**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 sistem yang mampu mengurangi beban kerja *programmer* 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**

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 Outline**

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 dan Simulasi

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 Kesimpulan

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***

* A software product line is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way (Clements & 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 (??)

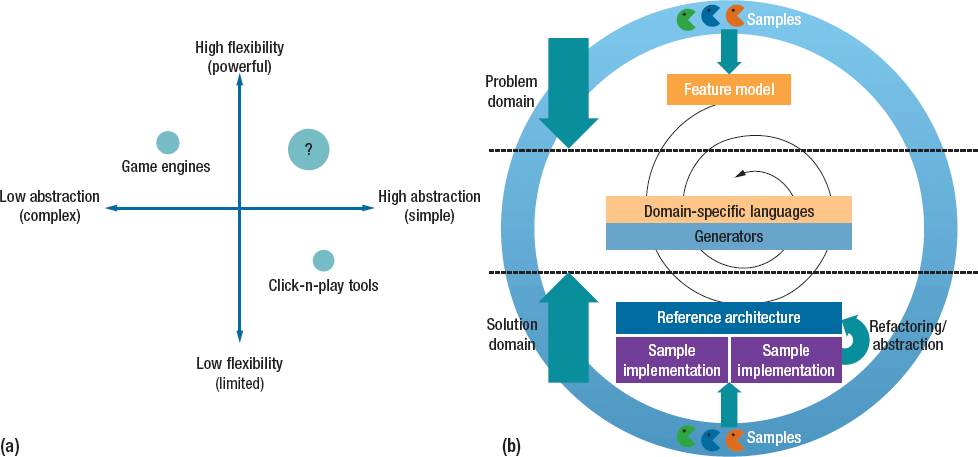
* Commonality and variability

*Software Product Line* memanfaatkan kesamaan fitur antara dua atau lebih perangkat lunak untuk menghemat *effort* yang dikeluarkan untuk mengembangkan sebuah perangkat lunak [citation needed]. Sebelum penerapan SPL, perangkat lunak yang akan dibuat *product line*-nya dianalisa terlebih dahulu mengenai *commonality* dan *variability*-nya [citation needed]. 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 [citation needed].

* Paradigm
  + 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*. (J.D. McGregor, 2004)
  + Teknik-teknik yang digunakan untuk mengembangkan program dengan paradigma berbasis obyek yang berkualitas menyediakan berbagai ???. McGregor mengatakan bahwa banyak dari *product line* yang ia pantau atau kerjakan menerapkan paradigma berbasis obyek dan menghasilkan banyak manfaat. (McGregor 2004)
* Advantage & disadvantage
* Furtado et al 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*. (Improving gamedev w/ SPL)
* 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. (Improving gamedev w/ SPL)

**2.2 Domain-Specific Language**

* Sebuah proses sistematik untuk mengeksploitasi *software product lines* untuk pengembangan *game* menawarkan *domain-specific languages* dan *generators* yang dibuat khusus untuk subdomain-subdomain *game*. (Improving gamedev w/ SPL)
* 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*.



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, DSL)

**2.3 Game Domain**

\*summary\*

Domain yang dimaksud adalah domain dalam konteks *software* secara umum, sehingga pengembangan *game* merupakan sebuah domain tersendiri.

**3.2.1 Membayangkan dan Menganalisis *Game Domain***

*Game domain* yang dibahas adalah domain dari Dungeon Crawl Stone Soup, yaitu *Role-playing game*. *Game domain* dianalisis dengan menjabarkan dan menjelaskan *game dimensions* inti yang terdapat dalam *game*. *Game dimensions* inti dari domain RPG dapat didefinisikan sebagai berikut:

1. Pemain
   1. Jumlah pemain

Sebagian besar *game* dalam domain RPG didesain untuk dimainkan oleh satu pemain, tetapi beberapa *game* memungkinkan lebih dari satu pemain bermain secara bersamaan dalam satu sesi permainan.

* 1. Mode *multiplayer*

*Game* yang menyediakan fitur *multiplayer* biasanya menyediakan mode-mode permainan tertentu untuk dimainkan oleh lebih dari satu orang.

1. Grafis
   1. Dimensi

Representasi visual dari *game* dapat berupa gambar (atau karakter ASCII untuk beberapa *game*) dua dimensi (2D) atau tiga dimensi (3D).



Gambar 3.3 Far Cry (2004) menggunakan grafis 3D

* 1. Resolusi

Untuk *game* yang menggunakan gambar dan model 3D, biasanya menggunakan gaya resolusi rendah (*pixelated*) atau resolusi tinggi (*high definition*).



Gambar 3.4 Minecraft (2010) menggunakan grafis resolusi rendah

1. Flow
   1. Level/*map*

Biasanya, permainan dalam *game* dipartisi menjadi beberapa level atau tingkatan. Sebuah level juga sering disebut sebagai *map* atau peta. Pada umumnya, setiap level atau *map* tidak saling berhubungan secara langsung dengan level dan *map* lainnya. Apa yang terjadi dalam suatu level tidak akan memengaruhi level lain. Sistem partisi dengan menggunakan level mengarahkan pemain langsung ke tujuan akhir permainan, berbeda dengan sistem *open-world* yang membiarkan pemain menjelajahi tempat manapun yang diinginkan. Hal ini membuat *game* yang menggunakan sistem level non-dinamis disebut bersifat *linear*.

* 1. World

Berbeda dengan sistem partisi *game* dengan level dan *map*, beberapa *game* justru memberikan sebuah dunia tanpa partisi yang dapat dijelajahi oleh pemain. Meskipun kadang dunia tersebut terkadang dipartisi untuk menghemat waktu dan memori saat komputer memuat aset-aset *game*, tetapi biasanya dunia-dunia yang dipartisi tersebut saling berhubungan secara dinamis.

1. Entities

*Entities* mencakup seluruh elemen benda yang terdapat dalam dunia *game* seperti senjata, *items*, musuh, serta *non-playable character*.

1. Events
   1. Player-controlled events

Terdapat *event-event* yang dipengaruhi oleh pemain. *Event-event* tersebut tidak memiliki ketentuan khusus yang ditetapkan oleh *developer*. Pemain yang menentukan terjadi atau tidaknya serta kapan *event* tersebut terjadi.

* 1. Scripted events

*Event* yang sudah dibuat oleh *developer* agar terjadi pada waktu dan batasan yang ditentukan.

1. Controller Input

*Controller input* mengatur input-input yang diterima oleh *game* melalui *keyboard*, *mouse*, ataupun *controller* lain seperti *joystick* dan *gamepad*.

1. Audio

*Audio* merupakan suara yang dihasilkan oleh *game*. Suara dapat berupa percakapan, musik, atau bunyi efek.

1. Physics

*Physics* merupakan elemen-elemen fisika yang berlaku di dalam dunia *game*, seperti gravitasi, gaya gesek, dan lain lain.

1. AI

*Artificial Intelligence* (AI) merupakan kecerdasan buatan yang menjalankan semua *entity* selain karakter pemain dalam *game*.

1. Networking
   1. Local

Beberapa *game* menyediakan fitur *multiplayer* dengan dua atau lebih *game instance* yang berjalan bersamaan dalam waktu yang sama. Biasanya, terdapat dua jenis jaringan untuk menyambungkan *game instances* tersebut. Jaringan *local* merupakan koneksi yang menghubungkan beberapa sistem dalam satu jaringan yang sama.

* 1. Online

*Online network* merupakan sistem koneksi yang menghubungkan dua atau lebih *game instance* yang tidak terhubung dalam sebuah jaringan *local*.

1. Character Progression

*Character progression* merupakan sistem yang mengatur bagaimana karakter pemain dapat berkembang. *Character progression* merupakan salah satu aspek penting yang mendefinisikan domain RPG.

1. Game Progression

*Game progression* merupakan sistem yang mengatur bagaimana *game* berjalan dan berkembang. Aspek ini berkaitan dengan aspek *flow* yang mengatur apakah sebuah *game* bersifat *linear* atau *open*. Sistem *gameprogression* secara umum menentukan bagaimana pemain dapat melaju ke tahap selanjutnya dalam permainan. Beberapa *game* seperti Assassin’s Creed memiliki *game progression* yang ditentukan oleh *character progression*, di mana cerita berjalan seiring dengan perkembangan karakter pemain.

1. World Mechanics

Aspek ini menjalankan banyak hal dalam dunia *game*, mulai dari kehidupan *non-playable character*, sistem ekonomi yang berlaku dalam dunia *game*, hingga sistem *quest* atau tugas bagi pemain.

**3.2.5 Menentukan dan Mengkaji Ulang Fitur *Game Domain***

*Game* bergenre RPG memiliki banyak fitur dan aspek. Dalam subbab ini, akan dibahas fitur-fitur apa saja dari domain RPG yang biasanya menjadi target penerapan SPL, baik fitur-fitur yang biasanya menjadi *commonality* atau memiliki *variability*. Berikut adalah fitur-fitur yang biasanya menjadi *commonality*:

1. Gameplay

*Game-game* RPG memiliki banyak perbedaan. *Game* yang berbeda dalam satu subdomainpun dapat memiliki perbedaan yang signifikan. Akan tetapi, karena secara umum subdomain-subdomain dari RPG dikelompokkan berdasarkan perbedaan *gameplay*, maka *gameplay* dari *game-game* yang terdapat dalam suatu subdomain memiliki banyak kesamaan, sehingga aspek *gameplay* cukup jarang menjadi perbedaan dalam pembuatan variasi. Sangat jarang terdapat sebuah *game* RPG yang memiliki beberapa tipe *gameplay* dari subdomain yang berbeda. Beberapa elemen dalam *gameplay* mungkin berbeda, namun inti *gameplay* seperti cara bermain, tujuan bermain, dan kondisi kemenangan biasanya tetap sama.

1. Story Theme/Emotion

Suatu *game* RPG biasanya memiliki tema atau latar cerita utama yang menjadi inti dari permainan. Meskipun dunia dan cerita dalam *game* RPG cenderung luas dan panjang, tetapi biasanya tetap dalam tema cerita dan emosi utama yang sama.

1. Character Progression

Sama seperti *gameplay*, sistem *character progression* merupakan sesuatu yang jarang diubah atau dijadikan perbedaan/variasi dalam sebuah *game*. *Character progression* sering menjadi identitas utama dari sebuah *game*, sehingga sistem tersebut dibiarkan sebagai sebuah *commonality* dalam *game*.

Sedangkan fitur-fitur yang biasanya memiliki *variablity* adalah:

1. Story Settings

Meskipun emosi dan tema utama cerita biasanya tidak menjadi perbedaan, latar cerita merupakan suatu aspek yang umum untuk dijadikan perbedaan. Perbedaan latar dapat berupa latar tempat atau waktu, tetapi tema cerita biasanya tetap sama.

1. Level/Map Design

Untuk menghindari kebosanan pemain, desain level dan dunia *game* biasanya dibuat berbeda-beda untuk setiap tahapan *progress* pemain. Level dan *map* yang berbeda-beda tersebut biasanya tetap memiliki *gameplay* dan cara bermain yang sama.

1. Character identitiy

*Game* RPG pada umumnya memiliki banyak karakter yang berbeda untuk peran yang berbeda, baik karakter yang dapat dimainkan oleh pemain, maupun karakter khusus AI yang tidak dapat dimainkan pemain atau *non-playable character* (NPC). Karena biasanya karakter-karakter tersebut memiliki tujuan dan sifat yang berbeda, maka identitas masing-masing karakterpun dibedakan. Perbedaan biasanya dilakukan berdasarkan sifat, seperti karakter baik atau karakter jahat. Beberapa karakter khusus yang memiliki tingkat kepentingan yang tinggi untuk alur cerita *game* biasanya juga dibedakan menjadi karakter-karakter yang unik.

1. Additional Features

Banyak *game* yang memiliki fitur-fitur tambahan seperti yang akan dibahas pada subbab berikutnya. Fitur-fitur tersebut biasanya merupakan *spin-off* dari fitur atau *gameplay* utama.

**2.4 Refactoring in Object-oriented paradigm**

* What to do (create responsibility, create class)
* Class-Responsibility-Collaboration Card
* How to refactor

**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:

* *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**

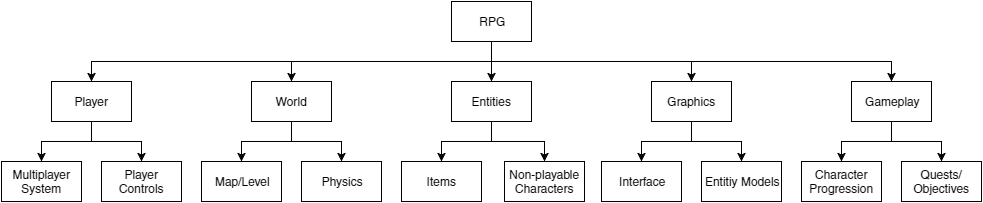
*Game* yang digunakan sebagai obyek studi kasus adalah Dungeon Crawl Stone Soup. Dungeon Crawl Stone Soup merupakan sebuah *game* yang pertama kali dikembangkan pada tahun 2006 sebagai upaya untuk menghidupkan kembali proyek *Linley’s Dungeon Crawl* yang dikembangkan oleh Linley Henzell pada tahun 1997.

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*. 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*.

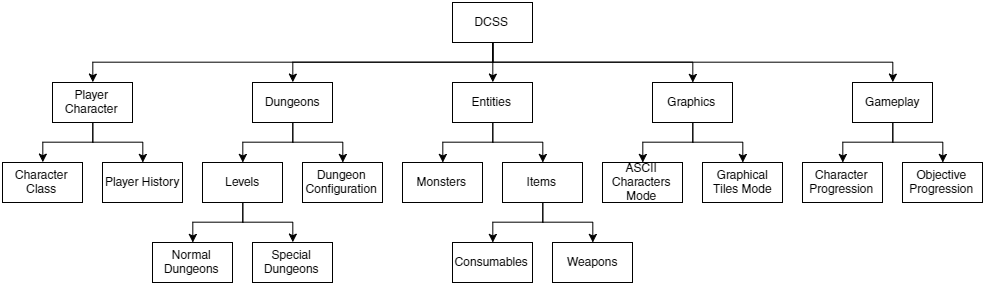


Gambar 3.1 Diagram fitur RPG secara umum

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.



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 *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.

**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***

Repositori Dungeon Crawl Stone Soup dapat diakses melalui tautan <https://github.com/crawl/crawl>.

**3.1.2.3 Bahasa**

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 Dream sheep commit (Entity Addition)**

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

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.

*Code changes:*

* 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.

Note: fungsi ini memiliki ~8000 baris code. Apa bisa dilakukan improvement agar code sebanyak itu tidak menumpuk ? Atau sudah best approach ?

* 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().

Pull Requests related to Dream Sheep

* 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 Dream Sheep: “As an emergency-flagged spell, dream sheep will directly cast Sleep instead of relying on 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 (sepertinya 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 (belum dibaca lebih lanjut apa itu “xv”, ada yang menuliskan sebagai “x-v”. Dari apa yang sudah dibaca (terdapat di crawl-ref/docs/develop/levels/advanced.txt) ‘xv’ merupakan command (sepertinya untuk Lua) untuk menampilkan deskripsi lengkap dari sebuah monster).

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. Fungsi ini telah dibahas di Materi Tambahan sebelumnya.

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*. Fungsi ini telah dibahas di Materi Tambahan sebelumnya.

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.

Note: 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.

Issues with Dream Sheep:

Isu-isu berikut merupakan komplain 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/) pada tanggal x.

* 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.

Development issues:

* 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.

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).

**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 Strategi untuk *refactoring***

*SPL* mengedepankan kemudahan navigasi *source code* sebagai salah satu usaha untuk memudahkan *programmer* bekerja. Jika dilihat, *source code* studi kasus walaupun menggunakan bahasa C++ masih tidak memanfaatkan paradigma *object-oriented* dengan baik. Untuk memudahkan navigasi, dilakukan *refactoring* sehingga *source code* lebih memanfaatkan paradigma *object-oriented* dan menerapkan *SPL* dengan baik. Dari *paper* yang dibahas di bab tinjauan literatur, dapat dilakukan beberapa langkah untuk membantu *refactoring*:

1. Menghilangkan *constant interface*

*Interface* yang digunakan untuk menyimpan konstanta dinilai sebagai sebuah *anti-pattern* yang tidak cocok dengan paradigma *object-oriented*. Karena penggunaannya memberikan beberapa dampak negatif, maka penggunaan *interface* untuk mendefinisikan konstanta dapat dihapus. Konstanta dapat didefiniskan secara langsung di dalam *class*.

1. Mengubah atribut *class* menjadi variabel lokal

Atribut-atribut yang didefinisikan di dalam *class* namun hanya digunakan oleh sebuah *method* dapat dipindahkan dan didefinisikan di dalam *method* tersebut.

1. Menghilangkan *inheritance relationship* yang redundan

Beberapa *class* yang terhubung dengan *inheritance* namun memiliki sangat sedikit kesamaan, dapat dihapus hubungan *inheritance*-nya.

1. Mengubah iterasi *loop* menjadi perbandingan dengan nol

Dalam sebuah iterasi, membandingkan suatu bilangan dengan nol lebih cepat dibandingkan dengan bilangan lain, sehingga *loop* yang membandingkan dengan sebuah bilangan bukan nol dapat diganti menjadi membandingkan dengan nol. Contoh:

for(int i=0; i<6; i++)

*Loop* 1. Membandingkan dengan bilangan bukan nol

for(int i=5; i>=0; i--)

*Loop* 2. Membandingkan dengan bilangan nol

Bentuk *loop* 1 dapat diganti menjadi *loop* 2 untuk mempercepat perbandingan.

Strategi ini tidak terlalu relevan dengan konsep optimisasi SPL, tetapi merupakan salah satu cara yang dapat diterapkan untuk meningkatkan performa *code*.

1. Menghilangkan *class* yang tidak perlu

Beberapa *class* yang memiliki sangat sedikit *method* dapat dihapus dengan cara memindahkan *functionality* yang terdapat dalam *class* tersebut ke *class* lain.

1. Menghilangkan *class* dan *method*yang sudah *obsolete*

*Method*-*method* yang tidak pernah dipakai, seperti *method* dari implementasi lama yang sudah *obsolete* dan tidak digunakan dalam implementasi baru dapat dihapus.

*Class* dan *method* yang sudah tidak terpakai namun tidak dihapus berpotensi membingungkan *programmer* yang bekerja dengan *code* tersebut.

1. Menghindari perbandingan *string* yang lambat

Perbandingan *string* dapat memakan waktu, jadi perbandingan *string* yang tidak perlu atau panjang sebaiknya dihindari.

1. Mendeklarasi *method*dan variabel sebagai *final*untuk mempercepat akses

Pendeklarasian *final* dapat membantu *compiler* bekerja lebih cepat. Selain itu, penggunaan *final* juga dapat mencegah pengubahan referensi *method* atau variabel yang tidak diinginkan. Penggunaan *final* dalam *code* yang dikerjakan oleh banyak kontributor juga membantu kontributor lain untuk mengetahui bahwa *method* atau variabel tersebut tidak untuk diubah-ubah.

1. Mengganti *resizeable vector* dengan *array*

*Vector* yang dialokasikan ke *heap* akan memakan waktu yang lebih lama dalam mengerjakan suatu proses dibandingkan dengan *array* yang dialokasikan ke *stack*. Untuk itu, *vector* dapat diganti dengan *array*.

1. Tidak *throwexception* melainkan me-*return* sebuah*null object*

*Exception* sebaiknya hanya digunakan untuk menangani atau mendeteksi *error*. Ketika menangani *object* yang diperkirakan akan menghasilkan *null* atau tidak menghasilkan apapun, sebaiknya me-*return* sebuah *null object*.

**4.2 Commonalities dan Variabilities**

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

**4.2.1 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.

**4.2.2 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.

**4.3 Perbedaan dalam *code***

Perubahan dilakukan dengan mengganti konstanta dan variabel yang memengaruhi aspek-aspek yang dijelaskan dalan subbab *variabilities*.

**4.3.1 *Refactoring* Umum**

*Refactoring* umum dilakukan berdasarkan strategi yang telah dijelaskan dalam subbab sebelumnya. Dari strategi-strategi yang yang diacu dari *paper* acuan tersebut, terdapat signifikansi dan relevansi yang berbeda untuk tiap strategi. Signifikansi dan relevansi strategi-strategi tersebut akan dibahas bersama dengan pertimbangan terhadap kondisi *code* studi kasus.

1. Menghilangkan *constant interface*

Strategi ini tidak terpakai dalam eksperimen, karena *code* studi kasus memang tidak mengandung *constant interface*.

1. Mengubah atribut *class* menjadi variabel lokal

Pemindahan sebuah atribut yang hanya dipakai oleh sebuah *method* dari dalam *class* ke dalam *method* memiliki signifikansi dalam memudahkan navigasi. Selain itu, menempatkan variabel spesifik dalam *method-method* tertentu yang menggunakannya membantu *programmer* melakukan kesalahan ketika membuat variasi. Penerapan SPL dalam membuat variasi menekankan bahwa masing-masing variasi memiliki sifat *fine-grained*, atau memiliki detail yang berbeda dengan variasi lainnya.

1. Menghilangkan *inheritance relationship* yang redundan

Strategi ini tidak terpakai dalam eksperimen, karena *code* studi kasus memang tidak mengandung *constant interface*.

1. Mengubah iterasi *loop* menjadi perbandingan dengan nol

Strategi ini memiliki signifikansi yang rendah terhadap performa dan efisiensi *code*, serta kurang relevan dengan konsep yang ditekankan SPL.

1. Menghilangkan *class* yang tidak perlu

Memindahkan *method-method* dari *class* yang hanya memilikis edikit method agar kelas tersebut dapat dihapus merupakan strategi yang relevan dengan penerapan SPL. Strategi ini membantu agar pengerjaan variasi tidak memiliki efek samping yang tidak diinginkan pada *class-class* di luar *class* yang sedang diubah.

1. Menghilangkan *class* dan *method*yang sudah *obsolete*

Menghilangkan *class* tertentu yang sudah tidak digunakan merupakan sebuah *good programming practice* yang tidak terlalu relevan dengan penerapan SPL, namun tetap memiliki dampak yang baik seperti untuk kemudahan navigasi.

1. Menghindari perbandingan *string* yang lambat

Strategi ini memiliki signifikansi yang rendah terhadap performa dan efisiensi *code*, serta kurang relevan dengan konsep yang ditekankan SPL.

1. Mendeklarasi *method*dan variabel sebagai *final*untuk mempercepat akses

Strategi ini memiliki signifikansi yang rendah terhadap performa dan efisiensi *code*, serta kurang relevan dengan konsep yang ditekankan SPL.

1. Mengganti *resizeable vector* dengan *array*

Strategi ini memiliki signifikansi yang rendah terhadap performa dan efisiensi *code*, serta kurang relevan dengan konsep yang ditekankan SPL.

1. Tidak *throwexception* melainkan me-*return* sebuah*null object*

Strategi ini memiliki signifikansi yang rendah terhadap performa dan efisiensi *code*, serta kurang relevan dengan konsep yang ditekankan SPL.

**4.4 CRC/Membuat *class***

* CRC

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.

Langkah-langkah menerapkan CRC-c:

Walk through the handling of a scenario of the use case, pointing to or picking up the cards, naming their responsibilities and how they handle and delegate each request. In a brainstorming session, add new cards as new functions are needed, or reallocate the responsibilities of the cards already on the table. It is not always necessary to name both the object type and the responsibility at the moment the card is put onto the table, as long as they are both written before the end. (Cockburn, 1999)

* Responsibilities

Menentukan *responsibilities* berdasarkan teknik *responsibility-driven modeling*.

* + Texts

Bertanggung jawab untuk setiap deskripsi teks mengenai monster

* + Behavior

Bertanggungjawab untuk pola gerakan, serangan, dan perilaku monster

* + Action

Bertanggung jawab untuk mekanika setiap aksi yang dilakukan oleh monster

* + Spell/ability

Bertanggungjawab untuk jurus-jurus yang dimilikioleh monster

* + Player interaction

Bertanggungjawab untuk hal-hal yang berkaitan dengan pemain yang dipengaruhi oleh monster

* Transform each responsibilities to class

Class monster\_action: this, that

monster\_behavior: this, that

monster\_ability: this, that

Note: beberapa class/responsibility sudah memiliki berkas sendiri di *source code* awal, tetapi terdapat beberapa fungsi yang tidak ditempatkan semestinya. Plus object-oriented programming belum diterapkan dengan optimal.

**4.5 Code Refactoring (hiscores.cc)**

* Menentukan responsibility-responsibility dari sistem skor
* Mengelompokkan responsibility-responsibility tersebut
  + Pencatatan dan tampilan High Scores
  + Pencatatan data dan statistic serta perhitungan skor akhir pemain
  + Fungsi XLog
* Mengubah tiap kelompok responsibility menjadi class
* Merefactor class yang telah dibuat menjadi OO yang ideal

**4.6 Code Refactoring (Entity Creation)**

Codes:

* mon-abil.cc
* mon-data.h
* mon-ench.cc
* mon-pick-data.h
* mon-place.cc
* mon-spell.h
* player-reacts.cc
* player.cc
* player.h

Original code 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 menjadi spell, behavior, data di mana masing-masing berkas berisi spell semua monster, behavior semua monster, dan data semua monster.

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

Ilustrasi code original:

Spell

{

Spell monster 1

Spell monster 2

Spell monster 3

}

Behavior

{

Pola gerakan monster 1

Pola serangan monster 1

Pola gerakan monster 2

Pola serangan monster 2

Pola gerakan monster 2

Pola serangan monster 2

}

Data

{

Data monster 1

Data monster 2

Data monster 3

}

Dapat diubah menjadi:

Monster 1

{

Spell monster 1

Pola gerakan monster 1

Pola serangan monster 1

Data monster 1

}

Monster 2

{

Spell monster 2

Pola gerakan monster 2

Pola serangan monster 2

Data monster 2

}

Monster 3

{

Spell monster 3

Pola gerakan monster 3

Pola serangan monster 3

Data monster 3

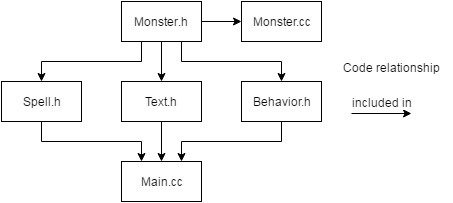
}

**BAB 5**

**EKSPERIMEN**

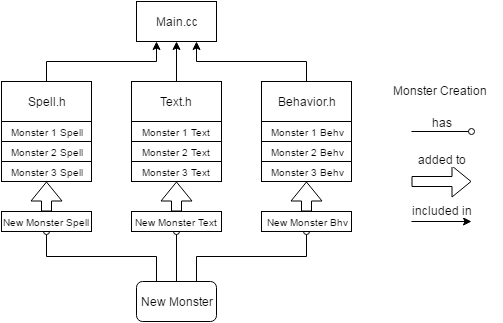
Old Design

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*.



Bentuk implementasi sebuah monster dalam struktur ini dapat dilihat pada Gambar. 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.

Ketika akan menambahkan sebuah monster baru ke dalam *game*, *contributor* perlu menambahkan fungsi-fungsi yang ingin diimplementasi ke dalam banyak berkas.



Bentuk *code* yang lama secara umum dapat digambarkan melalui beberapa contoh *code* sederhana yang melambangkan beberapa komponen monster. 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.

#ifndef MONSTER\_H

#define MONSTER\_H

#include <iostream>

#include <string>

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

class Monster

**{**

protected**:**

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

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

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

**};**

#endif

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

**}**

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. Implementasi Spell, Behavior, dan Text dapat dilihat pada berkas-berkas berikut.

#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**);**

**}**

**};**

Di dalam berkas Spell.h, terdapat fungsi-fungsi yang merepresentasikan *spell* dari masing-masing monster.

#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);**

**}**

**};**

Di dalam berkas Behavior.h, terdapat fungsi-fungsi yang merepresentasikan *behavior* dari masing-masing monster.

#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"**);**

**}**

**};**

Di dalam berkas Text.h, terdapat fungsi-fungsi yang merepresentasikan deskripsi dari masing-masing monster.

#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**();**

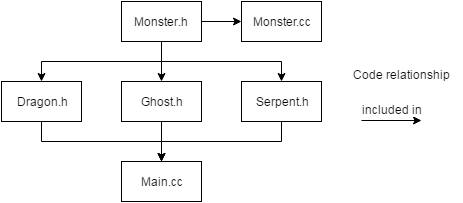
return 0;

}

Fungsi-fungsi yang berada di *class-class* Spell, Behavior, dan Text akan dijalankan oleh Main.cc.

Proposed Idea

Dalam penelitian ini, struktur *code* akan diubah untuk memanfaatkan paradigma pemrograman berbasis obyek secara lebih optimal, yaitu dengan mengelompokkan fungsi-fungsi yang ada berdasarkan *entity* yang menggunakan fungsi tersebut.



Bentuk implementasi dari struktur *code* yang telah di-*refactor* dapat dilihat pada Gambar. 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.

Dalam bentuk implementasi ini, dibuat sebuah *class* Monster yang merepresentasikan semua monster yang terdapat dalam *game*. Berikut adalah contoh berkas Monster.h yang merepresentasikan bentuk umum dari monster.

#ifndef MONSTER\_H

#define MONSTER\_H

#include <iostream>

#include <string>

#include "Spell.h"

#include "Behavior.h"

#include "Text.h"

class Monster

**{**

protected**:**

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

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

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

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

**};**

#endif

Berkas Monster.h tersebut 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 berkas Monster.cc.

#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**;**

**}**

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*.

#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"**);**

**}**

**};**

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.

#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"**);**

**}**

**};**

Untuk menjalankan fungsi-fungsi tersebut, *class* yang merepresentasikan sebuah monster perlu men-*extend* *class* Monster.

#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"**);**

**}**

**};**

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.

#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**()**

**{**

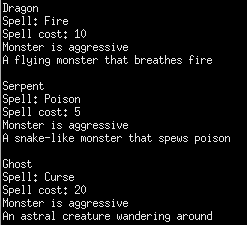
Print print**;**

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

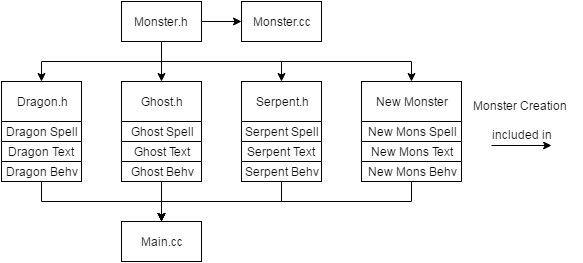
**return** 0**;**

**}**

Setelah dijalankan, *code* akan menghasilkan output seperti berikut:



Penambahan monster cukup dilakukan dengan membuat sebuah *class* baru yang men-*extend* *class* Monster, lalu *class* tersebut di-*include* di dalam berkas Main.cc.



**BAB 6**

**ANALISIS**

Low 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.

High 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.

Development Effort

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. Sebagai contoh, jika pada struktur *code* lama membutuhkan 4 buah berkas *code* teknis dan 5 berkas *code* yang merepresentasikan setiap komponen monster, maka 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. Gambaran perbandingan jumlah berkas dapat dilihat pada tabel berikut.