atribut *health.*
Kejadian Alternatif	-
Kondisi Akhir	Tampilan hasil permainan berupa nilai penambahan atribut.



Gambar 10.7. Diagram Aktivitas Memainkan Mini Game Health

3.1.5. Spesifikasi Kebutuhan Perangkat Lunak

Bagian ini berisi semua kebutuhan perangkat lunak yang diuraikan secara rinci dalam bentuk diagram kasus dan diagram aktivitas. Masing-masing diagram menjelaskan perilaku atau sifat dari sistem ini. Kebutuhan perangkat lunak dalam sistem ini mencakup kebutuhan fungsional saja. Pada bab ini juga dijelaskan tentang spesifikasi terperinci pada masing-masing kebutuhan fungsional. Rincian spesifikasi dari kasus penggunaan disajikan dalam bentuk tabel.

3.1.5.1. Kebutuhan Fungsional

Kebutuhan fungsional berisi proses-proses yang harus dimiliki sistem. Kebutuhan fungsional mendefinisikan layanan yang harus disediakan dan reaksi sistem terhadap masukan atau

pada situasi tertentu. Daftar kebutuhan fungsional dapat dilihat pada Tabel 3.8.

Tabel 10.8. Daftar Kebutuhan Fungsional Perangkat Lunak

Kode Kebutuhan	Kebutuhan Fungsional	Deskripsi
F-0001	Menampilkan *mini game Attack*	Sistem menampilkan *mini game* untuk dimainkan oleh pengguna. *Mini game* ini bertujuan untuk menaikkan atribut *attack.*
F-0002	Menampilkan *mini game Defense*	Sistem menampilkan *mini game* untuk dimainkan oleh pengguna. *Mini game* ini bertujuan untuk menaikkan atribut *defense.*
F-0003	Menampilkan *mini game Speed*	Sistem menampilkan *mini game* untuk dimainkan oleh pengguna. *Mini game* ini bertujuan untuk menaikkan atribut *speed.*
F-0004	Menampilkan *mini game Special Attack*	Sistem menampilkan *mini game* untuk dimainkan oleh pengguna. *Mini game* ini bertujuan untuk menaikkan atribut *special attack.*
F-0005	Menampilkan *mini game Special Defense*	Sistem menampilkan *mini game* untuk dimainkan oleh pengguna. *Mini game* ini bertujuan untuk menaikkan atribut *special defense.*
F-0006	Menampilkan *mini game Health.*	Sistem menampilkan *mini game* untuk dimainkan oleh pengguna. *Mini game* ini bertujuan untuk menaikkan atribut *health.*

3.2. Perancangan Sistem

Penjelasan tahap perancangan perangkat lunak dibagi menjadi dua bagian yaitu perancangan diagram kelas dan perancangan antarmuka.

3.2.1. Perancangan Diagram Kelas

Perancangan diagram kelas berisi rancangan dari kelas-kelas yang digunakan untuk membangun sistem. Pada subbab ini, hubungan dan perilaku antar kelas digambarkan dengan lebih jelas. Subbab ini dibagi menjadi tujuh bagian, yaitu diagram kelas untuk *mini game Attack, mini game Defense, mini game Speed, mini game Special Attack, mini game Special Defense, mini game Health, dan* hasil latihan.

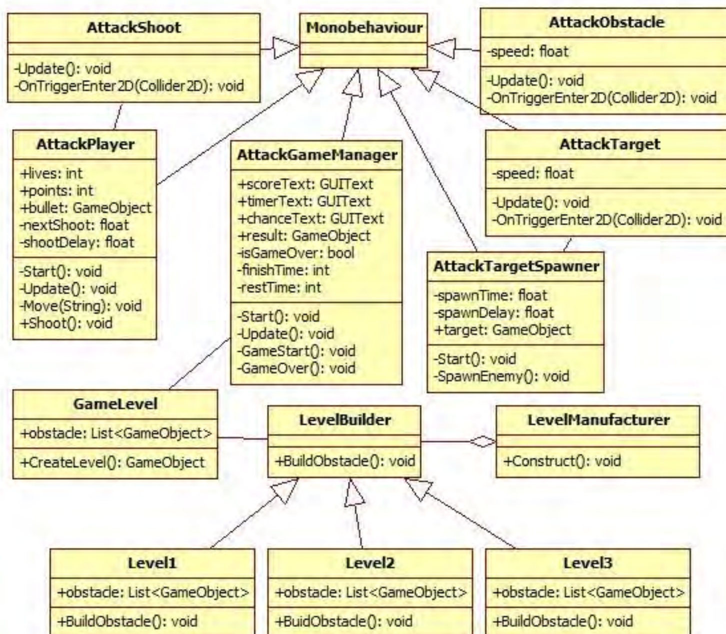
3.2.1.1. Diagram Kelas Mini Game Attack

Pada diagram kelas *mini game Attack* terdapat dua bagian besar, yaitu kelas-kelas yang menempel pada objek permainan dan kelas-kelas yang digunakan untuk membangun level dari permainan. Diagram kelas *mini game Attack* dapat dilihat pada Gambar 3.8.

Kelas-kelas yang menempel pada objek permainan yaitu kelas `AttackGameManager`, `AttackPlayer`, `AttackShoot`, `AttackTarget`, `AttackTargetSpawner`, dan `AttackObstacle`. Semua kelas tersebut merupakan kelas turunan dari *Monobehaviour.* Kelas `AttackGameManager` digunakan untuk untuk mengatur jalannya permainan. Kelas `AttackPlayer` berfungsi sebagai pengatur gerakan pemain pada permainan. Kelas `AttackShoot` merupakan pengatur tembakan pemain. Kelas `AttackTarget` digunakan untuk mengatur gerakan target dalam permainan. Sedangkan kelas `AttackTargetSpawner` digunakan untuk mengatur pembuatan objek target pada permainan. Kelas `AttackObstacle` berfungsi untuk mengatur gerakan dari objek rintangan di dalam permainan.

Untuk membangun level dari permainan digunakan *builder pattern.* Kelas-kelas yang digunakan yaitu kelas `GameLevel`, `LevelBuilder`, `LevelManufacturer`, `Level1`, `Level2`, dan `Level3`. Kelas `GameLevel` digunakan

untuk menyimpan daftar dari objek rintangan yang akan di tampilkan pada permainan. Kelas `LevelBuilder` merupakan kelas abstrak untuk membuat level. Sedangkan kelas `LevelManufacturer` berfungsi untuk membangun level permainan dengan menggunakan kelas `LevelBuilder`. Kelas `Level1`, `Level2`, dan `Level3` masing-masing merupakan kelas turunan dari `LevelBuilder` yang berfungsi untuk membangun level permainan secara spesifik.



Gambar 10.8. Diagram Kelas *Mini Game Attack*

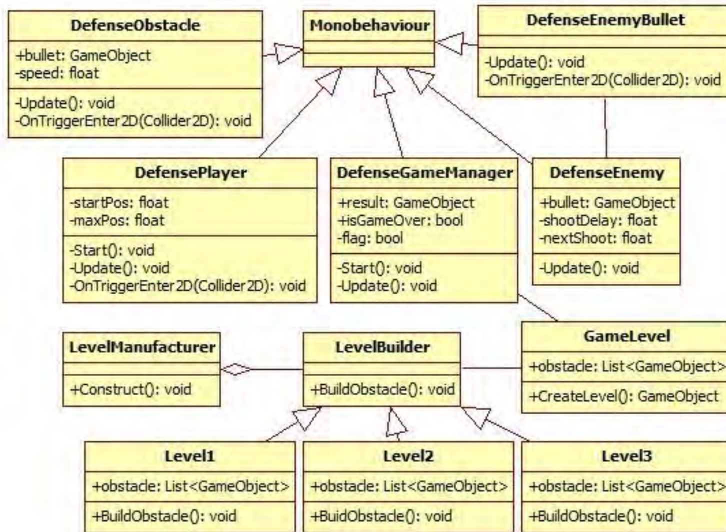
3.2.1.2. Diagram Kelas Mini Game Defense

Pada diagram kelas *mini game Defense* terdapat dua bagian besar, yaitu kelas-kelas yang menempel pada objek

permainan dan kelas-kelas yang digunakan untuk membangun level dari permainan. Diagram kelas *mini game Defense* dapat dilihat pada Gambar 3.9.

Kelas-kelas yang menempel pada objek permainan yaitu kelas `DefenseGameManager`, `DefensePlayer`, `DefenseEnemy`, `DefenseEnemyBullet`, dan `DefenseObstacle`. Semua kelas tersebut merupakan kelas turunan dari *Monobehaviour*. Kelas `DefenseGameManager` digunakan untuk untuk mengatur jalannya permainan. Kelas `DefensePlayer` berfungsi sebagai pengatur gerakan pemain pada permainan, sedangkan kelas `DefenseEnemy` berfungsi untuk mengatur gerakan dari musuh. Kelas `DefenseEnemyBullet` digunakan untuk mengatur gerakan peluru dari musuh. Kelas `DefenseObstacle` merupakan kelas untuk mengatur perilaku dari rintangan dalam permainan.

Untuk membangun level dari permainan digunakan *builder pattern*. Kelas-kelas yang digunakan yaitu kelas `GameLevel`, `LevelBuilder`, `LevelManufacturer`, `Level1`, `Level2`, dan `Level3`. Kelas `GameLevel` digunakan untuk menyimpan daftar dari objek rintangan yang akan di tampilkan pada permainan. Kelas `LevelBuilder` merupakan kelas abstrak untuk membuat level. Sedangkan kelas `LevelManufacturer` berfungsi untuk membangun level permainan dengan menggunakan kelas `LevelBuilder`. Kelas `Level1`, `Level2`, dan `Level3` masing-masing merupakan kelas turunan dari `LevelBuilder` yang berfungsi untuk membangun level permainan secara spesifik.



Gambar 10.9. Diagram Kelas *Mini Game Defense*

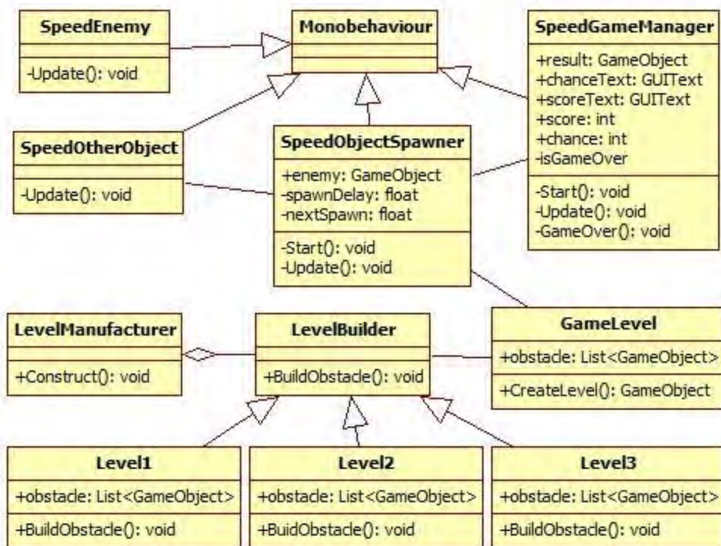
3.2.1.3. Diagram Kelas Mini Game Speed

Pada diagram kelas *mini game Speed* terdapat dua bagian besar, yaitu kelas-kelas yang menempel pada objek permainan dan kelas-kelas yang digunakan untuk membangun level dari permainan. Diagram kelas *mini game Speed* dapat dilihat pada Gambar 3.10.

Kelas-kelas yang menempel pada objek permainan yaitu kelas `SpeedGameManager`, `SpeedEnemy`, `SpeedObjectSpawner`, dan `SpeedOtherObject`. Semua kelas tersebut merupakan kelas turunan dari *Monobehaviour.* Kelas `SpeedGameManager` digunakan untuk untuk mengatur jalannya permainan. Kelas `SpeedEnemy` berfungsi untuk mengatur gerakan dari musuh, sedangkan kelas `SpeedOtherObject` berfungsi untuk mengatur gerakan dari objek lain dalam permainan. Kelas `SpeedObjectSpawner`

digunakan untuk mengeluarkan objek musuh dan objek lain di dalam permainan.

Untuk membangun level dari permainan digunakan *builder pattern.* Kelas-kelas yang digunakan yaitu kelas `GameLevel`, `LevelBuilder`, `LevelManufacturer`, `Level1`, `Level2`, dan `Level3`. Kelas `GameLevel` digunakan untuk menyimpan daftar dari objek rintangan yang akan di tampilkan pada permainan. Kelas `LevelBuilder` merupakan kelas abstrak untuk membuat level. Sedangkan kelas `LevelManufacturer` berfungsi untuk membangun level permainan dengan menggunakan kelas `LevelBuilder`. Kelas `Level1`, `Level2`, dan `Level3` masing-masing merupakan kelas turunan dari `LevelBuilder` yang berfungsi untuk membangun level permainan secara spesifik.



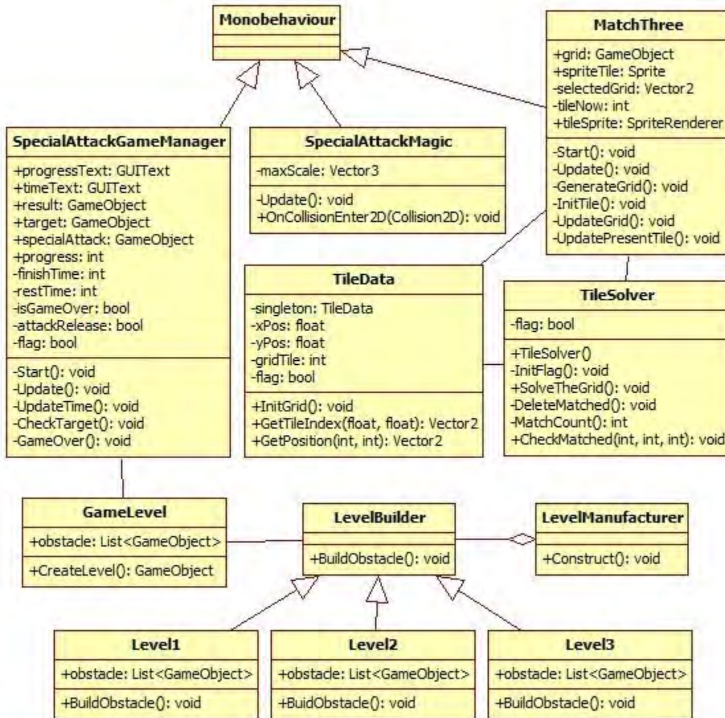
Gambar 10.10. Diagram Kelas *Mini Game Speed*

3.2.1.4. Diagram Kelas Mini Game Special Attack

Pada diagram kelas *mini game Special Attack* terdapat dua bagian besar, yaitu kelas-kelas yang menempel pada objek permainan dan kelas-kelas yang digunakan untuk membangun level dari permainan. Diagram kelas *mini game Special Attack* dapat dilihat pada Gambar 3.11.

Kelas-kelas yang menempel pada objek permainan yaitu kelas `SpecialAttackGameManager`, `MatchThree`, `TileData`, `TileSolver`, dan `SpecialAttackMagic`. Kelas `MatchThree`, `SpecialAttackGameManager`, dan `SpecialAttackMagic` merupakan kelas turunan dari *Monobehaviour*. Kelas `SpecialAttackGameManager` digunakan untuk untuk mengatur jalannya permainan. Kelas `SpecialAttackMagic` berfungsi untuk mengatur perilaku dari serangan pemain. Kelas `MatchThree` digunakan untuk menangani tampilan permainan *Match Three*. Kelas `TileData` merupakan kelas *singleton* yang berfungsi sebagai penyimpan data permainan sementara, sedangkan kelas `TileSolver` digunakan untuk menangani aturan dari *Match Three-Game*.

Untuk membangun level dari permainan digunakan *builder pattern*. Kelas-kelas yang digunakan yaitu kelas `GameLevel`, `LevelManufacturer`, `LevelBuilder`, `Level1`, `Level2`, dan `Level3`. Kelas `GameLevel` digunakan untuk menyimpan daftar dari objek rintangan yang akan di tampilkan pada permainan. Kelas `LevelBuilder` merupakan kelas abstrak untuk membuat level. Sedangkan kelas `LevelManufacturer` berfungsi untuk membangun level permainan dengan menggunakan kelas `LevelBuilder`. Kelas `Level1`, `Level2`, dan `Level3` masing-masing merupakan kelas turunan dari `LevelBuilder` yang berfungsi untuk membangun level permainan secara spesifik.



Gambar 10.11. Diagram Kelas *Mini Game Special Attack*

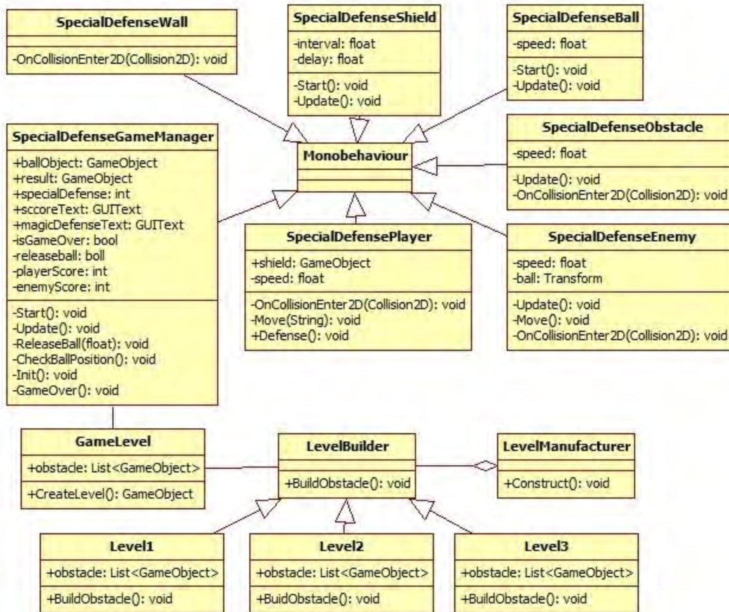
3.2.1.5. Diagram Kelas Mini Game Special Defense

Pada diagram kelas *mini game Special Defense* dapat dibagi menjadi dua bagian, yaitu kelas-kelas yang menempel pada objek permainan dan kelas-kelas yang digunakan untuk membangun level dari permainan. Diagram kelas *mini game Special Defense* dapat dilihat pada Gambar 3.12.

Kelas-kelas yang menempel pada objek permainan yaitu `SpecialDefensePlayer`, `SpecialDefenseEnemy`, `SpecialDefenseGameManager`, `SpecialDefenseWall`, `SpecialDefenseBall`, `SpecialDefenseShield`, dan

SpecialDefenseObstacle. Kelas SpecialDefenseGameManager digunakan untuk untuk mengatur jalannya permainan. Kelas SpecialDefensePlayer, SpecialDefenseEnemy, dan SpecialDefenseBall masing-masing berfungsi untuk mengatur gerakan pemain, musuh dan bola. Kelas SpecialDefenseWall digunakan untuk mengatur perilaku dari objek dinding pada permainan, sedangkan kelas SpecialDefenseObstacle digunakan untuk mengatur perilaku dari rintangan pada permainan. Kelas SpecialDefenseShield berfungsi sebagai pengatur perilaku objek perisai yang dimiliki pemain.

Untuk membangun level dari permainan digunakan *builder pattern.* Kelas-kelas yang digunakan yaitu kelas GameLevel, LevelManufacturer, LevelBuilder, Level1, Level2, dan Level3. Kelas GameLevel digunakan untuk menyimpan daftar dari objek rintangan yang akan di tampilkan pada permainan. Kelas LevelBuilder merupakan kelas abstrak untuk membuat level. Sedangkan kelas LevelManufacturer berfungsi untuk membangun level permainan dengan menggunakan kelas LevelBuilder. Kelas Level1, Level2, dan Level3 masing-masing merupakan kelas turunan dari LevelBuilder yang berfungsi untuk membangun level permainan secara spesifik.



Gambar 10.12. Diagram Kelas *Mini Game Special Defense*

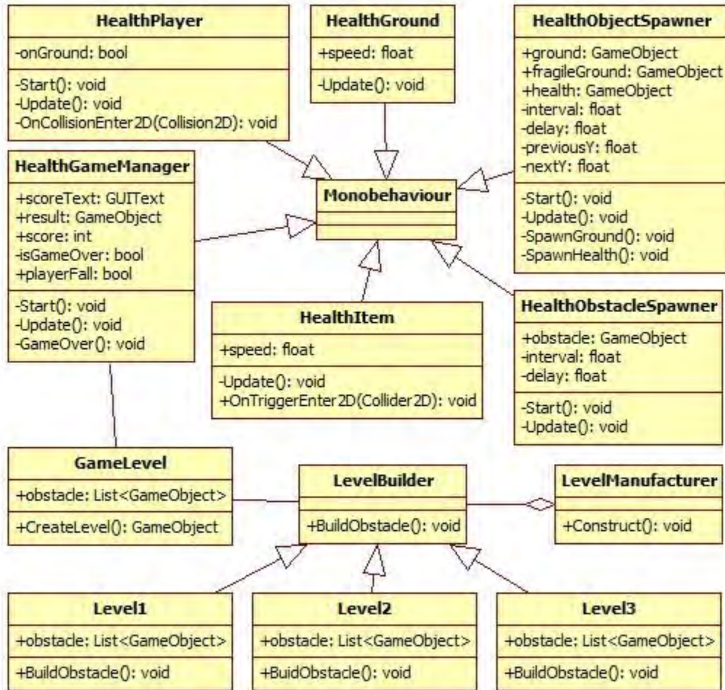
3.2.1.6. Diagram Kelas Mini Game Health

Pada diagram kelas *mini game Health* dapat dibagi menjadi dua bagian, yaitu kelas-kelas yang menempel pada objek permainan dan kelas-kelas yang digunakan untuk membangun level dari permainan. Diagram kelas *mini game Health* dapat dilihat pada Gambar 3.13.

Kelas-kelas yang menempel pada objek permainan yaitu kelas `HealthGameManager`, `HealthPlayer`, `HealthItem`, `HealthGround`, `HealthObjectSpawner`, dan `HealthObstacleSpawner`. Kelas `HealthGameManager` digunakan untuk untuk mengatur jalannya permainan. Kelas ini mengatur perolehan skor dari pemain serta kondisi saat permainan berakhir. Kelas

HealthPlayer berfungsi untuk mengatur gerakan pemain. Kelas ini juga mengatur perilaku pemain saat bertabrakan dengan objek lain. Kelas HealthItem dan HealthGround masing-masing digunakan untuk mengatur perilaku dari objek *Health* dan objek *Ground* pada permainan. Sedangkan kelas HealthObjectSpawner digunakan untuk mengeluarkan objek-objek yang digunakan dalam permainan dan kelas HealthObstacleSpawner digunakan untuk mengeluarkan objek rintangan.

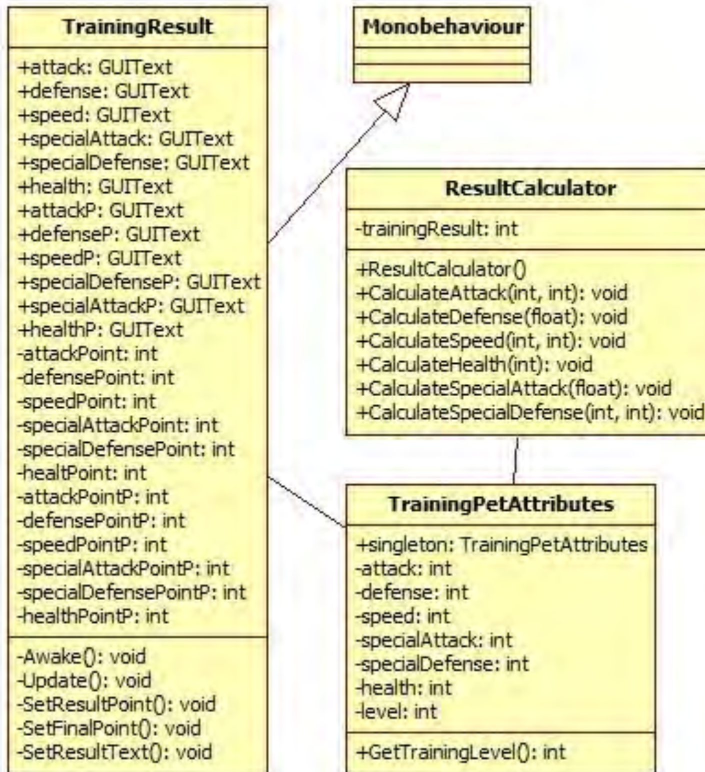
Untuk membangun level dari permainan digunakan *builder pattern.* Kelas-kelas yang digunakan yaitu kelas GameLevel, LevelManufacturer, LevelBuilder, Level1, Level2, dan Level3. Kelas GameLevel digunakan untuk menyimpan daftar dari objek rintangan yang akan di tampilkan pada permainan. Kelas LevelBuilder merupakan kelas abstrak untuk membuat level. Sedangkan kelas LevelManufacturer berfungsi untuk membangun level permainan dengan menggunakan kelas LevelBuilder. Kelas Level1, Level2, dan Level3 masing-masing merupakan kelas turunan dari LevelBuilder yang berfungsi untuk membangun level permainan secara spesifik.



Gambar 10.13. Diagram Kelas *Mini Game Health*

3.2.1.7. Diagram Kelas Hasil Latihan

Pada diagram kelas hasil latihan terdapat kelas `TrainingResult`, `TrainingPetAttributes`, dan `ResultCalculator`. Kelas `TrainingResult` merupakan kelas turunan dari *monobehaviour* yang digunakan pada objek permainan. Kelas ini berfungsi untuk menampilkan hasil dari latihan. Kelas `TrainingPetAttributes` merupakan kelas *singleton* yang berfungsi untuk menyimpan nilai atribut dari *pet* yang dilatih. Sedangkan kelas `ResultCalculator` digunakan untuk menghitung hasil dari latihan. Diagram kelas hasil latihan dapat dilihat pada Gambar 3.14.



Gambar 10.14. Diagram Kelas Hasil Latihan

3.2.2. Perancangan Antarmuka

Bagian ini membahas rancangan tampilan antar muka pada sistem. Pada sistem ini terdapat beberapa tampilan utama diantaranya tampilan menu dari *mini game,* tampilan *mini game Attack, mini game Defense, mini game Speed, mini game Special Attack, mini game Special Defense,* dan *mini game Health.* Penjelasan lebih lanjut dari tampilan tersebut akan dibahas pada subbab tersendiri.

3.2.2.1. Tampilan Menu *Mini Game*

Halaman ini merupakan tampilan utama saat pengguna memilih menu latihan dari permainan utama. Pada halaman ini terdapat tombol *Attack, Defense, Speed, Special Attack, Special Defense, Health,* dan *Home*. Tombol tersebut masing-masing berfungsi untuk menuju *mini game Attack, Defense, Speed, Special Attack, Special Defense,* dan *Health.* Sedangkan tombol *Home* berfungsi untuk kembali ke permainan utama. Rancangan tampilan menu *mini game* dapat dilihat pada Gambar 3.15, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.9.

Gambar 10.15. Rancangan Tampilan Menu *Mini Game*

Tabel 10.9. Spesifikasi Atribut Tampilan *Mini Game Attack*

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Tombol *Attack*	Tombol	Menuju *mini game Attack*	*Action*

2	Tombol *Defense*	Tombol	Menuju *mini game Defense*	*Action*
3	Tombol *Speed*	Tombol	Menuju *mini game Speed*	*Action*
4	Tombol *Special Attack*	Tombol	Menuju *mini game Special Attack*	*Action*
5	Tombol *Special Defense*	Tombol	Menuju *mini game Special Defense*	*Action*
6	Tombol *Health*	Tombol	Menuju *mini game Health*	*Action*
7	Tombol *Back*	Tombol	Menuju permainan utama	*Action*
8	Tombol *Help*	Tombol	Menuju halaman tutorial dari *mini game*	*Action*

3.2.2.2. Tampilan *Mini Game Attack*

Halaman ini ditampilkan setelah pengguna menekan tombol *Attack* pada menu *mini game.* Pengguna dapat menggerakkan pemain dalam permainan dengan menekan tombol arah. Pengguna juga dapat menembak target dengan menekan tombol tembak. Pada halaman ini terdapat teks kesempatan, waktu, serta skor. Rancangan tampilan *mini game Attack* dapat dilihat pada Gambar 3.16, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.10.

Gambar 10.16. Rancangan Tampilan *Mini Game Attack*

Tabel 10.10. Spesifikasi Atribut Tampilan *Mini Game Attack*

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Teks kesempatan	Teks	Menampilkan teks sisa kesempatan pemain	*String*
2	Teks waktu	Teks	Menampilkan teks sisa waktu permainan	*String*
3	Teks skor	Teks	Menampilkan skor yang didapat pemain	*String*
4	Tombol bergerak ke kiri	Tombol	Menggerakkan pemain ke arah kiri	*Action*
5	Tombol bergerak ke kanan	Tombol	Menggerakkan pemain ke arah kanan	*Action*

6	Tombol menembak	Tombol	Mengeluarkan tembakkan pemain	*Action*

3.2.2.3. Tampilan *Mini Game Defense*

Halaman ini ditampilkan setelah pengguna menekan tombol *Defense* pada menu *mini game*. Pengguna dapat menggerakkan pemain dalam permainan dengan cara menge-*tap* layar dari perangkat. Pada halaman ini tidak terdapat atribut lain yang berpengaruh terhadap permainan. Rancangan tampilan *mini game Defense* dapat dilihat pada Gambar 3.16.

Gambar 10.17. Rancangan Tampilan *Mini Game Defense*

3.2.2.4. Tampilan *Mini Game Speed*

Halaman ini ditampilkan setelah pengguna menekan tombol *Speed* pada menu *mini game*. Pada halaman ini terdapat teks kesempatan dan skor. Untuk kontrol terhadap permainan, pengguna dapat menge-*tap* layar dari perangkat untuk menghilangkan musuh. Rancangan tampilan *mini game Speed*

dapat dilihat pada Gambar 3.17, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.11.

Gambar 10.18. Rancangan Tampilan *Mini Game Speed*

Tabel 10.11. Spesifikasi Atribut Tampilan *Mini Game Speed*

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Teks kesempatan	Teks	Menampilkan teks sisa kesempatan pemain	*String*
2	Teks skor	Teks	Menampilkan skor yang didapat pemain	*String*

3.2.2.5. Tampilan *Mini Game Special Attack*

Halaman ini ditampilkan setelah pengguna menekan tombol *Special Attack* pada menu *mini game.* Pada halaman ini terdapat teks progres permainan dan teks sisa waktu. Untuk kontrol

terhadap permainan, pengguna dapat menge-*tap* layar dari perangkat untuk menempatkan objek pada tempatnya. Rancangan tampilan *mini game Attack* dapat dilihat pada Gambar 3.19, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.12.

Gambar 10.19. Rancangan Tampilan *Mini Game Special Attack*

Tabel 10.12. Spesifikasi Atribut Tampilan *Mini Game Special Attack*

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Teks progres	Teks	Menampilkan progres permainan	*String*
2	Teks waktu	Teks	Menampilkan sisa waktu permainan	*String*

3.2.2.6. Tampilan *Mini Game Special Defense*

Halaman ini ditampilkan setelah pengguna menekan tombol *Special Defense* pada menu *mini game.* Pada halaman ini terdapat teks jumlah *special defense* dan teks skor permainan. Pengguna dapat menggerakkan pemian dalam permainan dengan menekan tombol aksi. Rancangan tampilan *mini game Attack* dapat dilihat pada Gambar 3.20, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.13.

Gambar 10.20. Rancangan Tampilan *Mini Game Special Defense*

Tabel 10.13. Spesifikasi Atribut Tampilan *Mini Game Special Defense*

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Teks sisa *special defense*	Teks	Menampilkan teks sisa *special defense* pemain	*String*
2	Teks skor	Teks	Menampilkan skor permainan	*String*

3	Tombol bergerak ke atas	Tombol	Menggerakkan pemain ke arah atas	*Action*
4	Tombol bergerak ke bawah	Tombol	Menggerakkan pemain ke arah bawah	*Action*
5	Tombol aksi *special defense*	Tombol	Mengeluarkan *special defense* pemain	*Action*

3.2.2.7. Tampilan *Mini Game Health*

Halaman ini ditampilkan setelah pengguna menekan tombol *Health* pada menu *mini game*. Pada halaman ini terdapat teks skor pemain yang menandakan jumlah objek yang diambil. Untuk kontrol terhadap permainan, pengguna dapat menge-*tap* layar dari perangkat untuk melompat. Rancangan tampilan *mini game Attack* dapat dilihat pada Gambar 3.21, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.14.

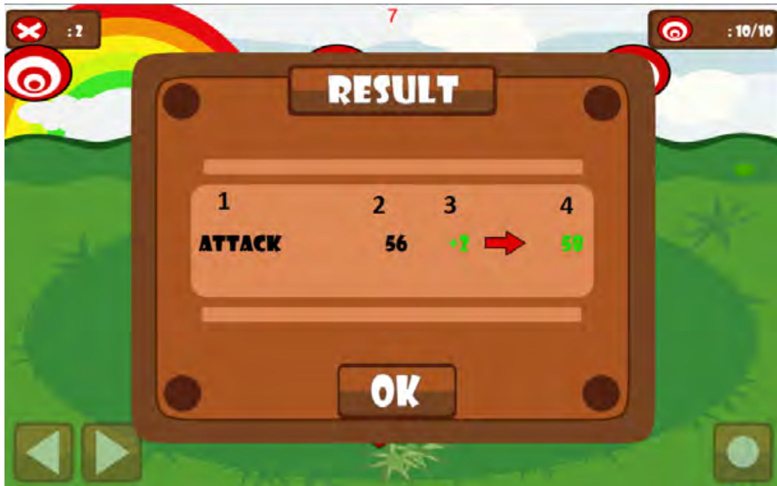
Gambar 10.21. Rancangan Tampilan *Mini Game Health*

Tabel 10.14. Spesifikasi Atribut Tampilan *Mini Game Health*

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Teks skor	Teks	Menampilkan skor yang didapat pemain	*String*

3.2.2.8. Tampilan Hasil Latihan

Tampilan ini merupakan *pop-up* yang muncul setiap permainan selesai. Tampilan ini memberikan informasi dari penambahan atribut hasil latihan. Pada tampilan ini terdapat teks atribut dari *pet* dan hasil penambahan dari latihan. Rancangan tampilan *mini game Attack* dapat dilihat pada Gambar 3.22, sedangkan spesifikasi dari tampilan dapat dilihat pada Tabel 3.15.



Gambar 10.22. Rancangan Tampilan Hasil Latihan

Tabel 10.15. Spesifikasi Atribut Tampilan Hasil Latihan

No.	Nama Atribut Antarmuka	Jenis Atribut	Kegunaan	Jenis Masukan / Keluaran
1	Teks atribut *pet*	Teks	Menampilkan informasi atribut dari *pet* yang dilatih	*String*
2	Teks nilai atribut sebelum latihan	Teks	Menampilkan informasi nilai atribut sebelum memainkan *mini game*	*String*
3	Teks nilai penambahan atribut	Teks	Menampilkan informasi nilai penambahan atribut yang didapatkan pemain	*String*
4	Teks hasil penambahan nilai atribut	Teks	Menampilkan informasi penambahan nilai atribut hasil dari latihan	*String*

BAB IV
IMPLEMENTASI

Bab ini membahas tentang implementasi dari perancangan sistem. Bab ini berisi proses implementasi dari setiap kelas pada semua modul. Bahasa pemrograman yang digunakan adalah bahasa pemrograman C#.

4.1. Implementasi *Builder Pattern*

Konsep *builder pattern* digunakan untuk implementasi level dari *mini game*. Kelas-kelas yang digunakan dalam *builder pattern* ini akan menangani pembuatan level dari masing-masing *mini game* yang dibuat. Dalam modul ini pembuatan level lebih spesifik pada pembuatan rintangan dari permainan. Kelas-kelas yang digunakan untuk mengimplementasikan *builder pattern* ini yaitu kelas `GameLevel`, `LevelBuilder`, `LevelManufacturer`, serta kelas-kelas turunan dari `LevelBuiler` yang berbeda tiap level dari *mini game*.

4.1.1. Kelas GameLevel

Kelas ini digunakan untuk menyimpan objek-objek rintangan yang akan ditampilkan dalam permainan. Objek-objek tersebut disimpan pada variabel *obstacle* yang tipe datanya adalah *List<GameObject>*. Kelas ini dapat dilihat pada Kode Sumber 4.1.

```
1   using UnityEngine;
2   using System.Collections;
3   using System.Collections.Generic;
4   namespace MiniGameModel
5   {
6       public class GameLevel
7       {
8           List<GameObject> obstacle;
9           public List<GameObject> Obstacle
10          {
11              get { return obstacle; }
```

```
12                set { obstacle = value; }
13            }
14            public List<GameObject> CreateLevel()
15            {
16                return obstacle;
17            }
18            public GameLevel()
19            {
20                obstacle = new List<GameObject>();
21            }
22        }
23    }
```

Kode Sumber 4.1. Kelas `GameLevel`

4.1.2. Kelas LevelBuilder

Kelas ini berupa kelas abstrak untuk membuat bagian-bagian dari level. Pada kelas ini hanya terdapat fungsi `BuildObstacle` yang bersifat virtual. Kelas ini dapat dilihat pada Kode Sumber 4.2.

```
1   using UnityEngine;
2   using System.Collections;
3   namespace MiniGameModel
4   {
5       public class LevelBuilder
6       {
7           public virtual void BuildObstacle() { }
8   
9            public GameLevel level;
10      }
11  }
```

Kode Sumber 4.2. Kelas `LevelBuilder`

4.1.3. Kelas LevelManufacturer

Kelas ini bertugas untuk membangun objek dengan menggunakan kelas `AttackLevelBuilder`. Fungsi ini dapat dilihat pada Kode Sumber 4.3.

```
1  public void Construct(LevelBuilder
2  levelBuilder)
3  {
4     levelBuilder.BuildObstacle();
5  }
```

Kode Sumber 4.3. Kelas `LevelManufacturer`

4.2. Implementasi *Mini Game Attack*

Mini game Attack merupakan permainan yang digunakan untuk menaikkan atribut *attack* dari *pet*. Pada bagian ini akan dijelaskan implementasi dari rancangan *mini game Attack*.

4.2.1. Kelas AttackGameManager

Kelas ini merupakan kelas yang digunakan untuk mengatur atribut permainan dari *mini game Attack*. Fungsi `Update` digunakan untuk menghitung dan menampilkan jumlah skor, kesempatan, dan waktu yang tersisa. Sedangkan fungsi `GameOver` dipanggil saat permainan berakhir. Saat fungsi ini dipanggil hasil dari penambahan atribut *attack* dihitung kemudian ditampilkan melalui objek *result*. Fungsi `Update` dan `GameOver` ini dapat dilihat pada Kode Sumber 4.1.

```
1  void Update()
2  {
3   if (!isGameOver)
4      {
5       restTime = finishTime - (int)Time.time;
6       if (restTime >= 0)
7          {
8           timerText.text = restTime.ToString();
```

```
9              }
10         else
11             {
12              timerText.text = "0";
13              GameOver();
14             }
15  chanceText.text = "Chance: " +
16  AttackPlayer.lives.ToString();
17   if (AttackPlayer.lives == 0)
18        GameOver();
19   scoreText.text = AttackPlayer.points + "/10";
20   if (AttackPlayer.points == 10)
21        GameOver();
22   }
23  }
24  void GameOver()
25  {
26   isGameOver = true;
27   Instantiate(result,
28  result.transform.position,
29  result.transform.rotation);
30
31  calculator.CalculateAttack(AttackPlayer.lives,
32  AttackPlayer.points);
33  audio.enabled = false;
34  }
```

Kode Sumber 4.4. Fungsi `Update` dan `GameOver` pada Mini Game Attack

4.2.2. Kelas AttackPlayer

Kelas ini digunakan untuk mengatur perilaku dari pemain. Kelas ini memiliki atribut *lives* dan *score* yang menyimpan nilai kesempatan dan skor pemain. Pada kelas ini terdapat fungsi `Start`, `Update`, `Move`, dan `Shoot`. Fungsi `Start` digunakan untuk inisialisasi variabel-variabel pada kelas ini saat pertama kali dijalankan. Fungsi `Update` digunakan untuk mengatur waktu jeda dari tembakan pemain. Fungsi `Move` digunakan untuk mengatur pergerakan dari pemain. Fungsi ini memiliki parameter yang digunakan untuk menentukan arah dari pergerakan pemain.

Sedangkan fungsi `Shoot` digunakan untuk mengeluarkan tembakan pemain saat tombol tembak pada layar ditekan oleh pengguna.

4.2.3. Kelas AttackTarget

Kelas ini digunakan untuk mengatur perilaku dari objek target pada permainan. Pada kelas ini terdapat fungsi `Update` dan `OnTriggerEnter2D`. Fungsi `Update` digunakan untuk menggerakkan objek target setiap *frame*-nya. Sedangkan fungsi `OnTriggerEnter2D` digunakan untuk mengatur objek target saat terjadi tabrakan dengan objek lain.

4.2.4. Kelas AttackTargetSpawner

Kelas ini digunakan untuk mengeluarkan objek target pada permainan. Pada kelas ini terdapat fungsi `Start` dan `SpawnEnemy`. Fungsi *Start* digunakan untuk menjalankan fungsi `InvokeRepeating` dari Unity saat pertama kali kelas ini dijalankan. Sedangkan fungsi `SpawnEnemy` digunakan untuk mengeluarkan objek target.

4.2.5. Kelas AttackShoot

Kelas ini digunakan untuk mengatur gerakan dari tembakan pemain. Pada kelas ini terdapat fungsi `Update` dan `OnTriggerEnter2D`. Fungsi `Update` digunakan untuk menggerakkan tembakan dari pemain. Sedangkan fungsi `OnTriggerEnter2D` digunakan untuk mengatur objek target saat terjadi tabrakan dengan objek lain.

4.2.6. Kelas AttackObstacle

Kelas ini digunakan untuk mengatur perilaku dari rintangan dalam permainan. Pada kelas ini terdapat fungsi *Update* dan `OnTriggerEnter2D`. Fungsi `Update` digunakan untuk

menggerakkan objek rintangan dalam permainan. Sedangkan fungsi `OnTriggerEnter2D` digunakan untuk mengatur objek target saat terjadi tabrakan dengan objek lain.

4.2.7. Kelas Level

Terdapat tiga kelas level pada *mini game Attack,* yaitu kelas `Level1`, `Level2`, dan `Level3`. Masing-masing kelas tersebut menangani pembuatan rintangan pada *mini game Attack.* Kelas-kelas tersebut menentukan daftar objek rintangan pada kelas `GameLevel`.

4.3. Implementasi *Mini Game Defense*

Mini game Defense merupakan permainan yang digunakan untuk menaikkan atribut *defense* dari *pet*. Pada bagian ini akan dijelaskan implementasi dari rancangan *mini game Defense.*

4.3.1. Kelas DefenseGameManager

Kelas ini merupakan kelas yang digunakan untuk mengatur atribut permainan dari *mini game Defense.* Pada kelas ini terdapat fungsi `Start` dan `Update`. Fungsi `Start` digunakan untuk inisialisasi variabel dan penentuan level permainan. Sedangkan fungsi `Update` digunakan untuk mengecek apakah permainan sudah berakhir. Jika permainan telah berakhir maka fungsi ini akan menampilkan hasil dari latihan.

4.3.2. Kelas DefensePlayer

Kelas ini bertujuan untuk mengatur perilaku dari pemain. Pada kelas ini terdapat variabel *startPos* dan *maxPos* yang digunakan untuk menyimpan nilai dari posisi pemain. Kelas ini memiliki fungsi `Start`, `Update`, dan `OnTriggerEnter2D`. Fungsi `Start` digunakan untuk inisialisasi variabel, sedangkan fungsi `Update` digunakan untuk mengatur gerakan pemain dengan mengecek inputan pengguna. Fungsi

`OnTriggerEnter2D` digunakan untuk mengecek saat terjadi tabrakan dengan objek lain.

4.3.3. Kelas DefenseEnemy

Kelas ini bertujuan untuk mengatur perilaku dari musuh. Pada kelas ini terdapat fungsi `Update` yang mengatur perilaku musuh untuk mengeluarkan tembakan. Fungsi ini dapat dilihat pada Kode Sumber 4.3.

```
1  void Update()
2  {
3     if (Time.time > nextShoot)
4        {
5          nextShoot = Time.time + shootDelay;
6          Instantiate(bullet, new
7  Vector2(transform.position.x - 1.1f,
8  transform.position.y), transform.rotation);
9        }
10 }
```

Kode Sumber 4.5. Fungsi `Update` pada kelas DefenseEnemy

4.3.4. Kelas DefenseEnemyBullet

Kelas ini digunakan untuk menggerakkan objek peluru dari musuh. Pada kelas ini terdapat fungsi `Update` yang digunakan untuk mengubah posisi dari objek peluru setiap *frame*-nya.

4.3.5. Kelas DefenseObstacle

Kelas ini digunakan untuk mengatur perilaku dari objek rintangan pada *mini game Defense*. Kelas ini memiliki fungsi `Update` yang digunakan untuk menggerakkan sekaligus mengatur tembakan dari objek rintangan.

4.3.6. Kelas Level

Terdapat tiga kelas level pada *mini game Defense,* yaitu kelas `Level1`, `Level2`, dan `Level3`. Masing-masing kelas tersebut menangani pembuatan rintangan pada *mini game Defense.* Kelas-kelas tersebut menentukan daftar objek rintangan pada kelas `GameLevel`.

4.4. Implementasi *Mini Game Speed*

Mini game Speed merupakan permainan yang digunakan untuk menaikkan atribut *speed* dari *pet*. Pada bagian ini akan dijelaskan implementasi dari rancangan *mini game Speed.*

4.4.1. Kelas SpeedGameManager

Kelas ini merupakan kelas yang digunakan untuk mengatur atribut permainan dari *mini game Speed.* Pada kelas ini terdapat fungsi `Start` dan `Update`. Fungsi `Start` digunakan untuk inisialisasi variabel dan penentuan level permainan. Sedangkan fungsi `Update` digunakan untuk mengecek masukkan dari pengguna serta mengecek apakah permainan sudah berakhir. Metode untuk mengecek masukkan dari pengguna menggunakan *RaycastHit2D*. fungsi ini dapat dilihat pada Kode Sumber 4.5. Jika permainan telah berakhir maka fungsi ini akan menampilkan hasil dari latihan.

```
1   RaycastHit2D tapEnemy =
2   Physics2D.Raycast(Camera.main.ScreenToWorldPoin
3   t(Input.mousePosition), Vector2.zero);
4   if ((Input.GetMouseButtonUp(0) ||
5   (Input.touchCount > 0 &&
6   Input.GetTouch(0).phase == TouchPhase.Began))
7   && tapEnemy.collider != null)
8   {
9     if (tapEnemy.collider.tag == "Enemy")
10        {
11  Destroy(tapEnemy.collider.gameObject);
12                          score++;
```

```
13          }
14
15     else if (tapEnemy.collider.tag ==
16 "OtherObject")
17             {
18 Destroy(tapEnemy.collider.gameObject);
19                         chance--;
20             }
21 }
```

Kode Sumber 4.6. Fungsi Pengecek Masukan Pengguna pada Kelas Mini Game Speed

4.4.2. Kelas SpeedEnemy

Kelas ini merupakan kelas yang digunakan untuk mengatur perilaku dari musuh. Pada kelas ini terdapat fungsi `Update` dan `OnCollisionEnter2D`. Fungsi `Update` digunakan untuk menggerakkan musuh. Sedangkan fungsi `OnCillisionEnter2D` digunakan untuk mengecek saat pengguna menge-*tap* musuh pada layar perangkat.

4.4.3. Kelas SpeedOtherObject

Kelas ini merupakan kelas yang digunakan untuk mengatur perilaku dari objek lain pada *mini game Speed*. Pada kelas ini terdapat fungsi `Update` dan `OnCollisionEnter2D`. Fungsi `Update` digunakan untuk menggerakkan objek. Sedangkan fungsi `OnCillisionEnter2D` digunakan untuk mengecek saat pengguna menge-*tap* objek pada layar perangkat.

4.4.4. Kelas SpeedObjectSpawner

Kelas ini digunakan untuk mengatur objek-objek yang dikeluarkan pada *mini game Speed.* Pada kelas ini terdapat fungsi `Start` dan `Update`. Fungsi `Start` digunakan untuk membaca objek dari *resource*. Fungsi `Update` digunakan untuk mengeluarkan objek dari *mini game Speed* secara berkala.

4.4.5. Kelas Level

Terdapat tiga kelas level pada *mini game Speed,* yaitu kelas `Level1`, `Level2`, dan `Level3`. Masing-masing kelas tersebut menangani pembuatan rintangan pada *mini game Speed.* Kelas-kelas tersebut menentukan daftar objek rintangan pada kelas `GameLevel`.

4.5. Implementasi *Mini Game Special Attack*

Mini game Special Attack merupakan permainan yang digunakan untuk menaikkan atribut *special attack* dari *pet*. Pada bagian ini akan dijelaskan implementasi dari rancangan *mini game Special Attack.*

4.5.1. Kelas SpecialAttackGameManager

Kelas ini digunakan untuk mengatur atribut permainan pada *mini game Special Defense.* Pada kelas ini terdapat fungsi `Start`, `Update`, `UpdateTime`, `CheckTarget`, dan `GameOver`. Fungsi `Start` digunakan untuk inisialisasi variabel yang digunakan pada kelas ini. Fungsi `Update` digunakan untuk memperbarui teks progres dan sisa waktu permainan. Fungsi `UpdateTime` digunakan untuk memperbarui waktu permainan. Fungsi `CheckTarget` digunakan untuk mengecek objek target apakah masih ada dalam permainan atau tidak. Pengecekan ini menggunakan *RaycastHit2D.* Sedangkan fungsi `GameOver` digunakan untuk menghentikan permainan dan menampilkan hasil latihan.

4.5.2. Kelas MatchThree

Kelas ini digunakan untuk menangani aturan dari permainan *match-three* yang ada pada *mini game Special Attack.* Pada kelas terdapat fungsi `Start`, `Update`, `GenerateGrid`, `InitTile`, `UpdateGrid`, dan `UpdatePresentTile`.

Fungsi `Start` digunakan untuk menentukan level dari permainan. Fungsi `Update` digunakan untuk mengecek masukkan dari pengguna untuk memasangkan objek pada *grid.* Fungsi pengecekan ini menggunakan *RaycastHit2D.* Fungsi `GenerateGrid` digunakan untuk mengeluarkan objek *grid* berupa kotak-kotak berukuran 6x7. Fungsi ini dapat dilihat pada Kode Sumber 4.7. Fungsi `InitTile` digunakan untuk menginisialisasi gambar *tile* yang terpasang pada objek *grid.* Sedangkan fungsi `UpdateGrid` digunakan untuk memperbarui gambar yang terpasang pada objek *grid.* Fungsi `UpdatePresentTile` digunakan untuk memperbarui objek *tile* yang bisa diletakkan pada objek *grid.* Fungsi ini dapat dilihat pada Kode Sumber 4.8.

```
1   void GenerateGrid()
2   {
3    for (int i = 0; i < 6; i++)
4         {
5          for (int j = 0; j < 7; j++)
6               {
7                Vector2 gridPosition =
8   tileData.GetPosition(j, i);
9                Instantiate(grid, gridPosition,
10  transform.rotation);
11              }
12        }
13  }
```

Kode Sumber 4.7. Fungsi `UpdateGrid` pada Kelas `MatchThree`

```
1   void UpdatePresentTile()
2   {
3    tileNow = Random.Range(0, 6);
4    Vector2 position = new Vector2(-3.4f, 3.7f);
5    RaycastHit2D presentTile =
6   Physics2D.Raycast(position, Vector2.zero);
7    if (presentTile.collider != null &&
8   presentTile.collider.tag == "Tile")
9      {
10
```

```
11        tileSprite =
12  presentTile.collider.gameObject.GetComponent<Sp
13  riteRenderer>();
14        tileSprite.sprite = spriteTile[tileNow];
15      }
    }
```

Kode Sumber 4.8. Fungsi `UpdatePresentTile` pada Kelas `MatchThree`

4.5.3. Kelas TileData

Kelas ini berperan sebagai kelas model dari permainan *match-three* pada *mini game Special Attack.* Kelas ini digunakan untuk menyimpan data *grid* dari permainan. Kelas ini menerapkan *singleton pattern* untuk menyimpan data *grid* tersebut agar dapat diakses dari kelas lain. Pada kelas ini terdapat fungsi `InitGrid`, `GetTileIndex`, dan `GetPosition`. Fungsi `InitGrid` digunakan untuk menginisialisasi nilai dari *grid* secara acak dalam rentang tertentu. Fungsi `GetTileIndex` digunakan untuk mendapatkan indeks dari *tile* pada koordinat tertentu. Sedangkan fungsi `GetPosition` merupakan kebalikan dari fungsi `GetTileIndex`. Fungsi ini digunakan untuk mengembalikan koordinat dari *tile* berdasarkan indeksnya.

4.5.4. Kelas TileSolver

Kelas ini berfungsi untuk menangani penyelesaian dari permainan *match-three.* Pada kelas ini terdapat fungsi `SolveTheGrid` yang digunakan untuk menyelesaikan data *grid* pada permainan *match-three*. Setiap kali pengguna meletakkan objek *tile* pada permainan fungsi ini akan dipanggil untuk mengecek apakah ada kondisi *match-three* yang terpenuhi. Fungsi ini dapat dilihat pada Kode Sumber 4.9. Untuk mengeceknya akan dipanggil fungsi `CheckMatched`. Fungsi ini mengimplementasi algoritma *Deep First Search* (DFS). Fungsi ini dapat dilihat pada Kode Sumber 4.10. Setelah itu dipanggil fungsi `MatchedCount` untuk mengetahui jumlah *tile* yang sama dan terhubung. Jika

jumlahnya tiga atau lebih maka *tile* tersebut dihapus dengan memanggil fungsi `DeleteMatched`.

```
1   void GenerateGrid()
2   {
3    for (int i = 0; i < 6; i++)
4         {
5          for (int j = 0; j < 7; j++)
6              {
7               Vector2 gridPosition =
8   tileData.GetPosition(j, i);
9               Instantiate(grid, gridPosition,
10  transform.rotation); } } }
```

Kode Sumber 4.9. Fungsi `SolveTheGrid` pada Kelas `TileSolver`

```
1   void CheckMatched(int i, int j, int value)
2   {
3     flag[i, j] = true;
4     if (i - 1 >= 0 && tileData.GridTile[i - 1, j]
5   == value && !flag[i - 1, j])
6     CheckMatched(i - 1, j, value);
7     if (j + 1 < 7 && tileData.GridTile[i, j + 1]
8   == value && !flag[i, j + 1])
9     CheckMatched(i, j + 1, value);
10    if (i + 1 < 6 && tileData.GridTile[i + 1, j]
11  == value && !flag[i + 1, j])
12    CheckMatched(i + 1, j, value);
13    if (j - 1 >= 0 && tileData.GridTile[i, j - 1]
14  == value && !flag[i, j - 1])
15    CheckMatched(i, j - 1, value); }
```

Kode Sumber 4.10. Fungsi `CheckMatched` pada Kelas `TileSolver`

4.5.5. Kelas SpecialAttackMagic

Kelas ini digunakan untuk mengatur perilaku dari objek *special attack*. Pada kelas ini terdapat fungsi `Update` dan `OnCollisionEnter2D`. Fungsi `Update` digunakan untuk menggerakkan objek *special attack*. Sedangkan fungsi

`OnCollisionEnter2D` digunakan untuk mengatur objek saat terjadi tabrakan dengan objek lain.

4.5.6. Kelas Level

Terdapat tiga kelas level pada *mini game Special Attack,* yaitu kelas `Level1`, `Level2`, dan `Level3`. Masing-masing kelas tersebut menangani pembuatan rintangan pada *mini game Special Attack.* Kelas-kelas tersebut menentukan daftar objek rintangan pada kelas `GameLevel`.

4.6. Implementasi *Mini Game Special Defense*

Mini game Special Defense merupakan permainan yang digunakan untuk menaikkan atribut *special defense* dari *pet*. Pada bagian ini akan dijelaskan implementasi dari rancangan *mini game Special Defense.*

4.6.1. Kelas SpecialDefenseGameManager

Kelas ini digunakan untuk mengatur jalannya permainan dari *mini game Special Defense*. Pada kelas ini terdapat fungsi `Start`, `Update`, `ReleaseBall`, `CheckBallPosition`, `Init`, dan `GameOver`. Fungsi *Start* digunakan untuk menentukan level dari permainan. Fungsi `Update` digunakan untuk memperbarui teks pada permainan. Fungsi `ReleaseBall` digunakan untuk mengeluarkan objek bola dalam permainan. Setelah objek bola dikeluarkan, fungsi `Update` akan memanggil fungsi `CheckBallPosition` untuk mengetahui posisi dari objek bola di setiap *frame*-nya. Setelah tujuan permainan tercapai atau pemain kalah, fungsi `GameOver` akan dipanggil untuk mengakhiri permainan.

4.6.2. Kelas SpecialDefensePlayer

Kelas ini digunakan untuk mengatur perilaku dari pemain dalam permainan. Pada kelas ini terdapat fungsi `Move`, `Defense`, dan `OnCollisionEnter2D`. Fungsi `Move` digunakan untuk mengatur pergerakkan dari pemain. Fungsi ini memiliki parameter utnuk menentukkan arah dari pergerakkan pemain. Fungsi `Defense` digunakan untuk mengeluarkan objek *special defense* dari pemain. Sedangkan fungsi `OnCollisionEnter2D` digunakan untuk mengatur pemain ketika bertabrakan dengan objek lain.

4.6.3. Kelas SpecialDefenseEnemy

Kelas ini digunakan untuk mengatur perilaku musuh dalam permainan. Pada kelas ini terdapat fungsi `Update`, `Move`, dan `OnCollisionEnter2D`. Fungsi `Update` digunakan untuk menjalankan fungsi `Move` setiap *frame* dari permainan. Fungsi `Move` mengatur pergerakan dari musuh. Musuh akan berusaha untuk mencegah bola masuk ke daerahnya dengan cara mengejarnya. Fungsi ini dapat dilihat pada Kode Sumber 4.12. Fungsi `OnCollisionEnter2D` digunakan untuk mengatur objek musuh saat bertabrakan dengan objek lain.

```
1   void Move(float x, float y)
2   {
3    if (GameObject.FindWithTag("Ball"))
4       {
5        ball =
6   GameObject.FindWithTag("Ball").transform;
7        if (ball.position.x > 0)
8            {
9             speed = Random.Range(0.01f, 0.1f);
10             if (y < ball.position.y &&
11  Mathf.Abs(y - ball.position.y) > 0.1f)
12                 transform.position = new
13  Vector2(x, y + speed);
14             if (y > ball.position.y &&
    Mathf.Abs(y - ball.position.y) > 0.1f)
```

```
15                     transform.position = new
16  Vector2(x, y - speed);
17              }
18          }
19  }
20
```

Kode Sumber 4.11. Fungsi `Move` pada Kelas `SpecialDefenseEnemy`

4.6.4. Kelas SpecialDefenseBall

Kelas ini digunakan untuk mengatur pergerakkan dari objek bola dalam permainan. Pada kelas ini terdapat fungsi `Start` dan `Update`. Fungsi `Start` digunakan untuk menginisialisasi gaya pada bola sehingga bola bisa bergerak. Fungsi `Update` digunakan untuk mengatur objek bola saat keluar dari arena permainan.

4.6.5. Kelas SpecialDefenseObstacle

Kelas ini digunakan untuk mengatur perilaku dari objek rintangan pada *mini game Special Defense*. Pada kelas ini terdapat fungsi `Update` dan `OnCollisionEnter2D`. Fungsi `Update` digunakan untuk menggerakkan objek *special attack*. Sedangkan fungsi `OnCollisionEnter2D` digunakan untuk mengatur objek saat terjadi tabrakan dengan objek lain.

4.6.6. Kelas SpecialDefenseShield

Kelas ini digunakan untuk mengatur perilaku dari objek perisai dalam *mini game Special Defense*. Pada kelas ini terdapat fungsi `Start` dan `Update`. Fungsi `Start` digunakan untuk inisialisasi variabel yang digunakan pada kelas ini. Fungsi `Update` digunakan untuk mengatur waktu dari objek perisai untuk menghilang.

4.6.7. Kelas SpecialDefenseWall

Kelas ini digunakan untuk mengatur perilaku dari objek dinding pembatas pada *mini game Special Defense.* Kelas ini memiliki fungsi `OnCollisionEnter2D` yang digunakan untuk memantulkan bola yang menabraknya. Fungsi ini dapat dilihat pada Kode Sumber 4.13.

```
1   void OnCollisionEnter2D(Collision2D collision)
2   {
3    if (collision.gameObject.rigidbody2D != null)
4       {
5        float velocity =
6   collision.gameObject.rigidbody2D.velocity.magni
7   tude;
8        collision.gameObject.rigidbody2D.velocity
9   = new
10  Vector2(collision.gameObject.rigidbody2D.veloci
11  ty.x, (collision.transform.position.y -
12  transform.position.y) * 4);
13       if
14  (collision.gameObject.rigidbody2D.velocity.magn
15  itude < velocity)
16         {
17         collision.gameObject.rigidbody2D.velocity
18  *= velocity /
19  collision.gameObject.rigidbody2D.velocity.magni
20  tude;
21         }
22      }
23  }
```

Kode Sumber 4.12. Fungsi `OnCollisionEnter` pada Kelas `SpecialDefenseWall`

4.5.7. Kelas Level

Terdapat tiga kelas level pada *mini game Special Defense,* yaitu kelas `Level1`, `Level2`, dan `Level3`. Masing-masing kelas tersebut menangani pembuatan rintangan pada *mini game*

Special Defense. Kelas-kelas tersebut menentukan daftar objek rintangan pada kelas `GameLevel`.

4.7. Implementasi *Mini Game Health*

Mini game Health merupakan permainan yang digunakan untuk menaikkan atribut *health* dari *pet*. Pada bagian ini akan dijelaskan implementasi dari rancangan *mini game Health.*

4.7.1. Kelas HealthGameManager

Kelas ini digunakan untuk mengatur jalannya permainan dari *mini game Health.* Pada kelas ini terdapat fungsi `Start`, `Update`, `Init`, dan `GameOver`. Fungsi `Start` digunakan untuk menentukan level dari *mini game Health.* Fungsi `Update` digunakan untuk memperbarui teks skor dari pemain dan kondisi permainan setiap *frame*-nya. Fungsi `Init` digunakan untuk inisialisasi variabel yang dipakai pada kelas ini. Sedangkan fungsi `GameOver` dijalankan apabila permainan telah berakhir.

4.7.2. Kelas HealthPlayer

Kelas ini digunakan untuk mengatur perilaku dari pemain dalam *mini game Health.* Pada kelas ini terdapat fungsi `Start`, `Update`, dan `OnCollisionEnter2D`. Saat fungsi `Start` dijalankan variabel *OnGround* diisi dengan nilai *true*. Variabel ini digunakan untuk mengecek apakah pemain menginjak tanah di dalam permainan. Sedangkan pada fungsi `Update` terdapat metode untuk mengecek masukan dari pengguna yang bertujuan untuk membuat pemain melompat. Pengecekan dilakukan dengan menggunakan *RaycastHit2D.* Fungsi `OnCollisionEnter2D` digunakan untuk mengatur pemain saat terjadi tabrakan dengan objek lain.

4.7.3. Kelas HealthGround

Kelas ini digunakan untuk mengatur pergerakan dari objek tanah pada permainan. Pergerakan tersebut diatur pada fungsi `Update`. Selain itu fungsi ini juga mengatur saat objek tanah keluar dari layar. Objek tanah akan di hilangkan dari permainan jika keluar dari layar.

4.7.4. Kelas HealthItem

Kelas ini digunakan untuk mengatur perilaku dari objek *health* pada permainan. Pada kelas ini terdapat fungsi `Update` dan `OnCollisionEnter2D`. Fungsi `Update` digunakan untuk menggerakkan objek *health* serta menghilangkannya saat keluar dari layar permainan. Fungsi `OnCollisionEnter2D` digunakan untuk mengatur objek *health* saat terjadi tabrakan dengan objek pemain.

4.7.5. Kelas HealthObjectSpawner

Kelas ini digunakan untuk mengatur keluarnya objek pada *mini game Health*. Pada kelas ini terdapat fungsi `Start`, `Update`, `SpawnGround`, dan `SpawnHealth`. Saat fungsi `Start` dijalankan, fungsi ini akan menginisialisasi objek tanah pada permainan. Untuk selanjutnya fungsi `Update` akan mengeluarkan objek tanah dan `health` dengan jeda waktu tertentu dengan memanggil fungsi `SpawnGround` dan `SpawnHealth`. Fungsi ini dapat dilihat pada Kode Sumber 4.15.

```
1  void SpawnGround(float x, float y)
2  {
3   Instantiate(ground, new Vector2(x, y),
4  transform.rotation);
5   previousY = nextY;
6   delay = 0;
7  }
8
9  void SpawnHealth(float x, float y)
```

```
10 {
11  int rand = Random.Range(0, 2);
12  if (rand == 1)
13  Instantiate(health, new Vector2(x, y),
14 transform.rotation);
15 }
```

Kode Sumber 4.13. Fungsi `SpawnGround` dan `SpawnHealth` pada Kelas `HealthObjectSpawner`

4.7.6. Kelas HealthObstacleSpawner

Kelas ini mengatur keluarnya objek-objek rintangan yang ada pada *mini game Health.* Pada kelas ini terdapat fungsi `Start` dan `Update`. Fungsi `Start` digunakan untuk inisialisasi variabel pada kelas ini. Fungsi `Update` digunakan untuk mengeluarkan objek rintangan pada permainan.

4.6.8. Kelas `Level`

Terdapat tiga kelas level pada *mini game Health,* yaitu kelas `Level1`, `Level2`, dan `Level3`. Masing-masing kelas tersebut menangani pembuatan rintangan pada *mini game Health.* Kelas-kelas tersebut menentukan daftar objek rintangan pada kelas `GameLevel`.

LAMPIRAN

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using AttackGameLevel;
using MiniGameModel;
namespace MiniGame
{
    public class AttackGameManager : MonoBehaviour
    {
        public GUIText timerText, chanceText,
scoreText;
        public GameObject result;
        int finishTime, restTime;
        bool isCompleted, isGameOver;
        TrainingPetAttributs petInfo;
        ResultCalculator calculator;
        void Start()
        {
            petInfo =
TrainingPetAttributs.CreateTrainingAtributSingleton
();
            calculator = new ResultCalculator();
            if (petInfo.GetTrainingLevel() != 1)
            {
                LevelManufacturer newManufacturer =
new LevelManufacturer();
                LevelBuilder levelBuilder = null;
                if (petInfo.GetTrainingLevel() ==
2)
                {
                    levelBuilder = new Level2();
                }
                else levelBuilder = new Level3();

newManufacturer.Construct(levelBuilder);
                foreach (GameObject i in
levelBuilder.Level.Obstacle)
                {
                    Instantiate(i,
i.transform.position, i.transform.rotation);
                }
```

```
            }

            GameStart();
        }
        void Update()
        {
            if (!isGameOver)
            {
                restTime = finishTime -
(int)Time.time;
                if (restTime >= 0)
                {
                    timerText.text =
restTime.ToString();
                }
                else
                {
                    timerText.text = "0";
                    GameOver();
                }
                chanceText.text = "Chance:  " +
AttackPlayer.lives.ToString();
                if (AttackPlayer.lives == 0)
                {
                    GameOver();
                }
                scoreText.text =
AttackPlayer.points + "/10";
                if (AttackPlayer.points == 10)
                {
                    isCompleted = true;
                    GameOver();
                }
            }
        }
        void GameStart()
        {
            isCompleted = false;
            isGameOver = false;

            finishTime = (int)Time.time + 30;
        }

        void GameOver()
```

```
            {
                isGameOver = true;
                Instantiate(result,
result.transform.position,
result.transform.rotation);

calculator.CalculateAttack(AttackPlayer.lives,
AttackPlayer.points);
                audio.enabled = false;
            }
        }
}
```

Kode Sumber 8.1. Kelas `AttackGameManager`

```
using UnityEngine;
using System.Collections;

namespace MiniGame
{
    public class AttackPlayer : MonoBehaviour
    {
        public static int lives, points;
        public GameObject bullet;
        public float bulletSpeed;
        float nextShoot, shootDelay;
        void Start()
        {
            shootDelay = 0.5f;
            nextShoot = 0;
            lives = 3;
            points = 0;
        }

        public float speed;
        void Update()
        {
            nextShoot += Time.deltaTime;
        }
        void Move(string param)
        {
            int direction = (param == "1" ? 1:-1);
            if ((transform.localPosition.x > -7.5f
&& direction == -1) || (transform.localPosition.x <
7.5f && direction == 1))
```

```
                {
                    transform.Translate(Vector2.right *
(speed * direction));
                }
            }

            void Shoot()
            {
                if (nextShoot > shootDelay)
                {
                    this.audio.Play();
                    nextShoot = 0;
                    Instantiate(bullet, new
Vector2(transform.position.x, transform.position.y
+ 1.2f), transform.rotation);
                }
            }
        }
}
```

Kode Sumber 8.2. Kelas `AttackPlayer`

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
namespace MiniGameModel
{
    public class GameLevel
    {
        List<GameObject> obstacle;

        public List<GameObject> Obstacle
        {
            get { return obstacle; }
            set { obstacle = value; }
        }
        public GameLevel()
        {
            obstacle = new List<GameObject>();
        }
    }
}
```

Kode Sumber 8.3. `Kelas GameLevel`

```
using UnityEngine;
using System.Collections;
using DefenseGameLevel;
using MiniGameModel;
namespace MiniGame
{
    public class DefenseGameManager : MonoBehaviour
    {
        public GameObject result;
        public static bool isGameOver, isComplete;
        bool flag;
        TrainingPetAttributs petInfo;
        ResultCalculator calculator;
        void Start()
        {
            flag = false;
            petInfo = 
TrainingPetAttributs.CreateTrainingAtributSingleton
();
            calculator = new ResultCalculator();
            if(petInfo.GetTrainingLevel() != 1)
            {
                LevelManufacturer newManufacturer =
new LevelManufacturer();
                LevelBuilder levelBuilder = null;
                if (petInfo.GetTrainingLevel()==2)
                    levelBuilder = new Level2();
                else levelBuilder = new Level3();
newManufacturer.Construct(levelBuilder);
                foreach(GameObject i in
levelBuilder.Level.Obstacle)
                {
                    Instantiate(i,
i.transform.position, i.transform.rotation);
                }
            }
            GameStart();
        }
        void Update()
        {
            if (isGameOver && !flag)
            {
                flag = true;
                audio.enabled = false;
```

```
                    Instantiate(result,
result.transform.position,
result.transform.rotation);
calculator.CalculateDefense(DefensePlayer.maxPos -
DefensePlayer.startPos);
                }
            }
            void GameStart()
            {
                isGameOver = false;
                isComplete = false;
            }
        }
}
```

Kode Sumber 8.4. Kelas `DefenseGameManager`

```
using UnityEngine;
using System.Collections;
using MiniGameModel;
namespace MiniGame
{
    public class DefensePlayer : MonoBehaviour
    {
        public static float startPos, maxPos;
        void Start()
        {
            startPos = maxPos =
transform.localPosition.x;
        }
        void Update()
        {
            if (Input.GetMouseButtonUp(0) ||
(Input.touchCount > 0 && Input.GetTouch(0).phase ==
TouchPhase.Began))
            {
                transform.Translate(Vector2.right *
0.2f);

if(transform.localPosition.x>maxPos)
                {
                    maxPos =
transform.localPosition.x;
                }
            }
```

```
        }

        void OnTriggerEnter2D(Collider2D collider)
        {
            if (collider.tag == "Shoot")
            {
                transform.Translate(Vector2.right *
-0.4f);
            }
            if (collider.tag == "Enemy")
            {
                DefenseGameManager.isComplete =
true;
                DefenseGameManager.isGameOver =
true;
            }
            if (collider.tag == "Boundary" ||
collider.tag == "Ball")
            {
                Destroy(gameObject);
                DefenseGameManager.isComplete =
false;
                DefenseGameManager.isGameOver =
true;
            }
        }
    }
}
```

Kode Sumber 8.5. Kelas `DefensePlayer`

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using SpeedGameLevel;
using MiniGameModel;
namespace MiniGame
{
    public class SpeedGameManager : MonoBehaviour
    {
        public GameObject result;
        public GUIText chanceText, scoreText;
        public static int score, chance;
        bool isGameOver;
        void Start()
```

```
        {
            TrainingPetAttributs petInfo =
TrainingPetAttributs.CreateTrainingAtributSingleton
();
            if(petInfo.GetTrainingLevel() != 1)
            {
                LevelManufacturer newManufacturer =
new LevelManufacturer();
                LevelBuilder levelBuilder = null;

                if (petInfo.GetTrainingLevel()==2)
                    levelBuilder = new Level2();
                else levelBuilder = new Level3();

newManufacturer.Construct(levelBuilder);

                foreach(GameObject i in
levelBuilder.level.Spawner)
                {
                    Instantiate(i,
i.transform.position, i.transform.rotation);
                }
            }
            score = 0;
            chance = 3;
            isGameOver = false;
        }
        void Update()
        {
            if (!isGameOver)
            {
                scoreText.text = score + "/30";
                chanceText.text = "Chance: " +
chance;

                if (score == 30 || chance <= 0)
                {
                    GameOver();
                }

                RaycastHit2D tapEnemy =
Physics2D.Raycast(Camera.main.ScreenToWorldPoint(In
put.mousePosition), Vector2.zero);
```

```
                if ((Input.GetMouseButtonUp(0) ||
(Input.touchCount > 0 && Input.GetTouch(0).phase ==
TouchPhase.Began)) && tapEnemy.collider != null)
                {
                    if (tapEnemy.collider.tag ==
"Enemy")
                    {
Destroy(tapEnemy.collider.gameObject);
                        score++;
                    }
                    else if (tapEnemy.collider.tag
== "OtherObject")
                    {
Destroy(tapEnemy.collider.gameObject);
                        chance--;
                    }
                }
            }
        }
        void GameOver()
        {
            isGameOver = true;
            audio.enabled = false;
            Instantiate(result,
result.transform.position,
result.transform.rotation);
            ResultCalculator calculator = new
ResultCalculator();
            calculator.CalculateSpeed(chance,
score);
        }
    }
}
```

Kode Sumber 8.6. Kelas `SpeedGameManager`

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using MiniGameModel;
using SAttackGameLevel;
namespace MiniGame
{
    public class MatchThree : MonoBehaviour
    {
```

```
        public GameObject grid;
        public Sprite[] spriteTile;
        Vector2 selectedGrid;
        int tileNow;
        SpriteRenderer tileSprite;
        TileData tileData;
        TileSolver solver;
        void Start()
        {
            tileData =
TileData.CreateTileSingleton();
            tileData.InitGrid();
            solver = new TileSolver();
            GenerateGrid();
            InitTile();
            UpdatePresentTile();
            solver.SolveTheGrid();
            UpdateGrid();
            TrainingPetAttributs petInfo =
TrainingPetAttributs.CreateTrainingAtributSingleton
();
            if (petInfo.GetTrainingLevel() != 1)
            {
                LevelManufacturer newManufacturer =
new LevelManufacturer();
                LevelBuilder levelBuilder = null;
                if (petInfo.GetTrainingLevel()==2)
                    levelBuilder = new Level2();
                else levelBuilder = new Level3();
newManufacturer.Construct(levelBuilder);
                foreach (GameObject i in
levelBuilder.level.Obstacle)
                {
                    int x = Random.Range(0, 7);
                    int y = Random.Range(0, 6);
                    Vector2 pos =
tileData.GetPosition(x, y);
                    Instantiate(i, new
Vector3(pos.x, pos.y, -1), i.transform.rotation);
                    tileData.GridTile[y, x] = 99;
                }
            }
        }
        void Update()
```

```
        {
            if (Input.GetMouseButtonUp(0) ||
(Input.touchCount > 0 && Input.GetTouch(0).phase ==
TouchPhase.Ended))
            {
                RaycastHit2D click =
Physics2D.Raycast(Camera.main.ScreenToWorldPoint(In
put.mousePosition), Vector2.zero);
                if (click.collider != null &&
click.collider.tag == "Grid")
                {
                    selectedGrid =
click.collider.gameObject.transform.position;
                    tileSprite =
click.collider.gameObject.GetComponent<SpriteRender
er>();
                    if (tileSprite.sprite == null)
                    {
                        tileSprite.sprite =
spriteTile[tileNow];
                        Vector2 tileIndex =
tileData.GetTileIndex(selectedGrid.x,
selectedGrid.y);

tileData.GridTile[(int)tileIndex.x,
(int)tileIndex.y] = tileNow;
                        UpdatePresentTile();
                        solver.SolveTheGrid();
                        UpdateGrid();
                    }
                }
            }
        }
        void GenerateGrid()
        {
            for (int i = 0; i < 6; i++)
            {
                for (int j = 0; j < 7; j++)
                {
                    Vector2 gridPosition =
tileData.GetPosition(j, i);
                    Instantiate(grid, gridPosition,
transform.rotation);
                }
```

```
            }
        }
        void InitTile()
        {
            for (int i = 0; i < 6; i++)
            {
                for (int j = 0; j < 7; j++)
                {
                    int tileNumber =
tileData.GridTile[i, j];
                    Vector2 tilePosition =
tileData.GetPosition(j, i);
                    RaycastHit2D selectedTile =
Physics2D.Raycast(tilePosition, Vector2.zero);
                    tileSprite =
selectedTile.collider.gameObject.GetComponent<Sprit
eRenderer>();
                    if (tileNumber < 6)
                    {
                        tileSprite.sprite =
spriteTile[tileNumber];
                    }
                    else tileSprite.sprite = null;
                }
            }
        }
        void UpdateGrid()
        {
            for (int i = 0; i < 6; i++)
            {
                for (int j = 0; j < 7; j++)
                {
                    int tileNumber =
tileData.GridTile[i, j];

                    if (tileNumber == 99)
                    {
                        RaycastHit2D selectedTile =
Physics2D.Raycast(tileData.GetPosition(j, i),
Vector2.zero);
                        if (selectedTile.collider
!= null && selectedTile.collider.tag == "Grid")
                        {
```

```
                                        tileSprite =
selectedTile.collider.gameObject.GetComponent<Sprit
eRenderer>();
                                        if (tileSprite.sprite
!= null)
                                        {
SpecialAttackGameManager.progress++;
                                         tileSprite.sprite =
null;
                                        }
                                  }
                            }
                      }
                }
          }
          void UpdatePresentTile()
          {
                tileNow = Random.Range(0, 6);
                Vector2 position = new Vector2(-3.4f,
3.7f);
                RaycastHit2D presentTile =
Physics2D.Raycast(position, Vector2.zero);
                if (presentTile.collider != null &&
presentTile.collider.tag == "Tile")
                {
                      tileSprite =
presentTile.collider.gameObject.GetComponent<Sprite
Renderer>();
                      tileSprite.sprite =
spriteTile[tileNow];
                }
          }
     }
}
```

Kode Sumber 8.7. Kelas `MatchThree`

```
using UnityEngine;
using System.Collections;
namespace MiniGameModel
{
     public class TileData
     {
          private TileData() { }
          private static TileData singleton;
```

```
        public static TileData
CreateTileSingleton()
        {
            if(singleton==null)
            {
                singleton = new TileData();
            }
            return singleton;
        }
        float[] xPos = { -7.3f, -6.0f, -4.7f, -
3.4f, -2.1f, -0.85f, 0.4f };
        float[] yPos = { 2.18f, 0.9f, -0.35f, -
1.6f, -2.9f, -4.2f };
        int[,] gridTile = new int[6, 7];
        bool[,] flag = new bool[6, 7];

        public int[,] GridTile
        {
            get { return gridTile; }
            set { gridTile = value; }
        }
        public void InitGrid()
        {
            for (int i = 0; i < 6; i++)
            {
                for (int j = 0; j < 7; j++)
                {
                    gridTile[i, j] =
Random.Range(0, 16);
                }
            }
        }
        public Vector2 GetTileIndex(float x, float
y)
        {
          Vector2 tileIndex = new Vector2(0, 0);
            for (int i = 0; i < 6; i++)
            {
               for (int j = 0; j < 7; j++)
               {
               if (xPos[j] == x && yPos[i] == y)
               tileIndex = new Vector2(i, j);
               }
            }
```

```
                return tileIndex;
            }
            public Vector2 GetPosition(int i, int j)
            {
               return new Vector2(xPos[i], yPos[j]);
            }
        }
    }
```

Kode Sumber 8.8. Kelas `TileData`

```
using UnityEngine;
using System.Collections;
namespace MiniGameModel
{
    public class TileSolver
    {
        TileData tileData;
        bool[,] flag = new bool[6, 7];
        public TileSolver()
        {
            tileData =
TileData.CreateTileSingleton();
        }
        void InitFlag()
        {
            for (int i = 0; i < 6; i++)
                for (int j = 0; j < 7; j++)
                    flag[i, j] = false;
        }
        public void SolveTheGrid()
        {
            for (int i = 0; i < 6; i++)
            {
                for (int j = 0; j < 7; j++)
                {
                 if (tileData.GridTile[i,j] < 6)
                    {
                        InitFlag();
                        CheckMatched(i, j,
tileData.GridTile[i, j]);
                        if (MatchCount() >= 3)
                        {
                            DeleteMatched();
                        }
```

```
                        }
                    }
                }
            }
            void DeleteMatched()
            {
                for (int i = 0; i < 6; i++)
                {
                    for (int j = 0; j < 7; j++)
                    {
                        if (flag[i, j])
                        {
                        tileData.GridTile[i, j] =99;
                        }
                    }
                }
            }
            int MatchCount()
            {
                int match = 0;
                for (int i = 0; i < 6; i++)
                {
                    for (int j = 0; j < 7; j++)
                    {
                        if (flag[i, j])
                            match++;
                    }
                }

                return match;
            }
            void CheckMatched(int i,int j,int value)
            {
                flag[i, j] = true;
                if (i - 1 >= 0 && tileData.GridTile[i-
1,j]==value && !flag[i-1,j])
                    CheckMatched(i - 1, j, value);
                if (j + 1 < 7 &&
tileData.GridTile[i,j+1]==value && !flag[i,j+1])
                    CheckMatched(i, j + 1, value);
                if (i + 1 < 6 && tileData.GridTile[i +
1, j] == value && !flag[i + 1, j])
                    CheckMatched(i + 1, j, value);
```

```
                if (j - 1 >= 0 &&
tileData.GridTile[i,j-1]==value && !flag[i,j-1])
                    CheckMatched(i, j - 1, value);
            }
        }
}
```

Kode Sumber 8.9. Kelas `TileSolver`

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using SDefenseGameLevel;
using MiniGameModel;
namespace MiniGame
{
    public class SpecialDefenseGameManager :
MonoBehaviour
    {
        public GameObject ballObject, result;
        Transform ball;
        public static int specialDefense;
        public GUIText scoreText, magicDefenseText;
        bool isGameOver, releaseBall;
        float delay, maxDelay;
        int playerScore, enemyScore;
        void Start()
        {
            TrainingPetAttributs petInfo =
TrainingPetAttributs.CreateTrainingAtributSingleton
();
            if (petInfo.GetTrainingLevel() != 1)
            {
                LevelManufacturer newManufacturer =
new LevelManufacturer();
             LevelBuilder levelBuilder = null;

                if (petInfo.GetTrainingLevel()==2)
                    levelBuilder = new Level2();
                else
                    levelBuilder = new Level3();

newManufacturer.Construct(levelBuilder);
                foreach (GameObject i in
levelBuilder.level.Obstacle)
```

```
                {
                    Instantiate(i,
i.transform.position, i.transform.rotation);
                }
            }
            Init();
        }
        void Update()
        {
            if (!isGameOver)
            {
                if (releaseBall)
                    ReleaseBall(delay +=
Time.deltaTime);
                CheckBallPosition();
                scoreText.text =
playerScore.ToString() + " - " +
enemyScore.ToString();
                magicDefenseText.text = "S.
Defense: " + specialDefense.ToString();
              if (playerScore >= 2 || enemyScore >=
2)
                {
                    GameOver();
                }
            }
        }
        void ReleaseBall(float time)
        {
            if (time >= maxDelay)
            {
                Instantiate(ballObject, new
Vector2(0f, 0f), transform.rotation);
                releaseBall = false;
                delay = 0f;
            }
        }
        void CheckBallPosition()
        {
            if (GameObject.FindWithTag("Ball"))
            {
                ball =
GameObject.FindWithTag("Ball").transform;
                if (ball.localPosition.x < -8)
```

```
                {
                    enemyScore++;
                    releaseBall = true;
                }
                if (ball.localPosition.x > 8)
                {
                    playerScore++;
                    releaseBall = true;
                }
            }
        }
        void Init()
        {
            playerScore = enemyScore = 0;
            specialDefense = 3;
            isGameOver = false;
            releaseBall = true;
            delay = 0f;
            maxDelay = 1.0f;
        }
        void GameOver()
        {
            isGameOver = true;
            audio.enabled = false;
            Instantiate(result,
result.transform.position,
result.transform.rotation);
            ResultCalculator calculator = new
ResultCalculator();

calculator.CalculateSpecialDefense(playerScore,
specialDefense);
        }
    }
}
```

Kode Sumber 8.10. Kelas `SpecialDefenseGameManager`

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using MiniGameModel;
using HealthGameLevel;
namespace MiniGame
{
```

```
    public class HealthGameManager : MonoBehaviour
    {
        public GUIText scoreText;
        public GameObject result;
        public static int score;
        bool isComplete, isGameOver, flag;
        public static bool playerFall;
        TrainingPetAttributs petInfo;
        ResultCalculator calculator;
        void Start()
        {
            petInfo =
TrainingPetAttributs.CreateTrainingAtributSingleton
();
            calculator = new ResultCalculator();
            HealthlevelManufacturer newManufacturer
= new HealthlevelManufacturer();
           HealthLevelBuilder levelBuilder=null;
            switch(petInfo.GetTrainingLevel())
            {
                case 1: levelBuilder = new
Level1(); break;
                case 2: levelBuilder = new
Level2(); break;
                case 3: levelBuilder = new
Level3(); break;
            }
newManufacturer.Construct(levelBuilder);
            foreach(GameObject i in
levelBuilder.level.Obstacle)
            {
                Instantiate(i,
i.transform.position, i.transform.rotation);
            }
            Init();
        }
        void Update()
        {
            if (score < 10)
            {
        scoreText.text = score.ToString()+"/10";
            }
            else
            {
```

```
                scoreText.text = "10/10";
                isComplete = true;
            }
            if ((score==10||playerFall)&&!flag)
            {
                flag = true;
                isGameOver = true;
                GameOver();
            }
        }
        void Init()
        {
            score = 0;
            isGameOver = false;
            isComplete = false;
            playerFall = false;
            flag = false;

        }
        void GameOver()
        {
            audio.enabled = false;
            Instantiate(result,
result.transform.position,
result.transform.rotation);
            calculator.CalculateHealth(score);
        }
    }
}
```

Kode Sumber 8.11. Kelas `HealthGameManager`

BAB VI
KESIMPULAN DAN SARAN

Pada bab ini dibahas kesimpulan dari perancangan, implementasi, uji coba, dan evaluasi sistem. Selain itu, dibahas pula saran untuk mendapatkan hasil yang lebih baik.

6.1. Kesimpulan

Dari hasil selama proses perancangan, implementasi, serta pengujian dapat diambil kesimpulan sebagai berikut.

1. Aturan main dari modul *mini game* berhasil diterapkan terhadap masing-masing *mini game.* Hasil pengujian fungsionalitas menunjukkan bahwa fungsionalitas dari masing-masing *mini game* dapat dijalankan dengan baik.
2. Konsep *builder pattern* berhasil diterapkan pada modul *mini game.* Kelas-kelas yang digunakan pada *builder pattern* dapat digunakan kembali jika terjadi penambahan level pada *mini game*.
3. Tingkat kesulitan berhasil diimplementasikan pada masing-masing *mini game.* Pada masing-masing *mini game* terdapat tiga level yang dapat dimainkan. Implementasi dari level tersebut menggunakan *builder pattern.*

6.2. Saran

Berikut saran-saran untuk pengembangan dan perbaikan sistem di masa yang akan datang. Untuk penggunaan modul *mini game* yang lebih *portable,* diperlukan metode pemrograman yang dapat menangani perubahan pada modul *mini game*. Perubahan yang dimaksud dapat berupa penambahan atau pengurangan jumlah *mini game.* Dengan demikian saat terjadi perubahan tersebut tidak diperlukan perubahan pada sisi modul permainan utama.

BIODATA PENULIS

Penulis, Afrizal merupakan anak pertama dari tiga bersaudara. Penulis dilahirkan di kota Magetan pada tanggal 27 Maret 1992. Penulis telah menempuh pendidikan formal di TK Pancasila (1997-1998), SDN Widorokandang 1 (1998-2004), SMPN 1 Panekan (2004-2007), SMAN 2 Magetan (2007-2010). Pada tahun 2010 penulis diterima di S1 Jurusan Teknik Informatika Institut Teknologi Sepuluh Nopember Surabaya.

Di Jurusan Teknik Informatika, penulis mengambil bidang minat Rekayasa Perangkat Lunak (RPL). Selama kuliah penulis pernah menjadi Asisten Dosen dari mata kuliah Analisis dan Struktur Data. Penulis juga beberapa kali mengikuti final kompetisi pemrograman, diantaranya ITB Programming Contest, Computer Festival, dan Indonesia National Programming.

Penulis dapat dihubungi melalui alamat email afrizal.kudo@gmail.com.