

**UNIVERSITAS INDONESIA**

**PENERAPAN *SOFTWARE PRODUCT LINE* DALAM PENGEMBANGAN *ROLE-PLAYING GAME***

**SKRIPSI**

**FARIZ ABIRAFDI BENARTO**

**1206278416**

**FAKULTAS ILMU KOMPUTER**

**PROGRAM STUDI ILMU KOMPUTER**

**DEPOK**

**DESEMBER 2016**



**UNIVERSITAS INDONESIA**

**PENERAPAN *SOFTWARE PRODUCT LINE* DALAM PENGEMBANGAN *ROLE-PLAYING GAME***

**SKRIPSI**

**Diajukan sebagai salah satu syarat untuk memperoleh gelar**

**Sarjana Ilmu Komputer**

**FARIZ ABIRAFDI BENARTO**

**1206278416**

**FAKULTAS ILMU KOMPUTER**

**PROGRAM STUDI ILMU KOMPUTER**

**DEPOK**

**DESEMBER 2016**

# **HALAMAN PERSETUJUAN**

**Judul** : Penerapan *Software Product Line* dalam Pengembangan *Role-Playing Game*

**Nama** : Fariz Abirafdi Benarto

**NPM** : 1206278416

Laporan Skripsi ini telah diperiksa dan disetujui.

19 Desember 2016

Dr. Ade Azurat

Pembimbing Skripsi

# **HALAMAN PERNYATAAN ORISINALITAS**

**Skripsi ini adalah hasil karya saya sendiri,**

**dan semua sumber baik yang dikutip maupun dirujuk**

**telah saya nyatakan dengan benar.**

**Nama : Fariz Abirafdi Benarto**

**NPM : 1206278416**

**Tanda Tangan :**

**Tanggal : 19 Desember 2016**

# **HALAMAN PENGESAHAN**

Skripsi ini diajukan oleh :

Nama : Fariz Abirafdi Benarto

NPM : 1206278416

Program Studi : Ilmu Komputer

Judul Skripsi : Penerapan *Software Product Line* dalam

Pengembangan *Role-Playing Game*

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Komputer pada Program Studi Ilmu Komputer, Fakultas Ilmu Komputer, Universitas Indonesia

DEWAN PENGUJI

Pembimbing : Dr. Ade Azurat (.............................)

Penguji 1 : Ir. Suryana Setiawan (.............................)

Penguji 2 : Dr. Ir. Eko K. Budiardjo (.............................)

Ditetapkan di : Depok

Tanggal : 3 Januari 2017

# KATA PENGANTAR

Puji syukur penulis panjatkan kehadirat Allah SWT, Tuhan Yang Maha Esa, karena berkat Rahmat dan Karunia-Nya penulis dapat menyelesaikan skripsi yang berjudul “Penerapan *Software Product Line* dalam Pengembangan *Role-Playing Game*”. Skripsi ini ditulis dalam rangka memenuhi salah satu syarat bagi penulis untuk dapat menyelesaikan pendidikan dan memperoleh gelar Sarjana Ilmu Komputer dari Jurusan Ilmu Komputer Universitas Indonesia.

Penulis menyadari bahwa skripsi ini tidak akan selesai tanpa bantuan dan dukungan dari berbagai pihak. Oleh karena itu, penulis ingin mengucapkan terima kasih kepada:

1. Bapak Dr. Ade Azurat selaku dosen pembimbing yang telah memberikan banyak sekali arahan, panduan, kritik, serta saran bagi penulis di tengah kesibukannya dan padatnya jadwal agar skripsi ini dapat diselesaikan.
2. Kedua orang tua, Mami dan Ayah, serta adik penulis, Farrel, atas bantuannya dalam berbagai bentuk mulai dari dukungan moral hingga material, serta untuk doa dan dukungan yang senantiasa diberikan kepada penulis.
3. Akhmad Agosto Prasetyo, Ilham Kusuma, Rahmat Hidayat, dan Rifqi Adiwidjaja selaku sahabat-sahabat penulis yang telah menemani penulis selama masa perkuliahan di Fakultas Ilmu Komputer Universitas Indonesia serta segala bentuk bantuan dan dukungan yang telah diberikan kepada penulis.
4. Muhammad Fadhlika selaku sahabat penulis sejak Sekolah Menengah Pertama yang selalu menjadi teman yang baik untuk bertukar pikiran dan pandangan selama 10 tahun terakhir.
5. Givana Sandita yang senantiasa memberikan dukungan kepada penulis, membantu penulis belajar untuk berbagai mata kuliah, menemani penulis mengerjakan tugas, serta senantiasa menemani penulis dalam penulisan skripsi ini.
6. Teman-teman sekelompok PPL C08 (Novie Kamalia, Ahmad Ghofari) termasuk Ilham Kusuma dan Givana Sandita yang telah menjadi lebih dari sekedar teman kelompok mata kuliah dan telah memberikan banyak bantuan dan dukungan kepada penulis sampai sekarang.
7. Pinta Kumbarani Nasution, Fandika Okdiba, dan Bagus Bagaskara selaku saudara-saudara sepupu penulis yang telah memberikan dukungan serta masukan kepada penulis selama penulisan skripsi ini.
8. ASTRO 2012 yang telah menjadi bagian dari hidup penulis selama menjalani masa perkuliahan.
9. Teman-teman yang pernah menjadi penghuni kos Graha Bouraq (Mas Pur, Mikael, Mas Pudang, Ardi, Japri) serta Tante dan Om Anshor selaku penjaga kos Graha Bouraq yang telah menemani penulis selama masa kos penulis.
10. Seluruh pihak yang tidak tersebutkan namanya, yang telah memberikan bantuan dan dukungan kepada penulis baik secara langsung maupun tidak.

Akhir kata, penulis berharap Allah SWT berkenan membalas segala kebaikan semua pihak yang telah membantu. Penulis menyadari bahwa masih terdapat banyak sekali kekurangan dalam penulisan skripsi ini. Penulis berharap semoga skripsi ini dapat membuka pintu bagi penelitian-penelitian lain yang serupa untuk dilakukan oleh mahasiswa-mahasiswa lain, serta menjadi acuan yang baik bagi mereka yang meneruskan penelitian ini.

Depok, 19 Desember 2016

Fariz Abirafdi Benarto

# HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama : Fariz Abirafdi Benarto

NPM : 1206278416

Program Studi : Ilmu Komputer

Fakultas : Ilmu Komputer

Jenis Karya : Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Royalti Noneksklusif (*Non-exclusive Royalty-Free Right*)** atas karya ilmiah saya yang berjudul:

Penerapan *Software Product Line* dalam Pengembangan *Role-playing Game*

beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Non-eksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/formatkan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di : Depok

Pada tanggal : 19 Desember 2016

Yang menyatakan

(.......................................................)

# **ABSTRAK**

Nama : Fariz Abirafdi Benarto

Program Studi : Ilmu Komputer

Judul : Penerapan *Software Product Line* dalam Pengembangan *Role-*

*playing Game*

Skripsi ini membahas penerapan *Software Product Line* (SPL) dalam pengembangan *game* bergenre RPG (*role-playing game)*. *Game* yang digunakan sebagai obyek studi kasus dalam skripsi ini adalah Dungeon Crawl Stone Soup. Eksperimen dari penelitian ini dilakukan dengan mengubah struktur *source code game* dari yang semula tidak menerapkan paradigma berbasis obyek dengan baik menjadi struktur *code* baru yang menerapkan paradigma berbasis obyek lebih baik agar dapat menerapkan konsep-konsep SPL dengan optimal. Eksperimen dilakukan berdasarkan pada contoh kasus penambahan sebuah *entity* baru ke dalam *game*, dan mencoba untuk membuat proses penambahannya lebih efisien.

Kata kunci:

*Software product lines*, *game development, role-playing game, software engineering*

# **ABSTRACT**

Name : Fariz Abirafdi Benarto

Study Program : Computer Science

Title : Software Product Lines Implementation in Role-playing Game

Development

This research discusses the implementation of Software Product Line (SPL) in Role-playing Game development. The game used as case study object is Dungeon Crawl Stone Soup. The experiment of this research is conducted by reworking the structure of the game source code which initially is not utilizing the object-oriented paradigm very well into a new code structure which utilizes it better so it can apply SPL concepts optimally. The experiment is done based on a case example of a new entity addition into the game, and is aimed at making the process more efficient.

Keywords:

*Software product lines*, *game development, role-playing game, software engineering*

DAFTAR ISI

[HALAMAN PERSETUJUAN ii](#_Toc472001629)

[HALAMAN PERNYATAAN ORISINALITAS iii](#_Toc472001630)

[HALAMAN PENGESAHAN iv](#_Toc472001631)

[KATA PENGANTAR v](#_Toc472001632)

[HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS vii](#_Toc472001633)

[ABSTRAK viii](#_Toc472001634)

[ABSTRACT ix](#_Toc472001635)

[DAFTAR ISI x](#_Toc472001636)

[DAFTAR *SOURCE CODE* xiv](#_Toc472001637)

[PENDAHULUAN 1](#_Toc472001638)

[**1.1 Latar Belakang** 1](#_Toc472001639)

[**1.2 Rumusan Masalah** 2](#_Toc472001640)

[**1.3 Tujuan Penelitian** 2](#_Toc472001641)

[**1.4 Manfaat Penelitian** 2](#_Toc472001642)

[**1.5 Ruang Lingkup Penelitian** 3](#_Toc472001643)

[**1.6 Metodologi Penelitian** 3](#_Toc472001644)

[**1.7 Sistematika Penulisan** 4](#_Toc472001645)

[TINJAUAN LITERATUR 5](#_Toc472001646)

[**2.1 *Software Product Line*** 5](#_Toc472001647)

[**2.1.1 *Commonality* dan *variability*** 5](#_Toc472001648)

[**2.1.2 Paradigma pemrograman** 5](#_Toc472001649)

[**2.1.3 Kelebihan dan kekurangan** 6](#_Toc472001650)

[**2.2 *Domain-specific Language*** 7](#_Toc472001651)

[**2.3 *Refactoring* dalam paradigma pemrograman berbasis obyek** 8](#_Toc472001652)

[STUDI KASUS 9](#_Toc472001653)

[**3.1 Dungeon Crawl Stone Soup** 9](#_Toc472001654)

[**3.1.1 *Game*** 9](#_Toc472001655)

[**3.1.2 *Source code*** 15](#_Toc472001656)

[**3.2 Penambahan *entity* Dream Sheep** 18](#_Toc472001657)

[**3.2.1 Ringkasan** 18](#_Toc472001658)

[**3.2.2 *Code* yang diubah** 19](#_Toc472001659)

[**3.2.3 *Pull requests* terkait penambahan Dream Sheep** 24](#_Toc472001660)

[**3.2.4 Masalah yang ditimbulkan oleh penambahan Dream Sheep** 27](#_Toc472001661)

[**3.2.5 Isu-isu pada pengembangan Dream Sheep** 29](#_Toc472001662)

[**3.2.6 Definisi *entity*** 30](#_Toc472001663)

[RANCANGAN EKSPERIMEN 31](#_Toc472001664)

[**4.1 *Commonalities* dan *variabilities* dari *game*** 31](#_Toc472001665)

[**4.1.1 *Commonalities*** 31](#_Toc472001666)

[**4.1.2 *Variabilities*** 32](#_Toc472001667)

[**4.2 Class-Responsibility-Collaboration Cards** 33](#_Toc472001668)

[**4.2.1 *Responsibilities*** 33](#_Toc472001669)

[**4.2.2 *Collaboration*** 33](#_Toc472001670)

[**4.3 Perbedaan struktur *code*** 34](#_Toc472001671)

[**4.4 Rancangan struktur *code*** 36](#_Toc472001672)

[**4.4.1 Desain struktur *code* lama** 36](#_Toc472001673)

[**4.4.2 Desain struktur *code* baru** 38](#_Toc472001674)

[**4.5 Integrasi rancangan *code* ke dalam *game*** 39](#_Toc472001675)

[**4.5.1 Pengubahan paradigma *code* lama menjadi berbasis obyek** 39](#_Toc472001676)

[**4.5.2 Penerapan rancangan desain *code* terhadap Dream Sheep** 39](#_Toc472001677)

[**4.5.3 Indikator keberhasilan eksperimen** 40](#_Toc472001678)

[**4.6 Rancangan produk variasi dari *game*** 40](#_Toc472001679)

[EKSPERIMEN 41](#_Toc472001680)

[**5.1 Memodelkan struktur *code* lama** 41](#_Toc472001681)

[**5.1.1 Monster.h** 41](#_Toc472001682)

[**5.1.2 Monster.cc** 42](#_Toc472001683)

[**5.1.3 Spell.h** 43](#_Toc472001684)

[**5.1.4 Behavior.h** 44](#_Toc472001685)

[**5.1.5 Text.h** 45](#_Toc472001686)

[**5.1.6 Main.cc** 46](#_Toc472001687)

[**5.1.7 *Output* dari struktur *code* lama** 46](#_Toc472001688)

[**5.2 Memodelkan struktur *code* baru** 47](#_Toc472001689)

[**5.2.1 Monster.h** 47](#_Toc472001690)

[**5.2.2 Monster.cc** 48](#_Toc472001691)

[**5.2.3 Dragon.h** 49](#_Toc472001692)

[**5.2.4 Serpent.h** 49](#_Toc472001693)

[**5.2.5 Ghost.h** 50](#_Toc472001694)

[**5.2.6 Main.cc** 50](#_Toc472001695)

[**5.2.7** ***Output* dari struktur *code* baru** 51](#_Toc472001696)

[**5.3 *Refactoring* terhadap rancangan struktur *code* baru** 51](#_Toc472001697)

[**5.3.1 *Interface*** 52](#_Toc472001698)

[**5.3.2 *Inheritance*** 52](#_Toc472001699)

[**5.3.3 *Polymorphism*** 52](#_Toc472001700)

[**5.4 *Refactoring* terhadap *source code* asli** 62](#_Toc472001701)

[**5.4.1 Sistem perhitungan skor** 62](#_Toc472001702)

[**5.4.2 Sistem pengelolaan *experience*** 63](#_Toc472001703)

[**5.4.3 Dream Sheep** 64](#_Toc472001704)

[**5.4.4 Pengaruh perubahan terhadap *source code*** 66](#_Toc472001705)

[**5.5 Menghasilkan variasi produk *game*** 66](#_Toc472001706)

[ANALISIS 68](#_Toc472001707)

[**6.1 Tingkat *Coupling*** 68](#_Toc472001708)

[**6.2 Tingkat *Cohesion*** 68](#_Toc472001709)

[**6.3 *Development Effort*** 69](#_Toc472001710)

[PENUTUP 71](#_Toc472001711)

[**7.1 Kesimpulan** 71](#_Toc472001712)

[**7.2 Tantangan dalam pengerjaan penelitian** 71](#_Toc472001713)

[**7.3 Saran** 71](#_Toc472001714)

[**7.4 *Future works*** 72](#_Toc472001715)

[DAFTAR PUSTAKA 73](#_Toc472001716)

**DAFTAR GAMBAR**

**Gambar 2.1** Grafik dan diagram *tools* serta penggunaan DSL dalam pengembangan *game*

**Gambar 3.1** Diagram fitur RPG secara umum

**Gambar 3.2** Diagram fitur Dungeon Crawl Stone Soup

**Gambar 4.1** Beberapa contoh pembagian berkas *code* pada *source code* asli

**Gambar 4.2** Hubungan antar berkas *code* pada struktur *code* lama

**Gambar 4.3** Ilustrasi penambahan monster baru pada struktur *code* lama

**Gambar 4.4** Hubungan antar berkas *code* pada struktur *code* baru

**Gambar 4.5** Ilustrasi penambahan monster baru pada struktur *code* baru

**Gambar 5.1** *Output* teks dari rangkaian *code* dengan struktur *code* lama

**Gambar 5.2** *Output* teks dari rangkaian *code* dengan struktur *code* baru

**Gambar 5.3** *Output* teks dari rangkaian *code* yang telah menerapkan *polymorphism*

**Gambar 5.4** Status *build* dari pengubahan yang dilakukan

# **DAFTAR *SOURCE CODE***

***Source Code* 4.1** Struktur *code* lama

***Source Code* 4.2** Struktur *code* baru

***Source Code* 5.1** Implementasi Monster.h pada struktur *code* lama

***Source Code* 5.2** Implementasi Monster.cc pada struktur *code* lama

***Source Code* 5.3** Implementasi Spell.h pada struktur *code* lama

***Source Code* 5.4** Implementasi Behavior.h pada struktur *code* lama

***Source Code* 5.5** Implemetasi Text.h pada struktur *code* lama

***Source Code* 5.6** Implementasi Main.cc pada struktur *code* lama

***Source Code* 5.7** Implementasi Monster.h dalam struktur *code* baru

***Source Code* 5.8** Implementasi Monster.cc dalam struktur *code* baru

***Source Code* 5.9** Implementasi Dragon.h dalam struktur *code* baru

***Source Code* 5.10** Implementasi Serpent.h dalam struktur *code* baru

***Source Code* 5.11** Implementasi Ghost.h dalam struktur *code* baru

***Source Code* 5.12** Implementasi Main.cc dalam struktur *code* baru

***Source Code 5.13***Potongan *code* dari *source code* *game* yang menggunakan *conditional*

***Source Code* 5.14** Implementasi Monster.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

***Source Code* 5.15** Implementasi Monster.cc yang mengakomodasi *polymorphism* dalam struktur *code* baru

***Source Code* 5.16** Implementasi Dragon.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

***Source Code* 5.17** Implementasi Serpent.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

***Source Code* 5.18** Implementasi Ghost.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

***Source Code* 5.19** *Class* NewSheep

**BAB 1**

# **PENDAHULUAN**

Bab ini menjelaskan latar belakang, rumusan masalah, tujuan penelitian, manfaat penelitian, ruang lingkup penelitian, dan sistematika penulisan.

## **1.1 Latar Belakang**

Industri *game* video telah berkembang sejak tahun 1970-an. Pada masa itu, biaya produksi *game*, kapasitas komputer, dan tinkat kompleksitas *game* pada umumnya masih rendah, sehingga sebuah *game* dapat dibuat oleh seorang *programmer* saja.

Memasuki era modern, industri *game* telah berkembang pesat, dari produksi berbiaya rendah dengan ukuran tim pengembang yang kecil, hingga produksi berbiaya jutaan *dollar* dengan tim yang beranggotakan ratusan orang. Proyek *game* berukuran besar melibatkan banyak orang dengan berbagai peran dan spesifikasi pekerjaan, termasuk para *programmer*. Sebuah *game* AAA, atau *game* yang diproduksi dengan biaya dan promosi yang sangat tinggi, biasanya melibatkan banyak sekali pengembang.

Beban kerja pengembang *game* sering kali ditentukan oleh genre dari *game* yang sedang dibuat. Salah satu *genre* yang paling berat untuk dikembangkan adalah genre *role-playing game*, melihat karakteristik dari genre tersebut yang biasanya memiliki banyak sekali aspek dan fitur yang berbeda.

Banyaknya *code* yang dikerjakan dan *programmer* yang berkontribusi sering kali menyebabkan *project hierarchy* tidak beraturan. Berbagai masalah sering dihadapi ketika berursan dengan *source code* yang besar, contohnya adalah ketika membuat sebuah perubahan pada *code* yang sudah ada. Mengubah *source code* yang hanya terdiri dari ratusan baris dan beberapa *file* mungkin bukan perkara sulit, tetapi ketika *code* yang dihadapi berjumlah puluhan hingga ratusan *file*, dengan tiap *file* memiliki hingga ribuan baris *code*, tentu akan menjadi sebuah pekerjaan yang rumit dan melelahkan. Banyak risiko yang dapat terjadi ketika mengubah *code*, salah satunya adalah kemungkinan bahwa terdapat *method* atau *class* yang gagal berfungsi karena perubahan yang dilakukan. Masalah akan menjadi semakin sulit untuk diatasi ketika *method* atau *class* tersebut berada dalam *file* yang berbeda dengan *file* yang diubah.

Dari kesulitan-kesulitan tersebut, dibutuhkan sebuah paradigma pengembangan yang mampu mengurangi beban kerja *developer* dengan membuat struktur *code* yang lebih terorganisasi, sehingga perubahan pada *code* dapat dilakukan dengan mudah dan lancar. *Software product line*, atau SPL, muncul sebagai sebuah paradigma *development* yang didesain untuk mengoptimalkan waktu pemasaran, biaya, produktivitas, kualitas, dan aspek *development* lainnya. Penerapan SPL dalam pembuatan *game* dapat membantu memudahkan pengerjaan dengan menekankan struktur *code* yang baik, sehingga dapat mengurangi waktu produksi dan biaya yang disebabkan oleh proses pengubahan *code*.

## **1.2 Rumusan Masalah**

Berdasarkan isu-isu mengenai pengembangan *game* yang telah dibahas sebelumnya seperti kesulitan-kesulitan dalam pengembangan serta efisiensi pengerjaan *game,* terdapat 2 rumusan masalah yang dapat dijawab oleh penelitian ini, yaitu:

1. Bagaimana SPL dapat diimplementasikan dalam pengembangan *game*?
2. Apakah bentuk implementasi tersebut lebih baik dibandingkan dengan paradigma pengembangan sebelum menggunakan SPL?

## **1.3 Tujuan Penelitian**

Berdasarkan pertanyaan-pertanyaan rumusan masalah, terdapat beberapa tujuan yang diharapkan dapat diraih oleh penelitian ini, yaitu:

1. Untuk mengetahui salah satu bentuk/contoh metode implementasi SPL dalam pengembangan game.
2. Untuk menganalisa kelebihan dan kekurangan metode yang diteliti.

## **1.4 Manfaat Penelitian**

Penelitian ini diharapkan dapat menjadi acuan bagi penelitian-penelitian yang akan datang berikutnya baik dalam bidang SPL maupun pengembangan game, serta membantu menerapkan sebuah standar dalam proses pengembangan *game*.

## **1.5 Ruang Lingkup Penelitian**

Ruang lingkup penelitian ini merupakan *game* sebagai sebuah domain. Ruang lingkup eksperimen yang dilakukan untuk menunjang penelitian ini merupakan penambahan sebuah *entity* baru ke dalam *game*, membuat monster baru dalam *game*.

## **1.6 Metodologi Penelitian**

Terdapat beberapa tahapan penelitian yang dilakukan dalam skripsi ini, yaitu:

1. Tinjauan Literatur

Dalam tahapan ini, penelitian-penelitian terkait dengan penelitian yang dilakukan dalam skripsi ini dipelajari dan dianalisis untuk dijadikan sebagai acuan dalam eksperimen.

1. Analisis Studi Kasus

Untuk mendalami lebih lanjut mengenai *game* studi kasus serta fitur/task yang akan diuji, dilakukan analisis terhadap berbagai aspek dari *game* studi kasus.

1. Merumuskan ide dan langkah eksperimen

Setelah melakukan tinjauan literatur dan analisis studi kasus, dirumuskan ide dan langkah-langkah yang akan dilakukan untuk eksperimen yang akan dilakukan.

1. Melakukan eksperimen

Dalam tahap ini, dilakukan eksperimen untuk menganalisa metodologi yang diteliti

1. Analisis hasil eksperimen

Dari eksperimen yang dilakukan, hasil yang didapat dicatat dan dianalisis mengenai kesesuaiannya dengan rumusan masalah, tinjauan literatur, serta analisis studi kasus.

1. Menarik kesimpulan

Dalam tahapan terakhir, yang dilakukan adalah menilai hasil dari eksperimen dan membuat kesimpulan sesuai dengan rumusan masalah.

## **1.7 Sistematika Penulisan**

Skripsi ini disusun menjadi lima bab dengan sistematika penulisan sebagai berikut:

* Bab 1 Pendahuluan

Bab ini menjelaskan latar belakang, rumusan masalah, tujuan penelitian, manfaat penelitian, ruang lingkup penelitian, metodologi penelitian dan sistematika penulisan.

* Bab 2 Tinjauan Literatur

Bab ini berisi tinjauan literatur yang digunakan sebagai acuan dari penelitian.

* Bab 3 Studi Kasus

Bab ini mendeskripsikan *game* serta fitur/*task* yang diacu sebagai studi kasus untuk di-*refactor* dalam penelitian.

* Bab 4 Rancangan Eksperimen

Bab ini menjelaskan rincian eksperimen yang dilakukan, beserta penjelasan dasar-dasar dari langkah yang dilakukan.

* Bab 5 Eksperimen

Bab ini memaparkan proses eksperimen yang dilakukan serta mencatat hasil yang didapat.

* Bab 6 Analisis

Bab ini berisi analisis terkait hasil eksperimen yang didapat dari bab sebelumnya.

* Bab 7 Penutup

Bab ini berisi kesimpulan yang dapat ditarik dari hasil eksperimen dan analisis dalam penelitian, serta saran untuk penelitian selanjutnya.

**BAB 2**

# **TINJAUAN LITERATUR**

## **2.1 *Software Product Line***

Menurut Clements dan Northrop (2001), *Software Product Line* adalah sebuah kumpulan sistem perangkat lunak dengan kesamaan fitur yang memenuhi suatu kebutuhan, dan dikembangkan dari suatu kumpulan *core asset* yang telah ditentukan. Secara umum, *Software Product Line* (SPL) merupakan bentuk penerapan sistem *product line* dalam pengembangan perangkat lunak. *Product line* diterapkan sebagai metode produksi yang memanfaatkan kesamaan dari dua buah produk yang berbeda untuk meminimalisasi biaya produksi kedua produk.

### **2.1.1 *Commonality* dan *variability***

*Software Product Line* memanfaatkan kesamaan fitur antara dua atau lebih perangkat lunak untuk menghemat *effort* yang dikeluarkan untuk mengembangkan sebuah perangkat lunak. Sebelum penerapan SPL, perangkat lunak yang akan dibuat *product line*-nya dianalisa terlebih dahulu mengenai *commonality* dan *variability*-nya. Commonality merupakan fitur yang sama antara dua atau lebih variasi perangkat lunak, sedangkan variability merupakan fitur yang membedakan setiap variasi perangkat lunak sehingga dibuat *product line* untuk perangkat lunak tersebut (Pohl et al, 2005).

### **2.1.2 Paradigma pemrograman**

Penerapan *Software Product Line* melibatkan berbagai pertimbangan dalam aspek *commonality* dan *variability*. Salah satunya adalah menentukan paradigma pemrograman yang sesuai untuk menerapkan konsep-konsep yang diusung oleh *Software Product Line*. Menurut McGregor (2004), paradigma berbasis obyek mendukung manajemen *variability* melalui berbagai teknik seperti analisis fitur dan domain, *inheritance*, dan *polymorphism*. Dengan menerapkan teknik-teknik paradigma pemrograman berbasis obyek, maka implementasi *commonality* dan *variability* dalam penerapan SPL akan menjadi lebih mudah.

Dalam *paper* yang ia tulis, McGregor mengatakan bahwa banyak dari *product line* yang ia pantau atau kerjakan menerapkan paradigma berbasis obyek dan menghasilkan banyak manfaat.

### **2.1.3 Kelebihan dan kekurangan**

Furtado et al (2011) mengatakan bahwa strategi-strategi *software reuse* dan *family-based production*, termasuk SPL, membuat rekayasa perangkat lunak semakin efektif dan produktif. Strategi-strategi tersebut mendorong para pengembang *software* untuk lebih menganalisa dan menerapkan sistem secara kolektif dibanding secara terpisah. Dengan demikian, *software life cycle* menjadi lebih terautomasi dengan memanfaatkan *domain assets* yang *reusable* seperti *application blocks*, *framework*, *patterns*, *domain-specific language*, *generator*, dan *tools*.

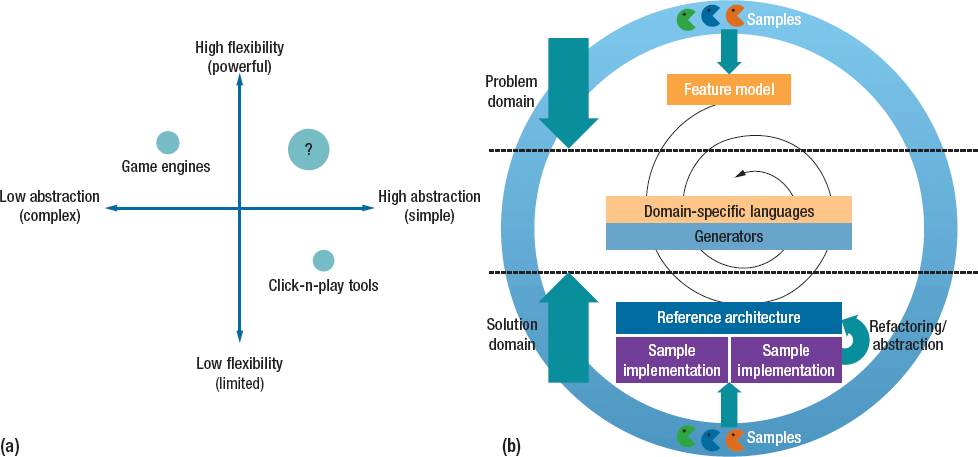
Meski telah berhasil diterapkan dalam pengembangan *software* secara umum, terdapat beberapa faktor yang menyebabkan SPL belum dapat diterapkan dengan baik dalam pengembangan *game*:

* Perekayasaan adalah proses tersulit dalam pengembangan *game*. Proses rekayasa *game* meliputi implementasi ide-ide desain berjalannya atau dimainkannya sebuah *game*, dari aspek mekanika permainan hingga estetika seperti grafis dan suara.
* Bidang pengembangan *game* dikarakterisasi oleh pengembangan *low-level* yang bersifat *ad hoc*.
* Pengembang *game* mengalami kesulitan dalam mengintegrasikan komponen-komponen dan mengelola kompleksitas arsitektural.
* Pengembangan *game* tidak sama dengan pengembangan *software* secara umum, sehingga rekayasa berdasarkan *requirements* tradisional tidak dapat diterapkan, serta konsep populer “*game genre*” yang terkesan ambigu dan tidak akurat dapat menyesatkan.

## **2.2 *Domain-specific Language***

Menurut Furtado (2011), sebuah proses sistematik untuk mengeksploitasi *software product lines* untuk pengembangan *game* menawarkan *domain-specific languages* dan *generators* yang dibuat khusus untuk subdomain-subdomain *game*.

Berikut adalah gambaran automasi pengembangan *game* yang dipaparkan oleh Furtado et al. Dalam diagram automasi tersebut, digambarkan bahwa bidang pengembangan *game* menghadapi ketidaktersediaan sebuah *tool* pengembangan *game* yang memiliki abstraksi dan fleksibilitas tinggi untuk memudahkan pengembang. Selain itu, terdapat *flow* yang menggambarkan pemanfaatan *domain-specific language* untuk menerapkan *software product line* dalam pengembangan *game*, seperti yang dapat dilihat pada Gambar 2.1 berikut.



**Gambar 2.1** Grafik dan diagram *tools* serta penggunaan DSL dalam pengembangan *game*

1. Gambaran diperlukannya sebuah *tool* dengan abstraksi dan fleksibilitas tinggi
2. Penggunaan DSL sebagai sebuah metode pendekatan untuk pengembangan *game* berbasis SPL

Dapat ditemukan beberapa bentuk penerapan *domain-specific language* di dalam *engine-engine* pengembangan *game* seperti UnrealScript (Dobbe, 2007)

## **2.3 *Refactoring* dalam paradigma pemrograman berbasis obyek**

Seperti yang dibahas pada pembahasan paradigma yang digunakan untuk penerapan SPL, dalam paradigm pemrograman berbasis obyek diperlukan implementasi dari konsep-konsep *inheritance* dan *polymorphism* untuk dapat menerapkan SPL dengan optimal. Menurut Fowler (2001), *polymorphism* dapat membantu menghindari penggunaan *conditional* ketika mengimplementasi obyek-obyek yang memiliki *behavior* atau sifat yang berbeda. Untuk dapat menunjang penerapan *polymorphism*, dapat dibuat sebuah struktur *code* yang juga menerapkan *inheritance*,dan *interface* untuk memudahkan penerapan *inheritance* dalam bahasa pemrograman berbasis obyek.

**BAB 3**

# **STUDI KASUS**

Bab ini mendeskripsikan *game* yang digunakan sebagai studi kasus untuk di-*refactor* dalam penelitian.

## **3.1 Dungeon Crawl Stone Soup**

Dalam subbab ini akan dijelaskan mengenai detail dari *game* studi kasus yang akan digunakan dalam eksperimen. Penjelasan akan dilakukan untuk sisi permainan dan struktur *project*.

### **3.1.1 *Game***

Dungeon Crawl Stone Soup dipilih sebagai obyek penelitian karena beberapa faktor, yaitu:

* *Source code* dari *game* ditulis dalam bahasa pemrograman yang mendukung penerapan paradigma pemrograman berbasis obyek, yaitu C++
* Pengembangan *game* belum didasarkan pada penerapan *software product line*, sehingga hasil penelitian merupakan suatu hal yang baru
* *Game* dapat diperoleh secara gratis dari situs resminya
* *Source code* dari *game* berbasis *open-source* dan tersedia untuk umum di platform GitHub
* Memiliki genre RPG yang memiliki banyak aspek implementasi yang dapat dioptimalkan dengan penerapan SPL
* Memiliki *player base* yang cukup besar di komunitas *game* serupa
* Pengembangan *game* masih berjalan

**3.1.1.1 Ringkasan umum Dungeon Crawl Stone Soup**

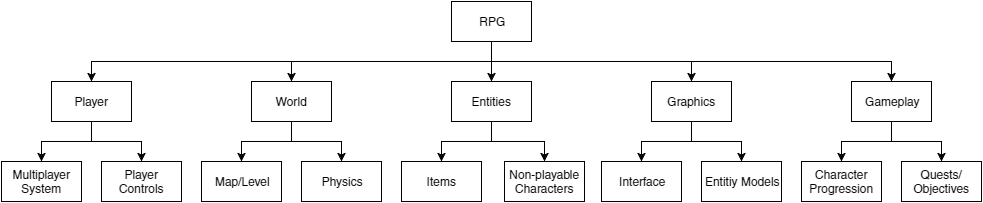
*Game* yang digunakan sebagai obyek studi kasus adalah Dungeon Crawl Stone Soup. Dungeon Crawl Stone Soup merupakan sebuah *game* yang pertama kali dikembangkan pada tahun 2006 sebagai upaya untuk menghidupkan kembali proyek *Linley’s Dungeon Crawl* yang dikembangkan oleh Linley Henzell pada tahun 1997.

Dungeon Crawl Stone Soup dapat dimenangkan dengan *win condition* tertentu, yaitu mendapatkan 3 dari 15 buah benda bernama *runes of zot*, mengambil sebuah benda bernama *orb of zot*, dan keluar dari *dungeon* dengan selamat.

**3.1.1.2 Genre dan mekanika *gameplay***

Pada umumnya, sebuah *game* dapat dikategorikan ke dalam sebuah genre. Genre dari sebuah *game* adalah jenis, tipe, atau kelompok dari *game* tersebut berdasarkan beberapa hal seperti cerita, mekanisme *gameplay*, model grafis, jumlah pemain, dan lain-lain. Dungeon Crawl Stone Soup dapat dikategorikan dalam genre RPG, atau *role-playing game*. RPG sendiri dapat didefinisikan sebagai sebuah *game* di mana pemain mengimitasi atau memerankan sebuah tokoh atau peran fiktif dalam sebuah cerita atau dunia, biasanya berlatar fantasi. Aslinya, RPG tidak terikat dalam *game* video. RPG berawal dari permainan papan (*board game*) atau permainan berbasis teks (*text-based*). Dalam studi kasus ini, RPG yang dimaksud adalah RPG dalam arti *game* video, di mana elemen-elemen *role-playing* yang terdapat dalam *game* tersebut berupa latar dan sistem yang terkomputerisasi.

*Game* yang termasuk dalam kategori genre RPG biasanya memiliki beberapa ciri khusus seperti pemain memainkan sebuah karakter, terdapat dunia—dinamis atau statis—yang dapat dijelajahi pemain, dan memiliki *character progression*, seperti yang dapat dilihat pada Gambar 3.1. Dalam Dungeon Crawl Stone Soup, pemain memainkan sebuah tokoh yang memiliki peran tertentu, di mana tokoh tersebut dapat menjelajahi sebuah dunia yang dinamis. Setiap aksi signifikan yang dilakukan oleh pemain menghasilkan elemen-elemen tertentu yang secara langsung mengubah karakteristik tokoh yang dimainkan. Hal tersebut merupakan sebuah sistem *character progression*.

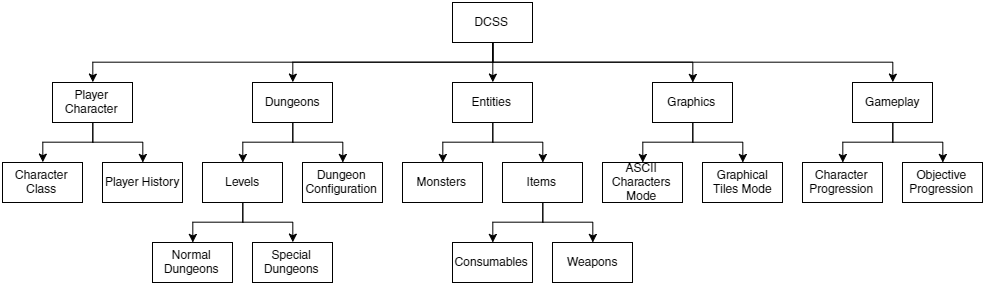


**Gambar 3.1** Diagram fitur RPG secara umum

Dalam genre RPG sendiri terdapat beberapa sub-genre yang mengkategorikan sistem permainan dan elemen *role-playing* lebih lanjut. Pembagian sub-genre untuk RPG pun terbagi menjadi beberapa dasar pengelompokan, seperti berdasarkan unsur atau elemen *game* yang paling menonjol, model skenario permainan, jumlah pemain, dan lain-lain. Dungeon Crawl Stone Soup dapat dikategorikan ke dalam sub-genre *roguelike*.

*Roguelike* merupakan sub-genre dari RPG yang mengandung unsur *dungeon crawler* dan biasanya terdiri dari sejumlah *level* yang dihasilkan secara prosedural. Beberapa karakteristik *roguelike* adalah *turn-based gameplay*, grafis *tile-based*, dan *permanent death* dari karakter yang dimainkan. Salah satu ciri khas *roguelike* adalah grafis yang ditampilkan dengan karakter ASCII dan hanya dapat dimainkan menggunakan *keyboard*, walaupun beberapa *game* bersub-genre *roguelike* modern telah menyediakan fitur *graphical tile* yang menggunakan gambar untuk merepresentasikan elemen-elemen dalam *game*. *Game* dengan sub-genre *roguelike* biasanya lebih mengedepankan kompleksitas mekanik *gameplay* dibandingkan dengan *interface* yang ramah dengan pemain. *Turn-based gameplay* pada *roguelike* biasanya merupakan sistem *real-time* yang dimodifikasi sehingga waktu dalam dunia *game* tersebut hanya akan berjalan jika pemain melakukan suatu aktivitas. Dalam Dungeon Crawl Stone Soup, setiap aktifitas dilakukan dengan satu ketikan tombol di *keyboard* atau *mouse*. Lama waktu yang berjalan dalam satu ketikan tersebut bergantung pada lama waktu aktivitas yang dilakukan. Selain *turn-based gameplay*, Dungeon Crawl Stone Soup juga memiliki sistem *permanent death*, di mana karakter yang terbunuh dalam *game* akan hilang selamanya, tanpa bisa di-*load* kembali seperti dalam *game* bergenre lain. Walaupun karakter yang telah terbunuh tidak dapat di-*load*, pemain dapat melakukan *save* dan *load* untuk karakter yang masih hidup, untuk menyimpan dan melanjutkan *progress* dalam *game*.

Unsur *dungeon crawl* yang terdapat dalam *roguelike* merupakan sebuah sistem atau skenario di mana tujuan atau aktivitas utama pemain adalah menjelajahi sebuah atau sejumlah *dungeon*. Dalam *dungeon* yang dijelajahi, pemain dapat bertarung dengan karakter-karakter musuh dan mengumpulkan benda-benda yang memiliki signifikansi dalam sistem *character progression*. *Game* dengan model skenario *dungeon crawl* biasanya memiliki sistem *loot*, di mana pemain dapat mengumpulkan benda atau barang-barang yang tersedia di dalam dunia *game* tersebut dengan berbagai cara, mulai dari mengambil barang yang tersedia di tempat-tempat tertentu seperti misalnya emas di dalam peti harta, atau benda yang didapat dari karakter musuh yang telah dikalahkan. Visualisasi fitur-fitur serta aspek-aspek *gameplay* dari Dungeon Crawl Stone Soup dapat dilihat pada Gambar 3.2.



**Gambar 3.2** Diagram fitur Dungeon Crawl Stone Soup

Seperti kebanyakan *game* dengan genre *roguelike*, Dungeon Crawl Stone Soup memiliki latar fantasi, di mana dunia yang dijelajahi merupakan sebuah dunia fiktif dengan karakter-karakter yang berasal dari berbagai mitologi. Dalam Dungeon Crawl Stone Soup, unsur cerita tidak terlalu ditonjolkan, hanya disebutkan mengenai tujuan permainan yang merupakan *win condition* dari *game* ini. Tingkah laku pemain tidak memiliki dampak terhadap cerita latar dan sebaliknya, cerita latar tidak memiliki dampak apapun terhadap permainan, kecuali sebagai penjelas *win condition*. Pemain tidak dapat mengubah cerita dan *win condition* dengan cara apapun.

Setiap sesi permainan Dungeon Crawl Stone Soup memiliki konfigurasi *dungeon* yang berbeda. Fitur ini dibuat agar *game* memiliki *replay value* yang tinggi, sehingga pemain tidak merasa bosan. Fitur *random/procedurally-generated levels* di *game-gameroguelike* juga dimaksudkan agar pemain tidak terdorong untuk menghafal pola *game* dan lebih menitikberatkan permainan pada strategi dan mempelajari kesalahan dari sesi-sesi sebelumnya.

Selain *dungeon level* yang *procedurally-generated*, fitur lain yang menaikkan *replay value* dan mengurangi unsur penghafalan pola adalah *scroll* dan *potion* yang memiliki efek yang di-*random* setiap sesinya. *Scroll* dan *potion* adalah salah satu elemen penting dalam *game* bertema fantasi dengan sistem *dungeon crawl*. *Scroll* merupakan gulungan perkamen yang memiliki efek sihir tertentu, sedangkan *potion* merupakan minuman obat atau pada kasus tertentu merupakan minuman racun. Dengan sistem seperti ini, pemain akan menjadi lebih berhati-hati ketika menggunakan *scroll* atau *potion*, karena harus memperhitungkan risiko yang terjadi jika benda yang digunakan memiliki efek samping yang buruk.

**3.1.1.3 Grafis**

Dungeon Crawl Stone Soup memiliki dua mode grafis, yaitu *console* dan *graphical tile*. Dalam mode *console*, *game* dijalankan di sebuah *console* dansemua elemen *game* direpresentasikan dalam karakter ASCII. Dalam mode *graphical tile*, *game* dijalankan dalam tampilan *user interface* interaktif, di mana semua elemen *game* memiliki representasi grafis berupa gambar 2D yang menyerupai deskripsi obyek yang direpresentasikan. Karena sistem yang digunakan adalah *tile-based*, maka elemen-elemen atau obyek yang berukuran lebih dari satu *tile* digambarkan melalui gabungan dari beberapa *tile*, bukan merupakan sebuah gambar tunggal.

**3.1.1.4 *User interface***

Mode *graphical tile* menyediakan tampilan *interface* untuk *inventory* pemain yang dapat diakses dan dilihat dengan mudah, langsung saat bermain. *Interface* mode *graphical tiles* secara umum memiliki lebih banyak tampilan, di mana tampilan yang terdapat di layar *game* tidak hanya karakter pemain dan dunia di sekitarnya, tetapi juga peta dari *level* dungeon yang sedang dijelajahi, visualisasi daftar *inventory* atau barang-barang yang dimiliki oleh pemain, daftar *skill* atau jurus yang dapat dilakukan oleh pemain, dan daftar *command* untuk membuat karakter pemain melakukan kegiatan tertentu secara otomatis.

**3.1.1.5 *Controls***

Masing-masing mode grafis yang disediakan Dungeon Crawl Stone Soup memiliki mekanisme *controls* yang berbeda. Mode *console* hanya dapat dikendalikan menggunakan *keyboard*, sedangkan mode *graphical tile* dapat dikendalikan dengan menggunakan *keyboard* dan *mouse*.

**3.1.1.6 Audio**

Dungeon Crawl Stone Soup menyediakan fitur suara yang dapat diaktifkan dengan mengubah *file* konfigurasi secara manual. Secara *default*, fitur suara tidak diaktifkan. Suara yang disediakan adalah efek-efek bunyi serta suara dari karakter-karakter dalam *game*.

**3.1.1.7 Platform**

Untuk saat ini, Dungeon Crawl Stone Soup tersedia untuk *operating system* Windows, Mac OS X, dan Linux. Pengembangan *game* untuk platform Android saat ini masih berlangsung, namun *unstable build* terbaru dapat diunduh dari situs resmi Dungeon Crawl Stone Soup. Dungeon Crawl Stone Soup dapat dimainkan secara *local*, melalui *executable file* di platform Windows dan melalui *terminal* di platform Linux. Selain *local*, tersedia juga versi *web browser* yang menyediakan Dungeon Crawl Stone Soup. Versi yang disediakan di *web browser* adalah *graphical tile version*.

Untuk eksperimen, pengerjaan dilakukan pada platform Linux karena kemudahan dalam proses *installation* dan *compile*.

**3.1.1.8 *Availability***

Dungeon Crawl Stone Soup dapat diunduh secara gratis melalui situs resmi. Situs Dungeon Crawl Stone Soup juga menyediakan tautan untuk mengakses dan mengunduh repositori *source-code* yang tersedia secara ­*open-source*. Dungeon Crawl Stone Soup dapat diunduh melalui tautan <https://crawl.develz.org/download.htm>.

**3.1.1.9 Lisensi**

Dungeon Crawl Stone Soup dirilis di bawah lisensi GNU General Public License v2.0 atau setelahnya, yang berarti semua orang berhak mengakses, mengunduh, menyebarkan, dan mengubah *game* ini dengan bebas.

### **3.1.2 *Source code***

Subbab ini menjelaskan *source-code* dari *game* studi kasus.

**3.1.2.1 *Overview***

*Source-code* Dungeon Crawl Stone Soup di-*host* di situs repositori GitHub, di mana *code* dapat diakses dan diubah dengan bebas oleh kontributor-kontributor yang telah terdaftar.

**3.1.2.2 *Repository***

Untuk memudahkan pengembangan *game* yang bersifat ­*open-source*, Dungeon Crawl Stone Soup memiliki repository *source code* yang dapat diakses melalui tautan <https://github.com/crawl/crawl>.

**3.1.2.3 Bahasa pemrograman**

Bahasa pemrograman yang digunakan dalam pengembangan Dungeon Crawl Stone Soup adalah C/C++. Meskipun format *code* yang digunakan adalah C++, tetapi paradigma *code* yang digunakan belum menerapkan konsep *object-oriented*.

**3.1.2.4 *Packages***

Dungeon Crawl Stone Soup memanfaatkan beberapa *package* yang tersedia secara *open-source*, yaitu:

* Lua scripting language, yang digunakan untuk fungsionalitas *in-game* dan *user macros*.
* PCRE library, yang digunakan untuk *regular expressions*.
* SQLite library, yang digunakan sebagai *engine* dari *database*.
* SDL and SDL\_image libraries, yang digunakan untuk tampilan *tile*.
* libpng library, yang digunakan untuk memuat gambar *tile*.

**3.1.2.5 *Project hierarchy***

Seluruh *source-code* utama yang membentuk *game* Dungeon Crawl Stone Soup terdapat dalam satu buah folder dengan direktori crawl/crawl-ref/source/. Setiap *file* yang mengandung *source-code* ditempatkan dalam satu *folder* tersebut, tanpa dipisahkan berdasarkan pengelompokan tertentu.

**3.1.2.6 Instalasi dan *building***

Karena bahasa yang digunakan dalam *source code* adalah C/C++, maka untuk kemudahan eksperimen, *game* akan di-*install* pada platform Linux. Untuk platform Linux, instalasi dilakukan dengan memasukkan beberapa *command* di Terminal sebagai berikut:

# Install the source repository

echo 'deb https://crawl.develz.org/debian crawl 0.18' | sudo tee -a /etc/apt/sources.list

# Install the DCSS signing key

wget https://crawl.develz.org/debian/pubkey -O - | sudo apt-key add -

# update your package list

sudo apt-get update

# install console version

sudo apt-get install crawl

*Command* di atas digunakan untuk melakukan instalasi Dungeon Crawl Stone Soup versi *console*. Instalasi untuk versi *graphical tile* dapat menggunakan *command* berikut:

# install tiles version

sudo apt-get install crawl-tiles

Setelah instalasi selesai dilakukan, Dungeon Crawl Stone Soup dapat di-*compile* dengan *command* berikut:

# go to file directory

cd crawl/crawl-ref/source/

# compile console version

make

*Command* di atas digunakan untuk *compilegame* versi *console*. Untuk *compile* versi *graphical tile* dapat menggunakan *command* berikut:

# compile tiles version

make TILES=y

Setelah selesai *compile*, Dungeon Crawl Stone Soup dapat dijalankan dengan *command*:

# launch the game

./crawl

**3.1.2.7 *Dependencies***

Untuk sistem Debian, *dependencies* dapat diunduh dengan *command* berikut:

apt-get install build-essential libncursesw5-dev bison flex liblua5.1-0-dev libsqlite3-dev libz-dev pkg-config libsdl2-image-dev libsdl2-mixer-devlibsdl2-dev libfreetype6-dev libpng-dev ttf-dejavu-core

## **3.2 Penambahan *entity* Dream Sheep**

Penambahan *entity* digunakan untuk mensimulasikan *refactoring* dari *game*. Untuk penelitian ini, diangkat sebuah contoh kasus penambahan *entity* berupa monster bernama Dream Sheep.

### **3.2.1 Ringkasan**

Dream Sheep merupakan sebuah *entity* baru yang ditambahkan pada DCSS oleh seorang kontributor bernama isloat pada Agustus 2016. Menurut deskripsi yang ditulis oleh isloat di halaman *commit*,ide mengenai *entity* ini dicetuskan oleh kontributor bernama PleasingFungus beberapa tahun lalu.

Dream Sheep merupakan sebuah karakter musuh yang muncul secara bergerombol pada level-level tingkat lanjut. Dream Sheep juga dapat muncul sebagai bagian dari gerombolan *entity* lain bernama Torpor Snail.

*Entity* ini menyerang pemain dengan cara memberikan *status effect* negatif pada pemain seperti *sleep*. Menurut isloat, *code* yang mengimplementasi kemampuan memberikan *sleep* tersebut terinspirasi dari *code* \_starcursed\_scream dan \_will\_starcursed\_scream milik kontributor lain bernama DracoOmega.

Kemampuan atau *ability* memberikan *sleep* yang dimiliki oleh Dream Sheep bernama Sleep Dust. Sleep Dusthanya dapat digunakan oleh lebih dari satu ekor Dream Sheep. Jika terdapat dua sampai empat ekor Dream Sheep di lingkup pandangan pemain, maka mereka akan mencoba untuk menidurkan pemain untuk waktu yang singkat. Sedangkan jika dalam lingkup pandangan pemain terdapat lima ekor atau lebih Dream Sheep, maka mereka akan mencoba untuk menidurkan pemain untuk waktu yang lama.

Sleep Dust memiliki *cooldown* yang relatif lama, yaitu 8 hingga 12 *turn*.*Ability* ini hanya dapat dilakukan jika semua Dream Sheep yang ada telah siap untuk melakukan *ability* tersebut. Tidak ada *damage* terhadap pemain yang dihasilkan dari *ability* ini.

Dream Sheep merupakan sebuah *entity* yang bersifat *recursive summoner*, yaitu dapat memunculkan atau memanggil *entity* lain dengan jenis yang sama dengan dirinya. Seekor Dream Sheep dapat memunculkan Dream Sheep lain ketika *health*-nya rendah. Tidak seperti *recursive summoner* lain, Dream Sheep hanya dapat memunculkan satu ekor Dream Sheep.

Secara kekuatan, Dream Sheep merupakan sebuah versi yang lebih kuat dari *entity* Sheep biasa. Dibandingkan dengan Sheep biasa, Dream Sheep sedikit lebih tangguh, lebih sering menghindar dari serangan pemain, dan memiliki *magic resistance* yang lebih tinggi.

### **3.2.2 *Code* yang diubah**

Untuk menambahkan Dream Sheep ke dalam *game*, terdapat beberapa berkas *source code* yang diubah:

* enum.h

Pada berkas ini, dilakukan penambahan *if-condition* untuk menangani kompatibilitas *entity* baru dengan *save profile* yang menggunakan versi lama yang tidak terdapat *entity* baru tersebut.

*Code* yang baru mengecek *tag* versi *game*. Jika *game* memiliki *tag* versi 34, maka monster Sheep dimunculkan. Jika *game* memiliki *tag* versi yang lebih tinggi dari versi 34, maka monster Dream Sheep yang dimunculkan. Penambahan juga dilakukan untuk menempatkan Dream Sheep ke dalam kelompok monster baru.

* godwrath.cc

Berkas ini berisi *function-function* terkait *god* atau dewa dalam *game*. Dungeon Crawl Stone Soup memiliki sistem penyembahan dewa di mana pemain dapat menjadi penyembah seorang dewa. *Function-function* tersebut sebagian besar menangani efek dari pemujaan dewa tertentu, serta konfigurasi monster-monster pengikut-pengikut setiap dewa. Pada berkas ini, tidak terdapat pengubahan yang mengubah baris *code*, pengubahan hanya terjadi pada sebuah baris *comment* terkait *entity* Sheep di mana kata ‘Sheep’ menjadi ‘Dream Sheep’.

* mgen-enum.h

Berkas ini berisi inisialisasi enum semua *band* monster yang terdapat dalam *game*.Pada berkas ini, dilakukan pengubahan pada pendefinisian *band* di mana *band* Sheep diubah menjadi *band* Dream Sheep. *Band* merupakan sejumlah *entity* yang sama yang beraktifitas secara bergerombol.

* mon-abil.cc

Pada berkas ini, dilakukan implementasi *ability* Dream Dust yang dimiliki oleh Dream Sheep. Ada beberapa *function* dan *method* yang ditambahkan.

Pertama adalah *method*\_sheep\_message() yang menghasilkan *output* berupa teks yang mendeskripsikan aksi-aksi yang dilakukan oleh Dream Sheep, atau efek-efek yang dialami oleh pemain ketika berinteraksi dengan Dream Sheep. Terdapat beberapa tipe teks yang ditambahkan. Setiap tipe teks memiliki variasi masing-masing sesuai dengan ketepatan *event* yang terjadi.

* + Teks untuk mendeskripsikan tingkat kekuatan*ability* yang dilancarkan kumpulan Dream Sheep.
  + Teks untuk mendeskripsikan efek yang dialami karakter non-pemain ketika terkena Dream Dust.
  + Teks untuk mendeskripsikan efek yang dialami pemain ketika terkena Dream Dust.

Kedua adalah *method*\_dream\_sheep\_sleep() yang memiliki beberapa fungsi:

* + Mengecek sasaran yang akan diserang oleh seekor Dream Sheep. Jika tidak ada sasaran atau sasaran yang diincar oleh Dream Sheep telah mati, maka *function* akan melakukan *return*.
  + Mengatur posisi dan gerakan gerombolan Dream Sheep, serta menghitung jumlah Dream Sheep dalam sebuah gerombolan
  + Men-*generate* kemungkinan berhasilnya sebuah gerombolan Dream Sheep untuk melakukan *ability*.
  + Memanggil *method*\_sheep\_message() untuk mengeluarkan *output* sesuai dengan hasil perhitungan dalam *method* ini.
  + Menentukan durasi Sleep yang dialami karakter sasaran ketika Dream Dust berhasil.

Ketiga adalah *case* MONS\_DREAM\_SHEEP dalam *method* bool mon\_special\_ability(monster\* mons) yang memiliki fungsi:

* + Mengecek apakah Dream Sheep tidak dalam kondisi bingung (*confused*) dan tidak sedang melarikan diri (*fleeing*), lalu mengecek apakah karakter sasaran tidak berada dalam sebuah *sanctuary*, sedang tidak tidur, dan tidak memiliki kekebalan terhadap Sleep.
  + Jika semua kondisi di atas terpenuhi, maka Dream Sheep memiliki peluang 20% untuk melakukan *ability*.
* mon-data.h

Pada berkas ini, Sheep ‘didaftarkan’ sebagai monster lama yang sudah tidak dipakai untuk versi baru, hanya disimpan untuk kompatibilitas versi lama. Selain itu, pengubahan juga banyak dilakukan untuk mengganti setiap *instance* dan konfigurasi yang terkait dengan Sheep agar menjadi Dream Sheep.

* mon-ench.cc

Secara umum, berkas ini berisi *function-function* yang mengatur karakteristik setiap *enchantment* yang terdapat dalam *game*, serta efeknya terhadap monster-monster yang ada. *Enchantment* merupakan jurus, atau *status effect* dari jurus yang dilancarkan oleh sebuah karakter atau *entity* dalam *game*, yang dapat memberikan efek tertentu pada sasaran yang menerima serangan jurus tersebut. Pada berkas ini, dilakukan pengubahan untuk *case*ENCH\_STICKY\_FLAME yang mengatur implementasi *spell* Sticky Flame. *Spell* tersebut memiliki keunikan berupa efektifitas yang sangattinggi terhadap monster dengan *genus*MONS\_SHEEP. *Genus* merupakan kelompok monster yang masih memiliki relasi biologis atau anatomis yang dekat, fiktif atau non-fiktif. Perubahan yang dilakukan adalah mengganti efek *spell* tersebut yang tadinya ditujukan secara spesifik untuk monster Sheep, menjadi untuk kelompok *genus* dari Sheep, yang mengandung Sheep dan Dream Sheep.

* mon-pick-data.h

Berkas ini berisi data mengenai kemunculan monster seperti lokasi dan peluang muncul. Data tersebut disimpan dalam beberapa *const* yang masing-masing mewakilkan kelompok monster berdasarkan lokasi *spawn* atau muncul. Berkas *header* ini dimanfaatkan oleh berkas mon-pick.cc. Pada berkas ini, dilakukan penambahan data untuk MONS\_DREAM\_SHEEP, dan penghapusan data untuk MONS\_SHEEP.Dream Sheep dan Sheep diprogram untuk *spawn* di kategori level *dungeon* yang bernama Lair.

* mon-place.cc

Berkas ini berisi *function-function* yang digunakan ketika menempatkan monster di dalam *dungeon*. Bagian yang diubah pada berkas ini merupakan:

* + *const*map<monster\_type, band\_set> bands\_by\_leader yang berisi daftar monster khusus yang memiliki peluang untuk *spawn* diikuti dengan sebuah *band* monster tersebut atau *band* monster lain. Pengubahan yang dilakukan adalah:
    - Mengganti konfigurasi kemunculan *band* Sheep menjadi untuk *band* Dream Sheep.
    - Mengganti perhitungan peluang 20% untuk kemunculan *band* Sheep yang mengikuti monster Cyclops menjadi peluang 100% untuk kemunculan *band* Dream Sheep.
  + *const*map<branch\_type, band\_weights> bands\_pickyang berisi penanganan *band-band* yang muncul mengikuti monster-monster dengan karakteristik kepemimpinan *band* khusus. Pengubahan yang dilakukan adalah:
    - Mengganti konfigurasi untuk kemunculan *band* Sheep yang mengikuti monster Torpor Snail menjadi untuk *band* Dream Sheep. Konfigurasi ini terdapat pada sebuah *case* tersendiri untuk MONS\_TORPOR\_SNAIL di instansiasi band\_type \_choose\_band().
  + *const*map<band\_type, vector<member\_possibilites>> band\_membership yang berisi daftar *band* monster beserta konfigurasi kemungkinan monster-monster yang terdapat dalam *band* tersebut. Pengubahan yang dilakukan adalah:
    - Mengganti konfigurasi komposisi *band* Sheep menjadi untuk *band* Dream Sheep.
* mon-spell.h

Berkas ini berisi daftar *spell* yang dimiliki oleh semua monster yang terdapat dalam *game*.*Spell* disusun berdasarkan monster yang memiliki *spell* tersebut.Pada berkas ini, terdapat penambahan berupa sebuah kategori *spell* yang dimiliki oleh Dream Sheep. Kategori tersebut berisi sebuah *spell* yang memiliki nama obyek SPELL\_SLEEP.

* player-reacts.cc

Berkas ini berisi *function-function* yang berhubungan dengan karakter pemain. Sebagian besar *function* yang terdapat dalam berkas ini digunakan untuk mengatur *enchantment* pada karakter pemain. Pada berkas ini, terdapat pengubahan *typo* untuk *command*you.awake() menjadi you.awaken(). *Command* ini berfungsi untuk membangunkan pemain setelah Sleep yang dialami pemain kadaluarsa.

* player.cc

Berkas ini berisi *function-function* yang berhubungan dengan pemain. Perbedaan dengan player-reacts.cc adalah player.cc berisi *function* untuk penggunaan yang lebih umum. Terdapat beberapa penambahan dan pengubahan pada berkas ini:

* + Penambahan *bool*player::can\_sleep yang memeriksa apakah pemain dapat dibuat tidur oleh *enchantment* Sleep. Fungsi ini me-return true jika pemain tidak memiliki kekebalan terhadap Sleep.
  + Pengubahan *typo* untuk *method*player::awake() menjadi player::awaken()
  + Pengubahan instansiasi nilai kekebalan pemain terhadap Sleep di player::awaken(). Sebelum diubah, pemain di-set untuk memiliki kekebalan selama 1 turn. Setelah pengubahan, pemain hanya di-set untuk memiliki kekebalan selama 1 turn jika pada saat fungsi dipanggil pemain tidak memiliki kekebalan sama sekali.
  + Pengubahan *typo* untuk pemanggilan *method*awake() menjadi awaken() di *method*player::check\_awaken() yang memeriksa apakah pemain sudah terbangun.
* player.h

Berkas ini merupakan *header* untuk player.cc. Isi dari berkas ini merupakan inisialisasi variabel-variabel yang digunakan di player.cc. Pengubahan yang dilakukan pada berkas ini adalah pengubahan *typo* untuk inisialisasi awake() menjadi awaken().

Daftar perubahan *code* ini tidak menyertakan berkas-berkas non-*source code* yang tidak relevan terhadap fokus penelitian seperti berkas-berkas .des yang merepresentasikan deskripsi monster atau berkas-berkas gambar yang merepresentasikan visualisasi monster dalam *game*.

### **3.2.3 *Pull requests* terkait penambahan Dream Sheep**

Selain *commit* utama yang dilakukan oleh isloat, terdapat *commit-commit* lain yang terdapat dalam beberapa *pull request* yang dibuat olehnya. *Pull requests* ini memiliki beberapa tujuan seperti untuk memindahkan letak beberapa fungsi *code* dari suatu berkas ke berkas lain, menghilangkan variabel-variabel yang tidak perlu, hingga menyiapkan fungsi yang sewaktu-waktu mungkin dipakai sebagai antisipasi terhadap ketidakseimbangan permainan yang dirasakan oleh pemain akibat penambahan Dream Sheep. Berikut adalah *pull requests* yang dibuat oleh isloat:

* Sheep no spell-sleep <https://github.com/crawl/crawl/pull/335>

Pull Request ini memiliki beberapa *commit*:

* + Remove emergency sleep spell from dream sheep

Dikutip dari deskripsi *pull request* tersebut, *spell* Sleep yang dimiliki oleh Dream Sheep merupakan *spell* ‘*emergency*’ yang justru dijalankan ketika seharusnya Dream Sheep lebih bergantung kepada *spell* Dream Dust. Dream Sheep dalam jumlah kecil sudah memiliki peluang untuk membuat pemain tertidur dengan menggunakan Dream Dust, sehingga *emergency* Sleep dirasa tidak perlu dan membingungkan pemain, terutama karena Sleep mengecek Magic Resistance sedangkan Dream Dust tidak.

Berkas *source-code* yang diubah:

* + - mon-spell.h

Fungsi MST\_DREAM\_SHEEP yang berisi Sleep dihapus dari *source code*.

* + Remove SPFLAG\_BATTLE

Menghilangkan fungsi SPFLAG\_BATTLE dari *source code* karena tidak terpakai. Fungsi ini tidak berhubungan dengan Dream Sheep.

Berkas *source code* yang diubah:

* + - spl-cast.h

Inisialisasi *enum* SPFLAG\_BATTLE dihapus dari *source code*.

* + - spl-data.h

SPFLAG\_BATTLE dihapus dari semua fungsi di *source code*.

* + Make dream dust a proper monspell

Memindahkan *code* implementasi Dream Dust dari mon-abil.cc ke mon-spell.h dan mon-cast.cc. “Dream Dust” dapat dilihat sebagai *natural ability* di layar ‘xv’.

Berkas *source code* yang diubah:

* + - enum.h

SPELL\_DREAM\_DUST ditambahkan ke dalam daftar *spell*.

* + - mon-abil.cc

Implementasi Dream Dust dihapus dari berkas ini.

* + - mon-cast.cc

Dream Dust diimplementasi di berkas ini. Penjelasan terdapat di beberapa Materi Tambahan sebelumnya.

* + - mon-spell.h

MST\_DREAM\_SHEEP berisi SPELL\_DREAM\_DUST beserta konfigurasinya ditambahkan dalam berkas ini.

* + - spl-data.h

Penambahan Dream Dust beserta data dan konfigurasinya.

* Make Clarity resist sleep <https://github.com/crawl/crawl/pull/331>

Untuk memberikan kegunaan lain pada Clarity yang cukup langka dan sulit untuk didapatkan, ditambahkan fungsi agar Clarity dapat memberikan kekebalan kepada pemain terhadap Sleep.

Berkas *source code* yang diubah:

* + actor.cc

Penambahan fungsi yang mengecek Clarity pada pemain yang menjadi sasaran Sleep.

* Remove chain sleeps <https://github.com/crawl/crawl/pull/334>

Untuk mencegah Sleep yang berkelanjutan atau berkepanjangan, ditambahkan sebuah fungsi yang membuat pemain mendapatkan kekebalan terhadap Sleep selama 2 hingga 4 *turn*.

Berkas *source code* yang diubah:

* + mon-abil.cc

Menghilangkan implementasi kekebalan yang lama di mana kekebalan diberikan dalam *turn* yang sama dengan saat pemain terkena Sleep. Kekebalan yang lama memiliki durasi 4 hingga 8 *turn* tanpa melihat durasi Sleep yang dialami pemain.

* + player.cc

Mengganti fungsi kekebalan Sleep yang lama menjadi fungsi baru dengan durasi kekebalan 2 hingga 4 *turn*.

* Reduce sleep damage bonus <https://github.com/crawl/crawl/pull/336>

Mengurangi *damage multiplier* terhadap karakter yang sedang terkena Sleep dari 2.5x menjadi 2.0x.

Fungsi ini tidak jadi dipakai. *Pull Request* ini dibuat sebagai antisipasi jika Dream Sheep dirasa terlalu kuat atau mengganggu. Menurut isloat *pull Request* ini tidak jadi dibutuhkan karena implementasi Pull Request #334 dan #335 telah berfungsi dengan baik untuk menanggulangi masalah *balancing* Dream Sheep.

Berkas *source code* yang diubah:

* + melee\_attack.cc

Pengubahan *damage* yang diterima oleh karakter yang sedang tertidur dari dikalikan dengan 5/2 menjadi dikalikan dengan 2.

### **3.2.4 Masalah yang ditimbulkan oleh penambahan Dream Sheep**

Isu-isu berikut merupakan keluhan dari komunitas pemain Dungeon Crawl Stone Soup terkait penambahan karakter Dream Sheep dalam *game* yang diambil dari laman *web* [https://www.reddit.com/r/dcss/comments/502w5p/please\_nerf\_or \_remove\_dream\_sheep\_preferably\_by/](https://www.reddit.com/r/dcss/comments/502w5p/please_nerf_or%20_remove_dream_sheep_preferably_by/) :

* Dream Dust tidak mengecek Magic Resistance

Menurut pemain-pemain yang telah menghadapi Dream Sheep dalam *game*, Dream Sheep terlalu mudah membuat pemain tertidur dengan *ability* Dream Dust-nya. Hal ini disebabkan oleh mekanika Dream Dust yang hanya mengecek Clarity, yaitu sebuah *intrinsic* (sebuah kekuatan khusus yang dimiliki pemain melalui cara tertentu) yang melindungi karakter pemain dari gangguan psikis. Banyak pemain yang beranggapan bahwa sebaiknya Dream Dust juga mengecek Magic Resistance pemain, yaitu ketahanan pemain terhadap jurus atau *ability* sihir musuh. Dengan demikian, pemain yang tidak memiliki Clarity memiliki cara atau kesempatan untuk sepenuhnya menangkal Dream Dust.

* Kekebalan terhadap Sleep belum terimplementasi dengan baik

Menurut salah satu *developer* Dungeon Crawl Stone Soup dengan *username* **amalloy** di *thread* Reddit di atas, seharusnya pemain menerima kekebalan terhadap Sleep selama beberapa *turn* setelah pemain mengalami Sleep, tetapi untuk sekarang fungsi tersebut belum bekerja.

* Mekanika yang belum diimplementasi

Perihal pengecekan Magic Resistance dalam implementasi Dream Dust sebenarnya sudah dibahas oleh **isloat** sebagai *contributor* Dream Sheep dalam deskripsi *commit­-*nya. **isloat** mengatakan bahwa jika pengecekan Clarity saja tidak cukup untuk mencegah Dream Sheep menjadi terlalu kuat atau mengganggu, maka pengecekan Magic Resistance akan ditambahkan dalam implementasi Dream Dust.

Kemungkinan besar pengecekan Magic Resistance tidak dilakukan sejak awal untuk menghindari perubahan *code* dalam jumlah besar, sehingga menyulitkan *contributor* dan memperpanjang masa *development*.

* Terlalu menyusahkan pemain

Karena poin-poin di atas, Dream Sheep terasa terlalu kuat atau terlalu menyusahkan pemain. Contoh kasus atau situasi di mana Dream Sheep terasa terlalu menyusahkan pemain adalah ketika *band* Dream Sheep muncul berdekatan dengan karakter musuh lainnya yang jauh lebih kuat. *Ability* Dream Dust dari Dream Sheep dapat dengan mudah membuat karakter pemain tertidur, sehingga karakter musuh lain dapat menyerang karakter pemain dengan leluasa tanpa perlawanan. Penempatan Dream Sheep di level yang cukup lanjut, di mana sebagian besar monster yang menghuni level tersebut memiliki kekuatan dan kemampuan kelas tinggi, sehingga ketika pemain dibuat tertidur, musuh-musuh tersebut dapat membunuh pemain hanya dengan sedikit serangan. Dalam beberapa kasus yang dialami pemain, musuh dapat membunuh pemain dengan sekali serang. Waktu bermain ketika berhadapan dengan Dream Sheep pun menjadi sangat sedikit karena ketika pemain terkena Sleep, pemain langsung dibunuh oleh karakter musuh lain.

* Penghapusan karakter Sheep dari *game*.

Penambahan Dream Sheep dalam *game* membuat *entity* Sheep biasa dihapus dari *game*. Hal ini menerima respon yang sangat bercampur di komunitas pemain. Beberapa pemain menyebutkan bahwa Sheep memang sepantasnya dihilangkan dari *game* karena merupakan karakter yang tidak berguna dan tidak memiliki efek atau pengaruh terhadap permainan. Di sisi lain, banyak pemain yang menyatakan kecewa terhadap penghapusan karakter Sheep dari *game*. Pemain-pemain tersebut menjelaskan bahwa Sheep memang sangat lemah dan tidak berguna, namun memiliki unsur hiburan tersendiri bagi pemain. Sheep seringkali menjadi bahan lelucon atau menjadi bulan-bulanan pemain, sehingga memberikan kesenangan atau kepuasan tersendiri bagi beberapa pemain yang berhadapan dengan Sheep.

### **3.2.5 Isu-isu pada pengembangan Dream Sheep**

* Kompleksitas *source code*

Pada umumnya, penambahan *entity* terutama karakter monster baru dalam *game* Dungeon Crawl Stone Soup mengharuskan *contributor* atau *developer* untuk mengubah lebih dari 20 buah berkas *source code* dan teks deskripsi. Kerumitan ini dapat membuat pengembang *game* ragu-ragu untuk mengimplementasi perubahan yang signifikan, atau hal-hal kecil yang implementasi *code*-nya tersebar dalam banyak berkas*.* Pada contoh kasus Dream Sheep, masalah ini dapat dilihat (atau diperkirakan/asumsikan terjadi) pada masalah pengecekan Magic Resistance yang masih berupa *To-Do* yang baru akan dikerjakan jika mekanika yang sudah diimplementasi dirasa kurang baik, di mana sebaiknya (atau seharusnya) pengecekan Magic Resistance diimplementasi sejak awal, hanya saja implementasi pengecekan Magic Resistance harus dilakukan dengan mengubah banyak berkas *code*.

* *Entity Balancing*

Berdasarkan poin-poin masalah yang dibahas sebelumnya, dapat ditarik kesimpulan bahwa salah satu isu yang signifikan dalam penambahan *entity* ke dalam *game* adalah *balancing*atau penyeimbangan karakteristik *entity* tersebut dengan lingkungan (dalam konteks ini, mekanika dan karakteristik *entity* lain dalam *game*). Keseimbangan yang dimaksud adalah ketika *entity* yang ditambahkan tidak memiliki perbedaan yang signifikan dengan *entity* lain dalam *game*, tidak mengubah pengalaman bermain yang dirasakan pemain secara signifikan, tidak terlalu kuat dan menyulitkan bagi pemain, serta tidak terlalu lemah dan tidak berefek terhadap permainan.

* *Gameplay Balancing*

Selain penyeimbangan *entity*, perlu dilakukan juga penyeimbangan mekanika *gameplay* terhadap *entity*. Untuk kasus penambahan *entity* secara masif atau signifikan (perlu contoh kasus, tapi belum ditemukan, tapi yakin ada), kurang efektif dan efisien jika penyeimbangan diberatkan pada *entity*, di mana setiap *entity* perlu menyesuaikan diri dengan mekanika yang ada. Untuk kasus tersebut, ada baiknya bila mekanika *gameplay* yang berubah mengikuti *entity-entity* baru yang ditambahkan. Pengubahan atau penyesuaian *gameplay* juga dapat dilakukan untuk men-*supportentity* yang ada demi pengalaman bermain yang lebih imersif. Contohnya adalah penggunaan *scripted difficulty* pada level terakhir Ace Combat: Assault Horizon, di mana pada umumnya semua karakter musuh memiliki *artificial intelligence* sendiri untuk bertempur, tetapi khusus untuk karakter musuh *boss* pada level terakhir, semua gerakan*boss*dan *event* yang berhubungan dengan *boss* tersebut sudah ditentukan dalam *script* untuk memberikan kesan *thrill* yang lebih kuat pada pemain.

### **3.2.6 Definisi *entity***

Dalam konteks penelitian dan studi kasus yang terkait, *entity* didefinisikan sebagai benda atau karakter yang memiliki signifikansi dalam *game*. Contoh dari *entity* dalam *game* ini adalah monster, *weapon* (senjata), dan *god* (dewa).

Berdasarkan studi kasus yang dipilih, yaitu Dream Sheep yang merupakan sebuah monster, maka eksperimen akan dilakukan pada proses penambahan *entity* berupa monster.

**BAB 4**

# **RANCANGAN EKSPERIMEN**

Bab ini menjelaskan rincian eksperimen yang dilakukan, beserta penjelasan dasar-dasar dari langkah yang dilakukan.

Eksperimen yang akan dilakukan dalam penelitian ini adalah melakukan perubahan pada *source code* dari *game* studi kasus. Perubahan yang akan dilakukan adalah membuat dua buah versi dari *game*, di mana masing-masing versi memiliki beberapa aspek atau fitur yang sama, tetapi untuk beberapa aspek atau fitur lain memiliki perbedaan konfigurasi.

## **4.1 *Commonalities* dan *variabilities* dari *game***

Untuk dapat menerapkan SPL, perlu didefinisikan *commonalities* dan *variabilities* dari *software* yang dikembangkan.

### **4.1.1 *Commonalities***

*Commonality* merupakan fitur-fitur atau atribut yang memiliki kesamaan. Berikut merupakan *commonalities* ysng dapat diangkat dari fitur penambahan *entity*.

1. Pengguna fungsi

Beberapa fungsi terkait monster yang diimplementasi dalam *source code game* dimiliki atau digunakan oleh monster yang sama. Contohnya *spell* X dan *ability* Y sama-sama dimiliki oleh monster Z.

1. Komponen monster

Secara umum, semua monster yang ada dalam *game* memiliki struktur komponen yang sama, seperti identitas nama, deskripsi teks, dialog teks, pola serangan, pola gerakan, kemampuan mengeluarkan jurus, dan lain lain.

### **4.1.2 *Variabilities***

*Variabilities* merupakan fitur-fitur atau atribut-atribut yang dibedakan dalam sebuah *product line*. Berikut adalah *variabilities* yang terdapat pada fitur penambahan monster.

1. Identitas

Monster-monster yang dibuat dapat memiliki identitas dan representasi visual maupun audio yang berbeda.

1. Konfigurasi kekuatan

Kekuatan yang dimiliki setiap monster dapat bervariasi. Terdapat monster yang secara umum lemah ataupun kuat. Terdapat pula monster yang memiliki kelebihan pada kecepatan, dan juga monster yang memiliki kelebihan pada ketahanan tubuh.

1. Pola gerakan dan serangan

Setiap monster memiliki ketentuan cara bergerak dan kecepatan masing-masing. Selain gerakan, monster-monster yang ada juga memiliki pola serangan yang berbeda.

1. *Spell*

Beberapa monster yang memiliki kekuatan magis dapat menggunakan *spell* untuk menyerang pemain dan monster lain. Terdapat bermacam *spell* yang berbeda untuk setiap monster.

1. *Ability*

*Ability* merupakan kekuatan khusus yang dimiliki beberapa monster yang biasanya memberikan suatu efek kepada monster-monster yang memilikinya. Terdapat bermacam-macam *ability* yang berbeda yang dapat dimiliki oleh monster.

Dengan membandingkan *commonalities* dan *variabilities* yang telah didefinisikan dari penambahan monster, dapat dirancang sebuah struktur *code* baru yang dibuat berdasarkan spesifikasi *commonality* dan *variability* untuk penerapan SPL.

Struktur *code* yang baru akan mengelompokkan fungsi-fungsi yang ada sesuai dengan *commonality* yang telah didefinisikan, yaitu monster pengguna fungsi dan komponen monster. Dengan begitu, *code* yang baru akan mengelompokkan fungsi-fungsi yang termasuk dalam *variabilities* dengan pengguna yang sama ke dalam sebuah kumpulan fungsi.

## **4.2 Class-Responsibility-Collaboration Cards**

Menurut Beck dan Cunningham (1989), CRC *cards* atau Class-Responsibility-Collaboration *cards* dapat digunakan sebagai alat penunjang pembelajaran pemrograman *object-oriented*. Selain sebagai metode pembelajaran, CRC *cards* juga dapat digunakan sebagai alat *brainstorming* menurut Cockburn (1999). Metode CRC *cards* ini akan digunakan untuk membagi fungsi-fungsi yang relevan dengan penambahan *entity* menjadi beberapa *class* yang terstruktur.

### **4.2.1 *Responsibilities***

Untuk eksperimen ini, *responsibilities*-*responsibilities* yang akan diimplementasi diasarkan pada komponen-komponen monster yang termasuk dalam aspek-aspek *varibilities*. Berikut adalah contoh sederhana dari pembagian *responsibilities*:

* + Texts

Bertanggung jawab untuk setiap deskripsi teks mengenai monster

* + Behavior

Bertanggungjawab untuk pola gerakan, serangan, dan perilaku monster

* + Spell

Bertanggungjawab untuk jurus-jurus yang dimiliki oleh monster

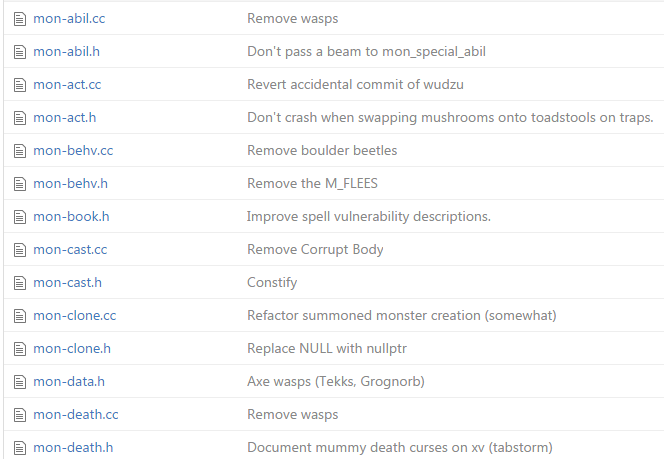
### **4.2.2 *Collaboration***

Dalam penerapan CRC, perlu didefinisikan *collaborator* yang mengimplementasikan *class-class* dan *responsibilities* yang telah dibuat. Untuk studi kasus ini, *collaborator* dapat ditentukan dengan melihat *class-class* yang di-*include* oleh setiap *class* yang menerapkan *responsibilities*.

*Collaboration* antara masing-masing *class* akan dimodelkan dalam subbab Rancangan Struktur Code.

## **4.3 Perbedaan struktur *code***

Struktur *code* yang asli sudah memiliki *syntax* pemrograman berbasis obyek, namun jika dilihat dari pengelompokan fungsi, struktur *code* masih mengelompokkan fungsi-fungsi berdasarkan kegunaan. Sebagai contoh, struktur yang original membagi *code* berdasarkan komponen-komponen monster menjadi mon-spell yang berisi *spell*, mon-behv yang berisi *behavior*, dan mon-data yang berisi data. Pada Gambar 4.1 dapat dilihat *screenshot* yang diambil dari *repository* Dungeon Crawl Stone Soup di <http://github.com/crawl/crawl/crawl-ref/source> yang menunjukkan pembagian berkas-berkas *code*.



**Gambar 4.1** Beberapa contoh pembagian berkas *code* pada *source code* asli

Struktur ini dapat diubah menjadi membagi code berdasarkan obyek seperti monster 1, monster 2, dan monster 3 yang masing-masing memiliki komponen masing-masing.

Ilustrasi *code* original dapat dilihat pada *code* berikut:

Spell

{

Spell monster 1

Spell monster 2

Spell monster 3

}

Behavior

{

Pola gerakan monster 1

Pola gerakan monster 2

Pola gerakan monster 2

}

Data

{

Data monster 1

Data monster 2

Data monster 3

}

***Source Code* 4.1** Struktur *code* lama

Struktur *code* tersebut dapat diubah menjadi struktur baru yang dapat dilihat pada *code* berikut:

Monster 1

{

Spell monster 1

Pola gerakan monster 1

Data monster 1

}

Monster 2

{

Spell monster 2

Pola gerakan monster 2

Data monster 2

}

Monster 3

{

Spell monster 3

Pola gerakan monster 3

Data monster 3

}

***Source Code* 4.2** Struktur *code* baru

## **4.4 Rancangan struktur *code***

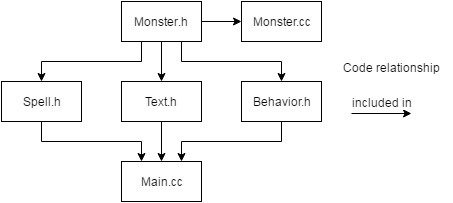
Dilihat dari isi direktori *game*, Dungeon Crawl Stone Soup terdiri atas lebih dari 15.600 berkas, di mana sekitar 15.500 di antaranya terdapat dalam *folder* “source” dan kurang lebih 622 di antaranya merupakan *code* C++ yang membangun fungsionalitas utama dari *game*. Dari 622 berkas *code* C++, sebagian besar di antaranya memiliki panjang lebih dari 1.000 *line of code*.

Berdasarkan pengetahuan yang didapat dari *commit* Dream Sheep, untuk menambahkan sebuah monster baru ke dalam *source code* *game* saat ini secara umum dibutuhkan perubahan pada 11 berkas *code*. Fungsi-fungsi yang ditambahkan banyak memiliki keterkaitan dengan fungsi lain yang menempati *class* yang berbeda. Selain itu, terdapat fungsi-fungsi yang berada di dalam sebuah *condition* atau *loop* yang tidak dapat dipisahkan dengan fungsi lainnya, kecuali dilakukan perubahan fundamental pada struktur *code* secara keseluruhan.

Untuk mensimulasikan penelitian ini, eksperimen dilakukan dengan program yang terdiri dari berkas-berkas *code* *mock-up* yang dibuat berdasarkan struktur *code* sebenarnya, dilihat dari pembagian fungsi dan *inheritance* terhadap berkas dan *class* lain.

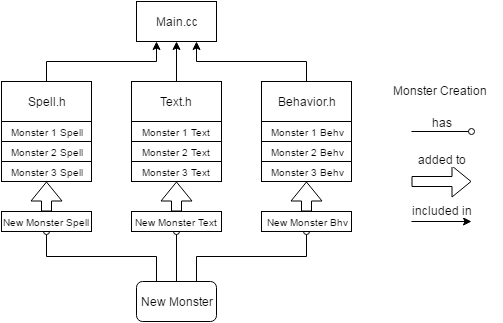
### **4.4.1 Desain struktur *code* lama**

Pada struktur *code* yang asli, fungsi-fungsi yang mengimplementasikan *game* dibagi-bagi ke dalam beberapa berkas berdasarkan jenis fungsi atau kegunaan. Contoh pembagian *code* dari struktur asli adalah *ability*, *data*, dan *behavior*.



**Gambar 4.2** Hubungan antar berkas *code* pada struktur *code* lama

Bentuk implementasi sebuah monster dalam struktur ini dapat dilihat pada Gambar 4.2. Sebagai perumpamaan, sebuah monster memiliki komponen berupa Spell, Text, dan Behavior. Masing-masing komponen dari sebuah monster dipecah ke dalam berkas *code* masing-masing yang berisi Spell dari semua monster, Text dari semua monster, dan Behavior dari semua monster. *Class* Monster merepresentasikan berkas-berkas dan *class*-*class* yang di-*include* atau di-*extend* oleh *class-class* yang merepresentasikan komponen-komponen monster.

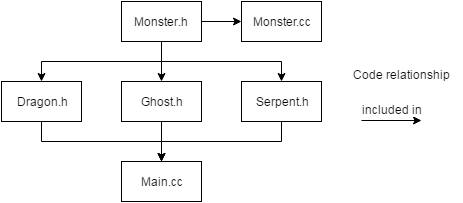


**Gambar 4.3** Ilustrasi penambahan monster baru pada struktur *code* lama

Ketika akan menambahkan sebuah monster baru ke dalam *game*, *contributor* perlu menambahkan fungsi-fungsi yang ingin diimplementasi ke dalam banyak berkas. *Flow* penambahan monster dapat pada struktur *code* ini dapat dilihat pada Gambar 4.3.

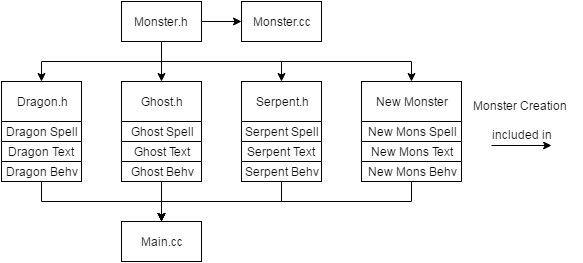
### **4.4.2 Desain struktur *code* baru**

Dalam penelitian ini, struktur *code* akan diubah menjadi bentuk susunan yang dapat memanfaatkan paradigma pemrograman berbasis obyek secara lebih optimal, yaitu dengan mengelompokkan fungsi-fungsi yang ada berdasarkan monster yang menggunakan fungsi tersebut. Dengan mengelompokkan fungsi-fungsi tersebut berdasarkan monster yang menggunakannya, seorang *contributor* tidak perlu mengubah banyak berkas *code* hanya untuk menambahkan sebuah monster.



**Gambar 4.4** Hubungan antar berkas *code* pada struktur *code* baru

Bentuk implementasi dari struktur *code* yang telah di-*refactor* dapat dilihat pada Gambar 4.4. Pembagian fungsi yang semula berdasarkan jenis fungsi diubah menjadi berdasarkan *entity* yang menggunakan fungsi. Pengubahan *code* ini dapat membantu mempermudah proses penambahan monster baru ke dalam *game*. Ketika sebuah monster baru akan ditambahkan ke dalam *game*, *contributor* hanya perlu membuat sebuah berkas atau *class* baru yang berisi seluruh fungsi dari monster yang akan ditambahkan. *Flow* penambahan monster dapat pada struktur *code* ini dapat dilihat pada Gambar 4.5.



**Gambar 4.5** Ilustrasi penambahan monster baru pada struktur *code* baru

## **4.5 Integrasi rancangan *code* ke dalam *game***

Setelah memodelkan bentuk *code* ideal yang ingin dicapai pada subbab sebelumnya, pada subbab ini akan dijelaskan penerapan dari bentuk tersebut pada *source code* asli *game*.

### **4.5.1 Pengubahan paradigma *code* lama menjadi berbasis obyek**

Agar *source code* *game* dapat lebih menerapkan konsep-konsep SPL, dilakukan *refactoring* yang mengubah struktur *code* dari elemen-elemen *game* selain monster menjadi lebih berbasis obyek. *Refactoring* ini juga dilakukan sebagai percobaan dasar sebelum memulai percobaan pengubahan *code* pada monster studi kasus yaitu Dream Sheep.

### **4.5.2 Penerapan rancangan desain *code* terhadap Dream Sheep**

Sesuai dengan penjelasan pada subbab sebelumnya, penerapan rancangan desain *code* dilakukan dengan mengelompokkan fungsi-fungsi terkait sebuah monster ke dalam sebuah *class*.

### **4.5.3 Indikator keberhasilan eksperimen**

Untuk memastikan bahwa pengubahan yang dilakukan terhadap *source code* dapat berjalan dengan baik, repository dari *source code* dihubungkan dengan Travis-CI agar setiap *build* dapat dites. Selain *build testing* menggunakan Travis-CI, dibuat pula perintah untuk memberikan *output* pada *terminal* setiap kali fungsi yang diubah dijalankan. Jika untuk setiap fungsi yang dijalankan terdapat *output* yang telah ditentukan pada *terminal*, maka dapat diasumsikan fungsi berjalan dengan baik.

## **4.6 Rancangan produk variasi dari *game***

*Software Product Line* dapat diterapkan dengan cara membuat variasi dari *game* yang ada. Berdasarkan fokus penelitian yang berpusat pada struktur *code* monster, maka ditentukan bahwa untuk penelitian ini variasi yang akan dibuat adalah sebuah *game* baru dengan monster Dream Sheep yang berbeda.

**BAB 5**

# **EKSPERIMEN**

Berdasarkan rancangan desain struktur *code* yang telah dibuat pada bab sebelumnya, dilakukan eksperimen untuk menguji penerapan desain struktur *code* yang baru tersebut. Eksperimen dilakukan dengan membuat perubahan pada struktur *code* yang lama dan menggantinya dengan struktur *code* yang baru.

## **5.1 Memodelkan struktur *code* lama**

Untuk membantu pengubahan, dilakukan pemodelan struktur *code*. Bentuk *code* yang lama secara umum dapat digambarkan melalui beberapa contoh *code* sederhana yang melambangkan beberapa komponen monster.

### **5.1.1 Monster.h**

Terdapat sebuah *class* Monster yang berisi representasi fungsi-fungsi yang terdapat pada *class*-*class* Spell, Behavior, dan Text. Untuk memudahkan representasi, setiap fungsi dari *class-class* tersebut ditampilkan dalam bentuk *input* teks. Isi dari Monster.h dapat dilihat pada *source code* berikut:

#include <iostream>

#include <string>

**using** **namespace** std**;**

class Monster

**{**

protected**:**

void createSpell**(**string name**,** int cost**);**

void addBehavior**(**bool attack**);**

void addText**(**string text**);**

**};**

***Source Code* 5.1** Implementasi Monster.h pada struktur *code* lama

Berkas *header class* Monster tersebut berisi deklarasi fungsi-fungsi yang akan dijalankan dalam *class-class* Spell, Behavior, dan Text sebagai bentuk representasi fungsinya.

### **5.1.2 Monster.cc**

Implementasi dari fungsi-fungsi yang dideklarasikan dalam Monster.h dapat dilihat pada berkas Monster.cc

#include "Monster.h"

void Monster**::**createSpell**(**string name**,** int cost**)**

**{**

cout **<<** "Spell: "**<<** name **<<** endl**;**

cout **<<** "Spell cost: " **<<** cost **<<** endl**;**

**}**

void Monster**::**addBehavior**(**bool attack**)**

**{**

**if(**attack**)**

**{**

cout **<<** "Monster is aggressive" **<<** endl**;**

**}**

**else**

**{**

cout **<<** "Monster is not aggressive" **<<** endl**;**

**}**

**}**

void Monster**::**addText**(**string text**)**

**{**

cout **<<** text **<<** endl **<<** endl**;**

**}**

***Source Code* 5.2** Implementasi Monster.cc pada struktur *code* lama

Seperti yang dapat dilihat pada *Source Code* 5.2, berkas Monster.cc tersebut berisi fungsi-fungsi yang mengeluarkan *output* berupa teks sesuai dengan nilai variabel yang nantinya akan ditentukan oleh fungsi-fungsi Spell, Behavior, dan Text. createSpell merupakan bentuk implementasi dari *spell* sebuah monster. addBehavior merupakan fungsi yang menentukan apakah sebuah monster agresif atau tidak. addText berisi deskripsi dari sebuah monster. Setiap implementasi komponen ditampilkan dalam bentuk teks.

### **5.1.3 Spell.h**

#include <iostream>

#include <string>

#include "Monster.h"

class Spell **:** protected Monster

**{**

protected**:**

void spell\_Dragon**()**

**{**

Monster**::**createSpell**(**"Fire"**,**10**);**

**}**

void spell\_Serpent**()**

**{**

Monster**::**createSpell**(**"Poison"**,**5**);**

**}**

void spell\_Ghost**()**

**{**

Monster**::**createSpell**(**"Curse"**,**20**);**

**}**

**};**

***Source Code* 5.3** Implementasi Spell.h pada struktur *code* lama

Di dalam berkas Spell.h, terdapat fungsi-fungsi yang merepresentasikan *spell* dari masing-masing monster. Masing-masing monster memiliki *method*-nya masing-masing yang berisi pemanggilan *method* createSpell yang berada pada *class* Monster dengan parameter yang telah diisi berdasarkan spesifikasi tiap monster. Isi dari Spell.h dapat dilihat pada *Source Code* 5.3.

### **5.1.4 Behavior.h**

#include <iostream>

#include <string>

#include "Monster.h"

class Behavior **:** protected Monster

**{**

protected**:**

void behavior\_Dragon**()**

**{**

Monster**::**addBehavior**(true);**

**}**

void behavior\_Serpent**()**

**{**

Monster**::**addBehavior**(true);**

**}**

void behavior\_Ghost**()**

**{**

Monster**::**addBehavior**(false);**

**}**

**};**

***Source Code* 5.4** Implementasi Behavior.h pada struktur *code* lama

Di dalam berkas Behavior.h, terdapat fungsi-fungsi yang merepresentasikan *behavior* dari masing-masing monster. Tidak jauh berbeda dengan Spell.h, Behavior.h berisi *method*-*method* yang dispesifikasikan ke masing-masing monster yang berisi pemanggilan *method* addBehavior dari *class* Monster yang telah diisi dengan parameter sesuai dengan spesifikasi masing-masing Monster. Isi dari Behavior.h dapat dilihat pada *Source Code* 5.4.

### **5.1.5 Text.h**

#include <iostream>

#include <string>

#include "Monster.h"

class Text **:** protected Monster

**{**

protected**:**

void text\_Dragon**()**

**{**

Monster**::**addText**(**"A flying monster that breathes fire"**);**

**}**

void text\_Serpent**()**

**{**

Monster**::**addText**(**"A snake-like monster that spews poison"**);**

**}**

void text\_Ghost**()**

**{**

Monster**::**addText**(**"An astral creature wandering around"**);**

**}**

**};**

***Source Code* 5.5** Implemetasi Text.h pada struktur *code* lama

Di dalam berkas Text.h, terdapat fungsi-fungsi yang merepresentasikan deskripsi dari masing-masing monster seperti yang dapat dilihat pada *Source Code* 5.5.

### **5.1.6 Main.cc**

#include "Spell.h"

#include "Behavior.h"

#include "Text.h"

#include "Monster.h"

class Print **:** protected Spell**,** protected Behavior**,** protected Text**,** protected Monster

**{**

public**:**

void create**()**

**{**

cout **<<** "Dragon" **<<** endl**;**

Spell**::**spell\_Dragon**();**

Behavior**::**behavior\_Dragon**();**

Text**::**text\_Dragon**();**

cout **<<** "Serpent" **<<** endl**;**

Spell**::**spell\_Serpent**();**

Behavior**::**behavior\_Serpent**();**

Text**::**text\_Serpent**();**

cout **<<** "Ghost" **<<** endl**;**

Spell**::**spell\_Ghost**();**

Behavior**::**behavior\_Ghost**();**

Text**::**text\_Ghost**();**

**}**

**};**

int main**()**

**{**

Print print**;**

print**.**create**();**

return 0;

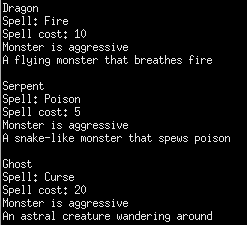
}

***Source Code* 5.6** Implementasi Main.cc pada struktur *code* lama

Fungsi-fungsi yang berada di *class-class* Spell, Behavior, dan Text akan dijalankan oleh Main.cc. Masing-masing fungsi dari setiap *class* akan dijalankan secara berurutan. Isi dari berkas Main.cc dapat dilihat pada *Source Code* 5.6.

### **5.1.7 *Output* dari struktur *code* lama**

Jika rangkaian *code* tersebut dijalankan, maka akan dihasilkan sebuah *output* berupa teks yang merepresentasikan setiap monster yang diimplementasi beserta komponen-komponen dari masing-masing monster. Contoh *output* dari pemodelan struktur *code* lama dapat dilihat pada Gambar 5.1.



**Gambar 5.1** *Output* teks dari rangkaian *code* dengan struktur *code* lama

## **5.2 Memodelkan struktur *code* baru**

Setelah memodelkan struktur *code* lama, dilakukan pemodelan serupa terhadap struktur *code* baru. Dalam bentuk implementasi ini, dibuat sebuah *class* Monster yang merepresentasikan semua monster yang terdapat dalam *game*.

### **5.2.1 Monster.h**

Berikut adalah contoh berkas Monster.h yang merepresentasikan bentuk umum dari monster. Berkas ini memiliki beberapa *method* yang bertanggungjawab atas komponen-komponen monster seperti yang dapat dilihat pada *source code* berikut:

#include <iostream>

#include <string>

class Monster

**{**

protected**:**

void newMonster **(**string monster**);**

void createSpell**(**string name**,** int cost**);**

void addBehavior**(**bool attack**);**

void addText**(**string text**);**

**};**

***Source Code* 5.7** Implementasi Monster.h dalam struktur *code* baru

### **5.2.2 Monster.cc**

Berkas Monster.h merupakan *header* dari berkas Monster.cc yang berisi implementasi dari fungsi-fungsi yang terdapat di Monster.h. Monster.h dan Monster.cc merupakan berkas-berkas yang merepresentasikan *class* Monster. Sebagai contoh dalam eksperimen, di dalam *class* Monster terdapat fungsi-fungsi yang menentukan identitas berupa nama monster, data mengenai *spell* yang dimiliki monster, perilaku monster, serta teks deskripsi yang menjelaskan monster tersebut. Implementasi fungsi-fungsi tersebut dapat dilihat pada *source code* berikut:

#include "Monster.h"

void Monster**::**newMonster**(**string monster**)**

**{**

cout **<<** monster **<<** endl**;**

**}**

void Monster**::**createSpell**(**string name**,** int cost**)**

**{**

cout **<<** "Spell: "**<<** name **<<** endl**;**

cout **<<** "Spell cost: " **<<** cost **<<** endl**;**

**}**

void Monster**::**addBehavior**(**bool attack**)**

**{**

/\*Behavior.canAttack = attack;\*/

**if(**attack**)**

**{**

cout **<<** "Monster is aggressive" **<<** endl**;**

**}**

**else**

**{**

cout **<<** "Monster is not aggressive" **<<** endl**;**

**}**

**}**

void Monster**::**addText**(**string text**)**

**{**

/\*Text.description = text;\*/

cout **<<** text **<<** endl **<<** endl**;**

**}**

***Source Code* 5.8** Implementasi Monster.cc dalam struktur *code* baru

Dalam perumpamaan ini, implementasi setiap fungsi direpresentasikan dengan *output* berupa teks. Nilai-nilai parameter fungsi yang diproses masih berupa variabel kosong. Variabel-variabel tersebut akan diubah berdasarkan nilai-nilai yang ditambahkan oleh *class-class* yang berisi implementasi monster-monster yang ada dalam *game*. Untuk eksperimen ini, digunakan tiga buah *class* Monster yang mewakilkan monster-monster dalam *game* yaitu *class* Dragon, Serpent, dan Ghost. Dalam *class* yang berisi representasi monster, terdapat sebuah *method* yang menjalankan fungsi-fungsi dari *class* Monster, namun dengan parameter fungsi yang sesuai dengan masing-masing monster. Untuk menjalankan fungsi-fungsi tersebut, *class* yang merepresentasikan sebuah monster perlu men-*extend* *class* Monster.

### **5.2.3 Dragon.h**

#include "Monster.h"

class Dragon **:** protected Monster

**{**

protected**:**

void add**()**

**{**

Monster**::**newMonster**(**"Dragon"**);**

Monster**::**createSpell**(**"Fire"**,**10**);**

Monster**::**addBehavior**(true);**

Monster**::**addText**(**"A flying monster that breathes

fire"**);**

**}**

**};**

***Source Code* 5.9** Implementasi Dragon.h dalam struktur *code* baru

Berkas Dragon.h berisi fungsi-fungsi yang mengatur komponen-komponen yang dimiliki oleh sebuah monster Dragon. Isi dari Dragon.h dapat dilihat pada *Source Code* 5.9.

### **5.2.4 Serpent.h**

#include "Monster.h"

class Serpent **:** protected Monster

**{**

protected**:**

void add**()**

**{**

Monster**::**newMonster**(**"Serpent"**);**

Monster**::**createSpell**(**"Poison"**,**5**);**

Monster**::**addBehavior**(true);**

Monster**::**addText**(**"A snake-like monster that spews

poison"**);**

**}**

**};**

***Source Code* 5.10** Implementasi Serpent.h dalam struktur *code* baru

Serpent.h memiliki isi yang tidak jauh berbeda dengan Dragon.h, seperti yang dapat dilihat pada *Source Code* 5.10. Fungsi-fungsi yang dimiliki oleh Serpent.h pada dasarnya sama dengan fungsi yang dimiliki oleh Dragon.h, hanya saja *value* yang diproses berbeda.

### **5.2.5 Ghost.h**

Isi dari Ghost.h memiliki bentuk yang sama dengan Dragon.h dan Serpent.h seperti yang dapat dilihat pada *source code* berikut:

#include "Monster.h"

class Ghost **:** protected Monster

**{**

protected**:**

void add**()**

**{**

Monster**::**newMonster**(**"Ghost"**);**

Monster**::**createSpell**(**"Curse"**,**20**);**

Monster**::**addBehavior**(true);**

Monster**::**addText**(**"An astral creature wandering

around"**);**

**}**

**};**

***Source Code* 5.11** Implementasi Ghost.h dalam struktur *code* baru

### **5.2.6 Main.cc**

Fungsi-fungsi di atas akan dijalankan oleh berkas Main.cc. Main.cc menjalankan sebuah *method* yang menjalankan *method-method* yang memanggil fungsi-fungsi dari *class* Monster. Isi dari Main.cc dapat dilihat pada *source code* berikut:

#include "Serpent.h"

#include "Dragon.h"

#include "Ghost.h"

class Print **:** protected Dragon**,** protected Serpent**,** protected Ghost

**{**

public**:**

void create**()**

**{**

Dragon**::**add**();**

Serpent**::**add**();**

Ghost**::**add**();**

**}**

**};**

int main**()**

**{**

Print print**;**

print**.**create**();**

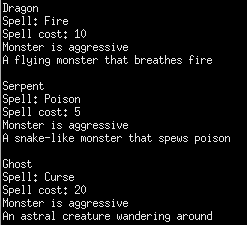
**return** 0**;**

**}**

***Source Code* 5.12** Implementasi Main.cc dalam struktur *code* baru

### **5.2.7** ***Output* dari struktur *code* baru**

Setelah dijalankan, *code* akan menghasilkan *output* yang sama dengan *output* dari struktur *code* yang lama:



**Gambar 5.2** *Output* teks dari rangkaian *code* dengan struktur *code* baru

## **5.3 *Refactoring* terhadap rancangan struktur *code* baru**

Setelah didapatkan bentuk *code* yang baru, dilakukan *refactoring* terhadap *code* tersebut untuk memastikan paradigma *object-oriented* benar-benar dimanfaatkan dengan baik untuk penerapan SPL. Menurut McGregor (2004), *Software Product Line* dapat diterapkan dengan optimal pada *code* berbasis obyek yang memiliki *inheritance* dan *polymorphism*.

Untuk menunjang penerapan *inheritance* dan *polymorphism*, dibuat sistem *interface* yang dapat di-*extend* oleh *class-class* yang membutuhkan. Bentuk penerapan *inheritance* dapat dilihat pada pemanggilan *method* yang dilakukan *class-class­* Dragon, Serpent, dan Ghost terhadap *method-method* dari *class* Monster, sedangkan *polymorphism* belum diterapkan dalam *code* tersebut.

### **5.3.1 *Interface***

*Interface* diterapkan dengan membuat *header file* dari *class* Monster yang berisi deklarasi dari fungsi-fungsi yang dapat di-*inherit* oleh *class-class* monster lain.

### **5.3.2 *Inheritance***

*Inheritance* diterapkan melalui *extension* yang dilakukan oleh *class-class* monster-monster terhadap *class* Monster. *Class-class* seperti Dragon, Serpent, dan Ghost dapat meng-*inherit* fungsi-fungsi yang terdapat pada *class* Monster untuk dijalankan dengan spesifikasi monster masing-masing.

### **5.3.3 *Polymorphism***

*Polymorphism* belum diterapkan dalam struktur *code* baru tersebut. Penerapan *Polymorphism* dapat dimodelkan dengan mengubah bentuk pemanggilan *method* struktur *code* yang baru. Dengan dilakukannya penerapan *polymorphism*, maka *code* yang baru akan lebih mengakomodasi implementasi fungsi-fungsi pada *source code game* yang memiliki banyak perbedaan, sesuai dengan pernyataan Fowler bahwa penerapan *polymorphism* dapat membantu menghindari penggunaan *conditional* untuk fungsi-fungsi yang memiliki implementasi berbeda. Contoh penggunaan *conditional* dalam *source code* asli yang dapat diubah dengan *polymorphism* dapat dilihat pada *Source Code* 5.13.

**switch** **(**spell\_cast**)**

**{**

**default:**

**break;**

**...**

**case** SPELL\_SPRINT**:**

mons**->**add\_ench**(**ENCH\_SWIFT**);**

simple\_monster\_message**(\***mons**,** “ puts on a burst of speed!”**);**

**return;**

**case** SPELL\_SILENCE**:**

mons**->**add\_ench**(**ENCH\_SILENCE**);**

invalidate\_agrid**(true);**

simple\_monster\_message**(\***mons**,** “’s surroundings become eerily quiet.”**);**

**return;**

**case** SPELL\_CALL\_TIDE**:**

**if** **(**player\_in\_branch**(**BRANCH\_SHOALS**))**

**{**

const int tide\_duration **=** BASELINE\_DELAY

**\*** random\_range**(**80**,** 200**,** 2**);**

mons**->**add\_ench**(**mon\_enchant**(**ENCH\_TIDE**,** 0**,** mons**,**

tide\_duration**));**

mons**->**props**[**TIDE\_CALL\_TURN**].**get\_int**()** **=** you**.**num\_turns**;**

**if** **(**simple\_monster\_message**(\***

mons**,**

“ sings a water chant to call the tide!”**))**

**{**

flash\_view\_delay**(**UA\_MONSTER**,** ETC\_WATER**,** 300**);**

**}**

**}**

**return;**

…

**Source Code 5.13** Potongan *code* dari *source code* *game* yang menggunakan *conditional*

*Conditional* tersebut merupakan bentuk implementasi dari *spell-spell* yang dimiliki oleh monster-monster. *Conditional* tersebut bekerja dengan cara mengecek *spell* apa yang dijalankan oleh monster yang menggunakannya, lalu menjalankan fungsi implementasi dari *spell* tersebut.

Setelah struktur *code* yang lama diubah menjadi struktur *code* baru yang membagi fungsi berdasarkan monster yang menggunakannya, maka masing-masing *case* dari *conditional* tersebut dapat dipecah menjadi *method-method* yang dijalankan oleh *class* monster masing-masing yang menjalankannya, dengan bentuk implementasi fungsi yang berbeda-beda.

5.3.3.1 Monster.h

Berkas Monster.h berisi deklarasi variabel-variabel yang nantinya dapat diturunkan ke *class-class* monster-monster yang meng-*extend* *class* Monster. Variabel-variabel tersebut akan berisi representasi komponen-komponen monster yang akan di-*print* seperti pada contoh struktur *code* di subbab sebelumnya, seperti yang dapat dilihat pada *Source Code* 5.14.

#include <iostream>

#include <string>

**using** **namespace** std**;**

class Monster

**{**

public**:**

string name**;**

string spellName**;**

int spellCost**;**

int spellDuration**;**

int spellDamage**;**

int spellSuccess**;**

bool behavior**;**

string description**;**

virtual void monsterName**(**string name**);**

virtual void monsterSpell**(**string spellName**,** int spellCost**,** int spellDuration**,** int spellDamage**,** int spellSuccess**);**

virtual void monsterBehavior**(**bool behavior**);**

virtual void monsterDescription**(**string description**);**

**};**

***Source Code* 5.14** Implementasi Monster.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

Fungsi-fungsi yang dideklarasikan menggunakan *virtual function* agar dapat di-*polymorph* oleh *class-class* yang menjadi *child* dari *class* Monster. Fungsi monsterSpell memiliki banyak parameter untuk mengakomodasi kebutuhan-kebutuhan masing-masing monster yang implementasi fungsinya membutuhkan parameter yang berbeda.

5.3.3.2 Monster.cc

Implementasi yang terdapat pada Monster.cc berisi fungsi cout yang mencetak *placeholder* yang nantinya akan diubah oleh *class-class* lain. Parameter dari fungsi-fungsi tersebut tidak digunakan dalam implementasi ini karena memang fungsi-fungsi tersebut tidak digunakan. Isi dari Monster.cc dapat dilihat pada *source code* berikut:

#include "Monster.h"

void Monster**::**monsterName**(**string name**)**

**{**

cout **<<** "Monster Name" **<<** endl**;**

**}**

void Monster**::**monsterSpell**(**string spellName**,** int spellCost**,** int spellDuration**,** int spellDamage**,** int spellSuccess**)**

**{**

cout **<<** "Spell Name" **<<** endl**;**

cout **<<** "Spell Cost" **<<** endl**;**

**}**

void Monster**::**monsterBehavior**(**bool behavior**)**

**{**

cout **<<** "Monster Behavior" **<<** endl**;**

**}**

void Monster**::**monsterDescription**(**string description**)**

**{**

cout **<<** "Monster Description" **<<** endl **<<** endl**;**

**}**

***Source Code* 5.15** Implementasi Monster.cc yang mengakomodasi *polymorphism* dalam struktur *code* baru

5.3.3.3 Dragon.h

*Class* Dragon dibuat menjadi berkas tunggal yang merupakan sebuah *header file* tanpa berkas implementasi dengan format .cc atau .cpp untuk mempermudah penjelasan. Isi dari Dragon.h dapat dilihat pada *source code* berikut:

#include "Monster.h"

class Dragon **:** public Monster

**{**

protected**:**

void createDragon**()**

**{**

**this->**name **=** "Dragon"**;**

**this->**spellName **=** "Fire"**;**

**this->**spellCost **=** 10**;**

**this->**spellDuration **=** 5**;**

**this->**behavior **=** **true;**

**this->**description **=** "A flying monster that breathes fire"**;**

**}**

void monsterName**(**string name**)**

**{**

cout **<<** name **<<** endl**;**

**}**

void monsterSpell**(**string spellName**,** int spellCost**,** int spellDuration**,** int spellDamage**,** int spellSuccess**)**

**{**

cout **<<** "Spell Name: " **<<** spellName **<<** endl**;**

cout **<<** "Spell Cost: " **<<** spellCost **<<** endl**;**

cout **<<** "The spell lasts for " **<<** spellDuration **<<** " seconds" **<<** endl**;**

**}**

void monsterBehavior**(**bool behavior**)**

**{**

**if(**behavior**)**

**{**

cout **<<** "Monster is Aggressive" **<<** endl**;**

**}**

**else**

**{**

cout **<<** "Monster is not Aggressive" **<<** endl**;**

**}**

**}**

void monsterDescription**(**string description**)**

**{**

cout **<<** description **<<** endl **<<** endl**;**

**}**

void add**()**

**{**

createDragon**();**

monsterName**(**name**);**

monsterSpell**(**spellName**,** spellCost**,** spellDuration**,** spellDamage**,** spellSuccess**);**

monsterBehavior**(**behavior**);**

monsterDescription**(**description**);**

**}**

**};**

***Source Code* 5.16** Implementasi Dragon.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

Perubahan utama yang dilakukan dalam berkas Dragon,h selain pemanfaatan *inheritance* adalah *polymorphism* yang dilakukan terhadap fungsi-fungsi dari *class* Monster. Fungsi-fungsi tersebut diubah agar dapat menghasilkan *output* sesuai dengan komponen-komponen yang dimiliki oleh monster Dragon.

5.3.3.4 Serpent.h

Secara umum, implementasi dari Serpent.h memiliki bentuk yang sama dengan Dragon.h. Hal yang membedakan Serpent.h dan Dragon.h adalah isi dari *method* monsterSpell yang menggunakan parameter berbeda dan *template* cetakan *output* yang berbeda. Perbedaan tersebut dimaksudkan untuk memodelkan kasus di mana *polymorphism* memiliki manfaat signifikan, karena pada *source code game* yang menggunakan struktur *code* lama, bentuk-bentuk *conditional* dan pengelompokan fungsi berdasarkan kegunaan dapat diperbaiki dengan dipecah menjadi fungsi-fungsi *polymorphism*. Isi dari Serpent.h dapat dilihat pada *source code* berikut:

#include "Monster.h"

class Serpent **:** protected Monster

**{**

protected**:**

void createSerpent**()**

**{**

**this->**name **=** "Serpent"**;**

**this->**spellName **=** "Poison"**;**

**this->**spellCost **=** 5**;**

**this->**spellSuccess **=** 85**;**

**this->**behavior **=** **true;**

**this->**description **=** "A snake-like monster that spews poison"**;**

**}**

void monsterName**(**string name**)**

**{**

cout **<<** name **<<** endl**;**

**}**

void monsterSpell**(**string spellName**,** int spellCost**,** int spellDuration**,** int spellDamage**,** int spellSuccess**)**

**{**

cout **<<** "Spell Name: " **<<** spellName **<<** endl**;**

cout **<<** "Spell Cost: " **<<** spellCost **<<** endl**;**

cout **<<** "The spell succeeds " **<<** spellSuccess **<<** "% of the time" **<<** endl**;**

**}**

void monsterBehavior**(**bool behavior**)**

**{**

**if(**behavior**)**

**{**

cout **<<** "Monster is Aggressive" **<<** endl**;**

}

**else**

**{**

cout **<<** "Monster is not Aggressive" **<<** endl**;**

**}**

**}**

void monsterDescription**(**string description**)**

**{**

cout **<<** description **<<** endl **<<** endl**;**

**}**

void add**()**

**{**

createSerpent**();**

monsterName**(**name**);**

monsterSpell**(**spellName**,** spellCost**,** spellDuration**,** spellDamage**,** spellSuccess**);**

monsterBehavior**(**behavior**);**

monsterDescription**(**description**);**

**}**

**};**

***Source Code* 5.17** Implementasi Serpent.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

5.3.3.5 Ghost.h

Berkas Ghost.h memiliki kesamaan dan perbedaan yang sama dengan Serpent.h dan Dragon.h. Isi dari Ghost.h dapat dilihat pada *source code* berikut:

#include "Monster.h"

class Ghost **:** protected Monster

**{**

protected**:**

void createGhost**()**

**{**

**this->**name **=** "Ghost"**;**

**this->**spellName **=** "Curse"**;**

**this->**spellCost **=** 20**;**

**this->**spellDamage **=** 50**;**

**this->**behavior **=** **false;**

**this->**description **=** "An astral creature wandering around"**;**

**}**

void monsterName**(**string name**)**

**{**

cout **<<** name **<<** endl**;**

**}**

void monsterSpell**(**string spellName**,** int spellCost**,** int spellDuration**,** int spellDamage**,** int spellSuccess**)**

**{**

cout **<<** "Spell Name: " **<<** spellName **<<** endl**;**

cout **<<** "Spell Cost: " **<<** spellCost **<<** endl**;**

cout **<<** "The spell inflicts " **<<** spellDamage **<<** " damage" **<<** endl**;**

**}**

void monsterBehavior**(**bool behavior**)**

**{**

**if(**behavior**)**

**{**

cout **<<** "Monster is Aggressive" **<<** endl**;**

**}**

**else**

**{**

cout **<<** "Monster is not Aggressive" **<<** endl**;**

**}**

**}**

void monsterDescription**(**string description**)**

**{**

cout **<<** description **<<** endl **<<** endl**;**

**}**

void add**()**

**{**

createGhost**();**

monsterName**(**name**);**

monsterSpell**(**spellName**,** spellCost**,** spellDuration**,** spellDamage**,** spellSuccess**);**

monsterBehavior**(**behavior**);**

monsterDescription**(**description**);**

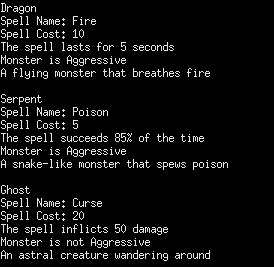
**}**

**};**

***Source Code* 5.18** Implementasi Ghost.h yang mengakomodasi *polymorphism* dalam struktur *code* baru

5.3.3.5 *Output* dari struktur *code* dengan *polymorphism*

Rangkaian *code* di atas akan menghasilkan output berupa teks sesuai dengan fungsi-fungsi yang diimplementasi pada *class-class* Dragon, Serpent, dan Ghost, berbeda dengan output yang dikeluarkan oleh fungsi asli pada *class* Monster, seperti yang dapat dilihat pada Gambar 5.3.



**Gambar 5.3** *Output* teks dari rangkaian *code* yang telah menerapkan *polymorphism*

## **5.4 *Refactoring* terhadap *source code* asli**

Pada subbab ini, dilakukan pengubahan langsung terhadap *source code* asli *game* sesuai dengan rancangan yang telah dipaparkan pada bab sebelumnya.

### **5.4.1 Sistem perhitungan skor**

Mekanisme perhitungan skor, seperti yang telah dijelaskan dalam Bab 3, terdiri dari berbagai faktor yang tersebar dalam berbagai fungsi, *class*, dan berkas. Untuk menghindari pengubahan pada arsitektur dasar *source code* serta menjaga agar ruang lingkup penelitian tetap terjangkau, eksperimen dilakukan pada bagian-bagian *code* yang murni mengatur *behavior* dari sebuah fungsi tanpa menyangkut data dan konfigurasi, atau membutuhkan pengubahan pada struktur fungsi, *class*, atau berkas lain yang memiliki relevansi rendah dengan sistem perhitungan skor.

Dalam penelitian ini, eksperimen akan dilakukan terhadap fungsi-fungsi yang menghitung *experience* dari pemain, serta poin yang dimiliki oleh pemain berdasarkan jumlah *rune* dan status kemenangan. Fungsi-fungsi tersebut berada dalam berkas hiscores.cc yang secara umum berfungsi untuk menampilkan skor pemain saat permainan selesai, serta menampilkan daftar *high score* setelahnya.

Berikut merupakan langkah-langkah *refactor* dari sistem perhitungan skor dari struktur yang lama menjadi struktur baru yang lebih menerapkan paradigma berbasis obyek:

1. Memisahkan fungsi-fungsi penghitung skor

Langkah ini dilakukan dengan cara memindahkan *method* \_award\_modified\_experience() ke dalam sebuah berkas baru bernama score.h, lalu memindahkan potongan fungsi dari *method* scorefile\_entry::init() yang berfungsi sebagai penghitung poin ke dalam berkas score.h dan mengubahnya sebagai sebuah *method* baru bernama calculate\_score() yang dipanggil oleh scorefile\_entry::init().

1. Membuat *class* baru

Setelah dipisahkan dari berkas awal, *method-method* tersebut dimasukkan ke dalam sebuah *class* baru yang merepresentasikan perhitungan skor. Dalam eksperimen ini, dibuat sebuah *class* bernama Score dalam berkas score.h yang mengakomodasi kedua *method* yang telah dipisahkan tersebut. Agar fungsi-fungsi yang telah dipindahkan ke dalam *class* Score dapat dipanggil oleh fungsi lain di hiscores.cc, maka pemanggilan fungsi yang dilakukan dalam berkas hiscores.cc diubah dari yang semula calculate\_score() dengan parameternya menjadi Score::calculate\_score() dengan parameternya.

1. Memisahkan *class header* dengan implementasi

Untuk dapat lebih meningkatkan pemanfaatan paradigma berbasis obyek, *class* Score dibagi menjadi dua buah berkas, yaitu berkas *header* dan berkas implementasi. *Header* dari Score yang berisi deklarasi fungsi disimpan dalam berkas score.h, sedangkan bagian implementasi dari fungsi-fungsi tersebut ditempatkan dalam berkas score.cc.

### **5.4.2 Sistem pengelolaan *experience***

Secara umum, e*xperience* merupakan poin yang didapat oleh pemain setiap kali pemain berhasil melakukan sesuatu dalam *game* seperti naik level pemain, naik level *skill*, atau membunuh monster. Dalam *game* Dungeon Crawl Stone Soup, *experience* juga dapat diperoleh oleh karakter-karakter musuh seperti monster. Pada *source code*, terdapat enam buah fungsi yang mengelola *experience* yang didapat oleh pemain dan musuh. Fungsi-fungsi tersebut terdapat pada berkas mon-death.cc yang secara umum mengelola proses kematian monster, termasuk menghitung *experience* yang didapat oleh pemain setelah membunuh monster.

Untuk menjaga agar ruang lingkup penelitian ini tidak melebar di luar jangkauan, maka fungsi-fungsi yang berhubungan dengan data dan konfigurasi nilai *experience* monster tidak di-*refactor*.

Berikut merupakan langkah-langkah *refactor* dari sistem pengelolaan *experience* dari struktur yang lama menjadi struktur baru yang lebih menerapkan paradigma berbasis obyek:

1. Memisahkan fungsi-fungsi pengelola *experience*.

Dari berkas mon-death.cc, diambil *method-method* yang mengelola *experience* yaitu:

* \_calc\_monster\_experience()
* \_give\_monster\_experience()
* \_beogh\_spread\_experience()
* \_calc\_player\_experience()
* \_give\_player\_experience()
* \_give\_experience()

Fungsi-fungsi tersebut dipindahkan ke sebuah berkas baru bernama experience.h yang di-include oleh mon-death.cc untuk memastikan bahwa fungsi-fungsi tersebut dapat dipanggil oleh fungsi lain di mon-death.cc.

1. Membuat *class* baru

Di dalam berkas experience.h dibuat sebuah *class* baru bernama Experience yang mengakomodasi fungsi-fungsi pengelola *experience* yang telah dipindahkan. Setelah fungsi-fungsi pengelola *experience* dimasukkan ke dalam *class* Experience, dilakukan pengubahan terhadap pemanggilan fungsi-fungsi tersebut yang dilakukan di berkas mon-death.cc dengan cara memberikan pemanggilan berdasarkan *class* seperti pada *refactoring* sistem skor, contohnya Experience::\_calc\_monster\_experience().

1. Memisahkan *class header* dengan implementasi

Pada kondisi sekarang, seluruh implementasi fungsi pengelolaan *experience* berada dalam satu berkas *header*. Seperti pada *refactoring* sistem skor, implementasi fungsi-fungsi tersebut akan dipisah ke dalam sebuah berkas lain bernama experience.cc, sehingga experience.h hanya akan berisi deklarasi fungsi.

### **5.4.3 Dream Sheep**

Setelah melakukan *refactoring* pada sistem perhitungan skor dan sistem pengelolaan *experience*, cara yang serupa dipraktikkan pada Dream Sheep untuk mengubah paradigma strukur *code* monster menjadi lebih berbasis obyek.

Seperti yang telah dijabarkan pada Bab 3, proses penambahan Dream Sheep terdiri dari pengubahan banyak berkas pada *source code*. Meski demikian, sebagian besar pengubahan dan penambahan tersebut merupakan *code-code* yang merepresentasikan data konfigurasi. Untuk menjaga ruang lingkup penelitian dan menghindari pengubahan pada data konfigurasi, maka pengubahan hanya dilakukan pada fungsi-fungsi yang murni hanya bertanggungjawab atas *behavior*.

1. Memisahkan fungsi-fungsi *spell* Dream Sheep

Fungsi-fungsi yang bertanggungjawab atas implementasi *spell* Dream Dust yang dimiliki oleh Dream Sheep terdapat dalam berkas mon-cast.cc yang secara umum berisi implementasi *spell-spell* lain yang dimiliki oleh monster-monster lain. Dream Dust diimplementasi oleh dua buah method bernama \_dream\_sheep\_sleep() dan \_sheep\_message(), serta sebuah variabel *int* bernama MIN\_DREAM\_SUCCESS\_POWER. Menggunakan cara yang sama dengan dua percobaan *refactoring* sebelumnya, fungsi-fungsi tersebut dipindahkan ke dalam sebuah berkas baru bernama dream\_sheep.h yang di-*include* oleh mon-cast.cc.

1. Membuat *class* baru

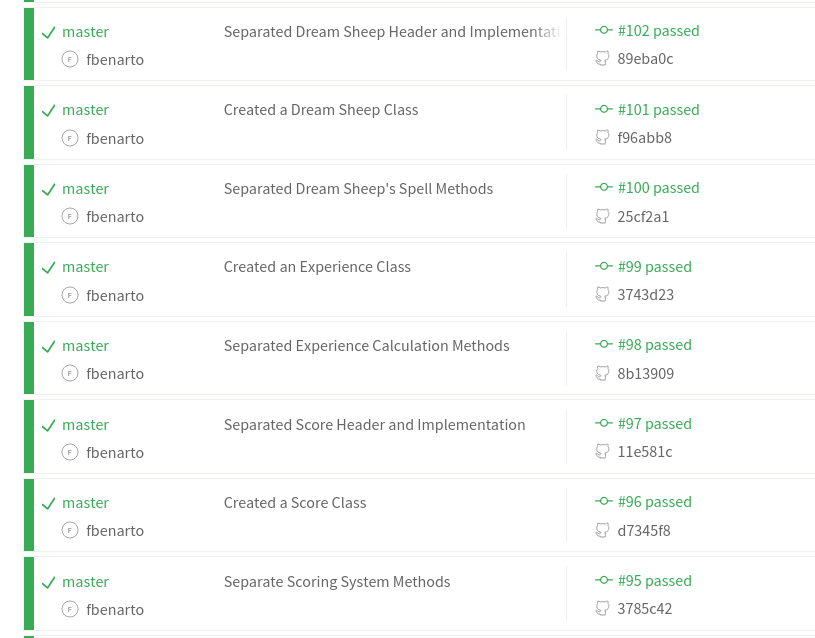
Langkah berikutnya merupakan pembuatan sebuah *class* baru sebagai representasi obyek Dream Sheep yang berisi fungsi-fungsi yang telah dipindahkan ke dalam dream\_sheep.h. Agar fungsi-fungsi yang terbungkus *class* baru tersebut dapat diakses oleh fungsi lain dari luar *class* dan berkas, maka pemanggilan fungsi-fungsi tersebut diubah agar menyertakan nama *class*, yaitu DreamSheep, sehingga pemanggilan \_dream\_sheep\_sleep() yang dilakukan dari berkas mon-cast.cc diubah menjadi DreamSheep::\_dream\_sheep\_sleep() beserta parameternya.

1. Memisahkan *class header* dengan implementasi

Setelah *class* DreamSheep selesai dibentuk, isi dari *class* tersebut dipisah menjadi dua bagian, yaitu *header* dan implementasi. Deklarasi fungsi yang berada di *header* akan tetap berada di berkas dream\_sheep.h, sedangkan implementasi dari fungsi-fungsinya ditempatkan dalam berkas dream\_sheep.cc.

### **5.4.4 Pengaruh perubahan terhadap *source code***

Untuk memastikan bahwa pengubahan yang dilakukan terhadap *source code* dapat berjalan dengan baik, repository dari *source code* dihubungkan dengan Travis-CI agar setiap *build* dapat dites. Bukti bahwa perubahan-perubahan yang dilakukan dapat di-*compile* dengan baik oleh *source code* dan tidak merusak struktur *code* secara keseluruhan dapat dilihat pada Gambar 5.4 yang merupakan hasil tes pada *build-build* pengubahan pada subbab sebelumnya.



**Gambar 5.4** Status *build* dari pengubahan yang dilakukan

Selain *build testing* menggunakan Travis-CI, dibuat pula perintah untuk memberikan *output* pada *terminal* setiap kali fungsi yang diubah dijalankan untuk memastikan bahwa fungsi tersebut tetap bekerja.

## **5.5 Menghasilkan variasi produk *game***

Produk *game* baru berupa *game* yang memiliki variasi Dream Sheep berbeda dibuat dengan cara membuat sebuah berkas baru yang berisi *class* NewSheep yang merupakan *subclass* dari Dream Sheep. *Class* tersebut men-*inherit* fungsi-fungsi yang dimiliki oleh Dream Sheep, tetapi memiliki karakteristik atau *behavior* fungsi yang berbeda. Bentuk implementasi dari NewSheep dapat dilihat pada ***Source code* 5.19** berikut:

class NewSheep **:** public Dream\_Sheep

**{**

public**:**

static const int MIN\_DREAM\_SUCCESS\_POWER**;**

static void \_sheep\_message**(**int num\_sheep**,** int sleep\_pow**,** actor**&** foe**);**

static void \_dream\_sheep\_sleep**(**monster**&** mons**,** actor**&** foe**);**

**};**

**Source code 5.19** *Class* NewSheep

Pengubahan *behavior* dari fungsi-fungsi atau variabel yang dimiliki oleh New Sheep agar berbeda dengan Dream Sheep dapat diimplementasikan dengan *polymorphism* seperti yang telah dimodelkan sebelumnya.

Dengan membuat sebuah *build* yang memiliki konfigurasi *compile* yang berbeda, maka dapat dihasilkan produk berupa *game* yang memiliki Sheep berbeda. Sebagai contoh, untuk *build* menggunakan perintah ‘make’ dengan konfigurasi *default*, maka program yang akan dihasilkan adalah *game* Dungeon Crawl Stone Soup dengan monster Dream Sheep, sedangkan untuk *build* dengan konfigurasi ‘new’, maka program yang akan dihasilkan adalah *game* Dungeon Crawl Stone Soup dengan monster New Sheep.

**BAB 6**

# **ANALISIS**

Dari ekseperimen yang telah dilakukan, terdapat beberapa hal yang dapat dibahas berdasarkan hasil eksperimen.

## **6.1 Tingkat *Coupling***

Struktur *code* yang baru akan memungkinkan *contributor* untuk melakukan pengubahan (dalam konteks penelitian ini penambahan) karena *contributor* hanya perlu mengubah satu buah *class* untuk setiap monster yang ingin diubah. Pada struktur *code* ini, semua fungsi yang berhubungan dengan sebuah monster dikelompokkan dalam sebuah *class*, dibandingkan dengan struktur *code* lama di mana *contributor* perlu mengubah semua berkas *code* yang mengandung fungsi yang ingin diubah.

Selain itu, *contributor* tidak perlu saling menunggu untuk membuat perubahan pada *code* ketika ingin menambahkan dua hal yang berbeda. Struktur *code* yang lama mengharuskan dua orang *contributor* yang ingin menambahkan dua monster yang berbeda untuk mengubah beberapa berkas yang sama. Jika tidak dikelola dengan *source control* yang baik, hal ini dapat menyebabkan banyak *conflict* perubahan *code*. Dengan menerapkan struktur *code* yang baru, dua orang *contributor* yang ingin menambahkan dua buah monster baru yang berbeda, secara garis besar hanya perlu berinteraksi dengan *class* baru masing-masing, dengan jumlah kemungkinan *conflict* yang sangat minimal seperti di berkas Main.cc.

## **6.2 Tingkat *Cohesion***

Meskipun pengelompokan fungsi diubah dari yang secara definisi dan kegunaan mirip menjadi dikelompokkan berdasarkan obyek yang menggunakan fungsi tersebut, namun *cohesion* dari *code* justru bertambah. Hal ini disebabkan struktur *code* yang lama meskipun fungsi-fungsi dikelompokkan berdasarkan definisi dan kegunaan, secara teknis fungsi-fungsi tersebut sangat berbeda mulai dari tipe *method*, tipe parameter, jumlah parameter, serta hasil *return*. Jurus dari sebuah monster dengan jurus monster lainnya, meskipun keduanya memiliki kegunaan yang sama (cara monster bertarung), kadang memiliki mekanika yang jauh berbeda, mulai dari nilai-nilai *input* yang diproses, nilai-nilai fungsi lain yang diubah, hingga pengaruh terhadap *code* lain.

Dengan menerapkan struktur *code* yang baru, fungsi-fungsi yang semula dikelompokkan dengan fungsi lain yang secara teknis berbeda akan dikelompokkan dengan fungsi-fungsi lain yang walaupun memiliki tugas yang berbeda, tetapi memiliki satu kesamaan dasar, yaitu berhubungan/memengaruhi monster yang menjadi *class* induk dari fungsi-fungsi tersebut.

## **6.3 *Development Effort***

Berdasarkan rancangan struktur *code* monster yang dijabarkan pada Bab 4 dan Bab 5, dengan membuat representasi *class* per monster, *effort* atau usaha yang dilakukan dalam implementasi *code* dilihat dari jumlah berkas yang dibuat menjadi relatif berdasarkan jumlah monster yang diimplementasi. Pada struktur *code* yang baru, seorang *contributor* hanya perlu membuat sebuah *class* baru, dan menambahkan sedikit penyesuaian untuk keperluan *inheritance* pada berkas yang lebih sedikit.

Di sisi lain, pada struktur *code* lama, sebanyak apapun monster yang diimplementasi, jumlah berkas yang dibuat tetap 9 berkas. Sedangkan pada *code* yang baru, berkas dibuat berdasarkan monster yang ditambahkan, sehingga setiap ada monster yang ditambahkan ke dalam *game*, maka akan dibuat sebuah berkas baru. Dengan struktur tersebut, maka jumlah berkas yang dibuat adalah 4 berkas *code* teknis ditambah dengan jumlah monster.

Meski jumlah berkas *code* pada struktur baru akan terus bertambah, jumlah *line of code* tidak akan mengalami perubahan yang signifikan, karena yang diubah hanyalah pembagian berkas *code* tempat masing-masing fungsi dikelompokkan.

Dengan membuat struktur *code* yang jumlah baris *code* per berkasnya statis, maka seorang *contributor* yang akan menambahkan sebuah monster baru akan lebih dimudahkan. Struktur *code* yang baru tidak mengharuskan seorang *contributor* untuk berinteraksi dengan banyak berkas *code* dengan jumlah baris *code* yang sangat banyak yang berpotensi membingungkan *contributor*.

Penerapan SPL pada *source code* ini akan memudahkan *contributor* yang akan membuat sebuah monster baru yang merupakan sebuah variasi dari monster yang sudah ada. *Contributor* tersebut tidak perlu membuat kembali fungsi-fungsi yang sama untuk monster tersebut, tetapi cukup men-*inherit* *class* monster yang merupakan *superclass* dari monster baru tersebut.

Meski demikian, pada kenyataannya prediksi tersebut tidak dapat direalisasikan dalam penelitian ini, karena dari pengubahan yang telah dilakukan pada Dream Sheep, hanya fungsi-fungsi dari berkas mon-cast.cc saja yang dapat dipindahkan ke dalam sebuah *class* dan berkas baru. Selain itu, implementasi *class* dalam bahasa pemrograman C++ yang baik membutuhkan dua buah berkas, yaitu berkas *header* berupa *file* .h dan berkas implementasi berupa *file* .cc atau .cpp. Jika setiap *class* monster membutuhkan dua buah berkas, maka jumlah keseluruhan berkas dalam *source code* akan bertambah dua untuk setiap monster yang dibuat, kecuali untuk kasus-kasus penurunan monster tertentu yang hanya perlu men-*inherit* *header* dari monster *superclass*-nya.

**BAB 7**

# **PENUTUP**

## **7.1 Kesimpulan**

Berdasarkan penelitian yang telah dilakukan, dapat disimpulkan bahwa SPL dapat diterapkan pada pengembangan *role-playing game* dengan memanfaatkan paradigma berbasis obyek dan mengeksploitasi *commonality* serta *variability* dari *game* tersebut. Dengan menerapkan SPL, *developer* dari *game* dapat lebih mudah membuat perubahan dalam *game* dan membuat *game* yang berbeda dengan *effort* yang minimal. Selain itu, pengubahan struktur *code* dari *game* yang semula menerapkan paradigma berbasis obyek dengan membagi *code* berdasarkan kesamaan kegunaan fungsi menjadi berdasarkan obyek yang menggunakan fungsi tersebut, menghasilkan sebuah struktur *code* yang memiliki tingkat *reusability* lebih tinggi, dinilai dari tingginya *cohesion* dan rendahnya *coupling*.

## **7.2 Tantangan dalam pengerjaan penelitian**

Dalam pengerjaannya, penelitian ini menemui beberapa kendala. Beberapa di antaranya adalah:

1. Mencari *game* dan contoh kasus yang sesuai dengan topik penelitian dengan ketersediaan *source code* yang memadai.
2. Struktur *code* asli dari *game* yang kurang teratur dan terlalu banyak *dependency* dengan berkas lain.
3. Pemodelan struktur *code* lama dari *source code* dan *code* baru yang dirancang untuk menggantikan struktur lama tersebut.

## **7.3 Saran**

Berdasarkan penelitian yang telah dilakukan serta kesimpulan-kesimpulan yang telah diambil dari penelitian tersebut, terdapat beberapa pertimbangan yang dapat menjadi saran bagi penelitian-penelitian berikutnya terkait topik penerapan SPL dalam pengembangan *game*:

1. Pastikan obyek studi kasus yang menjadi bahan penelitian memiliki ruang lingkup yang sesuai dengan rencana awal penelitian. Pada penelitian ini, ditemukan masalah ketika bekerja dengan *code* asli yang seharusnya hanya melingkupi berkas-berkas terkait penambahan monster baru, namun keterkaitan antara berkas-berkas tersebut dengan berkas lain dalam *source code* membuat eksperimen perlu dilakukan pada lingkup *source code* yang jauh lebih besar.
2. Dari hasil eksperimen yang telah dilakukan, dapat dilakukan penelitian dan eksperimen lebih lanjut terhadap metode penerapan SPL yang lebih efisien dan efektif dari metode pada penelitian ini, baik dengan memperbaiki metode yang telah dipaparkan, maupun mendesain metode baru dengan langkah-langkah yang berbeda.

## **7.4 *Future works***

Penelitian ini diharapkan dapat menjadi acuan atau pembuka jalan bagi penelitian-penelitian berikutnya mengenai penerapan *software product line* dalam pengembangan *game* secara umum, maupun untuk *genre* selain *role-playing game*. Mengingat lingkup penelitian ini hanya sebatas perancangan atau pemodelan pembuatan variasi produk dari sebuah *source code game*, diharapkan pada masa yang akan datang terdapat penelitian lain yang dapat meneruskan penelitian mengenai proses pembuatan variasi dengan cara yang lebih efektif dan efisien.

# **DAFTAR PUSTAKA**

1. Furtado, A. W. B., et al. (2011). Improving Digital Game Development with Software Product Lines. *IEEE Software, Vol. 28, Issue 5,* 30-37.
2. Clements, P. C., & Northrop, L. M. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001.
3. McGregor, J. D. (2004). Software Product Lines. *Journal of Object Technology, Vol. 3, No. 3*, 65-74.
4. Furtado, A. W. B. *Domain-Specific Game Development*. Universidade Federal de Pernambuco, 2012.
5. Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. Dobbe, J. *A Domain-specific Language for Computer Games*. Delft University of Technology, 2007.
7. Pohl, K., et al. *Software Product Line Engineering*. Springer, 2005
8. Beck, K. & Cunningham, W. (1989). A Laboratory for Teaching Object-Oriented Thinking. *OOPSLA’89 Conference Proceedings, Vol. 24, No. 10*.