Jurnal Teknologi Informasi dan Ilmu Komputer (JTIIK) DOI: 10.25126/jtiik.202294646
Vol. 9, No. 4, Agustus 2022, hlm. 743-750 p-ISSN: 2355-7699
Akreditasi KEMENRISTEKDIKTI, No. 36/E/KPT/2019 e-ISSN: 2528-6579

TOOL REFACTORING OTOMATIS UNTUK MENANGANI LAZY CLASS CODE SMELL DENGAN PENDEKATAN SOFTWARE METRICS

Umi Sa'adah[1], Desy Intan Permatasari[2], Fadilah Fahrul Hardiansyah[3], Andhik Ampuh Yunanto[*4], Jauari Akhmad Nur Hasim[5], Irma Wulandari[6], Muhammad Reza Pahlevi[7], Dufan Quraish Shihab[8]

[1,2,3,4,5,6,7,8]Politeknik Elektronika Negeri Surabaya, Surabaya
Email: [1]umi@pens.ac.id, [2]desy@pens.ac.id, [3]fahrul@pens.ac.id, [4]andhik@pens.ac.id, [5]jauari@pens.ac.id, [6]irma@pens.ac.id, [7]apahlevie@gmail.com, [8]dufanquraish@gmail.com
[*]Penulis Korespondensi

(Naskah masuk: 23 Januari 2021, diterima untuk diterbitkan: 18 Agustus 2022

Abstrak

Keberadaan lazy class sebagai code smell dapat meningkatkan jumlah class yang tidak begitu perlu pada perangkat lunak, sehingga meningkatkan biaya pemeliharaan dari segi waktu dan usaha. Ancaman tersebut dapat diatasi dengan restrukturisasi internal atau refactoring perangkat lunak. Namun, akibat keterbatasan tool, mengharuskan proses refactoring dilakukan secara manual, sehingga membutuhkan waktu dan biaya pemeliharaan yang tinggi. Penelitian ini mengajukan sebuah tool yang dapat mendeteksi dan me-refactoring lazy class secara otomatis. Penelitian yang diajukan ini bertujuan untuk menghindari refactoring lazy class secara manual. Input dari tool merupakan lokasi sebuah projek. Proses dimulai dari mendeteksi file dan class pada projek. Kemudian dilakukan proses deteksi lazy class dengan mengukur karakteristik perangkat lunak atau software metrics. Tahapan terakhir yaitu proses refactoring otomatis, yang dilakukan dengan membuat, me-replace, atau menghapus file, untuk menghasilkan projek yang telah di-refactor. Berdasarkan hasil percobaan, tool yang dikembangkan ini mampu mendeteksi dan me-refactoring lazy class dengan tingkat akurasi sama dengan manual dan proses kecepatannya hanya 5,71 detik. Sehingga hal ini menunjukkan bahwa tool dapat bekerja secara akurat dan lebih cepat dibandingkan dengan cara manual. Serta tool ini diharapkan dapat membantu para pengembang untuk meminimalisir effort dari segi biaya dan waktu dalam melakukan refactoring.

Kata kunci: *lazy class, code smell, refactoring* otomatis, *software metrics*

AUTOMATIC REFACTORING TOOL TO HANDLE LAZY CLASS CODE SMELL WITH SOFTWARE METRICS APPROACH

Abstract

The existence of lazy classes as code smells can increase the number of unnecessary classes in software, thus increasing maintenance costs in terms of time and effort. These threats can be overcome by internal restructuring or software refactoring. However, due to limited tools, the refactoring process is required to be done manually, which requires time and high maintenance costs. This research proposes a tool that can detect and refactor lazy class automatically. This research is proposed to avoid refactoring lazy class manually. The input of the tool is the location of a project. The process starts with detecting files and classes in the project. Then the lazy class detection process is carried out by measuring the characteristics of the software or software metrics. The final stage is the automatic refactoring process, which is done by creating, replacing, or deleting files, to produce a refactored project. Based on the experimental results, the tool developed is able to detect and refactoring lazy classes with the same accuracy level as manual and the process speed is only 5.71 seconds. So this shows that the tool can work accurately and faster than the manual method. And this tool is expected to help developers to minimize the effort in terms of cost and time in refactoring.

Keywords: *lazy class, code smell, automatic refactoring, software metrics*

1. PENDAHULUAN

Pengembangan perangkat lunak secara terus menerus dilakukan seiring dengan meningkatnya kebutuhan perangkat lunak, sehingga kode yang dihasilkan semakin kompleks dan sulit dibaca. Sulitnya pembacaan kode umumnya disebabkan oleh

desain kode yang buruk atau disebut dengan code smells. Code smell adalah karakteristik struktural perangkat lunak yang berpotensi memperlambat pengembangan dan meningkatkan risiko adanya bug atau celah keamanan di masa mendatang (Wikipedia, 2018). Lazy class merupakan salah satu jenis code smell yang memiliki class dengan fungsionalitas rendah. Lazy class dapat meningkatkan jumlah class yang tidak begitu perlu pada perangkat lunak, sehingga meningkatkan biaya pemeliharaan dari segi waktu dan usaha (Fowler et al., 1999). Lazy class dapat ditangani dengan proses refactoring pada kode dan permeriksaan ulang desain secara keseluruhan.

Refactoring menurut Martin Fowler dan Kent Beck adalah perubahan pada struktur internal perangkat lunak agar lebih mudah dipahami dan dimodifikasi tanpa mengubah alur perangkat lunak. Namun, keterbatasan tool pendukung yang ada, mengharuskan proses refactoring dilakukan secara manual. Proses refactoring secara manual seringkali sulit dilakukan akibat tidak tersedianya standar yang jelas untuk melakukan klasifikasi code smell. Hal tersebut membutuhkan waktu dan biaya pemeliharaan yang tinggi.

Beberapa metode penanganan lazy class telah diajukan dalam penelitian sebelumnya, baik menangani proses deteksi hingga proses refactoring. Salah satu penelitian (Aristyagama, 2016), menyediakan sebuah rancangan desain framework deteksi bad smell code semi otomatis untuk menangani permasalahan standarisasi bad smell code dalam pemrograman tim dengan memanfaatan expert system sebagai model knowledge base. Penelitian lain (Vale dan Figueiredo, 2015) mengusulkan metode untuk menurunkan ambang batas konteks SPL dalam penerapan dua deteksi code smell, yaitu God Class dan Lazy Class. SPL (software product line) adalah sekumpulan sistem perangkat lunak yang berbagi kumpulan komponen yang umum dan variabel.

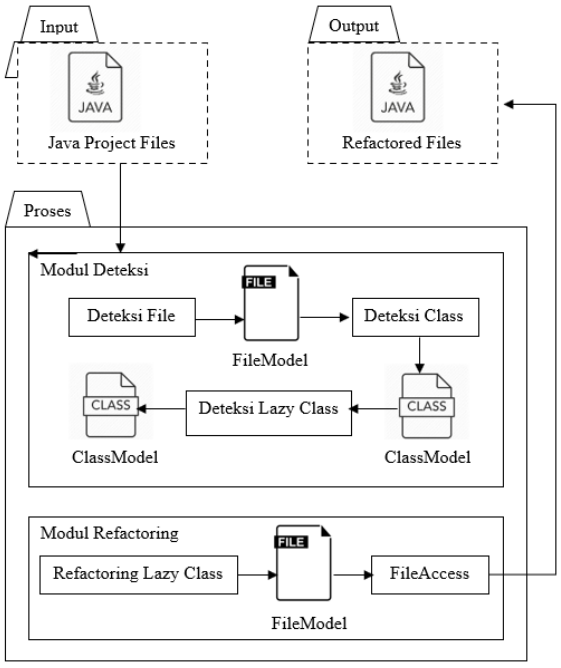
Pendeteksian (Kim et al., 2013) secara tepat dapat mendeteksi code smell dengan menggunakan OCL (Object Constraint Language) dan mempelajari cara mendeteksi mereka secara otomatis dengan menjalankan komponen OCL. Penelitian lain (Ibrahim et al., 2020) menyajikan pendekatan tentang cara mendeteksi dan me-refactor code smell dari kode aplikasi Android untuk mengurangi redundansi dalam pembuatan kasus uji.

Selain deteksi, terdapat juga penelitian yang membahas proses refactoring code smell (Szőke et al., 2015). Penelitian tersebut membangun sebuah toolset refactoring FaultBuster yang mendukung refactoring otomatis. Toolset ini menggunakan refactoring framework khusus untuk mengontrol analisis dan eksekusi algoritma otomatis.

Penelitian yang diajukan ini bertujuan untuk menghindari refactoring lazy class secara manual. Penelitian ini mengajukan sebuah automated tool untuk mendeteksi lazy class dan melakukan refactoring secara otomatis. Pendeteksian dilakukan dengan memasukkan aturan ke dalam sistem untuk menentukan apakah kode tersebut termasuk code smell atau bukan. Sedangkan refactoring dilakukan dengan membuat, me-replace, atau menghapus file dengan tujuan menghilangkan code smell pada program. Penelitian ini dapat bermanfaat bagi pengembang dalam menjaga dan meningkatkan kualitas desain kode sekaligus menghemat waktu dalam melakukan refactoring lazy class.

2. METODE PENELITIAN

Penelitian ini mengajukan dan mengembangkan sebuah *tool* yang dapat mendeteksi dan sekaligus melakukan *refactoring lazy class* secara otomatis. *Tool* dibangun dengan menggunakan *maven project* yang dapat dijalankan melalui *command line* secara langsung dan telah didukung oleh IDE populer seperti IntelliJ IDEA, Eclipse dan NetBeans. *Tool* terdiri dari bagian-bagian kecil atau modul yang memiliki fungsi spesifik, sehingga dapat dijadikan sebagai *depedency* antar modul. Secara garis besar, *tool* pada penelitian ini memiliki dua modul utama, yaitu modul deteksi dan modul *refactoring*. Desain sistem penelitian digambarkan pada Gambar 1.



Gambar 1. Desain sistem penelitian

Berdasarkan desain pada Gambar 1, *input* dari sistem merupakan lokasi sebuah projek dengan bahasa pemrograman Java. Projek Java tersebut dilakukan beberapa proses, dimulai dari analisa kode untuk mendeteksi file dan class pada projek. Setelah itu, dilakukan proses deteksi dengan menggunakan *software metrics* (Danphitsanuphan dan Suwantada, 2012), untuk menentukan *lazy class code smell* pada class. Tahapan terakhir yaitu proses *refactoring*,

yang menerapkan teknik *inline class* dan *collapse hierarchy*, untuk menghasilkan sebuah projek Java yang telah berhasil di-*refactor*.

Modul Deteksi

Tahapan awal dari sistem adalah proses deteksi. Input berupa projek Java akan dianalisa melalui beberapa proses. Proses tersebut diawali dengan deteksi file, deteksi class, hingga deteksi *lazy class code smell.*

1. Deteksi File

Proses ini bertujuan untuk membaca *file-file* yang ada pada *path* utama yang diinputkan. Diagram alur proses digambarkan pada Gambar 2 dan didapat berdasarkan penelitian sebelumnya (Putra *et al.*, 2019).

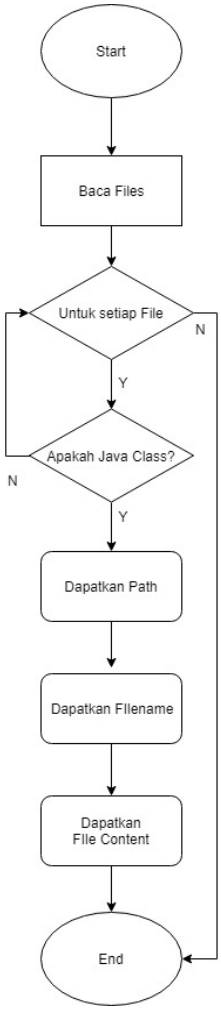
Berdasarkan diagram pada Gambar 2, proses deteksi dilakukan dengan menganalisa direktori projek, dan membaca setiap file satu persatu. Setiap file yang terbaca sebagai direktori, maka akan berpindah ke dalam direktori tersebut. Namun, apabila terbaca sebagai file, maka akan dilakukan proses pengecekan tipe file. Pengecekan tersebut untuk mencari file yang merupakan file Java. Apabila file tersebut ditemukan, maka selanjutnya adalah membaca *path, filename,* dan isi atau konten file.Proses deteksi file terus berlanjut hingga semua file pada projek selesai terbaca.

2. Deteksi Class

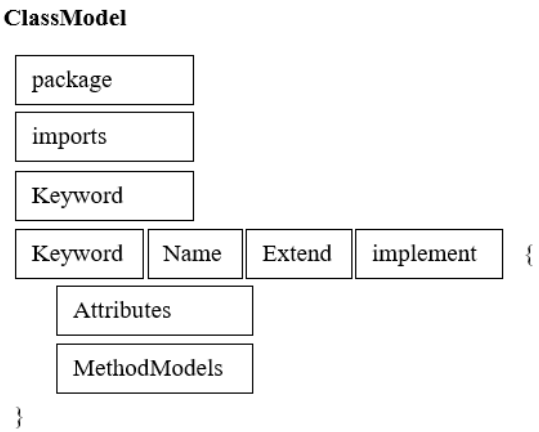
Proses ini berfungsi untuk melakukan analisa terhadap isi atau konten sebuah file java yang memiliki kelas di dalamnya. Proses analisa pada modul ini memecah satu kelas utuh menjadi beberapa bagian, sehingga dapat mempermudah proses penghitungan *software metrics.* Bagian atau atribut tersebut dapat dilihat pada Gambar 3.

Gambar 3 menunjukkan struktur sebuah class yang meliputi *package, import, keywords, name, extend, implement, attributes* dan *methods.* Masing masing atribut tersebut dijelaskan sebagai berikut.

- *Package,* yaitu nama package dari class.
- *Imports,* merupakan daftar *library* atau class lain yang dideklarasian sebelum deklarasi class.
- *Keyword,* merupakan konten yang terdapat setelah import dan sebelum pendeklarasian nama kelas. Isi dari atribut ini biasanya berupa *comment* ataupun *annotation* yang ditulis di bagian atas pendeklarasian class.
- *Name,* merupakan nama dari suatu class.
- *Extend,* merupakan nama class yang menjadi *parent* dari suatu class.
- *Implement,* merupakan nama dari class *interface* yang diimplementasikan pada suatu class.
- *Attributes,* merupakan daftar *field,* atribut, atau variabel yang dimiliki oleh sebuah class.
- *Method,* merupakan daftar method yang dimiliki oleh sebuah class.

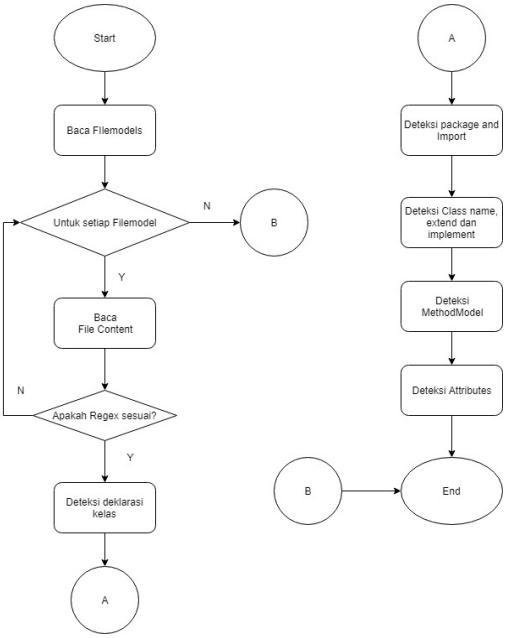


Gambar 2. Desain sistem deteksi file



Gambar 3. Struktur class

Selain atribut tersebut, terdapat atribut lain dari class untuk menyimpan hasil perhitungan *software metric*, yaitu LOC *(Lines of Code)*, NOM *(Number of Methods)*, dan NOF *(Number of Fields)*. Secara urut, atribut-atribut tersebut mewakili banyaknya baris class, banyaknya fungsi dalam class, dan banyaknya atribut class. Alur proses deteksi class ini dapat dilihat pada Gambar 4.



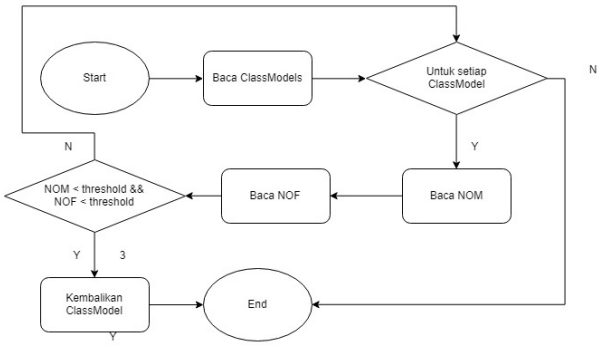
Gambar 4. Desain sistem deteksi class

Deteksi class dilakukan dengan membaca setiap file dari deteksi sebelumnya. Setelah itu, setiap file dilakukkan proses pembacaan *content* per barisnya. Deteksi class dilakukan menggunakan bantuan *regex*, sehingga didapatkan index yang memuat pendeklarasian class pada file. Kemudian dilakukan analisa untuk mendapatkan atribut yang ada pada class tersebut. Outputnya merupakan sebuah model yang memiliki atribut-atribut tersebut.

3. Deteksi *Lazy Class*

Proses ini merupakan proses lanjutan setelah deteksi class untuk mendeteksi adanya *lazy class code smell* pada sebuah class. Dari class yang didapat dari deteksi sebelumnya, akan dilakukan pendekatan melalui perhitungan *software metrics* untuk mengidentifikasi potensi keberadaan *lazy class*. *Software metric* merupakan sebuah ukuran dari karakteristik perangkat lunak yang dapat dihitung (Stringfellow, 2017). Sebuah penelitian (Danphitsanuphan dan Suwantada, 2012) memiliki dua buah *metrics* yang dijadikan sebagai acuan untuk menentukan *lazy class*, yaitu NOM dan NOF. Sebuah class dikategorikan sebagai *lazy class* apabila memiliki method dan atribut kurang dari lima (Danphitsanuphan dan Suwantada, 2012). Alur proses pendeteksian *lazy class* dapat dilihat pada Gambar 5.

Berdasarkan Gambar 5, hal yang dilakukan adalah membaca seluruh class yang terdeteksi atau *ClassModel* satu per satu dan menghitung metrik NOM dan NOF. Kedua nilai tersebut akan dibandingkan dengan *threshold* untuk menentukan *lazy class.* Apabila termasuk, maka class tersebut akan dimasukkan ke dalam daftar *lazy class.* Namun, apabila bukan, maka proses pengecekan dilanjutkan pada class selanjutnya. Output utama proses ini adalah menghasilkan *ClassModel* yang memiliki jumlah method dan atribut kurang dari *threshold* atau terindikasi sebagai *lazy class.*



Gambar 5. Desain sistem deteksi lazy class

Modul Refactoring

Setelah didapatkan class yang terindikasi sebagai *lazy class* dari hasil deteksi, proses dilanjutkan pada tahapan *refactoring* untuk memperbaiki class tersebut. *Refactoring lazy class* dilakukan dengan memindahkan semua *method* dan *field* yang ada pada *lazy class* kepada class lain. Oleh karena itu, dibutuhkan proses penentuan class target sebagai penampung *method* dan *field* dari class pendonor atau *lazy class.*

Penentuan class target dilakukan dengan mempertimbangkan hubungan hirarki dari *lazy class* dan banyaknya penggunaan *resource* dari *lazy class* oleh class lain. Ada dua aturan utama dalam penentuan target class sebagai berikut.

1. Apabila *lazy class* merupakan *child* dari sebuah class, maka target class-nya adalah class *parent.*
2. Apabila *lazy class* bukan merupakan *child*, maka target class akan ditentukan berdasarkan class mana dalam projek tersebut yang paling banyak mengakses *resource*, baik *method* ataupun *attribute* dari *lazy class*. Setiap pemanggilan *method* atau *attribute* akan dihitung 1 poin.

Selanjutnya, file yang dianggap sebagai *lazy class* akan dihapuskan, dan file class target akan diganti dengan file class yang baru dan telah terhindar dari *lazy class*..

3. HASIL DAN PEMBAHASAN

Pada tahap ini dilakukan pengujian terhadap sistem yang telah dibuat. Dalam pelaksanaan uji coba, peneliti menggunakan perangkat dengan spesifikasi pada Tabel 1.

Tabel 1. Spesifikasi perangkat uji coba

Sistem	Informasi
Sistem Operasi	Windows 10 Home Single Language 64bit
Processor	Intel(R) Core(TM) i7-6500U CPU @2.50GHz (4 CPUs), ~2.6GHz
RAM	8 GB
Storage	HDD 1 TB

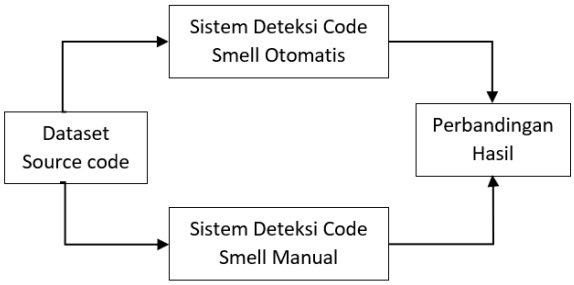
Peneliti menggunakan perangkat pribadi dengan spesifikasi sistem operasi Windows 10, processor Intel Core-i7 2.50GHz, RAM 8 GB, dan HDD 1 TB. Untuk menguji coba sistem deteksi *lazy class code smell*, maka diperlukan pencarian *code smell* pada beberapa kode program sebuah projek. Berikut pada Tabel 2 merupakan beberapa projek uji coba, yang merupakan projek *open-source* dengan bahasa pemrograman Java dari *repository* Github.

Tabel 2. Spesifikasi data projek uji coba

Nama Projek	Ukuran Class
Hotel-booking	21
Event-management-App	16
College-Management-Android-App	45
Bank Application	21

Berdasarkan Tabel 2, projek-projek yang dijadikan sebagai data uji coba memiliki ukuran class yang berbeda-beda. Jumlah class dalam project akan dilakukan analisa keterkaitan dengan waktu pendeteksian yang diperlukan untuk menentukan class mana yang termasuk *lazy class*.

Pencarian dilakukan dengan dua cara, yaitu dengan deteksi sistem secara otomatis dan deteksi secara manual. Desain pencarian dapat dilihat pada Gambar 6.



Gambar 6. Desain uji coba deteksi

Hasil pencarian kedua cara akan dibandingkan. Uji coba dikatakan berhasil apabila perbedaan hasil antara deteksi sistem dan deteksi manual tidak terlalu jauh ataupun tidak ada. Berikut merupakan hasil pengujian seberapa banyak *lazy class* ditemukan. Hasil pengujian akurasi deteksi dapat dilihat pada Tabel 3.

Tabel 3. Hasil uji coba akurasi deteksi lazy class

Nama Projek	Jumlah *Lazy Class*	
	Sistem	Manual
Hotel-booking	2	2
Event-management-App	3	3
College-Management-Android-App	8	8
BankApplication	4	4
Rata-rata	4,25	4,25

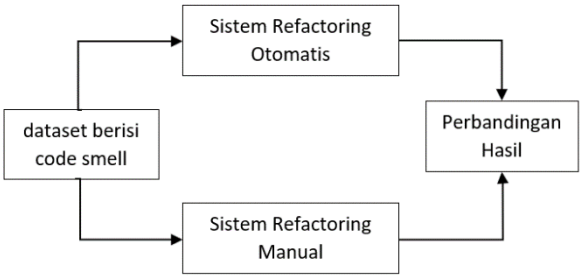
Dari hasil pengujian didapatkan jumlah yang sama untuk *lazy class* yang terdeteksi, baik menggunakan sistem maupun secara manual. Selanjutnya adalah membandingkan efektifitas waktu yang diperlukan untuk melakukan proses deteksi *lazy class* dengan dan tanpa menggunakan tool. Hasil pengujian ini dapat dilihat pada Tabel 4.

Tabel 4. Hasil uji coba waktu deteksi lazy class

Nama Projek	Waktu Deteksi (detik)	
	Sistem	Manual
Hotel-booking	2.22	94
Event-management-App	3.67	186
College-Management-Android-App	4.60	248
Bank Application	3.45	155
Rata-rata	3,485	170,75

Hasil pengujian di atas menunjukkan bahwa proses pendeteksian secara manual membutuhkan waktu yang jauh lebih lama dibanding proses pendeteksian yang dilakukan menggunakan sistem yang telah dikembangkan.

Untuk menguji coba sistem *refactoring* otomatis, maka perlu dilakukan pengembangan perangkat lunak menggunakan sistem dan tanpa sistem. Setelah perangkat lunak selesai dikembangkan melalui kedua cara pengembangan tersebut, selanjutnya dilakukan deteksi *lazy class code smell* pada perangkat lunak tersebut. Desain uji coba dapat dilihat pada Gambar 7.



Gambar 7. Desain uji coba refactoring

Hasil yang dibandingkan melalui uji coba adalah lamanya waktu yang dibutuhkan pengembang untuk melakukan *refactoring* saat proses pengembangan perangkat lunak, dan banyaknya *lazy class* yang masih tertinggal saat proses pengembangan perangkat lunak selesai dilakukan.

Uji coba pertama dilakukan dengan membandingkan akurasi dan efektifitas waktu yang diperlukan untuk melakukan proses *refactoring lazy class code smell* pada sebuah *project* dengan dan tanpa menggunakan *tool*. Hasil pengujian akurasi dapat dilihat pada Tabel 5.

Tabel 5. Hasil uji coba akurasi refactoring lazy class

Nama Projek	Total Masalah / *Error*	
	Sistem	Manual
Hotel-booking	12	12
Event-management-App	18	18
College-Management-Android-App	48	48
Bank Application	24	24
Rata-rata	25,5	25,5

Dari hasil pengujian Tabel 5 didapatkan jumlah yang sama untuk permasalahan atau *error* yang ditangani oleh sistem pada proses refactoring dengan *error* yang ditangani saat dilakukan *refactoring* secara manual. Hal ini menunjukkan bahwa sistem yang dibangun berhasil melakukan proses *refactoring lazy class* secara akurat. Untuk hasil pengujian waktu dapat dilihat pada Tabel 6.

Tabel 6. Hasil uji coba waktu refactoring lazy class

Nama Projek	Waktu *Refactoring* (detik)	
	Sistem	Manual
Hotel-booking	3.22	162
Event-management-App	5.61	203
College-Management-Android-App	8.62	549
Bank Application	5.40	275
Rata-rata	5,71	297,25

Hasil pengujian yang ditunjukkan pada Tabel 6 menunjukkan bahwa proses *refactoring* secara manual membutuhkan waktu yang jauh lebih lama dibanding dengan menggunakan sistem. Hasil rata-rata menunjukkan bahwa dengan menggunakan sistem, proses refactoring hanya memakan waktu 5,71 detik dimana prosesnya sekitar 55 kali lebih cepat dibandingkan dengan cara manual. Hal ini menunjukkan bahwa *refactoring* menggunakan sistem yang dibangun memiliki performa yang jauh lebih cepat dibanding proses *refactoring* secara manual.

Pada penelitian ini, fokus yang dikerjakan oleh peneliti diantaranya sebatas mengembangkan sebuah tool refactoring yang nantinya dapat membantu para peneliti atau pengembang lain dalam membangun suatu sistem yang bebas dari lazy class. Sehingga parameter kualitas yang diuji pada penelitian ini juga sebatas akurasi dan kecepatan proses dibandingkan dengan metode manual. Sehingga kenyataannya penelitian ini memang diawali di tahap teknis terlebih dahulu dimana mengembangkan sistem yang berbasis dasar teori dan referensi dari programming. Namun kedepannya penelitian ini akan terus dikembangkan dan akan diujikan dengan metode-metode ilmiah lainnya untuk mendapatkan hasil yang lebih optimal.

4. KESIMPULAN

Di dalam penelitian ini, telah dibangun sebuah tool yang mendukung proses deteksi dan refactoring lazy class secara otomatis. Dari hasil percobaan yang dilakukan, tool ini mampu mendeteksi dan refactoring lazy class secara akurat dan lebih cepat dibandingkan dengan cara manual dengan pendekatan software metrics. Berdasarkan hasil tersebut, tool ini dapat digunakan untuk meningkatkan kualitas kode pada perangkat lunak dan membantu para pengembang untuk meminimalisir effort dari segi biaya dan waktu dalam melakukan proses refactoring. Penelitian kedepan akan dilakukan pengembangan tool yang lebih kompleks seperti refaktoring kode duplikat melalui pendekatan *clean code architecture*.

DAFTAR PUSTAKA

FOWLER, M., & BECK, K. 1999. *Refactoring: Improving the Design of Existing Code*. Westford: Addison-Wesley Professional.

FONTANA, A. F., BRAIONEA, P., & ZANONIA, M. 2012. Automatic detection of bad smells in code: An experimental assessment. dalam *Journal of Object Technology, Vol. 11, No. 2.*

KIM, T. W., KIM. T. G., & SEU, J. H. 2013. Specification and Automated Detection of Code Smells using OCL. dalam *International Journal of Software Engineering and Its Applications, Vol. 7, No. 4, pp. 35-44.*

ARISTYAGAMA, Y. 2016. Framework Deteksi Bad Smell Code Semi Otomatis untuk Pemrograman Tim. *Institut Teknologi Bandung, Bandung.*

BLONSKI, H., PADILHA, J., BARBOSA, M., SANTANA, D., & FIGUEIREDO, E., 2013. ConcernMeBS: Metrics-based Detection of Code Smells. dalam *Congresso Brasileiro de Software (CBSoft)*, Brasília.

IBRAHIM, R., AHMED, M., NAYAK, R., & JAMEL, S. 2020. Reducing redundancy of test cases generation using code smell detection and refactoring, dalam *Journal of King Saud University - Computer and Information Sciences*, vol. 32, no. 3, pp. 367-374.

DONG, K. K. 2017. Finding Bad Code Smells with Neural Network Models. dalam *International Journal of Electrical and Computer Engineering (IJECE)*.

PUTRA, F. Z. P., PERMATASARI, D. I., SA'ADAH, U., HARDIANSYAH, F. F., & HASIM, J. A. N., 2019. Rancang Bangun Pustaka untuk Deteksi Otomatis Long Method Code Smell. dalam *The 11th National Conference on Information Technology and Electrical Engineering*, Yogyakarta.

VALE, G. & FIGUEIREDO, E. 2015. A Method to Derive Metric Thresholds for Software Product Lines. dalam *2015 29th Brazilian Symposium on Software Engineering (SBES)*.

MANTYLA, M. V., VANHANEN, J., & LASSENIUS, C. 2004. Bad Smells - Humans as Code Critics. dalam *20th IEEE International Conference on Software Maintenance (ICSM'04)*. Chicago.

KOLB, R., MUTHIG, D., PATZKE, T., & YAMAUCHI, K. 2005. A Case Study in Refactoring A Legacy Component For Reuse In A Product Line. dalam *Proceedings of*

International Conference on Software Maintenance. Budapest.

RATZINGER, J., FISCHER, M., & GALL, H. 2005. Improving Evolvability Through Refactoring. dalam *Proceedings of the international workshop on Mining software repositories*, pp. 1-5.

MOSER, R., SILITTI, A., & ABRAHAMSSON, P. 2006. Does Refactoring Improve Reusability? dalam *Proceedings of International Conference on Reuse of Off-the-Shelf Components*, pp. 287-297.

DANPHITSANUPHAN, P., & SUWANTADA, T. 2012. Code Smell Detecting Tool and Code Smell-Structure Bug Relationship. dalam *Spring Congress on Engineering and Technology 2012*, pp. 1-5.

SZOKE, G., NAGY, C., FULOP, L. J., FERENC, R., & GYIMOTHY, T. 2015. FaultBuster: An Automatic Code Smell. dalam *2015 IEEE 15th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Bremen.

PADILHA, J., PEREIRA, P., FIGUEIREDO, E., ALMEIDA, J., GARCIA, A., & SANT' ANNA, C. 2014. On the Effectiveness of Concern Metrics to Detect Code Smells: An Empirical Study. dalam *International Conference on Advanced Information Systems Engineering (CAiSE 2014)*, Thessaloniki.

PALOMBA, F. 2015. Textual Analysis for Code Smell Detection. dalam *IEEE/ACM 37th IEEE International Conference on Software Engineering*.

Code Smell. Diakses dari http://en.wikipedia.org/wiki/Code_smell. Tanggal 3 September 2018.

Dispensables, Refactoring Guru. Diakses dari https://refactoring.guru/refactoring/smells/dispensables. Tanggal 23 Januari 2019.

What Are Software Metrics and How Can You Track Them? Stringfellow, A. Diakses dari https://dzone.com/articles/what-are-software-metrics-and-how-can-you-track-th. Tanggal 27 Juni 2019.

Halaman ini sengaja dikosongkan