**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 *throw* *exception* 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**

Dua buah versi yang akan dibuat pada *game* didasarkan pada tingkat kesulitan atau *difficulty* dari permainan. Versi *casual* merupakan versi yang menawarkan tingkat kesulitan rendah, sedangkan versi *hardcore* merupakan versi yang tingkat kesulitannya lebih sulit. Variasi ini diangkat sebagai topik eksperimen karena beberapa hal, yaitu:

1. Pembuatan variasi tingkat kesulitan merupakan hal yang sangat umum dalam pengembangan *game*, melihat banyaknya *game* dari berbagai domain termasuk domain RPG yang memiliki variasi beberapa tingkat kesulitan.
2. Pengubahan tingkat kesulitan tidak mengubah inti permainan, dengan demikian tidak mengubah banyak struktur *source code*
3. Meskipun pembuatan variasi tingkat kesulitan tidak menyentuh banyak *code*, tetapi lingkup dari eksperimen meliputi cukup banyak aspek untuk menonjolkan dampak dari pemanfaatan SPL.

Dengan pertimbangan tersebut, ditentukan aspek-aspek *game* yang akan dibiarkan menjadi *commonality* dan aspek-aspek yang akan diubah menjadi *variant* yang memiliki *variability*.

**4.2.1 Commonalities**

1. Core gameplay

Inti dari permainan akan tetap sama. Segala hal yang menyangkut cara bermain, tujuan permainan, kondisi kemenangan, dan sebagainya akan tetap sama di kedua versi.

1. Controls

Konfigurasi *controls* akan tetap sama. Tidak ada perubahan dalam bagaimana cara pemain mengendalikan karakter.

1. Enemy types

Tipe-tipe musuh yang ada di dalam *dungeon* akan tetap sama. Meskipun konfigurasi kekuatan dan sifat musuh mungkin berbeda, tetapi bentuk, proporsi kekuatan, dan sifat secara umumnya akan tetap sama.

1. Weapon/skill types

Senjata dan *skill* yang tersedia untuk pemain akan tetap sama.

1. Dungeon design

Meskipun secara umum bentuk dan konfigurasi *dungeon* dibuat secara *random*, tetapi batasan-batasan dan kriteria pembuatan *dungeon* akan tetap sama.

**4.2.2 Variabilities**

1. Player freedom

Kebebasan pemain untuk versi *hardcore* akan dibatasi. Pemain tidak dapat melakukan beberapa aktifitas sebebas versi *casual*.

1. Enemy behavior

Untuk versi *hardcore*, musuh akan bergerak lebih cepat dan menyerang lebih sering.

1. Player capability

Kemampuan karakter pemain akan dikurangi untuk versi *hardcore*

1. Enemy capability

Kemampuan musuh akan ditambah untuk versi *hardcore*

1. Scoring system

Sistem penambahan skor akan berbeda untuk tiap versi. Versi *hardcore* akan memiliki variabel penambah skor yang lebih tinggi dari versi *casual*.

**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 *throw* *exception* 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.

**4.3.2 *Refactoring* Berdasarkan *Commonality* dan *Variability***

Untuk membuat variasi sistem skor dengan menerapkan konsep SPL, masing-masing variasi dari skor harus *fine grained*. Hal tersebut dicapai dengan cara memisahkan method untuk menentukan skor akhir untuk masing-masing versi.

Idealnya, untuk menerapkan SPL pada pembuatan variasi, masing-masing versi dibuat sebuah *class* yang mencakup masing-masing aspek yang berbeda antar versi.

Bentuk struktur *code* awal tidak menerapkan pendekatan SPL:

*Class*

*- Method A*

*- Function*

*- Method B*

*- Function*

*- Method C*

*- Function*

Bentuk ideal pendekatan SPL-nya adalah:

*Class 1 Class 2*

*- Method A1 - Method A2*

*- Function 1 - Function 2*

*- Method B1 - Method B2*

*- Function 1 - Function 2*

*- Method C1 - Method C2*

*- Function 1 - Function 2*

Struktur 1

Karena banyaknya *class* dalam *source code* yang berkaitan satu sama lain, pemisahan method menjadi beberapa *class* berdasarkan variasi menjadi suatu hal yang sangat sulit. Untuk itu, dalam studi kasus ini, penerapan SPL dilakukan dengan mengubah struktur menjadi:

*Class*

*- Method A1*

*- Function 1*

*- Method A2*

*- Function 2*

*- Method B1*

*- Function 1*

*- Method B2*

*- Function 2*

*- Method C1*

*- Function 1*

*- Method C2*

*- Function 1*

Struktur 2

Beberapa fungsionalitas juga memiliki keterkaitan dengan fungsi dan *method* lain, sehingga penerapan SPL dapat dilakukan dengan mengubah struktur menjadi:

*Class*

*- Method A*

*- Function 1*

*- Function 2*

*- Method B*

*- Function 1*

*- Function 2*

*- Method C*

*- Function 1*

*- Function 2*

Struktur 3

Karena banyaknya fungsionalitas dalam *game*, maka eksperimen hanya dilakukan pada bagian-bagian *code* yang paling representatif dengan aspek yang akan divariasikan dalam eksperimen ini.

1. Player freedom

Kebebasan pemain diubah dengan membatasi aksi-aksi yang dapat dilakukan oleh pemain. Karena saling terkaitnya satu fungsi dengan fungsi lain dalam *class* yang diubah, maka pengubahan dilakukan dengan menerapkan struktur 3. *Code* yang diubah adalah player.cc di mana bentuk *code* sebelum pengubahan adalah sebagai berikut:

*method*

*function*

*if*

*function*

*function*

Perubahan dilakukan dengan cara membuat sebuah *method* baru dengan bentuk yang sama, tetapi dengan konfigurasi *function* yang berbeda, sehingga masing-masing variasi akan memiliki *method* sendiri dengan konfigurasi yang berbeda. Bentuk *code* setelah dilakukan perubahan akan menjadi:

*method casual*

*function casual*

*if*

*function casual*

*function casual*

*method hardcore*

*function hardcore*

*if*

*function*  *hardcore*

*function hardcore*

1. Enemy behavior

Kekuatan pemain diubah dengan mengubah konfigurasi tingkah laku musuh. Karena saling terkaitnya satu fungsi dengan fungsi lain dalam *class* yang diubah, maka pengubahan dilakukan dengan menerapkan struktur 3. *Code* yang diubah adalah mon-behv.cc, di mana struktur *code* secara umum berbentuk seperti berikut:

*method*

*function*

*if*

*function*

*function*

Perubahan dilakukan dengan cara membuat sebuah *method* baru dengan bentuk yang sama, tetapi dengan konfigurasi *function* yang berbeda, sehingga masing-masing variasi akan memiliki *method* sendiri dengan konfigurasi yang berbeda. Bentuk *code* setelah dilakukan perubahan akan menjadi:

*method casual*

*function casual*

*if*

*function casual*

*function casual*

*method hardcore*

*function hardcore*

*if*

*function*  *hardcore*

*function hardcore*

1. Player capability

Kekuatan pemain diubah dengan mengubah kemampuan pemain untuk menyerang musuh. Karena saling terkaitnya satu fungsi dengan fungsi lain dalam *class* yang diubah, maka pengubahan dilakukan dengan menerapkan struktur 3.

Berkas yang diubah adalah attack.cc, di mana *code* awal memiliki struktur sebagai berikut:

*method*

*if* (pemain)

*function* pemain

*if* (musuh)

*function* musuh

Perubahan dilakukan dengan cara membuat sebuah variabel *flag* yang menandakan versi, kemudian membuat variasi pada masing-masing *function* yang diakses ketika variabel *flag* sesuai. Penerapan ini juga berlaku untuk aspek *enemy strength*.

Bentuk *code* setelah dilakukan perubahan akan menjadi:

*method*

*if* (pemain)

*function* pemain *casual*

*function* pemain *hardcore*

*if* (musuh)

*function* musuh *casual*

*function* musuh *hardcore*

Pembuatan variasi hingga *fine-grained* cukup sulit dilakukan karena banyaknya elemen-elemen dalam *method* dan *function* yang terhubung dengan *method* atau bahkan *class* lain, sehingga membutuhkan waktu yang sangat lama untuk *refactoring*.

1. Enemy capability

Kekuatan musuh diubah dengan mengubah kemampuan musuh untuk menyerang pemain. Penjelasan eksperimen mengacu pada penjelasan aspek sebelumnya.

1. Scoring system

Dalam aspek ini, unsur-unsur yang divariasikan adalah banyaknya skor akhir yang didapat oleh pemain untuk *experience* yang didapatkan. Untuk menghindari ketidakcocokan dengan *class* lain, maka pengubahan dilakukan dengan menerapkan struktur 2. *Code* yang diubah adalah hiscores.cc.

Bentuk *code* secara umum sebelum pengubahan adalah:

*method*

*function*

*function*

Perubahan dilakukan dengan cara membuat sebuah *method* baru dengan bentuk yang sama, tetapi dengan konfigurasi *function* yang berbeda, sehingga masing-masing variasi akan memiliki *method* sendiri dengan konfigurasi yang berbeda. Bentuk *code* setelah dilakukan perubahan akan menjadi:

*Method casual*

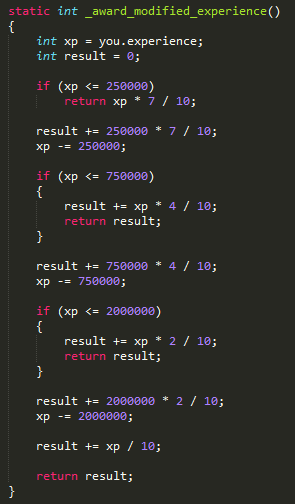
*Function casual*

*Function casual*

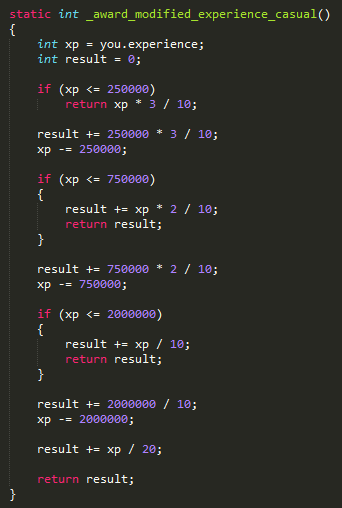
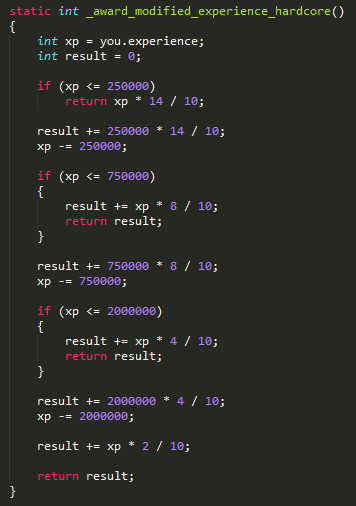
*Method hardcore*

*Function hardcore*

*Function hardcore*



Gambar 4.1 Bentuk *code* sistem skor sebelum pengubahan

Gambar 4.2 Bentuk *code* sistem skor setelah pengubahan

**4.4 Analisis**

Dari hasil eksperimen yang telah dilakukan, terdapat beberapa hal yang perlu diperhatikan:

1. Penerapan SPL pada *code* atau proyek yang sudah berjalan atau sudah jadi akan sangat berat untuk dilakukan
2. Pengerjaan tidak dapat dilakukan secara optimal karena banyaknya *class* dan *method* yang saling berhubungan
3. Sulit membuat perubahan pada *code* yang kurang menerapkan paradigma *object-oriented*