PENGEMBANGAN SISTEM DETEKSI GOD CLASS DAN BRAIN CLASS CODE SMELL

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
KEVIN AZWEGA
NIM:155150201111035

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2020

PENGESAHAN

PENGEMBANGAN SISTEM DETEKSI GOD CLASS DAN BRAIN CLASS CODE SMELL

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Kevin Azwega
NIM: 155150201111035

Skripsi ini telah diuji dan dinyatakan lulus pada
28 Juli 2020
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Adam Hendra Brata, S.Kom, M.T, M.Sc	Eriq Muhammad Adams Jonemaro, S.T, M.Kom
NIP: 19900105 201903 1 009	NIP: 19850410 201212 1 001

Mengetahui
Ketua Jurusan Teknik Informatika

Achmad Basuki, S.T, M.MG, Ph.D
NIP: 19741118 200312 1 002

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar referensi.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 1 Juli 2020

Kevin Azwega

NIM: 155150201111035

PRAKATA

Puji syukur kehadirat Allah SWT yang telah melimpahkan rahmat, taufik dan hidayah-Nya sehingga laporan skripsi yang berjudul "Pengembangan Sistem Deteksi *Code Smell God Class dan Brain Class*". Ini dapat terselesaikan.

Penulis menyadari bahwa skipsi ini tidak akan sukses tanpa bantuan dari beberapa pihak. Oleh karena itu, penulis ingin menyampaikan rasa hormat dan terima kasih kepada:

1. Kedua orang tua dan keluarga atas doa dan dukungan yang diberikan kepada penulis selama proses studi yang ditempuh penulis.
2. Bapak Adam Hendra Brata, S.Kom., M.T., M.Sc dan Bapak Eriq Muhammad Adams Jonemaro, S.T., M.kom selaku Pembimbing skripsi yang membimbing dan mengarahkan penulis sehingga dapat menyelesaikan skripsi ini.
3. Bapak dan Ibu dosen Fakultas Ilmu Komputer Universitas Brawijaya atas ilmu yang diberikan kepada penulis.
4. Teman-teman Teknik Informatika 2015 atas dukungan, motivasi dan memberikan waktu untuk berdiskusi selama ini.

Penulis menyadari bahwa dalam penyusunan skripsi ini masih banyak kekurangan, sehingga saran dan kritik yang membangun sangat penulis harapkan. Akhir kata penulis berharap skripsi ini dapat membawa manfaat bagi semua pihak yang menggunakannya.

Malang, 1 Juli 2020

Kevin Azwega
azwega.kevin@gmail.com

ABSTRAK

Kevin Azwega, Pengembangan Sistem Deteksi *God Class* dan *Brain Class Code Smell*

Pembimbing: Adam Hendra Brata, S.Kom., M.T., M.Sc dan Eriq Muhammad Adams Jonemaro, S.T., M.Kom

Perkembangan perangkat lunak tidak dapat dipisahkan dari kehidupan manusia. Hampir semua aspek memerlukan perangkat lunak. Hal ini membuat manusia terus mengembangkan perangkat lunak yang memiliki banyak fitur. Perangkat lunak yang memiliki banyak fitur memiliki desain yang kompleks. Desain perangkat lunak yang kompleks membuat struktur kode program yang sulit dimengerti. Salah satu upaya untuk memperbaiki hal tersebut dilakukan perubahan struktur kode program agar lebih mudah dipahami dan dilakukan perawatan biasa disebut *refactoring*. Untuk melakukan *refactoring* dilakukan proses pendeteksian *code smell* terlebih dahulu. *Code smell* merupakan kecacatan struktur kode program yang sulit dimengerti menyebabkan masalah perawatan sistem perangkat lunak. Pada penelitian ini membahas *code smell god class* dan *brain class*. *God class* merupakan sebuah kecacatan kode program membuat suatu kelas menjadi pusat kecerdasan sistem. Kelas tersebut melakukan sebagian besar pekerjaan dan mendelegasikan sedikit pekerjaan tersebut ke kelas lain. Sedangkan *brain Class* merupakan kecacatan struktur kode program yang membuat suatu kelas terdapat satu atau lebih *method* yang memiliki kompleksitas operasi yang tinggi sehingga menjadi pusat kecerdasan dari sistem. Deteksi *code smell* dapat dilakukan secara manual namun hal tersebut dapat menggunakan waktu yang lama jika mendeteksi ratusan kode pada sebuah sistem perangkat lunak. Sehingga dibutuhkan kakas bantu untuk mendeteksi *code smell god class* dan *brain class* secara otomatis. Untuk mengurangi usaha programer dalam mengatasi *code smell* terutama *god class* dan *brain class*. Sistem deteksi ini menggunakan perhitungan *software metrics* sebagai alat ukur untuk klasifikasi *code smell god class* dan *brain class.* Nilai perhitungan *software metrics* tersebut dijadikan dasar mendeteksi *code smell god class* dan *brain class*. Sistem ini diuji dengan pengujian unit menggunakan metode *whitebox*, pengujian integrasi dengan metode *bottom up integration* dan pengujian validasi dengan menggunakan metode *blackbox*. Sistem ini mampu mendeteksi *code smell god class* dan *brain class* dalam waktu kurang dari satu detik dan memiliki tingkat akurasi deteksi seratus persen.

Kata kunci: *code smell*, *god class*, *brain class*, *software metrics*.

ABSTRACT

Kevin Azwega, *Development of the God Class and Brain Class Code Smell Detection System*

Adviser: Adam Hendra Brata, S.Kom., M.T., M.Sc and Eriq Muhammad Adams Jonemaro, S.T., M.Kom

Software development is inseparable from human life. almost all aspect needs a software system. It makes people continue developing software that has many features. The software has a design complex that causes structure code difficult to understand. The solution for fixed it is refactoring. Before do refactoring is a detection code smell. A code smell is defect code in object-oriented programming it can cause a problem in the maintenance software system. Defect code cause class that tends to centralize the intelligence of the software system it means god class and brain class code smells. God class occurs because of functional complexity in class high, cohesion class high, and to many use data from the other class. Brain class occurs because it has a brain method in a software system. Detection code smell can be done manually, but it has a long time if detection hundreds code in a software system. That be the required tool for detection god class and brain class code smells automatic. to reduce programmer effort in overcoming god class and brain class code smells problem. This system uses software metrics to measure classification god class and brain class. The calculation value of the metrics software is used for detecting code smell God class and brain class. This system has been tested by unit test it used whitebox method, integration testing used bottom up integration method and validation testing used blackbox method. This system can be operated detection god class and brain class code smell less than one second and have detection accuracy one hundred percent.

***Keyword**: code smell, god class, brain class, software metrics.*

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar belakang

Perkembangan perangkat lunak tidak dapat dipisahkan dari kehidupan manusia. Hampir semua aspek memerlukan perangkat lunak. Hal ini membuat manusia terus mengembangkan perangkat lunak yang memiliki banyak fitur. Perangkat lunak yang memiliki banyak fitur memiliki desain yang kompleks. Desain perangkat lunak yang kompleks membuat struktur kode program yang sulit dimengerti. Yang mengakibatkan masalah pada proses perawatan dimasa mendatang (Aristyagama, 2016). Salah satu upaya untuk memperbaiki hal tersebut dilakukan perubahan struktur kode program agar lebih mudah dipahami dan dilakukan perawatan biasa disebut *refactoring*.

Untuk melakukan *refactoring* dilakukan proses pendeteksian *code smell* terlebih dahulu. *Code smell* merupakan kecacatan struktur kode program yang sulit dimengerti menyebabkan masalah perawatan sistem perangkat lunak (Olbrich, Cruzes dan Sjoøberg, 2010). Pada penelitian ini akan dilakukan penelitian tentang pendeteksian *code smells* jenis *god class dan brain class*. *God class* merupakan sebuah kecacatan kode program membuat suatu kelas menjadi pusat kecerdasan sistem. Kelas tersebut melakukan sebagian besar pekerjaan dan mendelegasikan sedikit pekerjaan tersebut ke kelas lain (Lanza dan Marinescu, 2006). Tiga karakteristik *god class*, yaitu kompleksitas fungsional sebuah kelas yang tinggi, kohesi kelas yang tinggi, dan terlalu banyak menggunakan data dari kelas lain (Lanza dan Marinescu, 2006).

Brain Class merupakan kecacatan struktur kode program yang membuat suatu kelas terdapat satu atau lebih *method* yang memiliki kompleksitas operasi yang tinggi sehingga menjadi pusat kecerdasan dari sistem. Pengertian tersebut memiliki kemiripan dengan pengertian *code smell* jenis *god class*. Perbedaan *brain class* dengan *god class* yaitu *brain class* tidak banyak mengakses data dari kelas lain dan lebih kohesif (Lanza dan Marinescu, 2006). Aturan deteksi dari *brain class* yaitu memiliki *method* yang dipengaruhi oleh *brain method*. Memiliki nilai *line of code* yang besar, lebih kohesif dan sangat kompleks. Jika sebuah kelas "monster" memiliki ukuran *line of code* yang besar serta memiliki nilai kompleksitas yang tinggi dan memiliki satu *method* yang dipengaruhi oleh *brain method*, maka dapat dianggap bahwa kelas tersebut terdapat *brain class* (Lanza dan Marinescu, 2006).

Proses pendeteksian *code smell* jenis *god class* dan *brain class* dapat dilakukan secara manual yaitu dengan menganalisis setiap struktur kode kemudian dilakukan pengukuran metriks. Hal ini dapat menyulitkan pengembang karena proses pendeteksian memerlukan waktu lama terlebih jika mendeteksi ratusan kode dalam sebuah program. Proses pendeteksian lebih lama dapat menyebabkan penggunaan biaya yang cukup besar dan persepsi dari setiap programer berbeda beda sehingga menyulitkan dalam proses pendeteksian jika dilakukan secara manual. Dalam hal ini lebih disarankan untuk menggunakan kakas bantu yang dapat mendeteksi sebuah *code smell*. Sehingga pada penelitian

ini akan mengusulkan sebuah sistem pendeteksian *code smells* jenis *god class* dan *brain class* secara otomatis dengan menggunakan perhitungan *software metrics* untuk mendapatkan hasil deteksi *god class* dan *brain class*. Dengan gambaran umumnya yaitu dengan membaca *source code* (*parsing*) dengan menggunakan *library* javalang pada setiap file pada sebuah program kemudian dilakukan kalkulasi *metrics,* beberapa *metrics* menggunakan kalkulasi dengan *library* lizard lalu disandingkan dengan nilai *threshold* sehingga didapatkan hasil pendeteksian apakah program atau aplikasi yang di periksa terdapat *code smell* jenis *god class* dan *brain class* atau tidak. Sehingga dapat meminimalisir dampak dari *code smell,* memudahkan proses *maintenance* program tidak menggunakan biaya yang cukup besar serta mengurangi usaha *developer*.

Penelitian yang dilakukan oleh Steffen M. Olbrich dan Daniela S. Cruzes (2010) yang menyelidiki tentang bahaya dari *code smell* jenis *god class* dan *brain class* data yang didetski merupakan tiga sistem perangkat lunak *open source* yaitu lucene, xerces dan log4j lalu dilakukan analisis. Dari hasil analisis dapat disimpulkan bahwa selama persentase *god class* dan *brain class* tidak terlalu besar. Terdapatnya god class dan brain class mungkin bermanfaat bagi sistem. Namun sistem tersebut perlu diselidiki lebih lanjut.

Penelitian ini terbatas pada pendeteksian *code smell god class* dan *brain class.* Dengan menggunakan aturan dan nilai ambang batas (*threshold)* yang berasal dari penelitian yang dilakukan oleh Steffen M. Olbrich dan Daniela S. Cruzes (2010). Hasil dari pendeteksian pada penelitian ini dapat dijadikan pengembang sistem sebagai bahan pertimbangan pada proses perbaikan sistem. Dengan adanya kontribusi penelitian ini diharapkan terdapat studi-studi selanjutnya tentang pendeteksian *code smell* terutama jenis *god class* dan *brain class*.

1.2 Rumusan masalah

Berdasarkan penjelasan latar belakang diatas, maka diperoleh beberapa rumusan masalah, sebagai berikut:

1. Bagaimana hasil rekayasa kebutuhan dalam membangun sistem pendeteksian *code smells* jenis *god class* dan *brain class*?
2. Bagaimana hasil perancangan dalam membangun sistem pendeteksian *code smell* jenis *god class* dan *brain class*?
3. Bagaimana hasil implementasi dalam membangun sistem pendeteksian *code smell* jenis *god class* dan *brain class*?
4. Bagaimana hasil pengujian dalam membangun sistem pendeteksian *code smell* jenis *god class* dan *brain class*?

1.3 Tujuan

Tujuan yang akan dicapai dalam penelitian ini terdapat beberapa point yaitu:

1. Pembangunan sistem deteksi *code smell* jenis *god class* dan *brain class* dibangun sesuai rekayasa kebutuhan sistem.
2. Pembangunan sistem deteksi *code smell* jenis *god class* dan *brain class* dibangun sesuai dengan perancangan sistem.
3. Pembangunan sistem deteksi *code smell* jenis *god class* dan *brain class* dapat diimplementasikan dengan baik.
4. Hasil penelitian ini diharapkan memberikan kemudahan dalam memahami *code smell* jenis *god class* dan *brain class*.

1.4 Manfaat

Pada penelitian ini diuraikan beberapa manfaat didapat dari penelitian sebagai berikut:

1. Hasil dari sistem ini digunakan untuk mendeteksi *code smell* jenis *god class* dan *brain class* pada sistem yang akan diidentifikasi
2. Pada skripsi ini memberikan pemahaman tentang *code smell* terutama jenis *god class* dan *brain class* kepada pembaca.
3. Hasil dari skripsi ini dapat meminimaisir dampak *code smell* sehingga sistem lebih mudah untuk di *maintenance*.
4. Pada skripsi ini dapat sebagai bahan untuk pembelajaran tentang *code smell* jenis *god Class* dan *brain class*.

1.5 Batasan masalah

Batasan masalah pada pembuatan sistem pendeteksian *code smell* jenis *god class* dan *brain class* adalah sebagai berikut:

1. *Source code* (kode sumber) yang akan dianalisis dan dihitung metriksnya merupakan kode sumber yang berasal dari *Java*.
2. Proses pembuatan sistem menggunakan bahasa pemrogaman phyton.
3. Bila ada dua atau lebih kelas pada berkas kode program maka dianggap satu kelas karena penelitian ini tidak mendeteksi pada level *inner class*.
4. Penelitian ini menggunakan jurnal penelitian Olbrich et all (2010) untuk aturan dan nilai ambang batas atau *threshold* pada pendeteksian *code smell god class* dan *brain class*.

1.6 Sistematika pembahasan

Sistematika penulisan laporan penelitian pengembangan sistem deteksi *code smell god class* dan *brain class* adalah sebagai berikut:

BAB 1 Pendahuluan

Bab ini berisi tentang uraian mengenai latar belakang, rumusan masalah, tujuan, manfaat, batasan masalah dan sistematika pembahasan.

BAB 2 Landasan Kepustakaan

Bab ini menjelaskan tentang dasar teori pendukung yang melandasi penelitian sistem deteksi *code smell* jenis *god class* dan *brain class*.

BAB 3 Metode Penelitian

Metode penelitian menjelaskan tentang uraian prosedur penelitian, untuk melakukan rekayasa kebutuhan, merancang dan mengimpelentasi *code smell* jenis *god class* dan *brain class*.

BAB 4 Rekayasa Kebutuhan

Bab ini menguraikan tentang analisis kebutuhan mengenai sistem deteksi *code smell* jenis *god class* dan *brain class*.

BAB 5 Perancangan dan Implementasi Sistem

Bab ini membahas tentang perancangan dan implementasi untuk sistem deteksi *code smell* jenis *god class* dan *brain class*.

BAB 6 Pengujian Sistem

Bab ini berisi tentang pegujian unit, pengujian validasi pada sistem pendeteksi *code smell* jenis *god class* dan *brain class*.

BAB 7 Penutup

Pada bab ini menjelaskan tentang kesimpulan dari hasil penelitian ini serta saran untuk pengembangan sistem lebih lanjut.

BAB 2 LANDASAN KEPUSTAKAAN

2.1 Kajian Pustaka

Dengan dilakukannya proses analisis dan kajian berdasarkan penelitian sebelumnya, Kajian pustaka yang dijadikan rujukan sebagai pendukung penelitian sistem deteksi *code smell god class* dan *brain class*. Terdapat empat penelitian yang menjadi fokus utama pembahasan. Sistem pendeteksian *code smell* jenis *god class* dan *brain class* berfokus pada pendeteksian *god class* dan *brain class*. Hasil dari pendeteksian pada penelitian ini dapat dijadikan pengembang sistem sebagai bahan pertimbangan untuk proses perbaikan sistem.

Penelitian pertama yang dilakukan oleh Martin Fowler (1999) yang berjudul "*Refactoring Improving the design of existing code*" membahas mengenai 22 tipe dari *code smell* dan dampak yang ditimbulkan *code smell* terhadap sistem. Hubungan penelitian pertama dengan penelitian ini ialah konsep dari *code smell*.

Penelitian kedua yang dilakukan oleh Michele dan Marinescu (2006) di dalam bukunya yang berjudul "Object-Oriented Metrics in Practice" dijelaskan bahwa menggunakan *software metrics* untuk karakterisasi, eveluasi, dan memperbaiki desain. Hubungan penelitian kedua dengan penelitian ini ialah konsep dari *code smell god class* dan *brain class*. dan konsep dari *code smell* jenis *god class* dan *brain class*.

Penelitian ketiga yang dilakukan Thanis Paiva, Amanda Damasceno et all (2017) yang berjudul "*On the Evaluation of Code Smells and Detection Tools*" tentang membandingkan nilai akurasi dari deteksi *code smell god class, god method* dan *feature envy* secara manual dengan deteksi melalui kakas bantu JDeodorant, PMD, JSpIRIT, dan inFusion. Kakas bantu ini menggunakan teknik deteksi *software metrics* kecuali JDeodorant. JDeodorant menggunakan teknik deteksi *refractoring opportunities*. Hasilnya ketiga kakas bantu tersebut mempunyai nilai akurasi yang lebih tinggi dibandingkan dengan JDeodorant yang menggunakan teknik deteksi *refractoring opportunities*. Hubungan penelitian ketiga dengan penelitian ini ialah bahan pertimbangan penggunaan *software metrics* pada penelitian ini.

Penelitian keempat yang dilakukan oleh Olbrich, Cruzes and Sjoøberg (2010) yang berjudul "*Are all Code Smell Harmful? A Study of God Classes and Brain Classes in the Evolution of three Open Source System*" menjelaskan bahwa pendeteksian *code smell* jenis *god class* dan *brain class* dapat dideteksi dengan metode *software metrics*. *Software metrics* yang digunakan untuk mendeteksi *code smell* jenis *god class* ada tiga yaitu *access to foreign data*, *tight class cohesion*, dan *weighted method count*. Lalu *software metrics* untuk mendeteksi code smell *brain class* yaitu nilai dari *code smell god class*, *line of code*, *weight method count*, *tight class cohesion*, nilai dari *brain method*. Hubungan penelitian keempat dengan penelitian ini ialah penggunaan aturan untuk mendeteksi *code smell god class* dan *brain class*. Dan menggunakan salah satu data uji yang digunakan pada penelitian ini yaitu apache lucene versi 2.9.2.

2.2 Rekayasa Perangkat Lunak

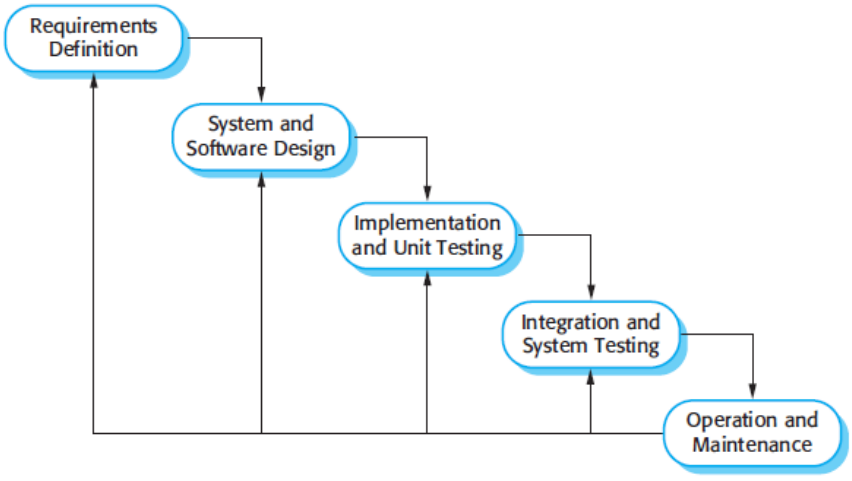
Rekayasa perangkat lunak adalah suatu disiplin ilmu yang mempelajari tentang semua aspek pembuatan perangkat lunak (Sommerville, 2011). Ilmu rekayasa perangkat lunak menjelaskan proses pengembangan perangkat lunak mulai dari spesifikasi sistem sampai proses perawatan setelah didistribusikan kepada pengguna. Produk perangkat lunak yang dihasilkan harus memiliki karakteristik seperti *reliable, efficient, testable, portable, usable, maintenanceable, reusable dan correct* (Leach, 2016). Pada perangkat lunak karakteristik ini didefinisikan pada kebutuhan non-fungsional.

2.3 Pengembangan Perangkat Lunak

Pengembangan perangkat lunak merupakan suatu proses konversi sistem spesifikasi menjadi sistem yang dapat dieksekusi (Sommerville, 2011). Dapat dikatakan bahwa pengembangan perangkat lunak adalah proses mengolah analisis kebutuhan menjadi sebuah *software*. Untuk membangun perangkat lunak berkualitas tinggi dan tepat waktu maka diperlukan proses pengembangan perangkat lunak. Pada penelitian ini, menggunakan siklus hidup pengembangan perangkat lunak *waterfall* pada penelitian ini menggunakan pendekatan berorientasi objek.

2.3.1 Model Waterfall

Pengembangan perangkat lunak memiliki pendekatan model *waterfall* yang dilakukan secara sistematis yaitu menyelesaikan satu fase lalu lanjut ke fase berikutnya secara berurutan (Pressman, 2010). Pendekatan model *waterfall* terdiri dari fase Model siklus hidup pengembangan perangkat lunak *waterfall* terdiri atas fase *requirement definition, system and software design, implementation and unit testing, integration and system testing, operation and maintenance*. Ilustrasi model *waterfall* ditujukan pada Gambar 2.1



Gambar 2.1 Waterfall Model

(Sumber: Sommerville, 2011)

1. Requirements Definition

   Fase ini dilakukan pendefinisian kebutuhuhan dan analisis sistem. Peneliti menggali kebutuhan dengan cara pengkajian pustaka, observasi dan *interview*. Menghasilkan infomasi layanan yang akan sistem sediakan. Dari kebutuhan tersebut di spesifikasi lebih detail untuk dilakukan proses perancangan sistem.

2. *System and Software Design*

   Fase ini dilakukan perancangan sistem yaitu menafsirkan kebutuhan sistem kedalam perancangan sistem perangkat lunak berdasarkan proses yang dilakukan pada tahap sebelumnya. Sehingga menghasilkan struktur data, representasi antarmuka sistem, arsitektur sistem dan perancangan algoritme. Hasil dari fase ini akan menjadi acuan untuk implementasi sistem.

3. *Implementation and Unit Testing*

   Fase ini dilakukan implementasi sistem yaitu menafsirkan perancangan sistem kedalam bahasa yang dimengerti oleh komputer. Selanjutnya komponen metode dan kelas dari kode program tersebut dilakukan pengujian unit. Pengujian unit melibatkan ver*if*ikasi setiap unit apakah sistem telah sesuai dengan spes*if*ikasi.

4. *Integration and System Testing*

   Fase ini dilakukan pengujian integrasi dari pengujian unit untuk menjamin sistem telah sesuai dengan kebutuhan. Sistem testing merupakan proses pengujian dengan menjalankan seluruh elemen sistem yang telah dikembangkan.

5. *Operation and Maintenance*

   Fase ini perangkat lunak telah didistribusikan kepada *user*. Proses *main*tenance merupakan proses perbaikan dan pemeliharaan sistem untuk meningkatkan kualitas perangkat lunak. Untuk memenuhi kebutuhan *user* biasanya pada tahap ini dilakukan penambahan fitur pada aplikasi.

2.4 Pendekatan Berorientasi Objek

Pendekatan dalam pengembangan perangkat lunak ada beberapa jenis salah satunya pendekatan berorientasi objek. Pendekatan berorientasi objek merupakan paradigma pembangunan yang mencangkup empat kerangka kerja yaitu *object orientied analysis, object orientied design, object orientied programing* dan *object orientied testing* (Pressman, 2010). Pada pendekatan berorientasi objek proses analisis kebutuhan sistem pada penelitian ini menguunakan metode *object orientied analysis* dengan menggunakan metode ini kebutuhan dianalisis dari prespektif *class* dan *object*. Selanjutnya perancangan

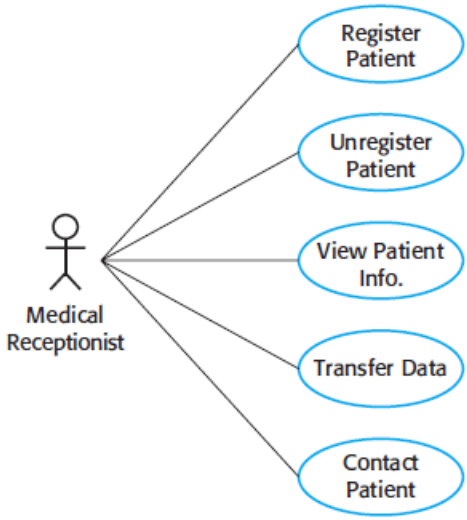
sistem pada penelitian ini menggunakan metode *object orientied design*. Dalam tahap ini dapat terlihat desain pada sistem yang akan dibuat. Setelah kebutuhan dan perancangan diperoleh selanjutnya implementasi ke dalam kode program dengan metode *object orientied programing*. Pada tahap ini sistem sudah dapat berjalan namun masih perlu untuk diuji agar sistem dapat berjalan dengan baik. Pengujian menggunakan *object orientied testing* yaitu pengujian unit, pengujian integrasi dan pengujian validasi (Pressman, 2010).

2.4.1 Permodelan Berorientasi Objek

Hasil kebutuhan yang diperoleh dari proses *object object orientied analysis* dan hasil perancangan yang diperoleh dari proses *object orientied design* divisualkan menggunakan *unified modeling language*. *Unified Modeling Language* adalah bahasa visual yang digunakan untuk memodelkan proses perangkat lunak yang berdasarkan desain arsitektur sistem dari sudut pandang pengguna (Pressman, 2001). UML digunakan dalam pengembangan sistem perangkat lunak yang menggunakan pendekatan berorientasi objek (Kurniawan, 2018). UML menyediakan banyak sekali diagram yang diperlukan untuk menjelaskan sistem yang sedang dikembangkan baik dari aspek statis maupun aspek dinamis. Tujuan dari UML adalah untuk memodelkan setiap proyek pengembangan sistem mulai dari tahap analisis sampai tahap implementasi. Di dalam *Unified Modeling Language* terdiri dari beberapa diagram, diantaranya:

2.4.1.1 Use case

Use case merupakan gambaran bagaimana pernggguna berinteraksi dengan sistem (Pressman, 2010). *Use case* juga salah satu diagram penting yang digunakan untuk mengilustrasikan kebutuhan (*requirements*) dari sebuah sistem. Setiap *use case* menyatakan spesifikasi perilaku fungsionalitas dari sistem yang sedang dijelaskan yang memang dibutuhkan oleh aktor untuk memenuhi tujuannya (Kurniawan, 2018). Terdapat 2 elemen penting yang harus digambarkan pada *use case*, diagram yaitu aktor dan *use case*. Tujuan dari *use case* diagram ini agar pembaca memahami fungsionalitas dan memahami gambaran umum dari sistem. Contoh gambar mengenai *use case diagram* ditunjukan pada gambar 2.2.



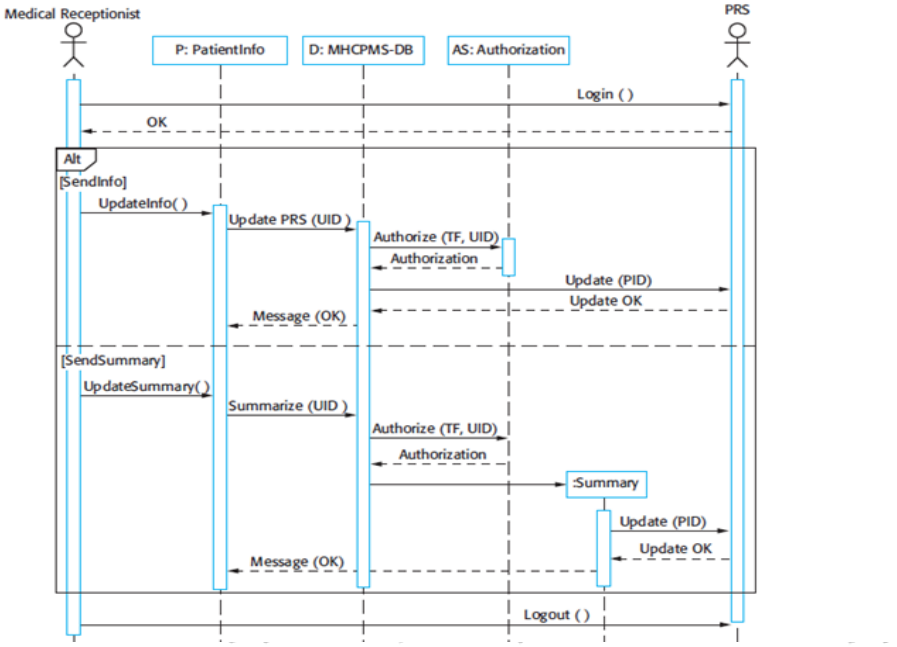
Gambar 2.2 *Use case Diagram*
Sumber : (Sommerville, 2011)

2.4.1.2 Use case Scenario

Use case scenario merupakan penjelasan secara tekstual dari sekumpulan skenario interaksi. Setiap skenario mendeskrepsikan urutan aksi/langkah yang dilakukan aktor ketika beriinteraksi dengan sistem, baik yang berhasil maupun tidak berhasil (Kurniawan, 2018). Misalnya skenario berhasil transaksi dengan mesin atm atau skenario gagal transaksi dengan mesin atm. Di dalam *use case* scenario terdapat. Aktor, Kondisi Awal, Skenario Utama (*Main flow*), proses altenat*if* (*Altenatif flow*), dan Kondisi Akhir (*post condition)*. Pembuatan *use case scenario* berdasarkan pada *use case diagram*.

2.4.1.3 Sequence diagram

Sequence diagram merupakan hubungan dinamis antar objek dalam urutan waktu (Pressman, 2010). Komunikasi suatu objek dengan objek lainya menggunakan pesan. Pesan (*message*) ini dilambangkan sebuah anak panah kanan ke kiri dari suatu objek ke objek lain. Isi pesan dari anak panah ini merupakan nama *method* biasanya lengkap dengan parameter, dan nilai balikan dari *method* tersebut. Garis vertikal pada *sequence diagram* merupakan waktu aktivitas suatu objek. Apabila objek telah menjalankan operasinya dan menghasilkan nilai balikan disebut dengan *return message*. Dilambangkan anak panak dari kiri ke kanan. Objek pada *sequence diagram* dilambangkan dengan persegi panjang atau simbol lainya seperti *entity*, *boundary*, dan *controll*. *Entity* merupakan kumpulan *class* yang digunakan untuk menyusun basis data. *Boundary* merupakan kumpulan kelas yang menjadi antarmuka sistem. *Controll* merupakan kumpulan *class* yang mengatur logika alur sistem. Aktor merupakan pengguna yang berinteraksi dengan perangkat lunak yang akan dibangun. Contoh gambar mengenai *sequence diagram* ditunjukan pada Gambar 2.3.

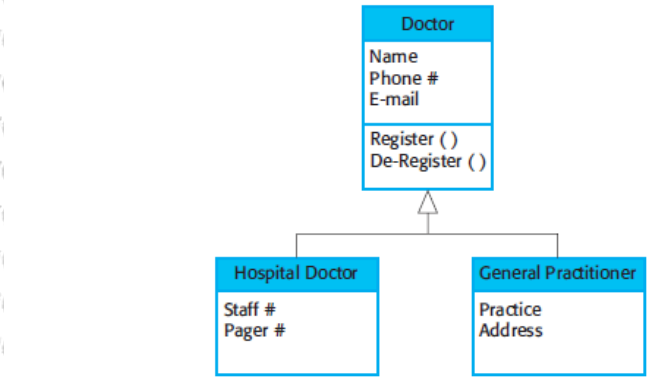


Gambar 2.3 *Sequence diagram*

Sumber : (Sommerville, 2011)

2.4.1.4 Class diagram

Class diagram merupakan gambar tampilan statis atau struktural dari suatu sistem (Pressman, 2010). Komponen utama pada *class diagram* ialah persegi panjang ini merupakan simbol yang mewakili kelas. Setiap persegi panjang dibagi menjadi dua pada bagian atas merupakan tempat untuk *attribute* dan bagian bawah untuk *operation*. *Operation* merupakan objek dari suatu kelas biasanya diimplementasikan sebagai *method* kelas. *Attributes* merupakan sesuatu yang dapat diketahui oleh objek kelas. *Attributes* diimplementasi sebagai *field* pada kelas. *Class diagram* mempunyai hubungan suatu kelas dengan kelas lain. Contoh hubungan antar kelas yaitu hubungan *subclass* ke *superclass* dilambangkan sebuah garis memiliki anak panah. Adapun hubungan lain seperti *association* yaitu menggambarkan hubungan umum antara dua *class*. Hubungan *composition* yaitu hubungan dimana sebuah *class* tidak bisa berdeiri sendiri dan harus bergantung dengan *class* lain. Hubungan *aggregation* merupakan hubungan antar dua *class* dimana sebuah *class* dapat berdiri sendiri meskipun bagian dari *class* lain. Hubungan *subclass* ke *superclass* disebut *generalisasi*. Contoh gambar mengenai *class diagram* ditunjukan pada Gambar 2.4.



Gambar 2.4 *Class diagram*
Sumber : (Sommerville, 2011)

2.4.2 Pemrograman Berorientasi Objek

Pemograman berbasis objek merupakan metode untuk mengimplementasikan perancangan dan desain yang telah didefinisikan sebelumnya kedalam kode program. Kode program tersebut telah dikelompokan sebagai objek yang masing masing mewakili sebuah *instance* (Baesens, Backiel, & Broucke 2015). Pada penelitian ini menggunakan bahasa pemrograman python untuk implementasi aplikasi.

2.4.2.1 Python

Python merupakan bahasa pemograman interpretatif yang berfokus pada *code readability*. Python bahasa pemograman multi platform yang dapat digunakan pada berbagai macam sistem operasi. Bahasa pemograman python mudah untuk dipelajari dan fleksibel. Keunggulan utama dari suatu bahasa pemograman python tidak membutuhkan pendeklarasian variabel, sehingga lebih fleksibel penggunaaannya. Berdasarkan *indexing* yang dilakukan TIOBE (2020). Python merupakan bahasa pemograman yang paling terpopuler peringkat ketiga dibawah satu tingkat peringkat dengan bahasa pemograman java. Memiliki rating 10,11% pada bulan Maret 2020 dibandingkan bahasa pemograman lain. Bahasa pemograman python menjadi populer dikarenakan banyak programer yang menggunakan bahasa python hal ini membuat bahasa python lebih banyak dilakukan pengembangan dan perbaikan sehingga membuat bahasa pemograman python lebih handal, efisien, dan mudah untuk diimplementasi.

2.4.3 Pengujian Pada Pendekatan Berorientasi Objek

Pengujian merupakan proses untuk menemukan sebuah kesalahan pada sistem yang dibuat secara tidak sengaja pada tahap perancangan dan pembangunan sistem (Pressman, 2010). Pengujian yang digunakan pada penelitian ini yaitu pengujian unit untuk menguji komponen atau unit pada

sebuah program. Selanjutnya pengujian integrasi untuk menguji komponen atau unit yang berhubungan dapat berjalan dengan baik. Pengujian validasi untuk memvalidasi fungsi sistem terhadap kebutuhan pengguna. Menurut Pressman (2010) strategi yang dapat digunakan dalam pengujian pada berorientasi objek sebagai berikut:

2.4.3.1 Pengujian Unit

Pengujian pengujian unit adalah pengujian unit terkecil dari komponen atau modul perangkat lunak (Pressman, 2010). Pada pemograman berorientasi objek pengujian dilakukan pada level terkecil yaitu pada tingkat kelas atau *method*. Pengujian pada level terkecil untuk mencegah pengaruh dari kompenen lainnya pada sistem, pengujian unit dilakukan secara terpisah dari komponen atau modul lainnya. Pengujian unit menggunakan metode *whitebox testing* dengan teknik *basis path testing*. *Whitebox* merupakan metode pengujian struktur kode program untuk mengetahui kesalahan yang terjadi pada unit terkecil. *Basis path testing* merupakan salah satu teknik dari metode *whitebox* dengan menghitung *control flow* dari suatu modul (Pressman, 2010).

2.4.3.2 Pengujian Integrasi

Pengujian integrasi merupakan teknik sistematis untuk menguji dua komponen yang saling berhubungan untuk menemukan kesalahan dan menjamin komponen yang saling berhubungan dapat berjalan dengan baik (Pressman, 2010). Terdapat beberapa teknik pengujian integrasi.

Pada penelitian ini menggunakan teknik *bottom-up integration*. Tenik pengujian integrasi dimana komponen komponen tingkat bawah akan diuji terlebih dahulu kemudian komponen komponen yang tingkat tinggi diuji. Jika komponen yang tingkat tinggi belum selesai dibangun maka pengujian menggunakan *driver* (Pressman, 2010). *Driver* merupakan alat bantu pengujian untuk mensimulasikan komponen tingkat tinggi yang belum selesai dibangun.

2.4.3.3 Pengujian Validasi

Pengujian validasi merupakan pengujian untuk mengetahui apakah perangkat lunak telah sesuai dengan kebutuhannya (Sommerville, 2011). Pengujian validasi harus menguji semua kebutuhan fungsional pada perangkat lunak. Pada penelitian ini pengujian validasi menggunakan metode *blackbox testing*. *Blackbox* merupakan sebuah metode pengujian yang berfokus pada pengujian kebutuhan fungsional sistem (Pressman, 2010).

Pengujian validasi kebutuhan non fungsional pada penelitian ini menggunakan pengujian peforma dan pengujian akurasi. Pengujian peforma merupakan jenis pengujian untuk menentukan dan memastikan bagaimana kinerja perangkat lunak dalam respons dan stabilitas dibawah beban kerja yang diharapkan (Permatasari, 2020). Fokus dari pengujian peforma pada penelitian ini yaitu *speed* untuk mengukur seberapa cepat aplikasi bekerja. Pengujian akurasi merupakan tingkat kedekatan pengukuran kuantitas yang dilakukan oleh sistem dengan pengukuran kuantitas secara manual (Rubal dan Arya, 2017).

Pengujian validasi dilakukan sebelum *software* dipasang dan digunakan. Pengujian validasi dilakukan dengan cara membuat *test case* yang dapat diuji pada setiap fungsi serta dirancang untuk menemukan kesalahan pada level validasi sehingga setiap *test case* diharapkan dapat menemukan kesalahan yang banyak dengan waktu minimal. Pengujian performa akan dilakukan dengan mengamati waktu yang dibutuhkan untuk mendeteksi *code smell*. Pada pengujian akurasi akan dilkukan dengan membandingkan nilai perhitungan *software metrics* secara manual dengan nilai perhitungan *software metrics* yang dilakukan oleh sistem.

2.5 Code smell

Code smell merupakan kecacatan struktur kode program yang sulit dimengerti menyebabkan masalah perawatan sistem perangkat lunak (Olbrich, Cruzes dan Sjoøberg, 2010). Menurut Martin dan Beck (1999) terdapat 22 kategori *code smell* yang sering ada pada program. Kategori tersebut diantaranya *long method*, *shotgun surgery*, *feature envy*, *large class* dan lain sebagainya.

Code smell bukan merupakan sebuah *bug error, code smell* tidak mencegah sebuah program untuk berfungsi. Namun dampak dari *code smell* dapat memperlambat proses pengembangan, meningkatkan resiko *bug*, celah keamanan dan kegagalan sistem. *Code smell* dapat diminimaisir dengan mendeteksi keberadaannya pada kode program dan melakukan *refactoring* atau mengganti struktur kode program.

Pada penelitian ini mendeteksi *code smell* yang termasuk kedalam kategori *large class* yaitu jenis *god class dan brain class*. Pendekteksian *god class* dan *brain class* dilakukan dengan cara parsing atau membaca kode sumber setelah itu kalkulasi *software matrics*. Setelah didapatkan hasil kalkulasi lalu dilakukan perbandingan nilai kalkulasi dengan nilai ambang batas atau *threshold* yang sudah ditentukan. Dibawah ini merupakan penjabaran mengenai aturan dan nilai *threshold* deteksi *code smell* dan dasar teori dari *god class* dan *brain class* yang di lakukan pada penelitian ini:

2.5.1 God Class

God class merupakan sebuah kecacatan kode program membuat suatu kelas menjadi pusat kecerdasan sistem. Kelas tersebut melakukan sebagian besar pekerjaan dan mendelegasikan sedikit pekerjaan tersebut ke kelas lain c. *God class* termasuk dalam kategori *large class*. Tiga karakteristik *god class*, yaitu kompleksitas fungsional sebuah kelas yang tinggi, kohesi *inner class* yang tinggi, dan terlalu banyak menggunakan data dari kelas lain (Lanza dan Marinescu, 2006). Apabila dilakukan *refactoring*, maka pada umumnya god class ini paling banyak mengalami perubahan jika dibandingkan dengan class yang lain. Untuk mendeteksi *god class* menggunakan aturan dan nilai *threshold*. Pada penelitian ini aturan dan nilai *threshold* berasal dari penelitian Olbrich et all (2010). Aturan *god class* dapat dilihat pada persamaan 2.1.

$$\boldsymbol{God\ Class}(c) = \begin{cases} \mathbf{1}, ((\boldsymbol{WMC}\ (c) \geq \mathbf{47}) \wedge (\boldsymbol{TCC}\ (c) < \mathbf{0.3}) \wedge (\boldsymbol{ATFD}\ (c) > \mathbf{5})) \\ \mathbf{0}, \boldsymbol{else} \end{cases} \quad 2.1$$

Keterangan:

C = Kelas

WMC = *Weighted method count*

TCC = *Tight class cohesion*

ATFD = *Access to foreign data*

2.5.2 Brain Class

Brain Class merupakan kecacatan struktur kode program yang membuat suatu kelas terdapat satu atau lebih *method* yang memiliki kompleksitas operasi yang tinggi sehingga menjadi pusat kecerdasan dari sistem (Lanza dan Marinescu, 2006). Perngertian tersebut memiliki kemiripan dengan pengertian *code smell* jenis *god class*. Perbedaan *brain class* dengan *god class* yaitu *brain class* tidak banyak mengakses data dari kelas lain dan lebih kohesif. Aturan deteksi dari *brain class* yaitu memiliki beberapa *method* yang dipengaruhi oleh *brain method*. Memiliki nilai LOC (*line of code*) yang besar, sedikit kohesif dan sangat kompleks. Jika sebuah kelas "monster" memiliki ukuran *line of code* (LOC) yang besar serta memiliki nilai kompleksitas yang tinggi (WMC) dan memiliki satu *method* yang dipengaruhi oleh *brain method*, maka dapat dianggap bahwa kelas tersebut terdapat *brain class* (Lanza dan Marinescu, 2006). Untuk mendeteksi *brain class* menggunakan aturan dan nilai *threshold*. Pada penelitian ini aturan dan nilai *threshold* berasal dari penelitian Olbrich et all (2010). Aturan *brain class* dapat dilihat pada persamaan 2.2.

$$\boldsymbol{BC}\ (c) = \begin{cases} \mathbf{1}, \boldsymbol{GC}(c) \wedge ((\boldsymbol{WMC}(c) \geq \mathbf{47}) \wedge (\boldsymbol{TCC}(c) < \mathbf{0.5})) \wedge ((\boldsymbol{NBM}(c) \equiv \mathbf{1}) \wedge (\boldsymbol{LOC}(c) \geq \mathbf{2.197})) \\ \quad \vee ((\boldsymbol{NBM}(c) > \mathbf{1}) \wedge (\boldsymbol{LOC}(c) \geq \mathbf{197})) \wedge (\boldsymbol{WMC}(c) \geq \mathbf{2.47}))) \\ \mathbf{0}, \boldsymbol{else} \end{cases} \quad 2.2$$

Keterangan :

BC = *Brain Class*

C = Kelas

GC = *God class*

WMC = *Weighted method count*

TCC = *Tight class cohesion*

NBM = *Number of brain method*

LOC = *Line of code*

2.5.3 Brain Method

Brain Method merupakan salah satu bagian dari *Brain class*. *Brain method* merupakan *method* yang memiliki kompleksitas operasi yang tinggi serta menjadi pusat kecerdasan dari sebuah sistem (Lanza dan Marinescu, 2006). Jika sebuah *class* terdapat lebih dari atau sama dengan satu *brain method* maka *class* tersebut terindikasi sebagai *brain class*. *Brain method* menggunakan *software metrics line of code in method* sebagai tolak ukur. Pada penelitian ini aturan dan nilai *threshold brain method* berasal dari penelitian Olbrich et all (2010). Aturan brain method dapat dilihat pada persamaan 2.3.

$$\boldsymbol{Brain\ method(c)} = \begin{cases} \mathbf{1}, ((\boldsymbol{LOC(M)} > \mathbf{65}) \wedge (\boldsymbol{CYCLO(M)/LOC(M)} \geq \mathbf{0.24}) \wedge \\ \quad (\boldsymbol{MAXNESTING(M)} \geq \mathbf{5}) \wedge (\boldsymbol{NOAV(M)} > \mathbf{8})) \\ \mathbf{0}, \boldsymbol{else} \end{cases} \quad 2.3$$

Keterangan:
M = *Method*
LOC = *Line of code*
CYCLO = *Cyclomatic complexity*
MAXNESTING = *Maximum nesting level*
NOAV = *Number of accessed variables*

2.6 Software Metrics

Software metrics merupakan indikator pengukuran kuantitat*if* pada sebuah sistem, komponen atau proses (Pressman, 2010). *Software metrics* perangkat lunak memainkan peran penting dalam memahami dan menganalisis modularitas sistem perangkat lunak. *Software metrics* perangkat lunak penting kegunaanya kerana dapat digunakan untuk mengukur kinerja perangkat lunak, sebagai alat untuk perancangan, mengukur produktivitas dan banyak kegunaan lainnya (Lanza dan Marinescu, 2006). *Software metrics* yang digunakan untuk pendeteksian *code smell* jenis *god class* dan brain class yaitu:

2.6.1 *Access To Foreign Data*

Access To Foreign Data merupakan kalkulasi jumlah *attributes* dari kelas lain yang diakses langsung oleh kelas atau dengan *accessor method* (Olbrich, Cruzes dan Sjoøberg, 2010). Untuk mendapatkan *metrics* ATFD menggunakan *metrics* lagi yaitu *number of accessor methods* (NOAM) atau *numberof public attributes* (NOPA). Pada setiap kelas tidak boleh memiliki lebih dari 5 *accessor methods* atau 3 *public attribute* (Marinescu, 2005). Dirumuskan pada persamaan 2.4.

$$\mathbf{ATFD\ (c)} = (\boldsymbol{NOAM} > \mathbf{5}) \vee (\mathit{NOPA} > 3) \quad 2.4$$

Keterangan:

ATFD (c) = *Access to foreign data in Class*
NOAM = Jumlah aksesor metode pada suatu kelas
NOPA = Jumlah public attribute

2.6.2 *Weighted Method Count*

Weighted Method Count merupakan kalkulasi jumlah total dari *cyclomatic complexity* dari semua *method* pada kelas (Olbrich, Cruzes dan Sjoøberg, 2010). WMC menggunakan *metrics* McCabe *cyclomatic complexity* untuk menghitung berat dari setiap *method* lalu kemudian dilakukan penjumlahan (Chidamber dan Kemerer, 1994). Dirumuskan pada persamaan 2.5.

$$\mathbf{WMC}\ (\mathbf{c}) = \sum_{i=1}^{n} \boldsymbol{Ci} \qquad 2.5$$

Keterangan:

WMC (c) = *Weight method count in class*

Ci = *Cyclomatic complexity in method*

n = Jumlah total dari *cyclomatic complexity*

2.6.3 *Tight Class Cohesion*

Tight Class Cohesion merupakan *metrics* untuk mengukur *cohesion* pada *public method* pada sebuah kelas. Dengan cara menghitung jumlah *method* yang terhubung pada suatu kelas. Dua *method* saling terhubung jika mereka mengakses variabel yang sama pada satu kelas (Olbrich, Cruzes dan Sjoøberg, 2010). Untuk mendapatkan metrics TCC yaitu dengan membagi nilai NDC yang merupakan jumlah method yang terhubung langsung pada suatu kelas, dan nilai NP yang merupakan jumlah method yang terhubung secara langsung maupun tidak langsung (Biema dan Kang, 1995). Dirumuskan pada persamaan 2.6.

$$\mathbf{TCC}\ (\mathbf{c}) = \frac{\boldsymbol{NDC}}{\boldsymbol{N} \times (\boldsymbol{N}-\mathbf{1}) \div \mathbf{2}} \qquad 2.6$$

Keterangan:

TCC (c) = *Tight class cohesion in class*

NDC (c) = jumlah metode yang terhubung langsung pada suatu kelas

N = Jumlah metode pada kelas

2.6.4 Line of code

Line of code adalah jumlah dari baris program pada suatu kelas. Jumlah ini meliputi seluruh kode program termasuk *comments* (Olbrich, Cruzes dan Sjoøberg, 2010).

2.6.5 Maxnesting

Maximum nesting level merupakan nilai maksimum *nesting* yang mengontrol struktur sebuah *method* (Lanza dan Marinescu, 2006).

2.6.6 Number of Accessed Variables

Number of Accessed Variables merupakan jumlah total variabel yang mengakses langsung ke metode (*method*) (Lanza dan Marinescu, 2006).

2.6.7 *Cyclomatic complexity*

Cyclomatic complexity merupakan metriks untuk menghitung kompleksitas baik kompleksitas kelas atau metode (*method*). *Cyclomatic complexity* menghitung *control flow* dari modul. Jika *control flow* besar maka semakin tinggi kompleksitas modul tersebut. Untuk mendapatkan nilai *control flow* dengan cara menghitung banyaknya *node* percabangan atau banyaknya *edge* dan *nodes* (Mccabe, 1976). Dirumuskan pada persamaan 2.7.

$$V(g) = E - N + 2 \quad (2.7)$$

Keterangan:

N = Jumlah *node*
E = Jumlah *edge*

2.7 Teknologi pengembangan sistem

2.7.1 Javalang

Javalang merupakan sekumpulan kode yang memiliki fungsi untuk membaca berkas kode java berasal dari python. Pada penelitian ini *javalang* digunakan untuk *parsing* atau memecah berkas kode java menjadi bagian bagian tertentu seperti *attribute*, *class*, *method* dan lain sebagainya. Digunakan sebagai bahan untuk menghitung *software metrics*.

2.7.2 tkInter

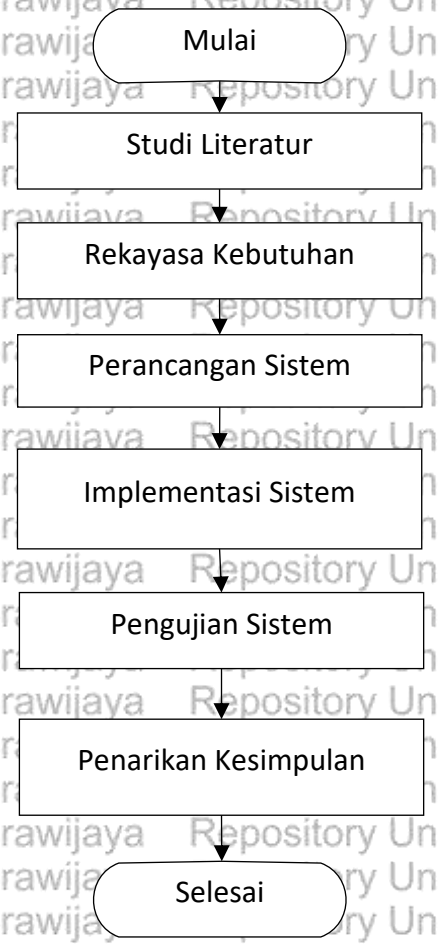
tkInter merupakan sekumpulan modul untuk membangun *grapichal user interface* yang berasal dari python. Pada penelitian ini *tkInter* digunakan untuk membangun sarana interaksi pengguna dengan sistem atau disebut dengan *grapichal user interface*.

2.7.3 Lizard

Lizard merupakan sekumpulan kode yang memiliki fungsi untuk menghitung *cyclomatic complexity, maxnesting, line of code* dan lain lain pada berkas kode java. Pada penelitian ini lizard digunakan untuk menghitung *software metrics cyclomatic complexity, maxnesting, line of code method* dan *line of code class*.

BAB 3 METODOLOGI

Metodologi merupakan bab yang membahas tentang langkah langkah dalam penelitian sistem deteksi *code smell god class* dan *brain class*. Tahapan metodologi penelitian yang digunakan pada penelitian skripsi ini dimulai dari studi literatur. Lalu selanjutnya tahap pengembangan perangkat lunak yang menggunakan *waterfall* yaitu tahap rekayasa kebutuhan setelah tahap rekayasa kebutuhan selesai dilakukan perancangan sistem, lalu dilakukan tahap implementasi sistem berdasarkan perancangan, selanjutnya dilakukan tahap pengujian. Tahap metodologi terakhir dilakukan penarikan kesimpulan berdasarkan tahapan metodologi sebelumnya.



Gambar 3.1 Diagram Alir Penelitian

3.1 Studi Literatur

Tahap studi literatur pada penelitian ini adalah mengumpulkan dasar teori terkait dengan penelitian ini yang bersumber dari buku, jurnal dan laporan penelitian sebelumnya. Kajian pustaka dan dasar teori pada penelitian ini telah dibahas pada bab 2. Dasar teori yang digunakan diantaranya adalah rekayasa perangkat lunak, pengembangan perangkat lunak, model *waterfall*, *unified modeling language* (UML), pengujian, *code smell* dan *software metrics*.

3.2 Rekayasa Kebutuhan

Pada tahap ini merupakan proses untuk mendapatkan seluruh kebutuhan *code smell* jenis *god Class* dan *brain class*. Selanjutnya kebutuhan dispesifikasi dan dimodelkan kedalam perancangan sistem. Tahap ini dilakukan untuk mengetahui kondisi khusus sehingga diketahui gambaran mengenai implementasi program. Yang dapat memberikan kemudahan dalam memahami program dan proses selanjutnya. Proses rekayasa kebutuhan pada perangkat lunak ini sebagai berikut:

1. Analisis Kebutuhan

   Pada tahap ini peneliti menggali kebutuhan yang diperlukan oleh sistem. Kebutuhan diperoleh dengan melakukan analisis dari penelitian yang dilakukan oleh Olbrich, Cruzes and Sjoøberg, (2010). Penelitian ini mengusulkan untuk mendeteksi *code smell god class* dan *brain class*.

2. Tahap Spesifikasi Kebutuhan

   Setelah proses tahap analisis kebutuhan selesai selanjutnya dilakukan proses tahap spesifikasi kebutuhan. Pada tahap ini dilakukan pendefinisian aktor aktor yang menggunakan sistem ini dan kebutuhan yang telah ditemukan dilakukan proses pemetaan kebutuhan fungsional dan kebutuhan non fungsional. Sehingga diketahui secara spesifik tentang apa yang dikerjakan oleh aktor maupun sistem.

3. Manajemen Kebutuhan

   Selanjutnya dilakukan tahap proses manajemen kebutuhan yang mana kebutuhan yang sudah dispesifikasi pada tahap sebelumnya diberi kode CSD-F-1 untuk kebutuhan fungsional dan kode CSD-NF-1 untuk kebutuhan non fungsional.

4. Permodelan Kebutuhan

   Setelah tahap manajemen kebutuhan selesai selanjutnya dilkaukan tahap terahir dari rekayasa kebutuhan. Kebutuhan yang sudah didefinisikan dilakukan proses permodelan kebutuhan dengan membuat *use case diagram* yang menggambarkan hubungan aktor dengan sistem. Setiap *use case* dijabarkan ke *use case scenario* sehingga proses interaksi aktor dengan sistem dapat lebih detail dan alternative apa yang ada pada sistem.

3.3 Perancangan

Tahap perancangan sistem menggunakan *object orientied design*. Tahap ini merupakan tahap lanjutan dari analisis kebutuhan. Pada tahap perancangan ini digunakan sebagai acuan untuk tahap selanjutnya yaitu implementasi dan pengujian. Tahap perancangan pada penelitian ini sebagai berikut:

1. Perancangan Arsitektur
   Perancangan arsitektur dilakukan permodelan menggunakan *unified modeling language* yaitu dengan pembuatan *sequence diagram* dan *class diagram*. Pada penelitian ini akan memaparkan *sequence diagram* yang merupakan fungsionalitas utama pada sistem ini.
2. Perancangan Komponen
   Pada perancangan komponen dituliskan beberapa *sample algorithm*. Pada penelitian ini menuliskan tiga *algorithm* utama dan *algorithm* ini ditulis dalam bentuk *pseudocode*. Hasil perancangan komponen ini akan digunakan sebagai implementasi kode program.
3. Perancangan Antarmuka
   Pada tahap perancangan antarmuka dilakukan perancangan tata letak komponen tampilan yang harus disediakan oleh perangkat lunak berdasarkan kebutuhan perangkat lunak. Pada penelitian ini akan dituliskan beberapa *sample* antarmuka utama. Perancangan ini menjadi dasar untuk implementasi antarmuka perangkat lunak.

3.4 Implementasi

Proses implementasi berdasarkan hasil perancangan yang dilakukan pada proses sebelumnya. Pada penelitian ini implementasi sistem dibangun menggunakan bahasa pemograman python. Proses implementasi sistem ini sebagai berikut:

1. Implementasi mengacu pada spesifikasi kebutuhan yang telah didefinisikan pada tahap sebelumnya.
2. Proses implementasi kode program dilakukan dengan cara menterjemahkan algoritme yang dibuat pada proses perancangan menjadi sebuah kode program. Dalam hal ini peneliti membuat aplikasi berbasis desktop menggunakan bahasa pemrograman python
3. Implementasi antarmuka dilakukan dengan membuat antarmuka yang mengacu pada hasil perancangan antarmuka. Implementasi ini menggunakan tkinter yang merupakan sebuah *library* dari bahasa pemograman python.

3.5 Pengujian

Pada tahap ini dilakukan pengujian terhadap sistem yang telah diimplementasi untuk mencari kesalahan pada program kemudian memperbaikinya dan mengecek apakah sistem sudah sesuai dengan kebutuhan yang telah didefinisikan sebelumnya. Pengujian pada sistem yang akan dibangun mempunyai tahapan sebagai berikut:

1. Pengujian Unit

   Pengujian unit dilakukan untuk pengecekan detail dari perancangan yang berfokus pada kode program. Pada pemograman berorientasi objek pengujian dilakukan pada level terkecil yaitu pada tingkat kelas atau *method*. Pengujian unit pada penelitian ini menggunakan metode pengujian *whitebox*. Metode *whitebox* dengan teknik pengujian menggunakan *basis path testing*. Tujuan dari pengujian unit untuk memastikan setiap struktur kode program terdapat kesalahan atau tidak.

2. Pengujian integrasi

   Pengujian integrasi dilakukan untuk menguji dua komponen yang saling berhubungan untuk menemukan kesalahan dan menjamin komponen yang saling berhubungan dapat berjalan dengan baik. Pengujian integrasi pada penelitian ini menggunakan teknik *bottom-up* dimana komponen tingkat bawah diuji terlebih dahulu kemudian komponen tingkat atas. Pengujian *bottom-up* ini menggunakan *driver* sebagai simulasi komponen tingkat atas.

3. Pengujian Validasi

   pengujian validasi dilakukan untuk pengecekan semua kebutuhan pada perangkat lunak. Pengujian ini bertujuan untuk validasi dan verifikasi sistem berjalan sesuai dengan kebutuhan yang sudah didefisinikan sebelumnya. Kebutuhan yang diuji mencangkup kebutuhan fungsional dan non fungsional. Pengujian validasi menggunakan teknik pengujian *blackbox*. Pada pengujian non fungsional akan dilakukan pengujian peforma dan akurasi. Pengujian performa akan dilakukan dengan mengamati waktu yang dibutuhkan untuk mendeteksi *code smell*. Pada pengujian akurasi akan dilkukan dengan membandingkan nilai perhitungan *software metrics* secara manual dengan nilai perhitungan *software metrics* yang dilakukan oleh sistem.

3.6 Penarikan Kesimpulan dan Saran

Penarikan kesimpulan dilakukan setelah tahap studi literatur, perancangan, implementasi, dan pengujian sistem yang telah dilaksanakan sebelumnya. Kesimpulan dilakukan untuk dapat menjawab rumusan masalah yang telah dirumuskan sebelumnya. Pemberian saran dilakukan berdasarkan hasil evaluasi dan kesalahan pada sistem serta pemberian masukan kepada penulis penelitian selanjutnya.

BAB 4 REKAYASA KEBUTUHAN

Rekayasa kebutuhan ialah aktifitas pertama yang dilakuakan pada proses pengembangan sistem. Rekayasa kebutuhan diawali dengan melakukan penggalian kebutuhan pada sistem yang hendak dibangun, identifikasi aktor serta spesifikasi kebutuhan akan digambarkan pada tahap proses perancangan sistem menggunakan *Unified Modeling Language* dalam bentuk *use case diagram* dan *use case scenario*. Permodelan *unfied modeling language* ini bertujuan untuk menyederhanakan pemahaaman akan setiap kebutuhan pada sistem.

4.1 Deskripsi Umum Sistem

Sistem pendeteksi *code smell* berjenis *god class* dan *brain class* ialah sistem perangkat lunak yang dibangun untuk memudahkan pengguna untuk mendeteksi *code smell* jenis *god class* maupun jenis *brain class* untuk memudahkan *maintenance* sistem. Pada sistem pendeteksi ini pengguna dapat memasukkan *source code java* kemudian sistem akan melakukan *parsing*. Selanjutnya sistem akan mengkalkulasi *software metrics*. Setelah sistem mengkalkulasi *software metrics* maka sistem dapat menentukan apakah program tersebut terdapat *code smell* jenis *god class* atau *brain class* di dalamnya.

Pada penelitian pengembangan sistem deteksi *code smell god class* dan *brain class* dengan menggunakan *metrics* kebutuhan diperoleh dari analisis pada penelitian Steffen, Daniela dan Dag I.K (2010) yang berjudul "Are all Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of three Open Source Systems" dari penelitian tersebut didapatkan aturan dan nilai *threshold* untuk mendeteksi *code smell god class* dan *brain class*.

4.2 Identifikasi Aktor

Pada penelitian pengembangan sistem deteksi *code smell god class* dan *brain class* menggunakan *metrics* dilakukan proses identifikasi aktor. Dengan tujuan untuk menentukan aktor yang akan berinteraksi dengan sistem. Pada penelitian pengembangan sistem deteksi *code smell god class* dan *brain class* terdapat satu aktor yaitu *User*. Yang dipaparkan pada Tabel 4.1.

Tabel 4.1 Daftar Aktor

Aktor	Deskripsi
User	*User* adalah pengguna sistem pendeteksi *code smell god Class* dan *brain class*.

4.3 Daftar Kebutuhan Fungsional Sistem

Kebutuhan fungsional ialah kumpulan pernyataan yang wajib ada dalam sistem yang mampu menyanggupi kebutuhan *user*. Kebutuhan fungsional diperoleh dari rekayasa kebutuhan kemudian dispesifikasi. Kebutuhan fungsional menjelaskan bagaimana sistem dapat berinteraksi dengan *user*. Pada setiap kebutuhan fungsional pada penelitian ini diberikan kode CSD-F-XX. CSD inisial dari sistem yang merupakan singkatan dari *Code Smell Detection* , F merupakan inisial dari kebutuhan fungsional, XX merupakan nomor kebutuhan Daftar kebutuhan pada penelitian ini digambarkan pada Tabel 4.2.

Tabel 4.2 Daftar Kebutuhan Fungsional Sistem

No	Kode Kebutuhan	Deskripsi Kebutuhan	*Use case*
1.	CSD-F-01	Sistem harus mampu melakukan deteksi *code smell god class* dan *brain class*.	Deteksi *code smell*

4.4 Kebutuhan Non-Fungsional

Kebutuhan non-fungsional merupakan kumpulan pernyataan terkait dengan batasan batasan fungsionalitas sistem. Kebutuhan non-fungsional biasanya berbentuk batasan pengembangan sistem, batasan waktu dan batasan standard sistem. Kebutuhan non-fungsional biasaya memaparkan kebutuhan yang terkait dengan karakteristik seperti peforma, akurasi, keamanan, *reliability*. Pada setiap keutuhan non-fungsional pada penelitian ini diberikan kode CSD-NF-XX. Dimana CSD merupakan inisial dari sistem yang merupakan singkatan dari *Code smell God Class*, NF merupakan inisial dari kebutuhan non-fungsional, XX merupakan nomor kebutuhan. Daftar kebutuhan pada penelitian ini digambarkan pada Tabel 4.3.

Tabel 4.3 Daftar Kebutuhan Non-Fungsional

No	Kode kebutuhan	Deskripsi Kebutuhan	Karakteristik
1	CSD-NF-01	Sistem harus mampu mendeteksi *code smell* jenis *god class* dan *barin class* dengan batas waktu kurang dari 1 s.	Peforma
2	CSD-NF-02	Sistem mampu mengkalkulasi *software metrics* dengan tingkat ketelitian minimal 90% pada setiap *class*.	Akurasi

4.5 Permodelan Kebutuhan

4.5.1 *Use case* Diagram

Use case merupakan gambaran bagaimana pernggguna berinteraksi dengan sistem dan menjelaskan tentang gambaran besar tentang fungsionalitas sistem. Pada penelitian ini terdapat satu *base use case*. *Base use case* itu adalah mendeteksi *code smell*. *Use case diagram* pada penelitian ini dapat dilihat pada Gambar 4.1

Gambar 4.1 *Use case Diagram*

4.5.2 *Use case* Scenario

Use case scenario merupakan penjelasan secara tekstual dari sekumpulan skenario interaksi. Setiap skenario mendeskrepsikan urutan aksi/langkah yang dilakukan aktor ketika beriinteraksi dengan sistem, baik yang berhasil maupun tidak berhasil. Di dalam *use case scenario* terdapat. Aktor, Kondisi awal, skenario utama (*Main flow*), proses altenat*if* (*Altenatif flow*), dan kondisi akhir (*post condition)*. Pembuatan *use case scenario* berdasarkan pada *use case diagram*. *Use case scenario* pada penelitian ini di paparkan pada Tabel 4.4.

Tabel 4.4 *Use case scenario* memasukkan kode program

***Use case Scenario* untuk memasukan kode program**	
Nomor *Use case*	CSD-F-01
Nama *Use case*	Mendeteksi *code smell*
Tujuan	Pengguna dapat mendeteksi *code smell*
Aktor	*User*
Kondisi Awal	*User* berada dihalaman beranda
Skenario Utama	1. *User* menekan tombol buka folder pada halaman beranda 2. Sistem menampilkan daftar folder 3. *User* memilih folder dan mengklik tombol *select folder*

	4. Sistem memasukkan *file source code* kedalam sistem 5. User menekan tombol deteksi pada halaman beranda 6. Sistem mengambil nilai perhitungan Access to foreign data pada setiap kelas. 7. Sistem mengambil nilai perhitungan Weighted method count pada setiap kelas. 8. Sistem mengambil nilai perhitungan Tight class cohesion pada setiap kelas. 9. Sistem mengambil nilai perhitungan Line of code pada setiap kelas. 10. Sistem mengambil nilai perhitungan Line of code method pada setiap method. 11. Sistem mengambil nilai perhitungan Maxnesting pada setiap method. 12. Sistem mengambil nilai perhitungan Cyclomatic complexity pada setiap method. 13. Sistem mengambil nilai perhitungan Number of accesed variable pada setiap method. 14. Sistem membandingkan semua nilai dengan nilai treshold setiap code smell. 15. User menekan tombol halaman hasil 16. Sistem mengambil nilai hasil dari kalkulasi software metrics dan pendeteksian code smell. 17. Sistem menampilkan hasil detail dari kalkulasi file tersebut dan deteksi lalu menampilkannya pada halaman hasil.
Alternatif	*1.* Jika *user* memasukkan *file* selain java maka sistem tidak akan membacanya.
Kondisi Akhir	*User* dapat mendeteksi *code smell* dan menampilkan hasil deteksi.

BAB 5 PERANCANGAN DAN IMPLEMENTASI

5.1 Perancangan Sistem

Tahap perancangan sistem merupakan salah satu tahap yang ada di dalam proses pengembangan perangkat lunak, tahap perancangan dilakukan setelah tahap rekayasa kebutuhan. Pada penelitian ini tahap perancangan terbagi menjadi tiga bagian yaitu, perancangan arsitektur, perancangan komponen dan perancangan antarmuka. Hasil dari tahap perancangan sistem ini sebagai acuan dasar untuk tahap implementasi sistem.

5.1.1 Perancangan Arsitektur

Perancangan arsitektur merupakan tahapan yang menjelaskan secara rinci mengenai *sequence diagram* dan *class diagram*. *Sequence diagram* akan menjelaskan alur interaksi atau pertukaran pesan antar objek pada sistem. *Class diagram* akan menjelaskan kelas, atribut, operasi dan hubungan antar kelas sebagai penyusun sistem. Pada penelitian ini menjelaskan satu *sequence diagram* yang mewakili fitur pada sistem ini. *Sequence diagram* tersebut adalah deteksi *code smell*.

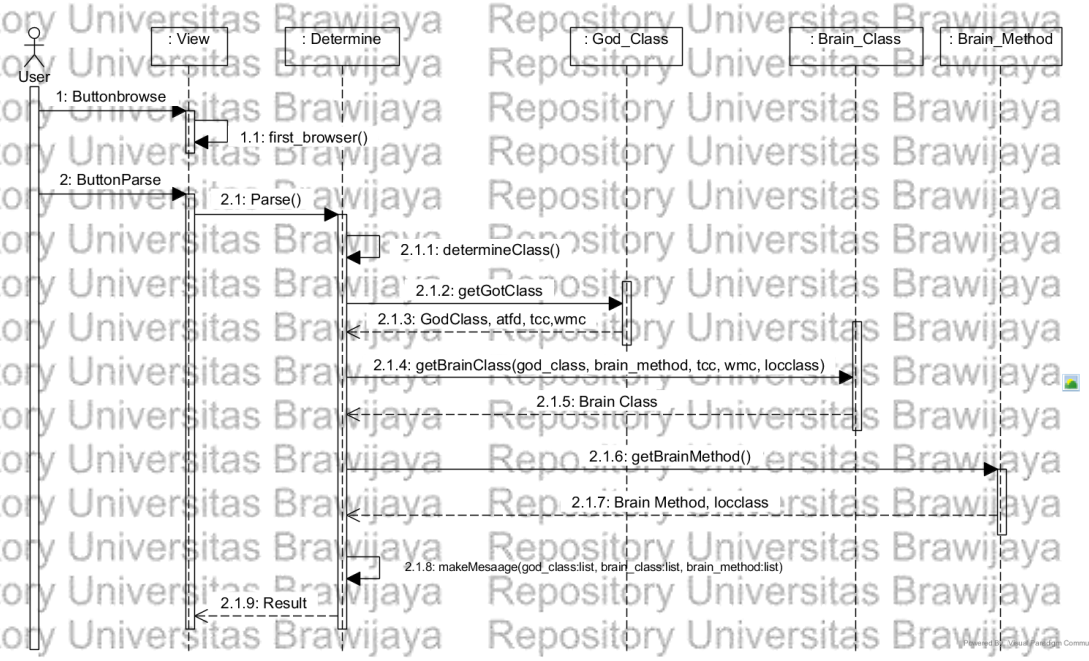
5.1.1.1 Sequence diagram deteksi *code smell*

Sequence diagram deteksi *code smell* ditunjukan pada Gambar 5.1. Pada *sequence diagram* ini menunjukan interaksi objek untuk mendeteksi *code smell god class* dan *brain class*. Tujuan *sequence diagram* ini untuk mendeteksi *code smell god class* dan *brain class*. Proses pendeteksian dilakukan dengan cara menghitung nilai semua metriks lalu dilakukan perbandingan nilai perhitungan dengan nilai batasan yang telah ditentukan, apabila nilai perhitungan mendekati atau sesuai dengan nilai batasan maka dapat dikategorikan sebagai *code smell*. Pada Gambar 5.2 terdapat empat objek yang berinteraksi. Empat objek tersebut diantaranya satu *boundary* view. Tiga *controller* yaitu *detection*, *calculation*, dan *parsing*.

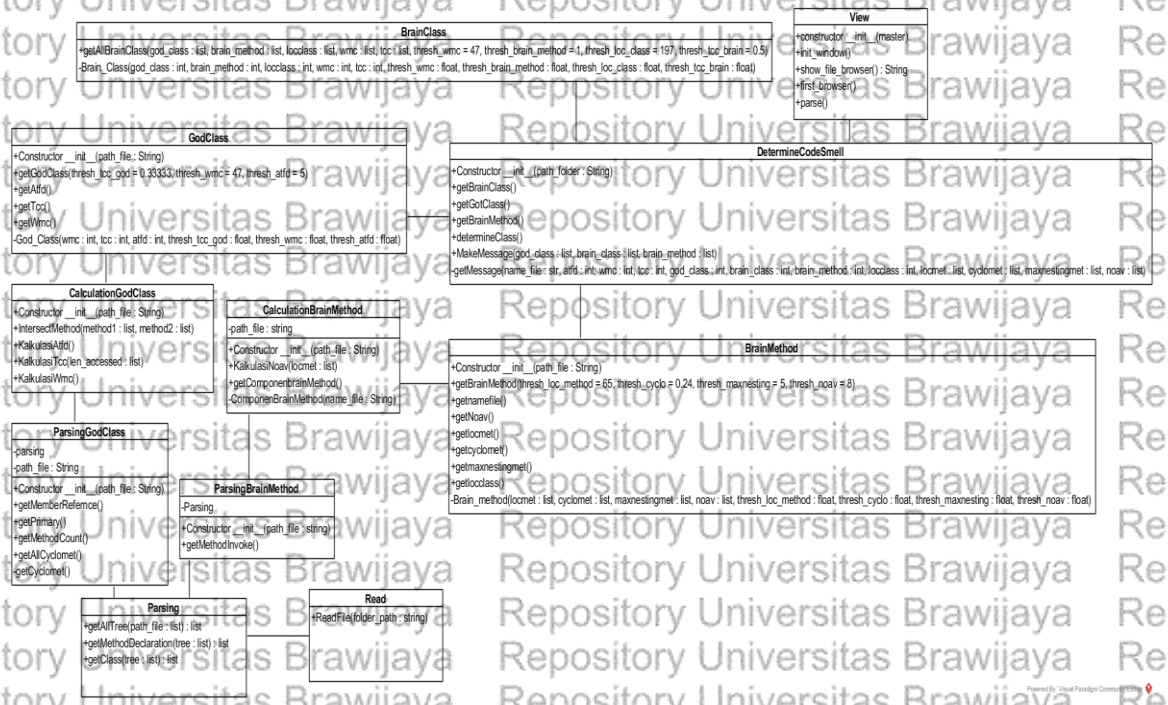
5.1.1.2 Class diagram

Class diagram pada sistem pendeteksian ini ditunjukan pada Gambar 5.2. *Class diagram* mempunyai hubungan relasi antar kelas yang jenis hubungannya berdasarkan kepada aturan penulisan *Class diagram*. Terdapat beberapa kelompok kelas yaitu kelompok kelas *view* yang berisi tampilan sistem. Kelas view berhubungan langsung dengan pada library tkinter. Kemudian ada kelas *parsing* kelas ini berisi alur logika untuk *parsing* atau memecah file java menjadi beberapa bagian seperti bagian *class*, *method*, *attribute* dan lain sebagainya. Kelas ini menggunakan *library* javalang dan os,glob. Javalang untuk proses *parsing,* os,glob untuk memilah berkas kode java dan membaca file hasil dari membaca tersebut digunakan untuk *parsing*. Selanjutnya terdapat kelas *calculation* kelas ini berisi alur logika perhitungan *software metrics*. Kelas ini menggunakan *library* lizard yang digunakan untuk menghitung *software metrics*

cyclomatic complexity, line of code method dan lain sebagainya. Selanjutnya terdapat kelas *detection* yang berisi alur logika untuk mendeteksi *code smell god class* dan *brain class*.



Gambar 5.1 *Sequence diagram* deteksi *code smell*



Gambar 5.2 *Class diagram* deteksi *code smell god class* dan *brain class*

5.1.2 Perancangan Komponen

Perancangan komponen akan menjelaskan rincian dari sub-sistem pada setiap komponen perangkat lunak. Pada perancangan komponen akan menjelaskan alur *algoritme* pada setiap komponen. Pada penelitian ini akan menjelaskan tentang tiga *algoritme method* yaitu *method readfile,method brain method* dan *method get variabletype*.

5.1.2.1 Perancangan Komponen *method readFile*

Nama *Class*: *Parsing*

Nama *method*: *readFile*

Tabel 5.1 Perancangan Komponen *readFile*

Pseudocode *Algoritme method readFile*	
1	Initialization array list
2	Looping filter file only java extention and get file directory
3	read file
4	Move result read file into array list
5	Move file directory into array list
6	Return to read file and file directory

Pada Tabel 5.1 merupakan pseudocode dari *method readFile* yang berfungsi untuk untuk membaca *file* yang telah dipilih oleh *user* untuk melakukan deteksi *code smell god class*, *brain method*, dan *brain class*.

5.1.2.2 Perancangan Komponen *method KalkulasiBrainMethod*

Nama *Class*: *Calculation*

Nama *Method*: *KalkulasiBrainMethod*

Tabel 5.2 Perancangan Komponen *method KalkulasiBrainMethod*

Pseudocode *Algoritme* method *KalkulasiBrainMethod*	
1	Initialization list
2	Analysis file with library
3	Call function to get line of code
4	Looping each file for calling function from library
5	Move result from function unqualified name into array list
6	Move result from function length into array list
7	Move result from function cyclomatic complexity into array list
8	Move result from function top nesting level into array list
9	End for

Pada Tabel 5.2 merupakan pseudocode dari *method KalkulasiBrainMethod* yang berfungsi untuk menghitung nilai kalkulasi *software metrics* yang dibutuhkan untuk mendeteksi *code smell brain method*.

5.1.2.3 Perancangan komponen *method getMethodDeclaration*

Nama *Class*: *Parsing*

Nama *Method*: *getMethodDeclaration*

Tabel 5.3 Perancangan Komponen *method getMethodDeclaration*

Pseudocode *algoritme* method *getMethodDeclaration*	
1	Initialization array list
2	Looping in each file for read file java source code
3	Move result method name to array list
4	End for
5	Return to variables

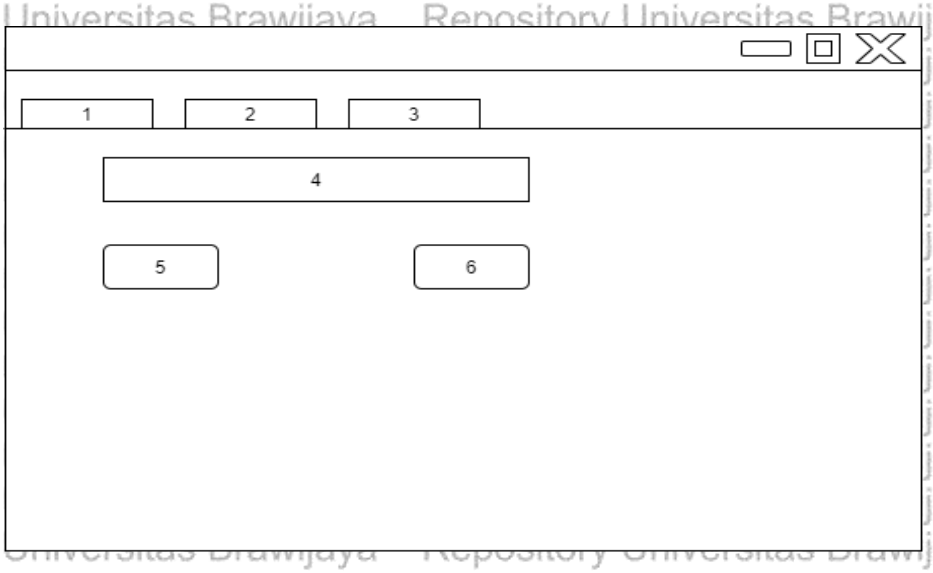
Pada Tabel 5.3 merupakan tabel pseudocode dari *method getMethodDeclaration* berfungsi sebagai proses untuk menghitung dan mendapatkan *method pada setiap kelas* dari file *source code* yang dideteksi. Nantinya digunakan untuk menghitung dan mengambil data *method*.

5.1.3 Perancangan Antarmuka

Perancangan antarmuka adalah sebuah *wireframe* tampilan dari sistem, sebagai sarana interaksi *user* dengan sebuah sistem. Dalam perancangan antarmuka akan menjelaskan beberapa *wireframe*. *Wireframe* pada penelitian ini ada tiga yaitu *wireframe* halaman beranda, hasil, kesimpulan.

5.1.3.1 Perancangan Antarmuka Halaman Beranda

Perrancangan antarmuka halaman *beranda* ditunjukan pada Gambar 5.3. Pada *layout* perancangan terdapat beberapa komponen yaitu *Button* dan *text box*. Komponen tersebut akan dijelaskan pada Tabel 5.4. Antarmuka ini digunakan untuk meng*input*kan atau memasukan *file source code* kedalam sistem dan untuk mendeteksi *file source code* yang telah dimasukkan oleh *user*.



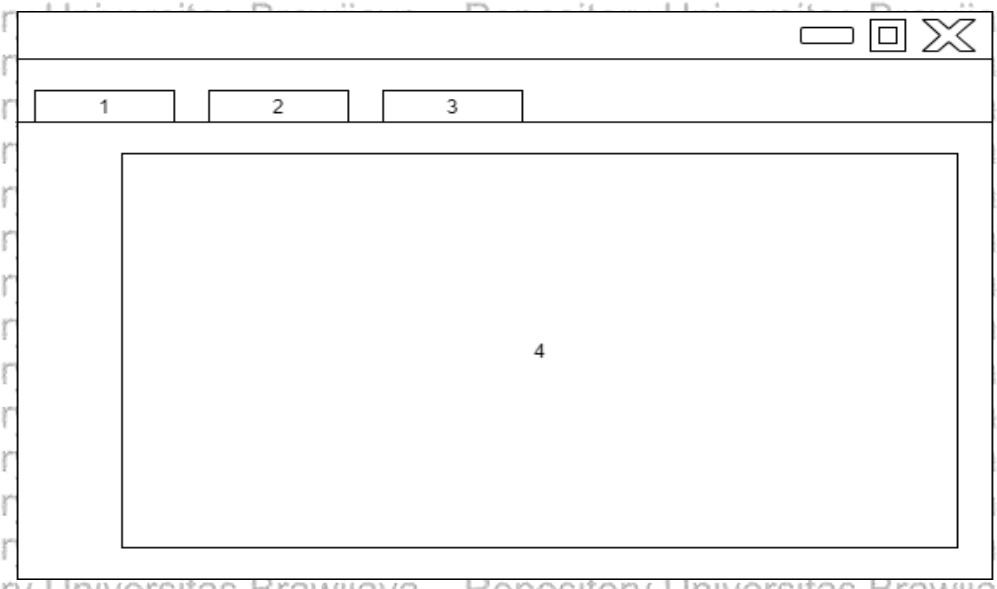
Gambar 5.3 Perancangan Antarmuka Halaman Beranda

Tabel 5.4 Perancangan Antarmuka Halaman Beranda

No.	Nama Objek	Tipe	Keterangan
1	Beranda	*Button*	Tombol yang digunakan untuk membuka halaman beranda.
2	Hasil	*Button*	Tombol yang digunakan untuk membuka halaman hasil.
3	Kesimpulan	*Button*	Tombol yang digunkan untuk membuka halaman kesimpulan.
4	*Text* Box	*Text*	*Text* yang digunakan untuk menampilkan direktori file yang dipilih.
5	Buka folder	*Button*	Tombol untuk membuka direktori folder dan memasukkan file kedalam sistem.
6	Deteksi	*Button*	Tombol untuk deteksi *code smell* pada *file source code* yang di*input*kan oleh *user* dan kalkulasi *metrics*.

5.1.3.2 Perancangan Antarmuka Halaman Hasil

Perancangan antarmuka halaman hasil ditunjukan pada Gambar 5.4. Pada *layout* perancangan terdapat beberapa komponen yaitu *Button* dan *Text box*. Komponen tersebut akan dijelaskan pada Tabel 5.5. Antarmuka ini digunakan untuk menampilkan hasil deteksi dan kalkulasi dari file yang telah dimasukkan oleh *user* dan menampilkan nilai hasil deteksi tersebut.



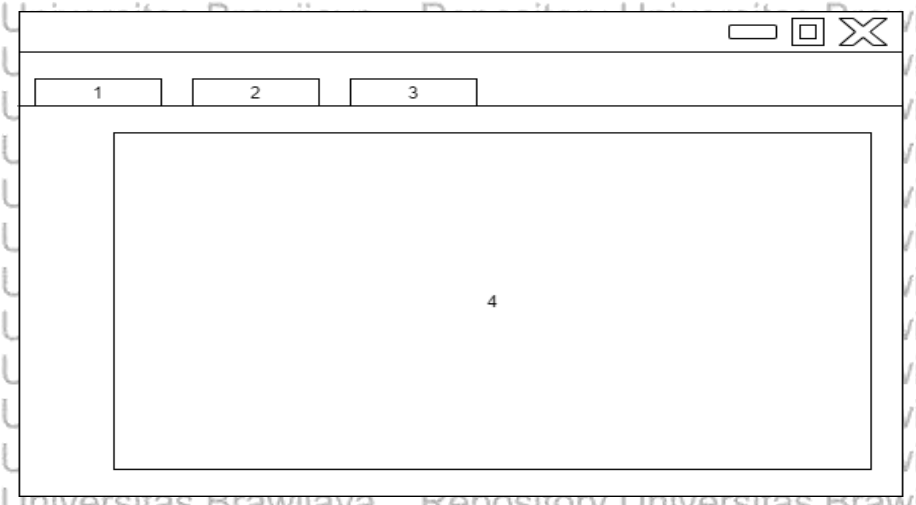
Gambar 5.4 Perancangan Antarmuka Halaman Hasil

Tabel 5.5 Perancangan Antarmuka Halaman Hasil

No.	Nama Objek	Tipe	Keterangan
1	Beranda	*Button*	Tombol yang digunakan untuk membuka halaman beranda.
2	Hasil	*Button*	Tombol yang digunakan untuk membuka halaman hasil.
3	Kesimpulan	*Button*	Tombol yang digunkan untuk membuka halaman kesimpulan.
4	*Text box*	*Text*	*Text* yang digunakan untuk menampilkan nilai kalkulasi dan hasil deteksi.

5.1.3.3 Perancangan Antarmuka Halaman Kesimpulan

Perrancangan antarmuka halaman kesimpulan ditunjukan pada Gambar 5.5. Pada *layout* perancangan terdapat beberapa komponen yaitu *Button* dan *Text box*. Komponen tersebut akan dijelaskan pada Tabel 5.6. Antarmuka ini digunakan untuk menampilkan penjelasan singkat mengenai sistem.



Gambar 5.5 Perancangan Antarmuka Halaman Kesimpulan

Tabel 5.6 Perancangan Antarmuka Halaman Kesimpulan

No.	Nama Objek	Tipe	Keterangan
1	Beranda	*Button*	Tombol yang digunakan untuk membuka halaman beranda.
2	Hasil	*Button*	Tombol yang digunakan untuk membuka halaman hasil.
3	Kesimpulan	*Button*	Tombol yang digunkan untuk membuka halaman kesimpulan.
4	*Text box*	*Text*	*Text* yang digunakan untuk menampilkan total nilai kalkulasi, total kelas yang di deteksi dan hasil deteksi keseluruhan.

5.2 Implementasi Sistem

Tahap implementasi sistem merupakan tahap yang dilakukan setelah tahap perancangan sistem. Implementasi berdasarkan hasil perancangan yang dilakukan pada proses sebelumnya. Implementasi mengacu pada spesifikasi kebutuhan yang telah didefinisikan pada tahap sebelumnya. Pada tahap ini perancangan algoritme di implementasikan menjadi kode program, implementasi antarmuka berdasarkan perancangan pada tahap sebelumnya dan spesifikasi sistem.

5.2.1 Spesifikasi Sistem

Pada bagian tahap spesifikasi sistem akan menjelaskan perangkat lunak dan perangkat keras yang digunakan untuk mengembangkan perangkat lunak.

Spesifikasi perangkat keras akan menjelaskan mengenai *prosesor*, *storage* (*secondary memory*), dan *main memory* (RAM). Spesifikasi perangkat lunak akan menjelaskan mengenai bahasa pemograman, IDE, sistem operasi dan perangkat penunjang lainnya yang digunakan. Spesifikasi perangkat keras ditunjukan pada Tabel 5.7 dan spesifikasi perangkat lunak ditunjukan pada Tabel 5.8.

Tabel 5.7 Spesifikasi Perangkat Lunak

Nama Komponen	Spesifikasi
Procesor	Intel Core i3 4010U @1.70GHz
Hard Disk Drive	500 GB
Main Memory (RAM)	8.0 GB DDR3

Tabel 5.8 Spesifikasi Perangkat Keras

Nama Komponen	Spesifikasi
Operating System	Windows 10
Programing Language	Python
IDE / *Text Editor*	Spyder v4.1.3
UI *Builder*	Tkinter v8.6
Editor Perancangan	StarUML v3.2.2
Editor Dokumentasi	Microsoft Word 2016

5.2.2 Implementasi Kode Program

Pada tahap implementasi kode program yang berdasarkan pada perancangan komponen *algoritme* yang telah didefinisikan. Pada tahap ini *algoritme* yang ada di tahap perancangan komponen di implementasikan menjadi bahasa pemograman. Pada penelitian ini akan menjelaskan tiga implementasi *method* yaitu *method inputfile*, *method brain_method* dan *getvariabletype*. Bahasa pemograman yang digunakan adalah python Programing *Language*. Berikut ini penjelasan mengenai implementasi kode program yang akan di paparkan pada sub-bab 5.2.2.1 sampai 5.2.2.3.

5.2.2.1 Implementasi kode program *method readFile*

Nama *Class*: *Parsing*
Nama *method*: *readFile*

Tabel 5.9 Kode sumber *method readFile*

Kode sumber method *readFile*

```
1   def readFile(self, folder_path:str)-> list:
2       containt_text = []
3       name_file = []
4       for filename in glob.glob(os.path.join(folder_path,
```

```
'*.java')):
        with open(filename, 'r') as f:
            containt_text.append(f.read())
            name_file.append(filename)
    return containt_text, name_file
```

Tabel 5.10 Penjelasan kode sumber *method readfile*

Penjelasan kode sumber *method* readFile	
1	Inisialisasi method public readFile dengan parameter folder_path tipe data string nilai kembali list
2	Inisialisasi variable containtext tipe data list
3	Inisialisasi variable name_file tipe data list
4	Menjalan perulangan untuk mendapatkan folder path dan file java
5	Menjalankan fungsi membaca
6	Hasil dari membaca disimpan kedalam variable contain_text
7	Hasil folderpath disimpan kedalam variable name_file
8	Mengembalikan contain_text, name_file

Pada Tabel 5.9 memaparkan implementasi dari *method readFile*. *Method readfile* digunakan untuk mengekstrasi *file source java* kedalam *string variable* nantinya digunakan untuk *parsing* atau memecah *source code* menjadi beberapa bagian lalu dari bagian tersebut diperoleh *variable, class* dan lain sebagainya. Selain mengekstrasi, *method* ini juga untuk mendapatkan *directory file*. Cara kerja *method inputfile* yaitu memanggil fungsi dari *library* os.glob untuk mengakases *file* lalu memilah berkas kode sumber java yang digunakan dan mendapatkan *file directory* pemyimpanan *device user*. Lalu dilakukan filter untuk memilih *file* yang memiliki ekstensi java saja. Isi dari *file* tersebut yang berupa *source code* dan *directory file* disimpan didalam *array list*.

5.2.2.2 Implementasi kode program *method* KalkulasiBrainMethod

Nama *Class* : *Calculation*

Nama *Method*: *KalkulasiBrainMethod*

Tabel 5.11 Kode seumber *method* KalkulasiBrainMethod

Kode sumber *method* KalkulasiBrainMethod

```
1  def __KalkulasiBrainMethod (self, name_file: str) -> dict:
2          name = list()
3          locmet = list()
4          cyclomet = list()
5          maxnestingmet = list()
6          analyze_files = lizard.analyze_file(name_file)
7          locclass = analyze_files.nloc
8          for func in analyze_files.function_list:
9              name.append(str(func.unqualified_name))
10             locmet.append(func.length)
11             cyclomet.append(func.cyclomatic_complexity)
12             maxnestingmet.append(func.top_nesting_level)
13         return {"name": name,
```

```
                "locmet": locmet,
                "cyclomet": cyclomet,
                "maxnestingmet": maxnestingmet,
                'locclass': locclass}
```

Tabel 5.12 Penjelasan kode sumber *method KalkulasiBrainMethod*

Penjelasan sumber *method* KalkulasiBrainMethod	
1	Inisialisasi method private dengan nama KalkulasiBrainMethod dengan parameter name_file tipe data string nilai kembali bertipe dict
2	Inisialisai variable name tipe list
3	Inisialisai variable locmet tipe list
4	Inisialisai variable cyclomet tipe list
5	Inisialisai variable maxnestingmet tipe list
6	Menjalankan fungsi read file menggunakan library lizard kemudian hasil disimpan kedalam variable analyze_files
7	Mendapatkan nilai line of code dari library lizard kemudian hasil disimpan kedalam variable locclass
8	Menjalankan perulangan untuk mendapatkan nama method, line of code method, cyclomatic complexity dan nilai maxnesting pada method
9	Memindahkan hasil nama method kedalam variable name
10	Memindahkan hasil line of code method kedalam variable locmet
11	Memindahkan hasil cyclomatic complexity kedalam variable cyclomet
12	Memindahkan hasil maxnesting kedalam variable maxnestingmet
13	Mengembalikan name, locmet, cyclomet, maxnestingmet,locclass

Pada Tabel 5.10 memaparkan implementasi dari *method* KalkulasiBrainMethod. *Method* ini digunakan untuk mendapatkan nilai kalkulasi *software metrics* untuk *code smell brain method*. Nilai kalkulasi tersebut berasal dari perhitungan yang dilakukan oleh *library* lizard. Perhitungan ini dilakukan dengan cara memanggil fungsi dari *library* kemudian hasilnya dimasukan kedalam *array* list.

5.2.2.3 Implementasi Kode Program *method getMethodDeclaration*

Nama *Class*: *Parsing*

Nama *Method*: *getMethodDeclaration*

Tabel 5.13 kode sumber *method getMethodDeclaration*

Kode sumber method *getMethodDeclaration*	
1	`def getMethodDeclaration(self,tree:list)-> list:`
2	`    Methods = []`
3	`    For path,node intree.filter(javalang.tree.MethodDeclaration):`
4	`        Methods.append(str(node.name))`
5	`    return Methods`

Tabel 5.14 Penejelasan kode sumber *method getMtehodDeclaration*

Penjelasan sumber *method* getMethodDeclaration	
1	Inisialisasi method public getMtehodDeclaration dengan parameter tree tipe data list nilai kembali list
2	Inisialisasi variable methods dengan tipe list
3	Menjalankan perulangan untuk mendapatkan nama method dari hasil membaca file yang dilakukan oleh library javalang
4	Memindahkan hasil nama method kedalam variable Methods
5	Mengembalikan method

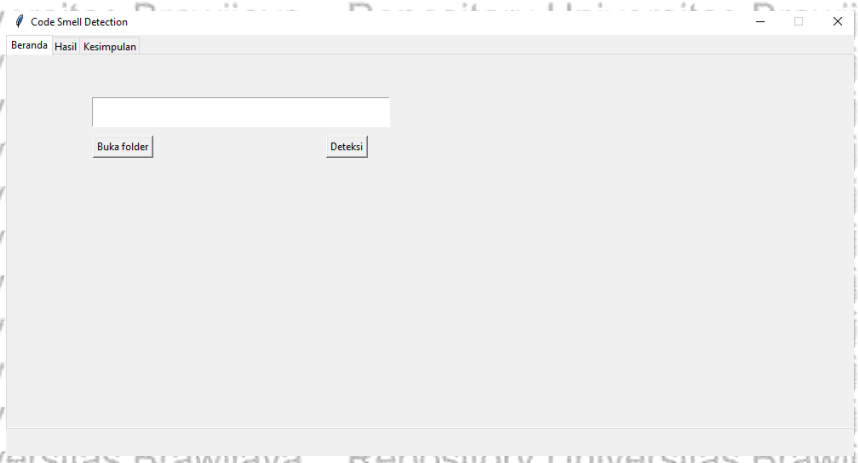
Pada Tabel 5.11 memaparkan implementasi dari *method getMethodDeclaration*. *Method* ini digunakan untuk mendapatkan *variables* beserta tipenya yang berasal dari *file source code* yang dideteksi. Cara kerja *method* ini dengan cara melakukan *parsing* pada setiap file yang dilakukan oleh *library* javalang. Lalu dilakukan seleksi untuk mengambil data *method*.

5.2.3 Implementasi Antarmuka

Implementasi antarmuka dilakukan dengan membuat antarmuka yang mengacu pada hasil perancangan antarmuka. Implementasi ini menggunakan tkinter yang merupakan sebuah *library* dari bahasa pemograman python. Pada penelitian ini akan memaparkan hasil implementasi antarmuka yaitu *home*, *result*, visualisasi. Implementasi ini akan dijelaskan pada sub-bab 5.2.3.1 sampai 5.2.3.3.

5.2.3.1 Implementasi Antarmuka Halaman Beranda

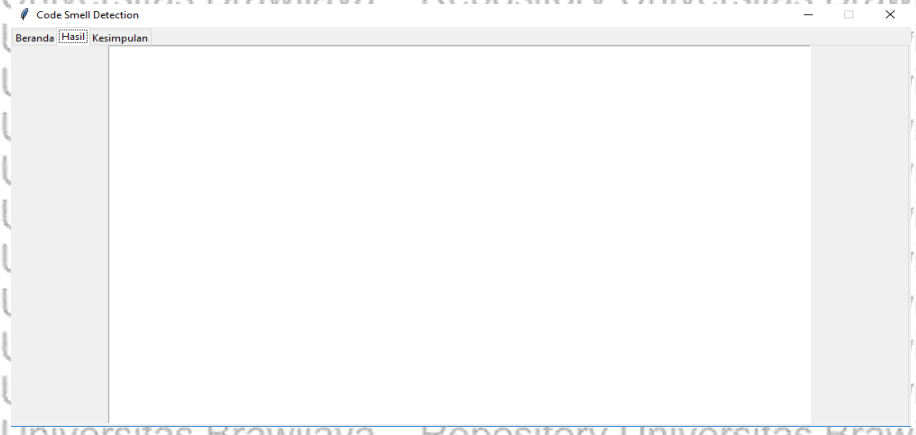
Pada Gambar 5.6 Merupakan implementasi antarmuka beranda. Implementasi antarmuka ini berdasarkan perancangan antarmuka yang telah dilakukan pada tahap perancangan. Pada implementasi antarmuka ini terdapat tombol "Buka Folder" yang berfungsi untuk memasukkan *file* dan terdapat tombol "Deteksi" yang berfungsi untuk kalkulasi semua *software metrics* dan mendeteksi *file* yang telah dipilih oleh *user*. Pada implementasi antarmuka halaman beranda ini juga terdapat *text box* yang berfungsi untuk menampilkan alamat *directory file* dari *file* yang dipilih oleh *user*.



Gambar 5.6 *Implementasi* Antarmuka Halaman Beranda

5.2.3.2 Implementasi Antarmuka Halaman Hasil

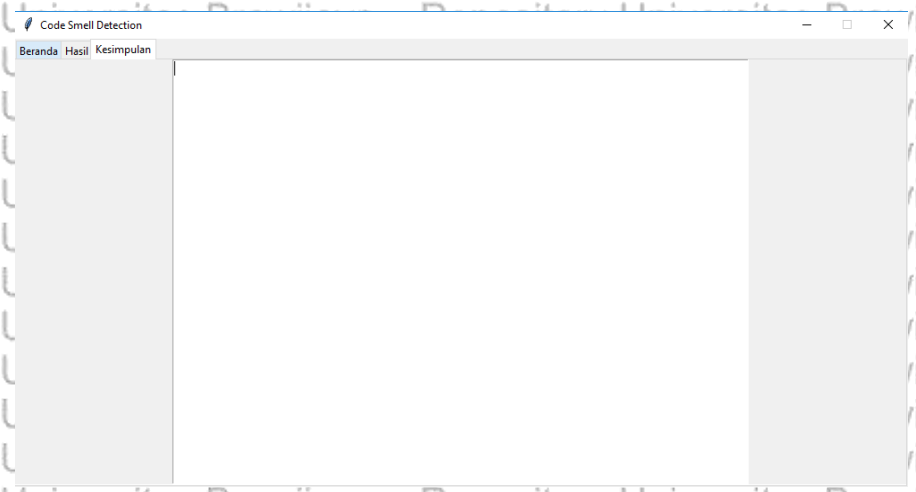
Pada Gambar 5.7 Merupakan implementasi antarmuka hasil. Implementasi antarmuka ini berdasarkan pada perancangan antarmuka yang telah dilakukan sebelumnya. Pada implementasi antarmuka ini terdapat "*Text box*" untuk menampilkan nilai dari hasil kalkulasi semua *software matrics* pada setiap kelas dan menampilkan hasil dari deteksi setiap kelas mengenai *code smell god class* dan *brain class*.



Gambar 5.7 *Implementasi* Antarmuka Halaman Hasil

5.2.3.3 Implementasi Antarmuka Halaman Kesimpulan

Pada Gambar 5.8 Merupakan implementasi antarmuka halaman kesimpulan. Implementasi antarmuka ini berdasarkan pada perancangan antarmuka yang telah dilakukan pada tahap sebelumnya yaitu tahap perancangan. Pada implementasi antarmuka ini terdapat "*text box*" untuk menampilkan nilai dari hasil kalkulasi semua *software matrics* pada semua kelas dan menampilkan hasil dari deteksi total semua kelas mengenai *code smell god class* dan *brain class* serta menampilkan total semua kelas yang dibaca.



Gambar 5.8 *Implementasi* Antarmuka Halaman Kesimpulan

BAB 6 PENGUJIAN

6.1 Pengujian Unit

Pengujian unit merupakan proses pengujian yang dilakukan pada setiap unit dari perangkat lunak yang dikembangkan. Proses pengujian unit menggunakan metode *whitebox testing* dengan menggunakan teknik *Basis Path testing*. Pengujian unit ini menguji tiga *method testing*. Tiga *method* yang diuji antara lain *method readFile*, *method* KalkulasiBrainMethod dan *getMethodDeclaration*.

6.1.1 Pengujian unit method *readFile*

1. Pseudecode

Nama *Class*: *Parsing*
Nama *method*: *readfile*

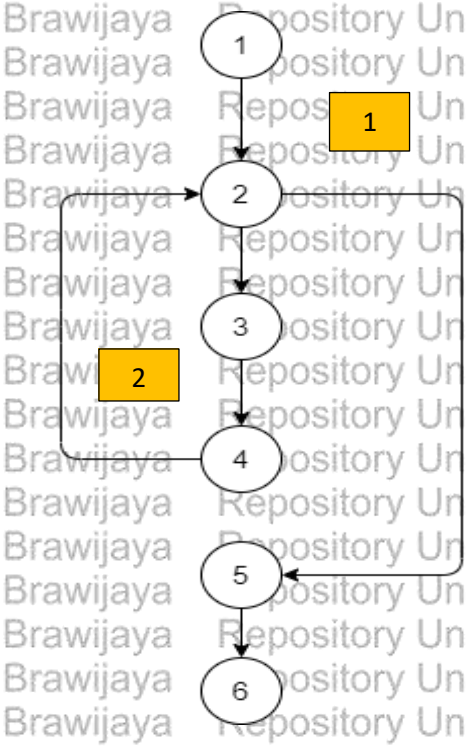
Tabel 6.1 Algoritme *method readfile*

Pseudocode *Algoritme method readFile*		
1	Initialization array list 1	
2	Looping filter only file java extention and get file directory 2	
3	Function read file 3	
4	Move result read file into array list	4
5	Move file directory into array list	
6	End for 5	
7	Return to read file and file directory 6	

2. Basis Path

Flow graph

Berdasarkan algoritme *method readFile* yang dipaparkan akan menjadi dasar pembuatan *flow graph* ditunjukan dalam Gambar 6.1.



Gambar 6.1 *Flow graph method readFile*

Cyclomatic Complexity

Berdasarkan gambar *flow graph readFile* diatas diketahui jalur independent melalui perhitungan *cyclomatic complexity*. Berikut ini merupakan hasil perhitungan *cyclomatic complexity* berdasarkan *flow graph readFile*.

- Jumlah *region* = 2
- Jumlah *edge* – jumlah *node* + 2 = 6 – 6 + 2 = 2
- Jumlah *predicated node* + 1 = 1 + 1 = 2

Independent Path

- Jalur 1: 1 – 2 – 3 – 4 – 2 – 5 – 6
- Jalur 2: 1 – 2 – 5 – 6

Berdasarkan *Independent Path* yang didapat akan dijadikan dasar kasus uji. Hasil pengujian unit dan kasus uji *method readfile* akan dipaparkan pada Tabel 6.2.

Tabel 6.2 Hasil pengujian unit *method readfile*

No	Kasus Uji	Hasil pengujian yang diharapkan	Hasil pengujian	Status
1	*Class Driver* memanggil *method readFile*() Dengan data: Terdapat berkas java di dalam folder	- Menjalankan *method readfile* dan *print arraylist* hasilnya menampikan isi dari file.	- Menjalankan *method readfile* dan *print arraylist* hasilnya menampikan isi dari file.	Valid
2	*Class Driver* memanggil *method readFile*() Dengan data: Tidak ada berkas java di dalam folder	- Menjalankan *method readfile* dan *print arraylist* hasilnya tidak menampikan isi dari file.	- Menjalankan *method readfile* dan *print arraylist* hasilnya tidak menampikan isi dari file.	Valid

6.1.2 Pengujian unit *method KalkulasiBrainMethod*

1. *Pseudecode*

Nama *Class*: *Calculation*

Nama *Method*: *KalkulasiBrainMethod*

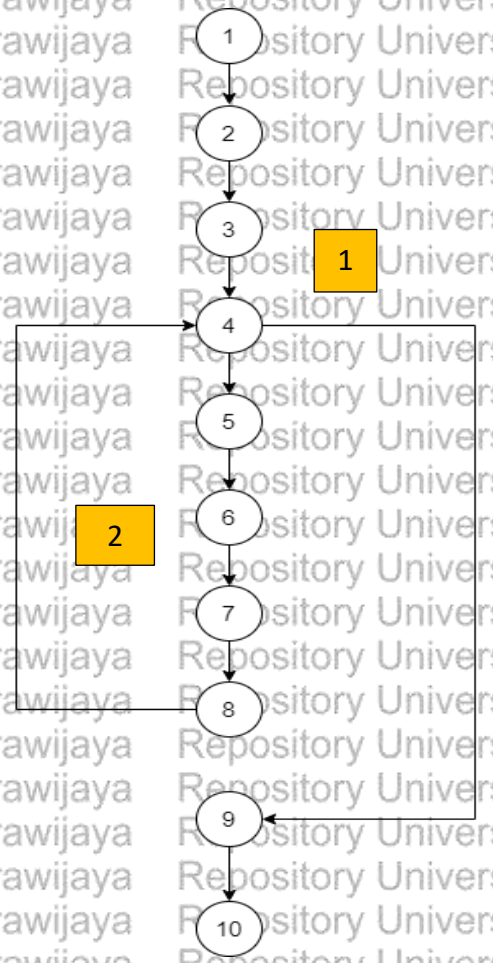
Tabel 6.3 Algoritme *method KalkulasiBrainMethod*

Pseudocode *Algoritme Method KalkulasiBrainMethod*	
1	Initialization list 1
2	Analysis file with library 2
3	Call function to get line of code 3
4	Looping each file for calling function from library 4
5	Move result from function unqualified name into array list 5
6	Move result from function length into array list 6
7	Move result from function cyclomatic complexity into array list 7
8	Move result from function top nesting level into array list 8
9	End for 9
10	Retun to unqualified name, length, cyclomatic complexity, top nesting level 10

2. *Basis Path*

Flow graph

Berdasarkan algoritme *method KalkulasiBrainMethod* yang dipaparkan akan menjadi dasar pembuatan *flow graph* ditunjukan dalam Gambar 6.2



Gambar 6.2 *Flow graph Method KalkulasiBrainMethod*

Cyclomatic Complexity

Berdasarkan gambar *flow graph KalkulasiBrainMethod* diatas diketahui jalur independent melalui perhitungan *cyclomatic complexity*. Berikut ini merupakan hasil perhitungan *cyclomatic complexity* berdasarkan *flow graph* method *KalkulasiBrainMethod*.

- Jumlah *region* = 2
- Jumlah *edge* – jumlah node + 2 = 10 – 10 + 2 = 2
- Jumlah predicated node + 1 = 1 + 1 = 2

Independent Path

- Jalur 1: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 4 – 9 – 10
- Jalur 2: 1 – 2 – 3 – 4 – 9 – 10

Berdasarkan *Independent Path* yang didapat akan dijadikan dasar kasus uji. Hasil pengujian unit dan kasus uji *method KalkulasiBrainMethod* akan dipaparkan pada Tabel 6.4.

Tabel 6.4 Hasil pengujian unit *method KalkulasiBrainMethod*

No	Kasus Uji	Hasil pengujian yang diharapkan	Hasil pengujian	Status
1	*Class Driver* memanggil *method KalkulasiBrainMethod* () Dengan data: File java	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya menampikan hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya menampikan hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Valid
2	*Class Driver* memanggil *method KalkulasiBrainMethod* () Dengan data: File java tanpa ada *method*	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya tidak terdapat nilai hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya tidak terdapat nilai hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Valid

6.1.3 Pengujian unit *method getMethodDeclaration*

1. Pseudecode

   Nama *Class*: Pasing

   Nama *Method*: getMethodDeclaration

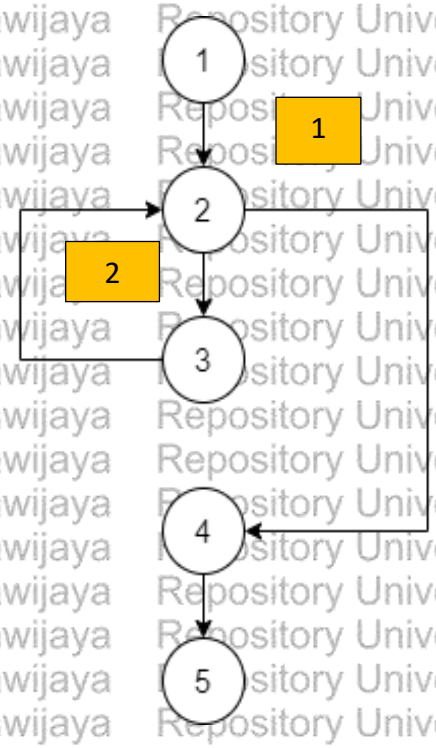
Tabel 6.5 Algoritme *method getMethodDeclaration*

Pseudocode *algoritme getMethodDeclaration*	
1	Initialization array list 1
2	Looping in each file for read file java source code 2
3	Move result method name to array list 3
4	End for 4
5	Return to variables 5

2. *Basis Path*

   Flow graph

Berdasarkan algoritme *method getMethodDeclaration* yang dipaparkan akan menjadi dasar pembuatan *flow graph* ditunjukan dalam Gambar 6.3



Gambar 6.3 *Flow graph method getMethodDeclaration*

Cyclomatic Complexity

Berdasarkan gambar *flow graph getMethodDeclaration* diatas diketahui jalur independent melalui perhitungan *cyclomatic complexity*.

Berikut ini merupakan hasil perhitungan *cyclomatic complexity* berdasarkan *flow graph method getMethodDeclaration*.

- Jumlah *region* = 2
- Jumlah *edge* – jumlah node + 2 = 5 – 5 + 2 = 2
- Jumlah predicated node + 1 = 1 + 1 = 2

Independent Path

- Jalur 1: 1 – 2 – 3 – 2 – 4 – 5
- Jalur 2: 1 – 2 – 4 – 5

Berdasarkan *Independent Path* yang didapat akan dijadikan dasar kasus uji. Hasil pengujian unit dan kasus uji *method getMethodDeclaration* akan dipaparkan pada Tabel 6.6

Tabel 6.6 Hasil Pengujian Unit *Method getMethodDeclaration*

No	Kasus Uji	Hasil pengujian yang diharapkan	Hasil pengujian	Status
1	*Class Driver* memanggil *method getMethodDeclaration*() Dengan data: File java	Menjalankan method getMethodDeclaration dan *print arraylist* hasilnya terdapat nama semua *method* di dalam kelas	Menjalankan method getMethodDeclaration dan *print arraylist* hasilnya terdapat nama semua *method* di dalam kelas	Valid
2	*Class Driver* memanggil *method getmethodDeclaration*() Dengan data: File java tanpa ada *method*	Menjalankan method getMethodDeclaration dan *print arraylist* hasilnya tidak terdapat nama semua *method* di dalam kelas	Menjalankan method getMethodDeclaration dan *print arraylist* hasilnya tidak terdapat nama semua *method* di dalam kelas	Valid

6.2 Pengujian Integrasi

Pengujian integrasi pada penelitian ini untuk memastikan bahwa interaksi antar kelas berjalan dengan baik. Pengujian integrasi pada penelitian ini dilakukan untuk menguji interaksi *class calculation* pada *method KalkulasiBrainMethod* dengan *class parsing* pada *method readFile*. Pengujian ini menggunakan teknik *bottom-up* yaitu dengan menguji modul tingkat bawah kemudian menguji modul tingkat atas. Dengan menggunakan *driver* untuk mensimulasi modul tingkat atas.

1. *Pseudecode*

Nama *Class*: *Calculation*

Nama *Method*: *KalkulasiBrainMethod*

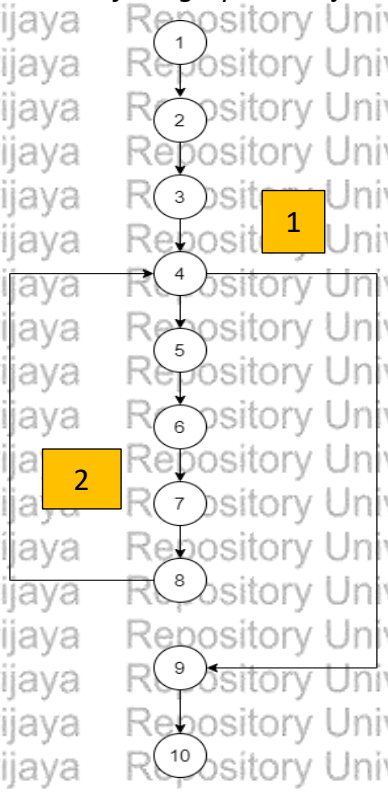
Tabel 6.7 Algoritme *method KalkulasiBrainMethod*

Pseudocode *Algoritme Method KalkulasiBrainMethod*	
1	Initialization list 1
2	Analysis file with library 2
3	Call function to get line of code 3
4	Looping each file for calling function from library 4
5	Move result from function unqualified name into array list 5
6	Move result from function length into array list 6
7	Move result from function cyclomatic complexity into array list 7
8	Move result from function top nesting level into array list 8
9	End for 9
10	Retun to unqualified name, length, cyclomatic complexity, top nesting level 10

2. *Basis Path*

Flow graph

Berdasarkan algoritme *method KalkulasiBrainMethod* yang dipaparkan akan menjadi dasar pembuatan *flow graph* ditunjukan dalam Gambar 6.4.



Gambar 6.4 *Flow graph Method KalkulasiBrainMethod*

Cyclomatic Complexity

Berdasarkan gambar *flow graph KalkulasiBrainMethod* diatas diketahui jalur independent melalui perhitungan *cyclomatic complexity*. Berikut ini merupakan hasil perhitungan *cyclomatic complexity* berdasarkan *flow graph* method *KalkulasiBrainMethod*.

- Jumlah *region* = 2
- Jumlah *edge* – jumlah node + 2 = 10 – 10 + 2 = 2
- Jumlah predicated node + 1 = 1 + 1 = 2

Independent Path

- Jalur 1: 1 – 2 – 3 – 4 – 5 – 6 – 7 – 8 – 4 – 9 – 10
- Jalur 2: 1 – 2 – 3 – 4 – 9 – 10

Berdasarkan *Independent Path* yang didapat akan dijadikan dasar kasus uji. Hasil pengujian unit dan kasus uji *method KalkulasiBrainMethod* akan dipaparkan pada Tabel 6.8.

Tabel 6.8 Hasil pengujian unit *method KalkulasiBrainMethod*

No	Kasus Uji	Hasil pengujian yang diharapkan	Hasil pengujian	Status
1	*Class Driver* memanggil *method KalkulasiBrainMethod* () yang memanggil *method readfile()* Dengan data: File java	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya menampikan hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya menampikan hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Valid

2	*Class Driver* memanggil *method KalkulasiBrainMethod* () yang memanggil method *readFile*() Dengan data: File java tanpa ada *method*	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya tidak terdapat nilai hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Menjalankan method *KalkulasiBrainMethod* dan *print arraylist* hasilnya tidak terdapat nilai hasil kalkulasi *line of class, line of class method, cyclomatic complexity, maxnesting level*	Valid

6.3 Pengujian Validasi

Pengujian validasi dilakukan pengujian setiap kebutuhan yang telah didefinisikan. Pengujian ini untuk memastikan bahwa setiap kebutuhan telah sesuai dengan skenario yang telah dibuat. Pengujian validasi akan menguji kebutuhan fungsional menggunakan metode *Blackbox*. Pada penelitian ini akan menguji tiga kebutuhan fungsional. Hasil pengujian dipaparkan pada sub bab 6.3.1.

6.3.1 Mendeteksi *code smell*

Pengujian validasi mendeteksi *code smell* untuk menguji kebutuhan fungsional kode CSD-F-02. Pengujian validasi mendeteksi *code smell* terdiri dari pengujian utama dan satu pengujian altenatif. Pengujian utama memastikan bahwa sistem berhasil mendeteksi *code smell*. Pada pengujian alternatif pertama sistem tidak bisa mendeteksi kode program selain java. Pengujian utama memasukan kode program java diuraikan pada Tabel 6.8. Pengujian alternatif satu diuraikan pada Tabel 6.9.

Tabel 6.9 Pengujian Validasi mendeteksi code smell

Pengujian validasi untuk mendeteksi code smell	
Nomor *Use case*	CSD-F-02
Nama *Use case*	Mendeteksi *code smell*
Tujuan	Menghitung nilai metriks dan mendeteksi *code smell*.
Data masukan	*file source code java*.

Prosedur Uji	1. *User* menekan tombol deteksi 2. Sistem mengambil nilai *Access to foreign data* pada setiap kelas. 3. Sistem mengambil nilai *Weighted method count* pada setiap kelas. 4. Sistem mengambil nilai *Tight class cohesion* pada setiap kelas. 5. Sistem mengambil nilai *Line of code* pada setiap kelas. 6. Sistem mengambil nilai *Line of code method* pada setiap *method.* 7. Sistem mengambil nilai *Maxnesting* pada setiap *method.* 8. Sistem mengambil nilai *Cyclomatic complexity* pada setiap *method*. 9. Sistem mengambil nilai *Number of accesed variable* pada setiap *method.* 10. Sistem membandingkan semua nilai dengan nilai *treshold*.
Hasil pengujian yang diharapkan	Sistem berhasil melakukan perhitungan nilai metriks dan mendeteksi *code smell*.
Hasil pengujian	Sistem berhasil melakukan perhitungan nilai metriks dan mendeteksi *code smell*.
Status	Valid

Tabel 6.10 Pengujian validasi mendeteksi *code smell* alternatif 1

Pengujian validasi untuk mendeteksi *code smell* alternatif 1	
Nomor *Use case*	CSD-F-02
Nama *Use case*	Mendeteksi *code smell*
Tujuan	Sistem tidak dapat membaca dan mendeteksi *code smell* dengan berkas selain java.
Data masukan	*file source code* selain *java*.

Prosedur Uji	1. *User* menekan tombol deteksi 2. Sistem membaca lalu melakukan perhitungan dan deteksi
Hasil pengujian yang diharapkan	Sistem tidak dapat membaca dan mendeteksi *code smell*.
Hasil pengujian	Sistem tidak dapat membaca dan mendeteksi *code smell*.
Status	Valid

6.3.2 Performa sistem

Pada pengujian performa sistem dilakukan dengan cara mengukur waktu yang dibutuhkan untuk mendeteksi *code smell god class* dan *brain class*. Waktu yang diambil adalah waktu sistem mendeteksi *code smell* pada setiap kelas. Berkas kode program yang dipakai adalah kode program yang berasal dari program lucene versi 2.9.2. Dengan mengambil sepuluh berkas dari 433 berkas pada program lucene. Pengujian validasi kebutuhan non fungsional peforma sistem diuraikan pada Tabel 6.11. dan hasil pengukuran waktu ditampilkan pada Tabel 6.12.

Tabel 6.11 Pengujian validasi peforma sistem

Pengujian validasi peforma sistem	
Nomor *Use case*	Kebutuhan non fungsional CSD-NF-01
Tujuan	Memastikan sistem harus mampu mendeteksi *code smell* jenis *god class* dan *barin class* dengan batas waktu kurang dari 1 s.
Data masukan	Sepuluh *file source code* lucene versi 2.9.2.
Prosedur Uji	1. klik tombol deteksi 2. Sistem menghitung waktu dari kalkulasi *software metrics* dan pendeteksian *code smell*.
Hasil pengujian yang diharapkan	Sistem berhasil menghitung waktu dari deteksi *code smell*. Dengan batas waktu kurang dari 1 s.
Hasil pengujian	Sistem berhasil menghitung waktu dari deteksi *code smell*. Dengan batas waktu kurang dari 1 s.
Status	Valid

Tabel 6.12 Hasil pengujian peforma sistem

No	Atfd	Wmc	Tcc	Loc	Hasil Deteksi	Waktu Deteksi	Status
1	1	9	0.05	45	Tidak terdapat *code smell*	0.0285s	Valid
2	30	91	0.07	348	*God class*	0.2681s	Valid
3	7	7	0.47	33	Tidak terdapat *code smell*	0.0275s	Valid
4	26	51	0.15	224	*God class*	0.2091s	Valid
5	104	65	0.12	359	*God class*	0.2952s	Valid
6	83	138	0.03	559	*God class*	0.3907s	Valid
7	93	257	0.03	984	*God class*	0.8259s	Valid
8	64	72	0.12	291	*God class*	0.2531s	Valid
9	236	230	0.07	892	*God class*	0.6234s	Valid
10	57	127	0.04	455	*God class*	0.3172s	Valid

6.3.3 Pengujian Akurasi

Pengujian akurasi pada penelitian ini dilakukan dengan cara membandingkan perhitungan yang dilakukan manual dengan perhitungan yang dilakukan oleh sistem. Pengujian akurasi ini menggunakan dua berkas dari sistem lucene 2.9.2. berkas tersebut dilakukan perhitungan *software metrics* secara manual dan dilakukan perhitungan secara otomatis. Lalu hasil perhitungan keduanya dibandingkan. Pengujian validasi kebutuhan non fungsional akurasi diuraikan pada Tabel 6.13 dan hasil uji pada Tabel 6.14 sampai 6.15.

Tabel 6.13 Pengujian validasi akurasi sistem

Pengujian validasi akurasi sistem	
Nomor *Use case*	Kebutuhan non fungsional CSD-NF-02
Tujuan	Sistem mampu mengkalkulasi *software metrics* dengan tingkat ketelitian minimal 90% pada setiap *class*.
Data masukan	*file source code java*.
Prosedur Uji	1. Melakukan deteksi manual 2. Sistem menghitung kalkulasi *software metrics*.

Hasil pengujian yang diharapkan	Sistem mampu mengkalkulasi *software metrics* dengan tingkat ketelitian minimal 90% pada setiap *class*.
Hasil pengujian	Sistem mampu mengkalkulasi *software metrics* dengan tingkat ketelitian minimal 90% pada setiap *class*.
Status	Valid

Tabel 6.14 Hasil perhitungan nilai kalkulasi *software metrics* dan deteksi secara manual

No	Class	Atfd	Wmc	Tcc	Loc	Nilai *Brain Method*	Hasil Deteksi
1	AttributeSource	48	75	0.2077922	301	0	Terdapat god class
2	FieldComparator	31	138	0.0100616	590	0	Terdapat *god class*

Tabel 6.15 Hasil perhitungan nilai kalkulasi *software metrics* dan deteksi secara otomatis

No	Class	Atfd	Wmc	Tcc	Loc	Nilai *Brain Method*	Hasil Deteksi
1	AttributeSource	48	75	0.21	301	0	Terdapat god class
2	FieldComparator	31	138	0.01	590	0	Terdapat *god class*

6.4 Pembahasan Pengujian

Hasil pengujian unit, pengujian integrasi dan pengujian validasi telah dilakukan menghasilkan status valid pada seluruh kasus uji. Pengujian unit dilkakukan pada ketiga *method* yaitu *readfile, KalulasiBrainMethod* dan *getMethodDeclaration* yang menghasilkan 6 kasus uji. Enam kasus uji tersebut terdiri dari dua kasus uji untuk *method readfile,* dua kasus uji untuk *method KalkulasiBrainMethod* dan dua kasus uji untuk *method getMethodDeclaration*. Dari ke enam kasus uji tersebut menghasilkan status valid.

Pada pengujian integrasi yang dilakukan pada method KalkulasiBrainMethod yang berada pada kelas *calculation*. Integrasi dengan kelas *parsing* pada *method readFile* menghasilkan dua kasus uji dan semua kasus uji tersebut menghasilkan status valid.

Progam deteksi *code smell god class* dan *brain class* memiliki kebutuhan fungsional yang telah didefinisikan pada proses perancangan. Terdapat satu kebutuhan fungsional diantaranya adalah mendeteksi *code smell*. Kebutuhan fungsional tersebut telah diuji pada pengujian validasi dan menghasilkan status valid pada ketiga kasus uji. Kebutuhan non fungsional pada program deteksi ini telah diuji pada pengujian validasi performa yang menghasilkan status valid pada kasus uji mendeteksi *code smell* menggunakan waktu kurang dari 1 detik pada setiap kelas. Pengujian validasi akurasi pada sistem ini menghasilkan status valid akurasi 100 persen dengan membandingkan nilai kalkulasi dan deteksi secara manual dengan nilai kalkulasi dan deteksi secara otomatis.

BAB 7 PENUTUP

1.1 Kesimpulan

Berdasarkan hasil rekayasa kebutuhan, perancangan, implementasi, dan pengujian pada penelitian ini dapat disimpulkan bahwa:

1. Pada tahap rekayasa kebutuhan pengembangan sistem deteksi *code smell god class* dan *brain class* diperoleh kebutuhan fungsional dan kebutuhan non fungsional. Yang dapat membantu *developer* untuk membantu mendeteksi *code smell* jenis *god class* dan *brain class* untuk kode sumber java. Kebutuhan fungsional yang dihasilkan yaitu mendeteksi *code smell*. Kebutuhan fungsional telah didefinisikan dan dimodelkan dalam bentuk *use case diagram* dan *use case scenario*.
2. Pada tahap perancangan diperoleh hasil perancangan arsitektur berupa *sequence Diagram*, dan *class diagram*. Pada proses perancangan data diperoleh perancangan algoritme yang akan digunakan pada sistem. Pada proses perancangan antarmuka diperoleh *layout* perancangan *user interface* yang akan digunakan untuk implementasi sistem.
3. Pada tahap implementasi diperoleh spesifikasi sistem, implementasi kode program dan implementasi antarmuka berdasarkan hasil perancangan yang telah dilakukan pada tahap perancangan.
4. Pada tahap pengujian diperoleh hasil pengujian unit, pengujian validasi, pengujian peforma dan pengujian akurasi yang menghasilkan nilai 100% valid.

1.2 Saran

Saran yang dieberikan untuk pengembangan lebih lanjut dari pengembangan sistem deteksi *code smell* jenis *god class* dan *brain class* antara lain:

1. Nilai akurasi dapat ditingkatkan lagi dengan cara mendeteksi pada *inner class*.
2. Sistem dapat dikembangkan kembali dengan mendeteksi menggunakan berkas selain kode program java.

DAFTAR PUSTAKA

Anderson, J., McRee, J. dan Wilson, R., 2010. *Effective UI: The Art of Building Great User*.Canada:*O'Reilly Media*.

Aristyagama, Y.H., 2016. Framework Deteksi Bad Smell Code Semi Otomatis untuk Pemrograman Tim.[Online] tersedia di <https://www.researchgate.net/publication/303405055 Framework Deteksi Bad Smell Code Semi Otomatis untuk Pemrograman Tim> [Diakses 2 Februari 2020].

Biema, J.M. dan Kang, B.-K., 1995. *ACM SIGSOFT Software Enginering Notes.* Cohesion and Reuse in an OO System.,[e-Journal] 20. Tersedia melalui ACM Digital Library <https://dl.acm.org/doi/10.1145/223427.211856>. [Diakses 5 Februari 2020].

Chidamber, S.R. dan Kemerer, C.F., 1994. *IEEE Transactions on Software Engineering*. A Metrics Suite for Object Oriented Design., [E-Journal] 20(6), hal.476–493. Tersedia melalui IEEE Xplore < https://ieeexplore.ieee.org/document/295895 >. [Diakses 10 Februari 2020]

Kurniawan, T.A., 2018. Jurnal Teknologi Informasi dan Ilmu Komputer. Pemodelan Use Case (UML): Evaluasi Terhadap beberapa Kesalahan dalam Praktik.[E-journal] 5(1), hal.77. Tersedia pada Universitas Brawijaya < http://jtiik.ub.ac.id/index.php/jtiik/article/view/610> [Diakses 15 Mei 2020]

Lanza, M. dan Marinescu, R., 2006. *Object-Oriented Metrics In Practice*. Berlin : Springer . Tersedia di < https://www.springer.com/gp/book/9783540244295> [Diakses 11 Februari 2020]

Leach, R.J., 2016. *Introduction to Software Engineering*. 2 ed. Washington: CRC Press.

Marinescu, R., 2005. *IEEE International Conference on Software. Maintenance* Measurement and quality in object-oriented design., hal.701–704. Tersedia melalui IEEE Xplore <https://ieeexplore.ieee.org/document/1510177> [Diakses 13 Februari 2020]

Marinescu, R., Ratiu, D. dan Gırba, T., 2004. Software Maintenance and Reengineering. Using History Information to Improve Design Flaws Detection. Tersedia melalui IEEE Xplore < https://ieeexplore.ieee.org/document/1281423> [Diakses 5 Maret 2020]

Mccabe, T.J., 1976. *IEEE Transactions on Software Engineering*. A Complexity Measure. SE-2(4), hal.308–320. Tersedia melalui IEEE Xplore < https://ieeexplore.ieee.org/document/1702388> [Diakses 21 Mei 2020].

Olbrich, S.M., Cruzes, D.S. dan Sjoøberg, D.I.K., 2010. *IEEE International Conference on Software Maintenance.* Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. Tersedia melalui IEEE Xplore < https://ieeexplore.ieee.org/document/5609564 > [Diakses 3 Maret 2020]

Permatasari, D.I., 2020. *Jurnal Sistem dan Teknologi Informasi (JUSTIN).* Pengujian Aplikasi menggunakan metode Load Testing dengan Apache

JMeter pada Sistem Informasi Pertanian. [E-journal] 8(1), hal.135. Tersedia melalui <http://jurnal.untan.ac.id/index.php/justin/article/view/34452> [Diakses 17 Maret 2020]

Pressman, R.S., 2010. *Software Engineering Seventh Edition* .New York:McGraw-Hill

Rubal, C. dan Arya, A., 2017. *International Journal for Research in Applied Science & Engineering Technology*. Optical Character Recognition.[E-Journal] 5(VI), hal.2321–9653. Tersedia melalui < http://dx.doi.org/10.23956/ijarcsse/V7I7/0158 > [Diakses 20 Maret 2020]

Sommerville, 2011. *Software Engineering Ninth Edition*. London:*Addison-Wesley*

TIOBE 2020. TIOBE *Index for* Maret 2020. Tersedia di < <https://www.tiobe.com/tiobe-index/> [Diakses 19 Maret 2020].

Paiva Thanis, Damasceno Amanda et all., 2017. *Journal of Software Engineering Research and Development*. *On the Evaluation of Code Smells and Detection Tools*. [E-Journal] tersedia melalui < https://link.springer.com/article/10.1186/s40411-017-0041-1> [Diakses 14 August 2020].