PENGEMBANGAN SISTEM PERHITUNGAN KOMPLEKSITAS KODE SUMBER BERDASARKAN METRIK *HALSTEAD* DAN *CYCLOMATIC COMPLEXITY*

SKRIPSI

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Fredy Nendra Pranata
NIM: 115060801113001

PROGRAM STUDI INFORMATIKA
PROGRAM TEKNOLOGI INFORMASI DAN ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2016

PENGESAHAN

PENGEMBANGAN SISTEM PERHITUNGAN
KOMPLEKSITAS KODE SUMBER BERDASARKAN
METRIK *HALSTEAD* DAN *CYCLOMATIC COMPLEXITY*

SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Fredy Nendra Pranata
NIM: 115060801113001

Skripsi ini telah diuji dan dinyatakan lulus pada
21 Januari 2016
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Fajar Pradana, S.ST., M.Eng.	Tri Astoto Kurniawan, ST., MT., Ph.D
NIP. 19871121 201504 1 004	NIP. 19710518 200312 1 001

Mengetahui
Ketua Program Studi Informatika

Drs. Marji, M.T.
NIP. 19670801 199203 1 001

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 pustaka.

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, 12 Januari 2016

Fredy Nendra Pranata

NIM: 115060801113001

KATA PENGANTAR

Dengan menyebut nama Allah Yang Maha Pengasih dan Penyayang. Segala puji bagi Allah Subhanahu wa ta'ala karena hanya atas limpahan nikmat dan rahmat karunia-Nya, penulis dapat menyelesaikan skripsi yang berjudul "Pengembangan Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity*".

Penyusunan skripsi tidak lepas dari bantuan semua pihak yang terus memberikan bimbingan, kritik, saran, dukungan, motivasi maupun doa dari orang tua. Oleh karena itu, ucapan terima kasih penulis sampaikan kepada:

1. Ayahanda Mujianto dan Ibunda Umi Musarofah selaku orang tua penulis dan seluruh keluarga yang telah memberikan doa dan dukungan penuh dalam penyusunan skripsi ini.
2. Bapak Fajar Pradana, S.ST., M.Eng. selaku pembimbing 1 dan Bapak Tri Astoto Kurniawan, ST., MT., Ph.D selaku pembimbing 2, atas segala bimbingan dan waktu yang telah diluangkan serta kritik dan saran yang diberikan kepada penulis.
3. Seluruh bapak dan ibu dosen Fakultas Ilmu Komputer Universitas Brawijaya atas segala bimbingan dan ilmu yang telah diajarkan kepada penulis.
4. Teman-teman penulis khususnya Galuh Yudha Mahardika, Bima Surya Nugraha, Dito Rizki Pramudeka, Kenneth Setiawan Sarashadi, S.Kom, Fathra Primadhana, S.Kom dan Yudhistira Wirayudha Pratama, S.Kom yang memberikan dukungan, semangat dan motivasi.
5. Seluruh mahasiswa Program Studi Informatika angkatan 2011.
6. Semua pihak yang tidak dapat disebutkan satu per satu yang terlibat baik secara langsung maupun tidak langsung demi terselesaikannya skripsi ini.

Semoga jasa dan amal baik mendapatkan balasan dari Allah Subhanahu wa ta'ala. Penulis menyadari bahwa skripsi ini masih jauh dari sempurna serta banyak kekurangan disebabkan oleh keterbatasan kemampuan dan ilmu yang dimiliki penulis. Semoga laporan skripsi ini bermanfaat bagi pembaca dan pengembang selanjutnya.

Malang, 12 Januari 2016

Penulis

ABSTRAK

Perangkat lunak memainkan peran penting di dunia saat ini, namun banyak terjadi kegagalan pada perangkat lunak yang menyebabkan permintaan perangkat lunak berkualitas semakin tinggi. Pengukuran kompleksitas kode sumber diperlukan untuk mendukung proses pendeteksian cacat sedini mungkin pada perangkat lunak dan menjamin kualitas perangkat lunak. Informasi kompleksitas kode sumber dapat digunakan sebagai indikator kemungkinan cacat pada perangkat lunak. Dalam penelitian ini menggunakan dua metode pengukuran kompleksitas kode sumber yaitu, *Halstead's Volume* dan *Cyclomatic Complexity*. *Metrik Halstead* digunakan untuk mengevaluasi dan melakukan pengukuran kode sumber berdasarkan pada *operator* dan *operand* sedangkan, *Cyclomatic Complexity* digunakan untuk mengukur dan mengontrol jalur alur program. Pengukuran atau perhitungan kompleksitas kode sumber dilakukan dengan cara melakukan proses *parsing code* menggunakan *library Java Parser and AST. Parsing code* terhadap *file* kode sumber dilakukan untuk mendapatkan *predicate nodes*, *operand* dan *operator* kemudian dihitung kompleksitasnya menggunakan metode *Halstead's Volume* dan *Cyclomatic Complexity*. Data dari hasil perhitungan kompleksitas tersebut dapat digunakan untuk mengetahui kemungkinan cacat pada perangkat lunak. Dari hasil perhitungan akurasi kompleksitas kode sumber terhadap 30 *class* data uji dengan 96 *method* didalamnya, diperoleh hasil akurasi untuk metode *Halstead's Volume* sebesar 87.5% dan metode *Cyclomatic Complexity* sebesar 100%. Berdasarkan hasil pengujian unit, integrasi dan validasi sistem menunjukan keberhasilan serta telah memenuhi kebutuhan fungsional dan non fungsional.

Kata kunci: Kompleksitas Kode Sumber, *Halstead's Volume*, *Cyclomatic Complexity*, Deteksi Cacat Perangkat Lunak

ABSTRACT

The software plays an important role in today's world, however much a failure of software that causes the demand for increasingly high-quality software. Measurement of the complexity of the source code needed to support the process of defect detection as early as possible in the software and ensure software quality. Information complexity of the source code can be used as an indicator of the possibility of defects in software. In this study using two measurement methods, namely the source code complexity, Halstead's volume and Cyclomatic Complexity. Halstead metrics used to evaluate and take measurements based on the source code of the operator and operand while, Cyclomatic Complexity used to measure and control the flow path of the program. Measurement or calculation of the complexity of the source code is performed by the parsing code uses a Java library parser and AST. Parsing code to the source code file is done to get the predicate nodes, operand and the operator is then calculated using the method complexity Halstead's volume and Cyclomatic Complexity. Data from the complexity of the calculation results can be used to determine the possibility of defects in software. From the calculation accuracy of the complexity of the source code for class 30 with 96 test data method, in which the accuracy of the results obtained for the method of Halstead's volume amounted to 87.5% and the method Cyclomatic Complexity of 100%. Based on the results of the test unit, integration and system validation indicates success and have met the functional and non-functional requirements.

Keywords: *Source Code Complexity, Halstead Volume, Cyclomatic Complexity, Software Defect Detection*

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Dunia *Information Technology* (IT) berkembang sangat pesat hingga kini. Beberapa aspek kehidupan tidak dapat lepas dari kebutuhan teknologi informasi, mulai dari yang paling sederhana hingga yang paling kompleks di dunia bisnis. Berbagai instansi pemerintahan maupun swasta, baik di dalam maupun luar negeri menggunakan perangkat lunak (*sofware*) sebagai kebutuhan esensial. Perangkat lunak memainkan peran penting di dunia saat ini, namun banyak terjadi kegagalan pada perangkat lunak. Sebuah survei terbaru dari 800 manajer IT mengatakan bahwa 62% dari total perangkat lunak gagal, 49% perangkat lunak yang gagal tersebut menderita biaya anggaran yang melebihi batas, 47% memiliki biaya pemeliharaan lebih tinggi dari yang diharapkan dan 41% gagal untuk memberikan nilai bisnis yang diharapkan (Dalal, 2012). Hal tersebut menyebabkan permintaan untuk perangkat lunak berkualitas semakin tinggi.

Satuan ukuran kualitas perangkat lunak pada saat proses rekayasa dapat meliputi kompleksitas program, modularitas yang efektif dan besarnya program (Fathoni, 2009). Dalam siklus hidup pengembangan perangkat lunak ada 2 teknik yang digunakan untuk menjamin kualitas dan keandalan perangkat lunak yaitu, *software testing* dan *software metrics* (Madhavji, 1991). *Software testing* adalah istilah yang digunakan untuk menemukan kesalahan saat pengembangan atau pembangunan perangkat lunak. Sedangkan, *software metrics* digunakan untuk meningkatkan kualitas perangkat lunak dengan memantau kompleksitas perangkat lunak (Madhavji, 1991). Kearney (1986) mendefinisikan kompleksitas perangkat lunak sebagai sejauh mana sistem atau komponen memiliki desain atau implementasi yang sulit dimengerti dan diverifikasi. Salah satu metode yang paling sering digunakan oleh tim pengembangan untuk mengukur kualitas internal perangkat lunak adalah analisis statis dari kode sumber (Tomas, 2013). Analisis statis kode merupakan seperangkat teknik analisis dimana perangkat lunak dikaji tidak dieksekusi (Tomas, 2013).

Maka dari itu, dalam penelitian ini menitikberatkan pada pengukuran kompleksitas kode sumber untuk mendukung memenuhi kebutuhan perangkat lunak yang berkualitas. Kode sumber (*source code*) merupakan setiap deskripsi perintah yang dapat dieksekusi oleh perangkat lunak (Harman, 2010). Selain itu, kode sumber juga diperlukan sebagai informasi untuk mengidentifikasi dan memperbaiki masalah tentang sebuah *bug* (Grant, 2013). Kompleksitas kode sumber dapat diukur dengan beberapa metrik yaitu LOC *(Line of Code)*, *McCabe's Cyclomatic Complexity* dan *Halstead's Volume* (Zhang, 2007). Dalam penelitian ini, kompleksitas kode sumber dihitung menggunakan dua metode metrik yaitu *Halstead's Volume* dan *McCabe's Cyclomatic Complexity*. *McCabe's Cyclomatic Complexity* dan *Halstead's Volume* adalah atribut yang secara umum digunakan untuk menghitung tingkat kompleksitas kode program (Zhang, 2007).

Metrik *Halstead* digunakan untuk mengevaluasi dan melakukan pengukuran kode sumber berdasarkan pada *operator* dan *operand*. Sedangkan, *Cyclomatic Complexity* merupakan pendekatan ukuran kompleksitas yang dilakukan untuk mengukur dan mengontrol jumlah alur melalui program (McCabe, 1976).

Kompleksitas kode program adalah salah satu informasi dimana dapat digunakan sebagai indikator kemungkinan terjadinya cacat (*defect*) pada sebuah kode program (Priyambadha, 2013). Pengukuran kompleksitas diperlukan untuk mendukung proses pendeteksian cacat sedini mungkin pada perangkat lunak dan menjamin kualitas perangkat lunak. Dengan pengukuran kompleksitas tersebut dapat diketahui kemungkinan cacat pada perangkat lunak. Jika terdapat kecacatan pada perangkat lunak, maka otomatis akan semakin meningkatkan biaya *maintenance* perangkat lunak dalam jangka panjang. Perangkat lunak yang memiliki cacat akan mengakibatkan menurunya kualitas dari perangkat lunak, hal itu tentu tidak diinginkan dan harus dihindari. Oleh sebab itu, dibututuhkan sebuah *tools* yang dapat mengukur kompleksitas kode sumber, terutama pada pemrograman berorientasi objek khususnya bahasa java.

Java merupakan bahasa pemrograman yang paling populer dan masih favorit digunakan oleh *developer* dibandingkan dengan bahasa yang lainnya. Selain itu, bahasa pemrograman java juga merupakan *development tools* yang fleksibel dan *powerfull* (Cahyono, 2006). Menurut indek TIOBE 2011 bahasa Java berada pada posisi sebagai bahasa yang terbanyak digunakan dengan presentase sebesar 19,711% (Tomas, 2013). Maka, penelitian ini menggunakan bahasa Java sebagai pembuatan *tools* dan objek studi kasusnya.

1.2 Rumusan Masalah

1. Bagaimana mengembangkan sistem perhitungan kompleksitas kode sumber menggunakan metode *Halstead's Volume* dan *Cyclomatic Complexity*?
2. Bagaimana menerapkan metode *Halstead's Volume* dan *Cyclomatic Complexity* untuk perhitungan kompleksitas kode sumber?
3. Bagaimana mengukur validitas algoritma metode *Halstead's Volume* dan *Cyclomatic Complexity*?

1.3 Batasan Masalah

Penelitian ini dibatasi oleh hal-hal sebagai berikut.

1. Pembahasan difokuskan untuk memberikan informasi kompleksitas kode sumber menggunakan metode *Halstead* dan *Cyclomatic Complexity.*
2. Data atau kode sumber yang dianalisis dan dihitung kompleksitasnya adalah kode sumber Java.
3. Hasil penelitian berupa *tools* berbasis Java.

1.4 Tujuan

Berdasarkan latar belakang, tujuan dari penelitian ini antara lain:

1. Membuat sistem perhitungan kompleksitas kode sumber.

2. Menerapkan metode *Halstead's Volume* dan *Cyclomatic Complexity* untuk perhitungan kompleksitas kode sumber.

1.5 Manfaat

Adapun manfaat dari penelitian ini adalah sebagai berikut.

1. Hasil dari penelitian ini dapat membantu memudahkan perhitungan kompleksitas kode sumber pada perangkat lunak berdasarkan metrik *Halstead's Volume* dan *Cyclomatic Complexity*.
2. Hasil pengukuran kompleksitas dapat digunakan untuk mengetahui kemungkinan cacat (*defect*) pada perangkat lunak.

1.6 Sistematika Penulisan

Sistematika penulisan dalam penelitian ini adalah sebagai berikut.

BAB 1 PENDAHULUAN

Bab ini menjelaskan mengenai latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat dan sistematika penulisan.

BAB 2 LANDASAN KEPUSTAKAAN

Bab ini membahas tentang teori, temuan dan bahan penelitian sebelumnya yang diperoleh dari berbagai referensi yang dijadikan dasar melakukan penelitian.

BAB 3 METODOLOGI

Bab ini berisi metode atau langkah-langkah yang digunakan dalam penelitian skripsi pengembangan sistem perhitungan kompleksitas.

BAB 4 REKAYASA KEBUTUHAN

Bab ini berisi analisis kebutuhan sistem perhitungan kompleksitas kode sumber berdasarkan metrik *Halstead* dan *Cyclomatic Complexity*.

BAB 5 PERANCANGAN DAN IMPLEMENTASI

Bab ini berisi penjelasan mengenai perancangan dan implementasi perangkat lunak yang dikembangkan.

BAB 5 PENGUJIAN

Bab ini menjelaskan mengenai hasil pengujian dan analisis yang dilakukan terhadap sistem perangkat lunak.

BAB 5 PENUTUP

Bab ini berisi kesimpulan yang diperoleh berdasarkan hasil penelitian dan saran untuk pengembangan penelitian dimasa yang akan datang.

BAB 2 LANDASAN KEPUSTAKAAN

Beberapa dasar teori yang terkait dalam penelitian ini adalah pengukuran kompleksitas kode sumber, *software process model* dan UML (*Unified Modelling Language*).

2.1 Pengukuran Kompleksitas Kode Sumber

Pengukuran merupakan dasar dari setiap disiplin rekayasa dan berlaku juga dalam perekayasaan perangkat lunak. Menurut *IEEE Standard Glossary of Software Engineering Terminology* (1990), pengukuran adalah ukuran kuantitatif dari tingkat dimana sebuah sistem, komponen atau proses memiliki atribut tertentu. Sedangkan, mengukur adalah mengindikasikan kuantitatif dari luasan, jumlah, dimensi dan kapasitas. Pengukuran kompleksitas adalah pengukuran yang berdasarkan kode dan nilai metrik perangkat lunak, pengukuran kompleksitas dapat digunakan untuk memahami kompleksitas perangkat lunak (Sharma, 2010).

Kode sumber merupakan setiap deskripsi perintah yang dapat dieksekusi oleh perangkat lunak (Harman, 2010). Kode sumber memungkinkan *programmer* untuk berkomunikasi dengan komputer menggunakan beberapa perintah yang telah terdefinisi. Selain itu, kode sumber juga diperlukan sebagai informasi untuk mengidentifikasi dan memperbaiki masalah tentang sebuah *bug* (Grant, 2013). Sedangkan, kompleksitas kode program adalah salah satu informasi dimana dapat digunakan sebagai indikator kemungkinan terjadinya cacat (*defect*) pada sebuah kode program (Priyambadha, 2013). Kompleksitas kode sumber merupakan salah satu acuan dalam prediksi cacat pada perangkat lunak.

Terdapat beberapa metode pengukuran kompleksitas pada perangkat lunak, yaitu LOC *(Line of Code)*, *McCabe's Cyclomatic Complexity* dan *Halstead's Volume* (Zhang, 2007). LOC adalah suatu teknik pengukuran dengan cara menghitung jumlah keseluruhan baris kode pada seluruh *file* kode sumber, sedangkan *McCabe's Cyclomatic Complexity* dan *Halstead's Volume* adalah atribut yang secara umum digunakan untuk menghitung tingkat kompleksitas kode sumber (Zhang, 2007). Metrik *Halstead* digunakan untuk mengevaluasi dan melakukan pengukuran kode sumber berdasarkan pada *operator* dan *operand*. Dan *Cyclomatic Complexity* merupakan pendekatan ukuran kompleksitas yang dilakukan untuk mengukur dan mengontrol jumlah alur melalui program (McCabe, 1976). Semakin tinggi angka *Cyclomatic Complexity* akan meningkatkan *error* serta pemeliharaanya (Ankita, 2014). Sehingga, semakin rendah kompleksitas maka semakin tinggi kualitasnya. Perhitungan hasil kompleksitas berdasarkan metrik *Halstead's Volume* dan *Cyclomatic Complexity* tersebut berperan sebagai pendukung proses pendeteksian cacat secara dini pada perangkat lunak. Dalam penelitian ini, pengembangan sistem perhitungan kompleksitas kode sumber menggunakan *library Java Parser and AST*.

2.1.1 *Halstead Metrics*

Metrik *Halstead* merupakan metrik yang digunakan untuk mengevaluasi dan melakukan pengukuran kode sumber berdasarkan pada *operator* dan *operand*. Dengan mengidentifikasi *operator* dan *operand* dari program, kemudian mengidentifikasi unik *operator* dan *operand* maka dapat diidentifikasi atribut untuk perangkat lunak tertetu seperti, *program length*, *difficult level* dan sebagainya (Ankita, 2014). Kompleksitas *Halstead* adalah perangkat lunak metrik yang diklarifikasikan sebagai kompleksitas metrik komposit.

Perangkat lunak metrik tersebut diperkenalkan oleh Maurice Howard Halstead pada tahun 1977 sebagai bagian dari ilmu empiris pengembangan perangkat lunak. Halstead membuat pengamatan bahwa metrik perangkat lunak harus mencerminkan pelaksaan atau ekspresi algoritma dalam bahasa yang berbeda, tetapi independen eksekusinya pada platform spesifik. Oleh karena itu, metrik dihitung statis dari kode (Chhabra, 2014).

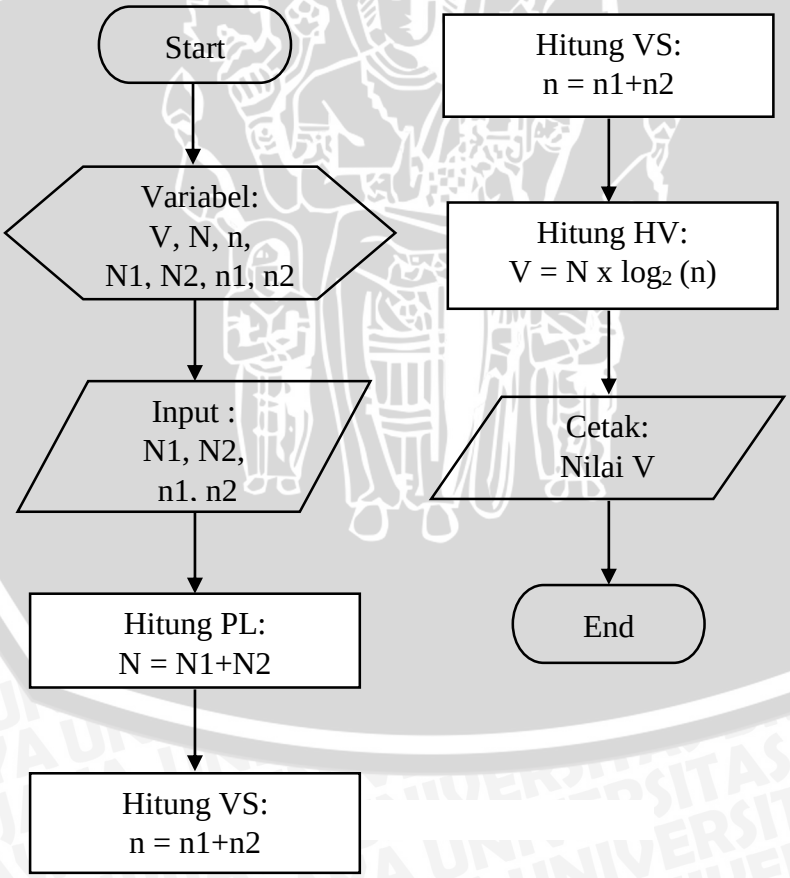
Metrik *Halstead* didefinisikan sebagai berikut.

1. *Program Length* (N)
   Program length (N) adalah jumlah dari total *operator* dan *operand* dalam program.
   N = N1 + N2
   Dimana:
   N1 = Total jumlah kemunculan *operator* dalam program
   N2 = Total jumlah kemunculan *operand* dalam program
2. *Vocabulary Size* (n)
   Vocabulary size (n) adalah jumlah dari total *operator* dan *operand* yang unik.
   n = n1 + n2
   Dimana:
   n1 = Total *operator* unik dalam program
   n2 = Total *operand* unik dalam program
3. *Program Volume* (V)
   Program Volume (V) adalah isi informasi dari program, diukur dalam bit matematika. Dihitung berdasarkan hasil perkalian dari *program length* dan logaritma basis 2 *vocabulary size*.
   $V = N \times \log_2 (n)$
4. *Difficulty Level* (D)
   Difficulty level adalah tingkat kesulitan atau kesalahan rawan dari progam.
   Dihitung dengan rumus:
   D = (n1/2) x (N2/n2)
   Jika *operand* yang sama digunakan berkali-kali dalam program, maka akan lebih rentan terhadap *errors*.
5. *Program Level* (L)
   Program level (L) adalah kebalikan dari *difficulty level* (D).
   Dihitung dengan rumus:
   L = 1 / D

Program tingkat rendah lebih rentan terhadap kesalahan dari program tingkat tinggi.

6. *Effort to Implement* (E)
   Effort to implement atau *understand a program* diperoleh dari perkalian nilai *program volume* dan *difficulty level*.
   E = V x D
7. *Time to Implement* (T)
   T = E / 18
   Halstead telah menemukan bahwa *time to implement* diperoleh dari pembagian *effort* dengan 18 perkiraan waktu dalam detik.
8. *Number of Delivered Bugs* (B)
   Number of delivered bugs (B) berkorelasi dengan kompleksitas keseluruhan dari perangkat lunak.
   Dihitung dengan rumus:
   $B = (E^{2/3}) / 3000$

Rumus *Halstead Metrics* yang digunakan dalam menghitung kompleksitas kode sumber pada penelitian ini antara lain adalah *Program Length*, *Vocabulary Size* dan *Program Volume* atau *Halstead's Volume*. Berikut ini *flowchart* dari proses perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.



Gambar 2.1 *Flowchart* Metode *Halstead's Volume*

Mengacu pada Chhabra (2014)

Berdasarkan pada Gambar 2.1 maka dapat dijabarkan perhitungannya seperti dibawah ini:

V = *Program Volume* (*Halstead's Volume*)

N = *Program Length*

n = *Vocabulary Size*

N1 = Total jumlah kemunculan *operator* dalam program

N1 = Total jumlah kemunculan *operand* dalam program

n1 = Total *operator* unik dalam program

n2 = Total *operand* unik dalam program

$V = N \times \log_2 (n)$

$V = (N1+N2) \times \log_2 (n1+n2)$

$V = (27+34) \times \log_2 (11+9)$

$V = 61 \times \log_2 (20)$

$V = \sim 263.6376137881291$

2.1.2 *Cyclomatic Complexity*

McCabe 1967 mengusulkan bahwa ukuran dari kompleksitas adalah jumlah kemungkinan jalur (*path*) yang dilalui perangkat lunak saat dieksekusi. Salah satu strategi pengujian yang disebut dasar jalur pengujian (*basis path testing*) oleh McCabe yang pertama kali diusulkan untuk menguji setiap linear jalur independen program (Agarwal, 2013). Kompleksitas *Cyclomatic* juga dikenal sebagai kompleksitas struktural menghitung jumlah dari jalur independen program.

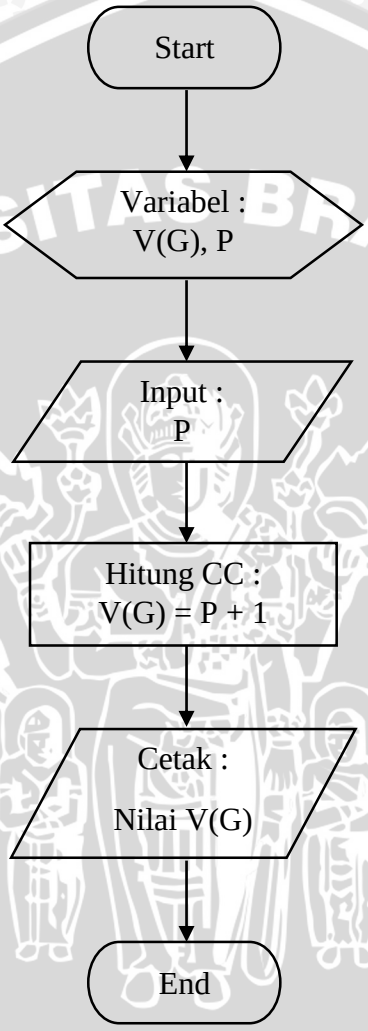
Cyclomatic Complexity merupakan perangkat lunak metrik yang menyediakan ukuran kuantitatif dari kompleksitas logikal suatu program. Pendekatan ukuran kompleksitas dilakukan untuk mengukur dan mengontrol jumlah alur melalui program (McCabe, 1976). Ketika digunakan dalam konteks metode uji coba berbasis alur, nilai yang dikomputasi untuk *Cyclomatic Complexity* mendefinisikan jumlah *independent path* dalam himpunan basis suatu program dan menyediakan batas atas sejumlah uji coba yang harus dilakukan untuk memastikan bahwa seluruh perintah telah dieksekusi sedikitnya satu kali (Saini, 2014).

Cyclomatic Complexity dapat dihitung dengan salah satu dari 3 formula antara lain:

1. Jumlah *region* dari *graf* alur sesuai dengan *Cyclomatic Complexity*.
2. *Cyclomatic Complexity* untuk *graf* alur G didefinisikan:
   V(G) = E – N + 2
   Dimana E = jumlah *edge*, dan N = jumlah *node*

3. *Cyclomatic Complexity* untuk *graf* alur G didefinisikan:
   V(G) = P + 1
   Dimana P = jumlah *predicate nodes*.

Formula *Cyclomatic Complexity* yang digunakan dalam menghitung kompleksitas kode sumber pada penelitian ini menggunakan jumlah *predicate nodes* untuk graf alur G. Berikut ini *flowchart* dari proses perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.



Gambar 2.2 *Flowchart* Metode *Cyclomatic Complexity*

Mengacu pada McCabe (1976)

Berdasarkan pada Gambar 2.2 maka dapat dijabarkan perhitungannya seperti berikut ini.

V(G) = *Cyclomatic Complexity*

P = Jumlah *Predicate Nodes*

V(G) = P + 1

V(G) = 5 + 1

V(G) = 6

Beberapa fakta membuktikan jika progam dengan jumlah *Cyclomatic Complexity* yang lebih besar dari 10 maka program memilik probabilitas yang lebih tinggi terhadap *errors* dan cacat. Semakin tinggi angka *Cyclomatic Complexity* akan meningkatkan *error* serta pemeliharaanya (Ankita, 2014). *Software Engineering Institute*, mengkategorikan evaluasi resiko sesuai dengan *Cyclomatic Complexity* sebagai berikut.

Tabel 2.1 Evaluasi Resiko *Cyclomatic Complexity*

Cyclomatic Complexity	***Risk Evaluation***
1-10	*Software* bebas dari resiko.
11-20	*Software* memiliki resiko sedang.
21-50	*Software* memiliki resiko tinggi.
> 51	*Software* sangat beresiko.

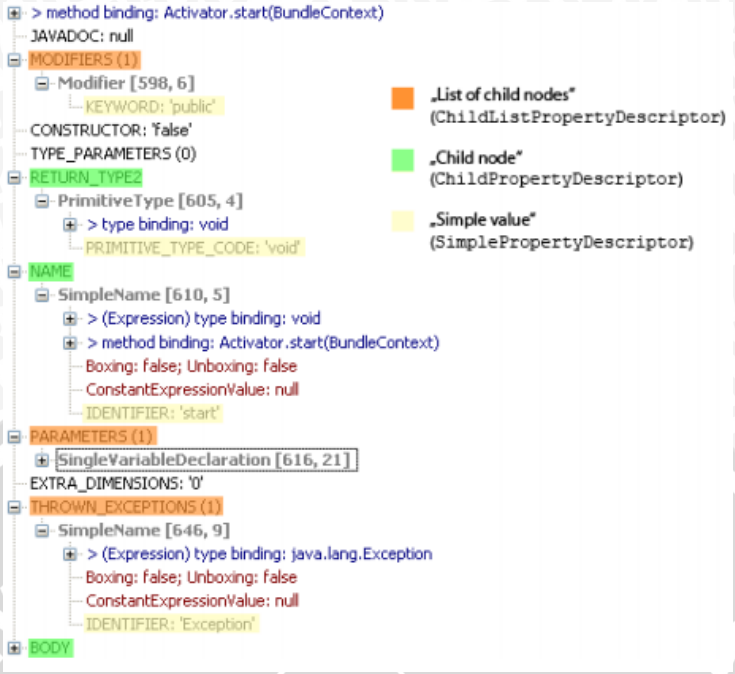
Sumber**:** Diadaptasi dari Ankita (2014)

2.1.3 *Java Parser and AST (Abstract Syntax Tree)*

Java parser adalah sebuah *tool* yang dikembangkan untuk mengindeks program java berdasarkan konsep ontology Java. *Java Parser* dikembangkan menggunakan *framework* dari *Eclipse Abstract Syntax Tree*. *Framework* ini menghasilkan sebuah *Abstract Syntax Tree* (AST) yang benar-benar dapat mewakili *source program* (Hosseini, 2013).

Abstract Syntax Tree (AST) terdiri dari beberapa node, setipa set berisi informasi yang dikenal sebagai *structural properties* (Hosseini, 2013). Gambar 2.3 menunjukan *structural properties* dari *method declaration* berikut ini.

```
public void start(BundleContext context) throws Exception {
      super.start(context);
}
```



Gambar 2.3 *Structural Properties Method Declaration*

Sumber: (Hosseini, 2013)

Tabel 2.2 Hasil *Output Java Parser*

Source	Output
`public void start(BundleContext context) throws Exception { super.start(context); }`	PublicAccessSpecifier, MethodDefinition, VoidDataType, FormalMethodParameter, ThrowsSpecification, ExceptionClass, SuperReference, SuperclassMethodCall, ExpressionStatement

Sumber: Diadaptasi dari Hosseini (2013)

Tabel 2.2 menunjukan konsep *output* Java Parser untuk fragmen kode pada *method declaration*. Setelah proses membangun pohon (*tree*) menggunakan *Eclipse* AST API selesai, *parser* melakukan analisis sematik menggunakan informasi dalam setiap *node*. Informasi tersebut digunakan untuk mengidentifikasi indeks detail dari *source program* (Hosseini, 2013).

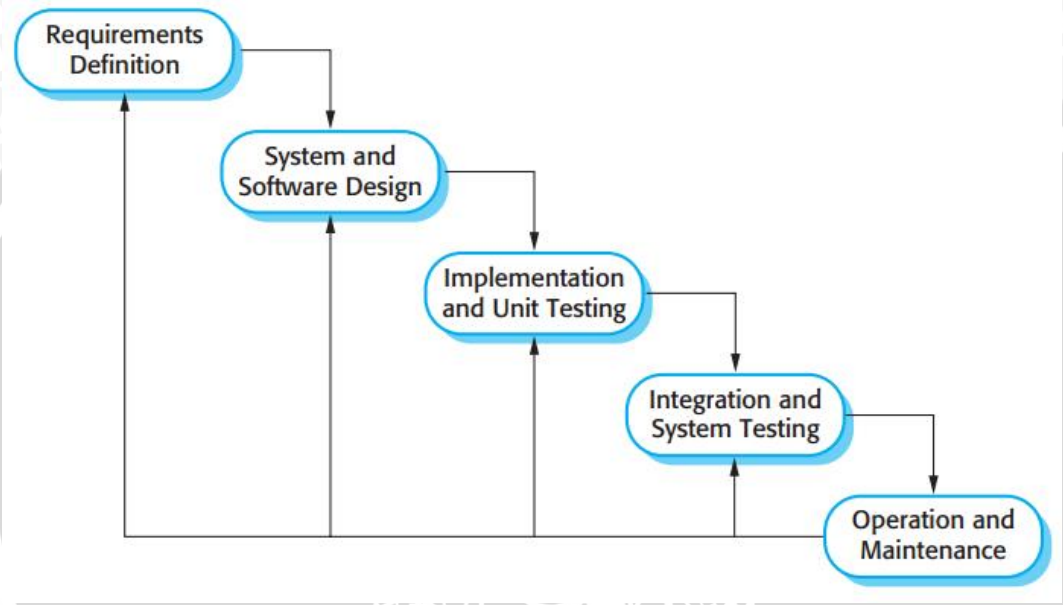
2.2 *Software Process Model*

Model proses perangkat lunak adalah representasi yang disederhanakan dari proses perangkat lunak. Setiap model proses merupakan proses dari perspektif tertentu dan hanya menyediakan informasi parsial mengenai proses tersebut. Sejumlah model proses yang sangat umum kadang disebut sebagai paradigma proses dan penyajiannya dari perspektif arsitektur. Artinya, melihat pada kerangka proses tetapi tidak melihat rincian aktivitas yang spesifik. Secara umum ada 3 model proses perangkat lunak yaitu *waterfall model*, *incremental development*, dan *reuse-oriented software engineering* (Sommerville, 2011).

Adapun model proses perangkat lunak yang digunakan dalam penelitian ini adalah *waterfall model*. Kelebihan dari *waterfall model* meliputi, kemudahan untuk dimengerti, mudah digunakan, *requirement* dari sistem bersifat stabil, baik dalam manajemen kontrol, serta bekerja dengan baik ketika kualitas lebih diutamakan dibandingkan dengan biaya dan jadwal (*deadline*) (Fahrurrozi, 2010).

2.2.1 *Waterfall Model*

Waterfall model mengambil proses dasar spesifikasi, pengembangan, validasi dan evolusi. Tahapan proses *waterfall model* terdiri dari spesifikasi kebutuhan, perancangan perangkat lunak, implementasi, pengujian dan sebagainya (Sommerville, 2011).



Gambar 2.4 *Waterfall Model*

Sumber: Sommerville (2011)

Gambar 2.3 menunjukan ilustrasi dari *waterfall model* atau dikenal dengan *software life cycle*. *Waterfall model* adalah contoh dari proses yang bergerak berdasarkan rancangan tahapannya, kita harus merancang dan menjadwalkan semua proses sebelum mulai mengerjakan. Tahapan utama dari aktivitas pembangunan yang mendasar dari *waterfall model* antara lain (Sommerville, 2011):

1. Analisis Kebutuhan dan Definisi

   Layanan sistem, kendala dan tujuan ditetapkan dengan berkonsultasi dengan pengguna sistem. Kemudian, didefinisikan secara rinci sebagai spesifikasi sistem.

2. Sistem dan Desain Perangkat Lunak

   Proses desain sistem mengalokasikan kebutuhan baik perangkat keras atau perangkat lunak sistem dengan membentuk arsitektur sistem secara keseluruhan.

Desain perangkat lunak melibatkan identifikasi dan menggambarkan abstraksi sistem perangkat lunak yang mendasar dan relasinya.

3. Implementasi dan Pengujian Unit

Dalam tahap ini, desain perangkat lunak direalisasikan sebagai serangkaian program atau unit program. Pengujian unit melibatkan verifikasi setiap unit untuk memenuhi spesifikasi.

4. Integrasi dan Pengujian Sistem

Unit program individu atau program diintegrasikan dan diuji sebagai sistem yang lengkap untuk memastikan bahwa kebutuhan perangkat lunak telah dipenuhi. Setelah pengujian, sistem perangkat lunak diserahkan kepada pengguna.

5. Operasi dan Pemeliharaan

Umumnya, tahapan ini adalah fase terpanjang dalam *software life cycle*. Pemeliharaan melihatkan kesalahan yang tidak ditemukan di awal tahap *software life cycle*, sehingga meningkatkan unit sistem dan layanan sistem sebagai kebutuhan baru.

2.3 *Unified Modelling Language*

UML (*Unified Modelling Language*) adalah metode pemodelan secara visual sebagai sarana untuk menganalisis dan merancang perangkat lunak berorientasi objek. UML sendiri juga memberikan standar penulisan sebuah sistem *blue print*, yang meliputi konsep bisnis proses, penulisan *class-class* dalam bahasa program yang spesifik, skema *database* dan komponen-komponen yang diperlukan dalam sistem perangkat lunak. Diagram UML dibagi menjadi 3 kategori (Sommerville, 2010):

1. *Structural diagram*: *diagram* yang menggambarkan elemen dari spesifikasi yang mengabaikan waktu. Terdiri dari *Class Diagram*, *Object Diagram*, *Component Diagram*, *Deployment Diagram*, *Composite Structure Diagram* dan *Package Diagram*.
2. *Behavior diagram*: *diagram* yang menggambarkan ciri-ciri *behavior/method/function* dari sebuah sistem atau *business process*. Terdiri dari *Use Case Diagram*, *Activity Diagram* dan *State Machine Diagram*.
3. *Interaction diagram*: merupakan bagian dari *behavior diagram* yang menggambarkan *object interactions*. Terdiri dari *Communication Diagram*, *Interaction Overview Diagram*, *Sequence Diagram* dan *Timing Diagram*.

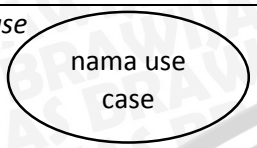
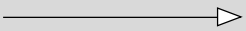
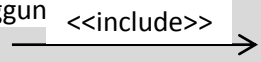
Adapun beberapa model UML yang digunakan dalam analisis dan perancangan sistem dalam penelitian ini yaitu *use case diagram*, *class diagram* dan *sequence diagram*.

2.3.1 *Use Case Diagram*

Use case diagram merupakan pemodelan untuk menggambarkan tingkah laku (*behavior*) sistem informasi yang akan dibuat. *Use case* mendeskripsikan sebuah interaksi antara satu atau lebih aktor dengan sistem informasi yang akan dibuat. Syarat penamaan pada *use case* adalah nama didefinisikan sesimpel

mungkin dan dapat dipahami. Ada dua hal utama yang harus dipahami pada *use case* yaitu pendefinisian apa yang disebut aktor dan *use case*. Tabel 2.2 berikut adalah simbol-simbol yang ada pada diagram *use case* (Rosa, 2014).

Tabel 2.3 Simbol *Use Case Diagram*

Simbol	Deskripsi
Use case 	Fungsionalitas yang disediakan sistem sebagai unit-unit yang saling bertukar pesan antar unit atau aktor, biasanya dinyatakan dengan menggunakan kata kerja diawal frase nama *use case*.
Aktor / *actor*  nama aktor	Orang, proses, atau sistem lain yang berinteraksi dengan sistem informasi. Jadi, walaupun simbol dari aktor adalah gambar orang, tetapi aktor belum tentu merupakan orang, biasanya dinyatakan menggunakan kata benda di awal frase nama aktor.
Asosiasi / *association*	Komunikasi antara aktor dan *use case* yang berpartisipasi pada *use case* atau *use case* memiliki interaksi dengan aktor.
Ekstensi / *extend* 	Relasi *use case* tambahan ke sebuah *use case* dimana *use case* yang ditambahkan dapat berdiri sendiri walau tanpa *use case* tambahan itu. Mirip dengan prinsip *inheritance* pada pemrograman berorientasi objek, biasanya *use case* tambahan memiliki nama depan yang sama dengan *use case* yang ditambahkan.
Generalisasi / *generalization* 	Hubungan generalisasi dan spesialisasi (umum-khusus) antara dua buah *use case* dimana fungsi yang satu adalah fungsi yang lebih umum dari lainnya.
Menggun 	Relasi *use case* tambahan ke sebuah *use case* dimana *use case* yang ditambahkan memerlukan *use case* ini untuk menjalankan fungsinya atau sebagai syarat dijalankan *use case* ini.

Sumber: Diadaptasi dari Rosa (2014)

2.3.2 *Class Diagram*

Class diagram menggambarkan struktur sistem dari segi pendefisian *class-class* yang akan dibuat untuk membangun sistem. *Class* memiliki apa yang disebut atribut dan metode atau operasi.

1. Atribut merupakan variabel-variabel yang dimiliki oleh suatu *class*.
2. Operasi atau metode adalah fungsi-fungsi yang dimiliki oleh suatu *class*.

Class-class yang ada pada struktur sistem harus dapat melakukan fungsi-fungsi sesuai dengan kebutuhan sistem. Susunan struktur *class* yang baik pada *class diagram* sebaiknya memiliki jenis-jenis *class* berikut (Rosa, 2014):

1. *Class main*
   Class yang memiliki fungsi awal dieksekusi ketika sistem dijalankan.

2. *Class* yang menangani tampilan sistem (*view*)
   Class yang mendefinisikan dan mengatur tampilan ke pengguna.
3. *Class* yang diambil dari pendefinisian *use case* (*controller*)
   Class yang menangani fungsi-fungsi yang harus ada diambil dari pendefinisian *use case*, *class* ini biasanya disebut dengan c*lass* proses yang menangani proses bisnis pada perangkat lunak.
4. *Class* yang diambil dari pendefinisian data (*model*)
   Class yang digunakan untuk memegang atau membungkus data menjadi sebuah kesatuan yang diambil maupun akan disimpan ke basis data.

Tabel 2.4 berikut adalah simbol-simbol yang ada pada *class diagram*.

Tabel 2.4 Simbol *Class Diagram*

Simbol	Deskripsi
Class nama_kelas +atribut +operasi()	*Class* pada struktur sistem.
Antarmuka / *interface* nama_interface	Sama dengan konsep *interface* dalam pemrograman berorientasi objek.
Asosiasi / *association*	Relasi antar *class* dengan makna umum, asosiasi biasanya juga disertai dengan *multiplicity*.
Asosiasi berarah / *directed association*	Relasi antar *class* dengan makna *class* yang satu digunakan oleh *class* yang lain, asosiasi biasanya juga disertai dengan *multiplicity*.
Generalisasi	Relasi antar *class* dengan makna generalisasi-spesialisasi (umum-khusus).
Kebergantungan / *dependency*	Relasi antar *class* dengan makna kebergantungan antar *class*.

Tabel 2.4 Simbol *Class Diagram* (lanjutan)

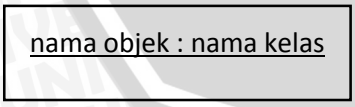
Agregasi / *aggregation* 	Relasi antarkelas dengan makna semua-bagian (*whole-part*).

Sumber: Diadaptasi dari Rosa (2014)

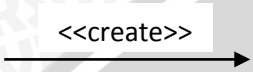
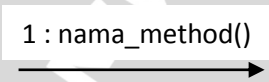
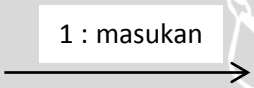
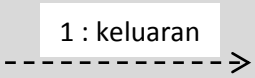
2.3.4 *Sequence Diagram*

Sequence diagram menggambarkan tingkah laku objek pada *use case* dengan mendeskripsikan waktu hidup objek dan *message* yang dikirimkan dan diterima antar objek. Oleh karena itu, untuk menggambarkan *sequence diagram* maka harus diketahui objek-objek yang terlibat dalam sebuah *use case* berserta metode-metode yang dimiliki kelas yang diinstansiasi menjadi objek itu. Membuat *sequence diagram* juga dibutuhkan untuk melihat skenario *use case*. Banyaknya *sequence diagram* yang harus digambar adalah minimal sebanyak pendefinisian *use case* (Rosa, 2014). Tabel 2.5 berikut adalah simbol-simbol yang ada pada *sequence diagram*.

Tabel 2.5 Simbol *Sequence Diagram*

Simbol	Deskripsi
Aktor  nama aktor	Orang, proses, atau sistem lain yang berinteraksi dengan sistem informasi. Biasanya dinyatakan menggunakan kata benda diawal frase nama aktor.
Garis hidup / *lifeline*	Menyatakan kehidupan suatu objek.
Objek 	Menyatakan objek yang berinteraksi pesan.

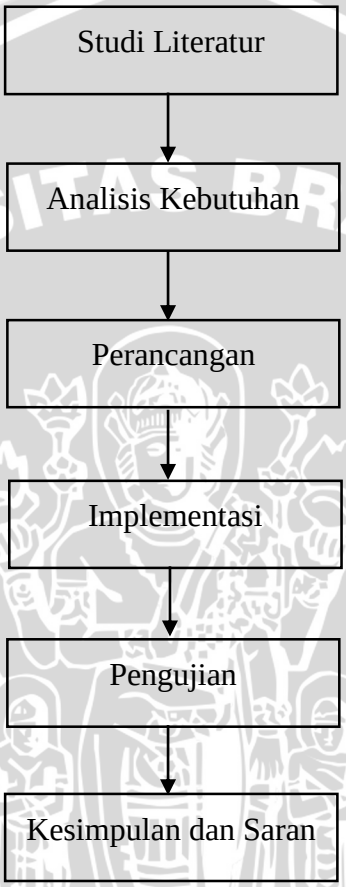
Tabel 2.5 Simbol *Sequence Diagram* (lanjutan)

Waktu aktif 	Menyatakan objek dalam keadaan aktif dan berinteraksi.
Pesan tipe *create* 	Menyatakan suatu objek membuat objek lain, arah panah mengarah pada objek yang dibuat.
Pesan tipe *call* 	Menyatakan suatu objek memanggil operasi/metode yang ada pada objek lain atau dirinya sendiri. Arah panah mengarah pada objek yang memiliki operasia atau metode, karena ini memanggil operasi atau metode maka operasi/metode yang dipanggil harus ada pada *class diagram* sesuai dengan *class* objek yang berinteraksi.
Pesan tipe *send* 	Menyatakan bahwa suatu objek mengirimkan data/masukan/informasi ke objek lainya. Arah panah mengarah pada objek yang dikirim.
Pesan tipe *return* 	Menyatakan bahwa suatu objek yang telah menjalankan suatu operasi atau metode menghasilkan suatu kembalian ke objek tertentu. Arah panah mengarah pada objek yang menerima kembalian.
Pesan tipe *destroy* 	Menyatakan suatu objek mengakhiri hidup objek yang lain. Arah panah mengarah pada objek yang diakhiri, sebaiknya jika ada *create* maka ada *destroy*.

Sumber: Diadaptasi dari Rosa (2014)

BAB 3 METODOLOGI

Tipe penelitian ini adalah implementatif – pengembangan (*development*). Langkah-langkah atau metode yang dilakukan dalam penelitian ini yaitu, studi literatur, analisis kebutuhan, perancangan, implementasi, pengujian, kesimpulan dan saran. Adapun diagram alir metodologi penelitian seperti pada Gambar 3.1.



Gambar 3.1 Diagram Alir Metodologi Penelitian

3.1 Studi Literatur

Studi literatur dilakukan untuk mendapatkan teori pendukung dalam penelitian dan ilmu-ilmu dasar yang akan digunakan untuk perancangan serta pengembangan perangkat lunak sistem perhitungan kompleksitas kode sumber. Penelusuran literatur dasar pengetahuan mengenai hal-hal yang berkaitan dengan penelitian ini diperoleh dari buku, paper, *ebook*, jurnal-jurnal ilmiah dan media literatur lainnya. Beberapa referensi yang terkait dalam penelitian ini, diantaranya adalah:

1. Pengukuran Kompleksitas Kode Sumber
   a. *Halstead Metrics*
   b. *Cyclomatic Complexity*
   c. Java Parser and AST

2. *Software Process Model*
   a. *Waterfall Model*
3. UML (*Unified Modelling Language*)
   a. *Use Case Diagram*
   b. *Class Diagram*
   c. *Sequence Diagram*

3.2 Analisis Kebutuhan

Analisis kebutuhan perangkat lunak (*software requirement*) merupakan proses mempelajari kebutuhan pengguna untuk mendapatkan definisi kebutuhan sistem. Tujuan dari tahap analisis kebutuhan adalah untuk memahami dengan sesungguhnya kebutuhan dari sistem perangkat lunak. Metode yang digunakan dalam analisis kebutuhan pada sistem perhitungan kompleksitas kode sumber ini adalah analisis berorientasi objek (*object oriented analysis*) dengan menggunakan pemodelan UML (*Unified Modelling Language*). Sasaran analisis berorientasi objek adalah untuk mendefinisikan sekelompok kebutuhan yang harus dipenuhi oleh perangkat lunak.

Adapun analisis kebutuhan dalam penelitian ini terdiri dari 5 tahapan yaitu, identifikasi aktor, analisis kebutuhan fungsional, pemodelan *use case diagram*, skenario *use case* dan analisis kebutuhan non fungsional. Identifikasi aktor adalah tahap untuk mengidentifikasi aktor-aktor yang berinteraksi dengan sistem, selanjutnya dilakukan analisis kebutuhan fungsional untuk menggambarkan fungsionalitas sistem atau layanan-layanan sistem dan dimodelkan dengan *use case diagram* untuk menggambarkan aktor dan interaksi mereka dalam suatu sistem. Kemudian dibuat skenario *use case* untuk menjelaskan lebih detail dari masing-masing *use case* yang terdapat pada *use case diagram*. Di tahap akhir, didefinisikan analisis kebutuhan non fungsional untuk menentukan batasan eksternal yang harus dipenuhi oleh perangkat lunak.

3.3 Perancangan

Perancangan perangkat lunak merupakan tahap yang mengambarkan bagaimana suatu sistem dibentuk. Perancangan menghubungkan spesifikasi kebutuhan dan implementasi dengan menekankan mengenai cara sistem memenuhi kebutuhan. Tujuan dari tahap perancangan adalah memenuhi kebutuhan dan memberikan gambaran yang jelas dalam rancang bangun sistem perangkat lunak. Metode perancangan yang digunakan pada sistem perhitungan kompleksitas kode sumber ini adalah perancangan berorientasi objek (*object oriented design*) dengan mengggunakan pemodelan UML (*Unified Modelling Language*).

Adapun perancangan perangkat lunak dalam penelitian ini terdiri dari 3 tahapan yaitu, perancangan arsitektur, perancangan komponen dan perancangan *user interface*. Perancangan arsitektur berkaitan dengan bagaimana sistem harus terorganisir dengan cara merancang struktur keseluruhan sistem, selanjutnya sub-sistem pada perancangan arsitektur didekomposisi menjadi lebih detail di

perancangan komponen. Dan di tahap akhir, dibuat perancangan *user interface* yang akan digunakan sebagai mekanisme komunikasi antara *user* dan sistem.

3.4 Implementasi

Implementasi perangkat lunak merupakan kegiatan memperoleh dan mengintegrasikan sumber daya fisik dan konseptual yang menghasilkan suatu sistem yang bekerja. Tujuan dari tahap implementasi adalah mewujudkan semua model hasil perancangan sistem berdasarkan pada kebutuhan-kebutuhan yang sudah dispesifikasikan sebelumnya. Implementasi perangkat lunak menggunakan pemrograman berorientasi objek yaitu bahasa pemrograman Java dengan *software* Netbeans IDE 8.0.2.

Adapun implementasi perangkat lunak sistem perhitungan kompleksitas kode sumber berdasarkan metrik *Halstead* dan *Cyclomatic Complexity* ini antara lain: spesifikasi lingkungan sistem, implementasi *class* dan implementasi *user interface*. Spesifikasi lingkungan sistem menjelaskan perangkat keras dan perangkat lunak yang digunakan dalam pengembangan sistem, sedangkan implementasi *class* adalah hasil dari pemodelan *class diagram* yang telah dibuat sebelumnya pada perancangan arsitektur begitu juga implementasi *user interface* yang berdasarkan pada perancangan *user interface*.

3.5 Pengujian

Pengujian perangkat lunak merupakan proses menjalankan dan mengevaluasi sebuah sistem secara manual maupun otomatis. Tujuan dari tahap pengujian adalah untuk menguji apakah sistem perangkat lunak sudah memenuhi kebutuhan atau belum dengan membandingkan antara hasil yang diharapkan dengan hasil yang sebenarnya.

Adapun metode pengujian perangkat lunak yang digunakan adalah *white box testing* dan *black box testing*. Pendekatan strategi pengujian yang dilakukan meliputi: pengujian unit, pengujian integrasi dan pengujian validasi. Pengujian unit atau komponen difokuskan untuk verifikasi pada unit terkecil dari perancangan perangkat lunak yang beroritentasi pada *white box testing*. Sedangkan, pengujian integrasi difokuskan untuk menguji gabungan dari unit-unit sistem yang membentuk kesatuan fungsional. Pengujian integrasi dapat berorientasi pada *white box testing* maupun *black box testing*. Selanjutnya, dilakukan pengujian validasi untuk memastikan bahwa perangkat lunak yang dibuat telah memenuhi kebutuhan dengan serangkaian uji *black box*.

3.6 Kesimpulan dan Saran

Proses pengambilan kesimpulan dan saran dilakukan setelah tahapan implementasi dan pengujian selesai dikerjakan. Kesimpulan diambil dari hasil pengujian dan analisis terhadap sistem. Sedangkan, saran disajikan dari evaluasi sistem dan keterbatasan-keterbatasan yang dimiliki dalam penelitian untuk memberikan masukan dalam pengembangan sistem dan penelitian dimasa yang akan datang.

BAB 4 REKAYASA KEBUTUHAN

4.1 Gambaran Umum Sistem

Gambaran umum Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity* dibagi menjadi dua bagian yaitu, deskripsi umum sistem dan lingkungan sistem.

1. Deskripsi Umum Sistem

Perangkat lunak yang dibuat dalam penelitian ini adalah "Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity*". Pengukuran kompleksitas kode sumber *java* menggunakan 2 metode metrik yaitu *Halstead's Volume* dan *Cyclomatic Complexity*. Data *input* yang diproses adalah *file* kode sumber *java*. Data *input* tersebut kemudian dihitung kompleksitasnya setelah melalui proses *parsing code*. *File java* yang dijadikan *input* proses dapat lebih dari satu *file* atau beberapa *file java class* sekaligus yang didalamnya memuat lebih dari satu *method* dalam setiap *class*. *Output* dari sistem ini berupa *log* hasil perhitungan kompleksitas masing-masing dari kedua metode yang disimpan dalam bentuk *file text* (.txt). Hasil dari perhitungan selanjutnya dapat digunakan sebagai data untuk mendeteksi kemungkinan cacat (*defect*) pada perangkat lunak yang dianalisis.

2. Lingkungan Sistem

Sistem perhitungan kompleksitas kode sumber ini membutuhkan lingkungan yang digunakan sistem untuk berjalan. *Tools* perhitungan kompleksitas kode sumber ini berbasis *java* sehingga membutuhkan JRE (*Java Runtime Environment*) untuk menjalankannya. *Tools* ini dapat dijalankan di sistem operasi manapun (*java write once run anywhere*) jika mesin sudah ter-*install* JRE minimal versi 1.7.0.

4.2 Analisis Kebutuhan

Analisis kebutuhan perangkat lunak pada penelitian ini terdiri dari identifikasi aktor, analisis kebutuhan fungsional, pemodelan *use case diagram*, skenario *use case* dan analisis kebutuhan non fungsional.

4.2.1 Identifikasi Aktor

Tabel 4.1 berikut menjelaskan aktor yang terlibat dalam interaksi sistem serta deskripsinya.

Tabel 4.1 Identifikasi Aktor

Aktor	Deskripsi
User	*User* adalah orang yang dapat menggunakan semua fungsi sistem tanpa terkecuali.

4.2.2 Analisis Kebutuhan Fungsional

Tabel 4.2 berikut menunjukan spesifikasi kebutuhan fungsional sistem perhitungan kompleksitas kode sumber.

Tabel 4.2 Daftar Kebutuhan Fungsional

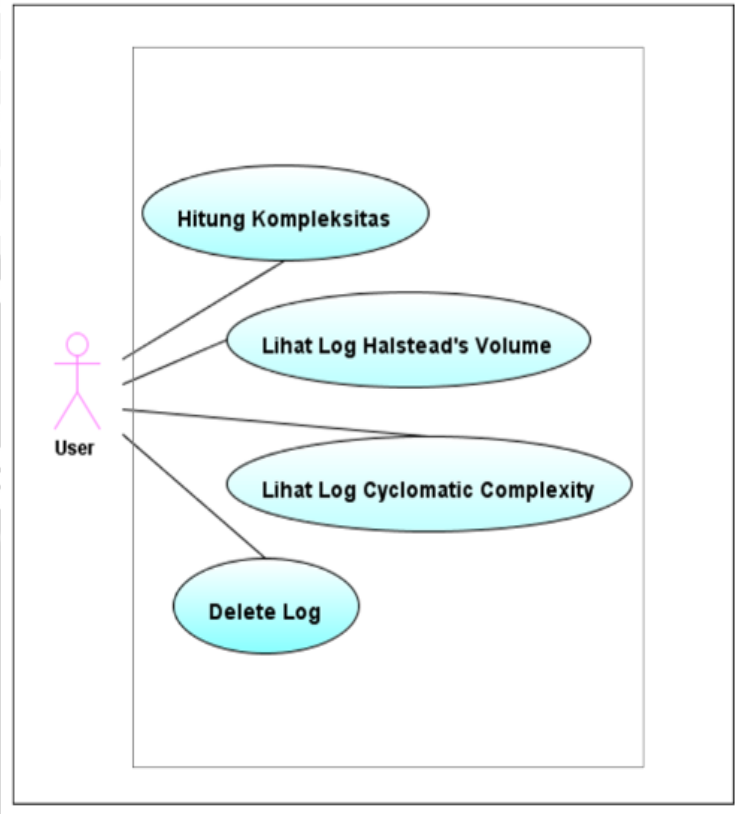
Nomor SRS_F	Kebutuhan	*Use Case*
SRS_F001	Sistem harus mampu melakukan perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume* dan *Cyclomatic Complexity*. Sistem harus mampu menyediakan fitur bagi *user* agar dapat memilih dan membuka *file* kode sumber sebagai *input data* untuk proses *parsing code* dan perhitungan kompleksitas kode sumber.	Hitung Kompleksitas
SRS_F002	Sistem harus mampu menampilkan *log* hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.	Lihat *Log Halstead's Volume*
SRS_F003	Sistem harus mampu menampilkan *log* hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.	Lihat *Log Cyclomatic Complexity*
SRS_F004	Sistem harus mampu menghapus *file log* dari hasil perhitungan kompleksitas metode *Halstead's Volume* dan *Cyclomatic Complexity*.	*Delete Log*

4.2.3 Pemodelan *Use Case Diagram*

Dalam sistem ini terdapat 4 fitur (*use case*) utama yang dapat dioperasikan oleh *user*. Berikut ini penjelasan dari *use case* tersebut dan pemodelan *use case diagram* yang ditunjukan pada Gambar 4.1.

1. Hitung Kompleksitas
   User dapat melakukan perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume* dan *Cyclomatic Complexity*. Diawali dengan memilih dan membuka *file* kode sumber satu atau beberapa *file* (*open multiple file*) kode sumber ke dalam sistem dengan ekstensi *file* .java sebagai *input data*. Selanjutnya, dilakukan proses *parsing code* terhadap *file* kode sumber tersebut untuk dihitung kompleksitasnya berdasarkan metode *Halstead's Volume* dan *Cyclomatic Complexity*.
2. Lihat Log *Halstead's Volume*
   User dapat melihat informasi hasil perhitungan *Halstead's Volume* yang meliputi nama *class*, nama *method* dan nilai *Halstead's Volume*.
3. Lihat Log *Cyclomatic Complexity*
   User dapat melihat informasi hasil perhitungan *Cyclomatic Complexity* yang meliputi nama *class*, nama *method,* nilai *Cyclomatic Complexity* dan *risk evaluation*.

4. *Delete Log*
   User dapat menghapus *file log* hasil perhitungan kompleksitas kode sumber metode *Halstead's Volume* dan *Cyclomatic Complexity*.



Gambar 4.1 *Use Case Diagram*

4.2.4 Skenario *Use Case*

Skenario *use case* menjelaskan lebih detail dari masing-masing *use case* yang terdapat pada Gambar 4.1 *Use Case Diagram*.

4.2.4.1 Skenario *Use Case* Hitung Kompleksitas

Skenario *use case* Hitung Kompleksitas pada Tabel 4.3 berikut ini menjelaskan rincian informasi kasus pada sistem dan reaksi sistem terhadap *use case* hitung kompleksitas.

Tabel 4.3 Skenario *Use Case* Hitung Kompleksitas

Skenario Kasus Pada Sistem	
Nomor *Use Case*	SRSF_001
Nama	Hitung Kompleksitas
Tujuan	Menghitung kompleksitas kode sumber berdasarkan metode *Halstead's Volume* dan *Cyclomatic Complexity*.

Tabel 4.3 Skenario *Use Case* Hitung Kompleksitas (lanjutan)

Deskripsi	*Use case* ini menjelaskan bahwa sistem melakukan proses perhitungan kompleksitas yang diawali dengan melakukan *open file* dan *parsing code* terhadap *file* kode sumber.
Aktor	*User*
Kondisi Awal	*User* telah menjalankan sistem dan halaman utama sistem ditampilkan.
Skenario Utama	
Aksi Aktor	**Reaksi Sistem**
User menekan tombol "Hitung Kompleksitas"	Sistem menampilkan *dialog browse file*.
User memilih *file* kode sumber lalu menekan tombol "Open"	Sistem menampilkan hasil *parsing code* dan perhitungan kompleksitas setiap *method*.
Alur Alternatif	
Alternatif 1: Jika *input file* yang dipilih oleh *user* tidak *valid* (*file* bukan bertipe *java*).	
Aksi Aktor	**Reaksi Sistem**
	Sistem menampilkan pesan kesalahan kepada *user* bahwa "*file* tidak *valid*, silahkan pilih *file* bertipe java".
User menekan tombol "OK"	Sistem menampilkan *dialog browse file*.
Kondisi Akhir	Sistem menampilkan hasil *parsing code* dan perhitungan kompleksitas setiap *method* dalam *class*.

4.2.4.2 Skenario *Use Case* Lihat *Log Halstead's Volume*

Skenario *use case* Lihat *Log Halstead's Volume* pada Tabel 4.4 berikut ini menjelaskan rincian informasi kasus pada sistem dan reaksi sistem terhadap *use case* Lihat *Log Halstead's Volume*.

Tabel 4.4 Skenario *Use Case* Lihat *Log Halstead's Volume*

Skenario Kasus Pada Sistem	
Nomor *Use Case*	SRS_F002
Nama	Lihat *Log Halstead's Volume*
Tujuan	Menampilkan hasil perhitungan kompleksitas metode *Halstead's Volume*.
Deskripsi	*Use case* ini menjelaskan bahwa sistem melakukan proses *read* pada isi *file* hasil perhitungan kompleksitas berdasarkan metode *Halstead's Volume*.
Aktor	*User*
Kondisi Awal	*User* telah menjalankan sistem dan halaman utama sistem ditampilkan.
Skenario Utama	
Aksi Aktor	**Reaksi Sistem**
User menekan tombol "Halstead"	Sistem menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.

Tabel 4.4 Skenario *Use Case* Lihat *Log Halstead's Volume* (lanjutan)

Alur Alternatif	
Alternatif 1: Jika *file log* hasil *Halstead's Volume* tidak ditemukan.	
Aksi Aktor	**Reaksi Sistem**
	Sistem menampilkan informasi bahwa "*file log Halstead's Volume* tidak ditemukan".
User menekan tombol "OK"	
Kondisi Akhir	Sistem menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.

4.2.4.3 Skenario *Use Case* Lihat *Log Cyclomatic Complexity*

Skenario *use case* Lihat *Log Cyclomatic Complexity* pada Tabel 4.5 berikut ini menjelaskan rincian informasi kasus pada sistem dan reaksi sistem terhadap *use case* Lihat *Log Cyclomatic Complexity*.

Tabel 4.5 Skenario *Use Case* Lihat *Log Cyclomatic Complexity*

Skenario Kasus Pada Sistem	
Nomor *Use Case*	SRS_F003
Nama	Lihat *Log Cyclomatic Complexity*
Tujuan	Menampilkan hasil perhitungan kompleksitas metode *Cyclomatic Complexity*.
Deskripsi	*Use case* ini menjelaskan bahwa sistem melakukan proses *read* pada isi *file* hasil perhitungan kompleksitas berdasarkan metode *Cyclomatic Complexity*.
Aktor	*User*
Kondisi Awal	*User* telah menjalankan sistem dan halaman utama sistem ditampilkan.
Skenario Utama	
Aksi Aktor	**Reaksi Sistem**
User menekan tombol "Cyclomatic"	Sistem menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.
Alur Alternatif	
Alternatif 1: Jika *file log* hasil *Cyclomatic Complexity* tidak ditemukan.	
Aksi Aktor	**Reaksi Sistem**
	Sistem menampilkan informasi bahwa "*file log Cyclomatic Complexity* tidak ditemukan".
User menekan tombol "OK"	
Kondisi Akhir	Sistem menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.

4.2.4.4 Skenario *Use Case Delete Log*

Skenario *use case Delete Log* pada Tabel 4.6 berikut ini menjelaskan rincian informasi kasus pada sistem dan reaksi sistem terhadap *use case Delete Log*.

Tabel 4.6 Skenario *Use Case Delete Log*

Skenario Kasus Pada Sistem	
Nomor *Use Case*	SRS_F004
Nama	*Delete Log*
Tujuan	Menghapus *file log* hasil perhitungan kompleksitas kode sumber metode *Halstead's Volume* dan *Cyclomatic Complexity*.
Deskripsi	*Use case* ini menjelaskan bahwa sistem melakukan proses *delete* pada *file* hasil perhitungan kompleksitas kode sumber dari kedua metode (*Halstead's Volume* dan *Cyclomatic Complexity*).
Aktor	*User*
Kondisi Awal	*User* telah menjalankan sistem dan halaman utama sistem ditampilkan.
Skenario Utama	
Aksi Aktor	**Reaksi Sistem**
User menekan tombol "Del Log"	Sistem menampilkan *dialog* konfirmasi *delete* dengan opsi *YES*/*NO*.
User memilih opsi *YES*	Sistem menghapus *file* hasil perhitungan kompleksitas metode *Halstead's Volume* dan *Cyclomatic Complexity*.
Alur Alternatif	
Alternatif 1: Jika *file log* hasil *Halstead's Volume* tidak ditemukan.	
Aksi Aktor	**Reaksi Sistem**
	Sistem menampilkan informasi bahwa "*file log* tidak ditemukan".
User menekan tombol "OK"	
Alternatif 2: Jika *file log* hasil *Halstead's Volume* ditemukan tetapi *file log* hasil *Cyclomatic Complexity* tidak ditemukan.	
Aksi Aktor	**Reaksi Sistem**
	Sistem menampilkan informasi bahwa "*file log* tidak ditemukan".
User menekan tombol "OK"	
Alternatif 3: Jika *user* memilih opsi *NO*.	
Aksi Aktor	**Reaksi Sistem**
	Sistem tidak menghapus *file log* hasil perhitungan kompleksitas kode sumber kedua metode.
Kondisi Akhir	Sistem menghapus *file log* hasil perhitungan kompleksitas metode *Halstead's Volume* dan *Cyclomatic Complexity*.

4.2.5 Analisis Kebutuhan Non Fungsional

Tabel 4.7 menunjukan spesifikasi daftar kebutuhan non fungsional sistem.

Tabel 4.7 Daftar Kebutuhan Non Fungsional

Nomor SRS_NF	Deskripsi Kebutuhan	Parameter
SRS_NF001	Sistem harus memiliki tingkat akurasi perhitungan kompleksitas minimal 50% pada setiap metode *Halstead's Volume* dan *Cyclomatic Complexity*.	Akurasi

BAB 5 PERANCANGAN DAN IMPLEMENTASI

5.1 Perancangan

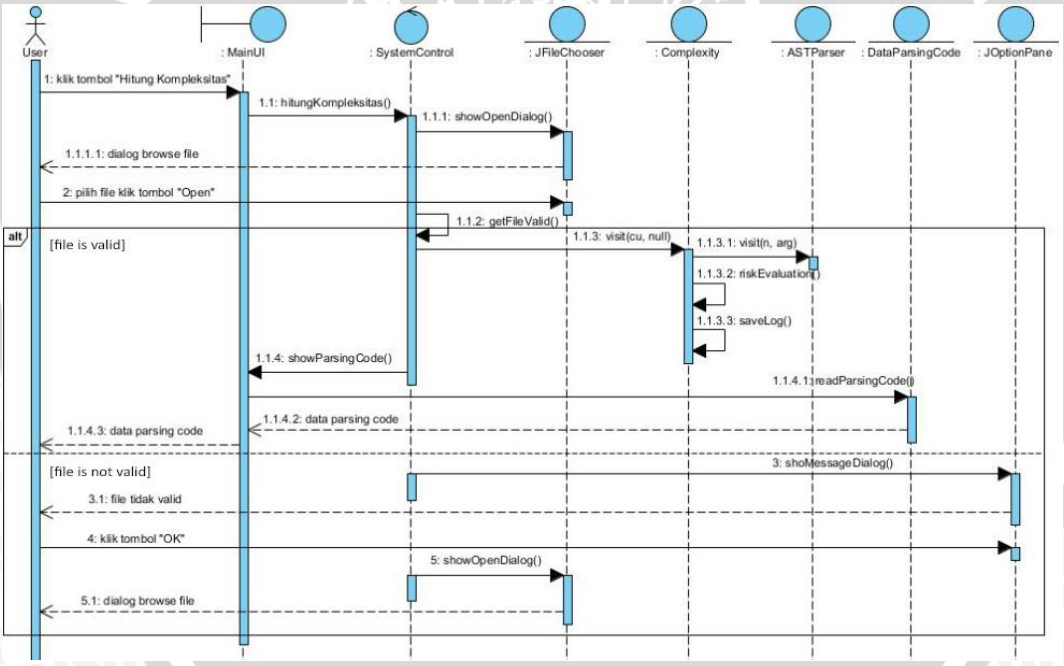
Tahapan dari perancangan perangkat lunak dalam penelitian ini terdiri dari perancangan arsitektur, perancangan komponen dan perancangan *user interface*.

5.1.1 Perancangan Arsitektur

Perancangan arsitektur adalah tahap pertama dalam proses perancangan perangkat lunak. *Output* dari perancangan arsitektur adalah *class diagram* yang berdasarkan dari identifikasi objek menggunakan *sequence diagram*.

5.1.1.1 Pemodelan *Sequence Diagram* Hitung Kompleksitas

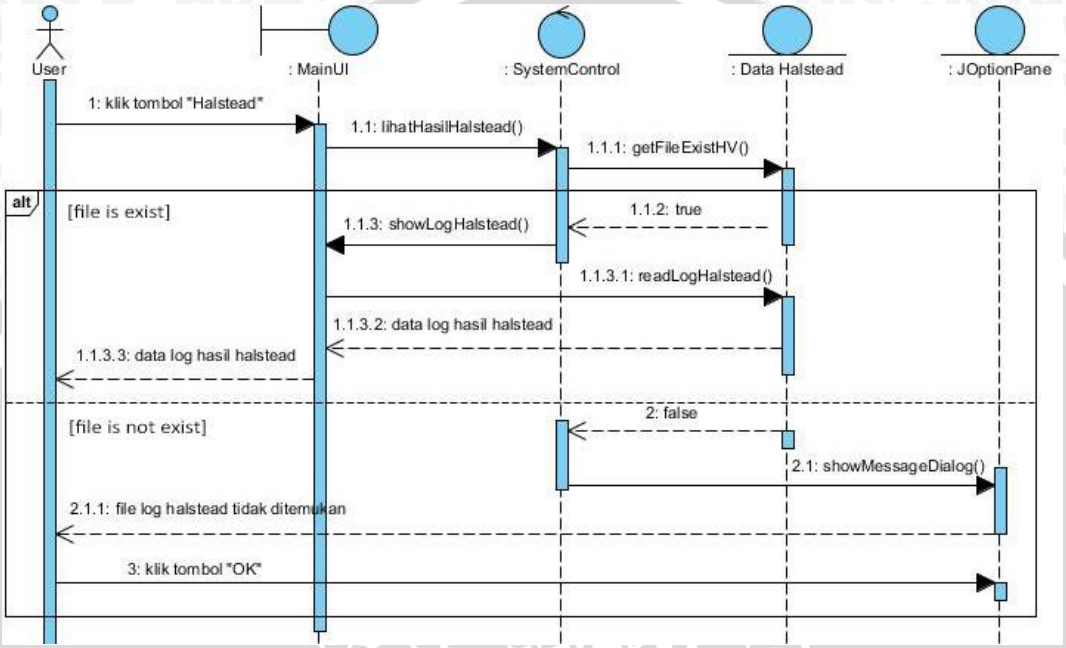
Sequence diagram Hitung Kompleksitas menjelaskan model interaksi *user* dan sistem dalam melakukan proses perhitungan kompleksitas kode sumber. Beberapa objek yang terkait dalam proses diantaranya adalah *User*, *Main UI*, *System Control*, *JFile Chooser*, *Complexity*, *AST* Parser, *Data Parsing Code* dan *JOption Pane* yang ditunjukan pada Gambar 5.1.



Gambar 5.1 *Sequence Diagram* Hitung Kompleksitas

5.1.1.2 Pemodelan *Sequence Diagram* Lihat *Log Halstead's Volume*

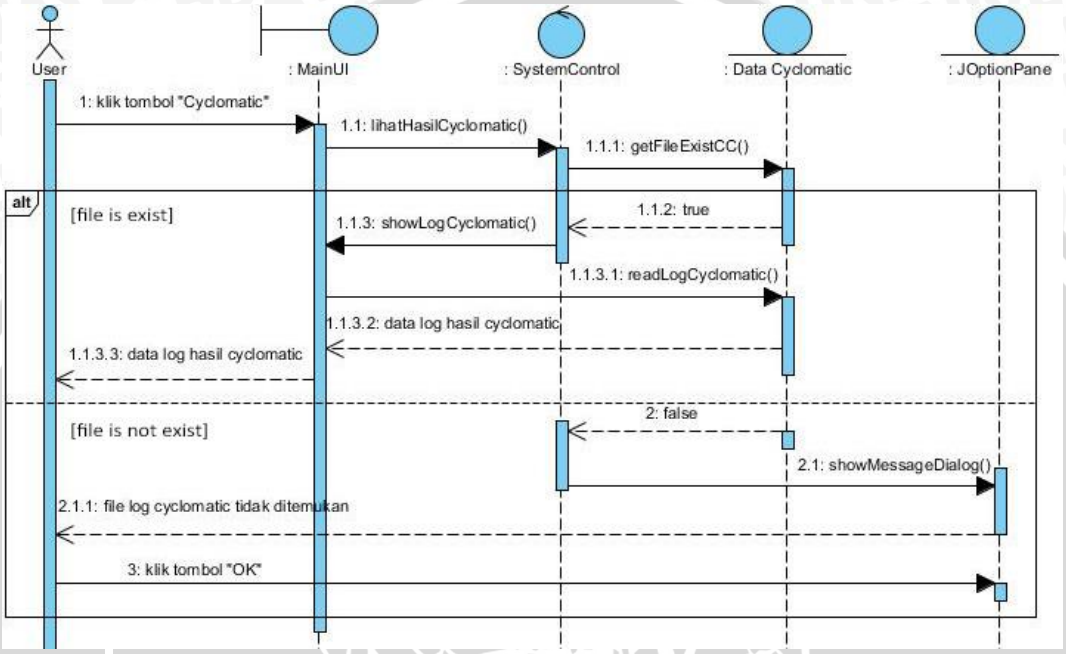
Sequence diagram Lihat *Log Halstead's Volume* menjelaskan model interaksi *user* dan sistem dalam melakukan proses *read* data dari hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*. Beberapa objek yang terkait dalam proses diantaranya adalah *User*, *Main UI*, *System Control*, *Data Halstead* dan *JOption Pane* yang ditunjukan pada Gambar 5.2.



Gambar 5.2 *Sequence Diagram* Lihat *Log Halstead's Volume*

5.1.1.3 Pemodelan *Sequence Diagram* Lihat *Log Cyclomatic Complexity*

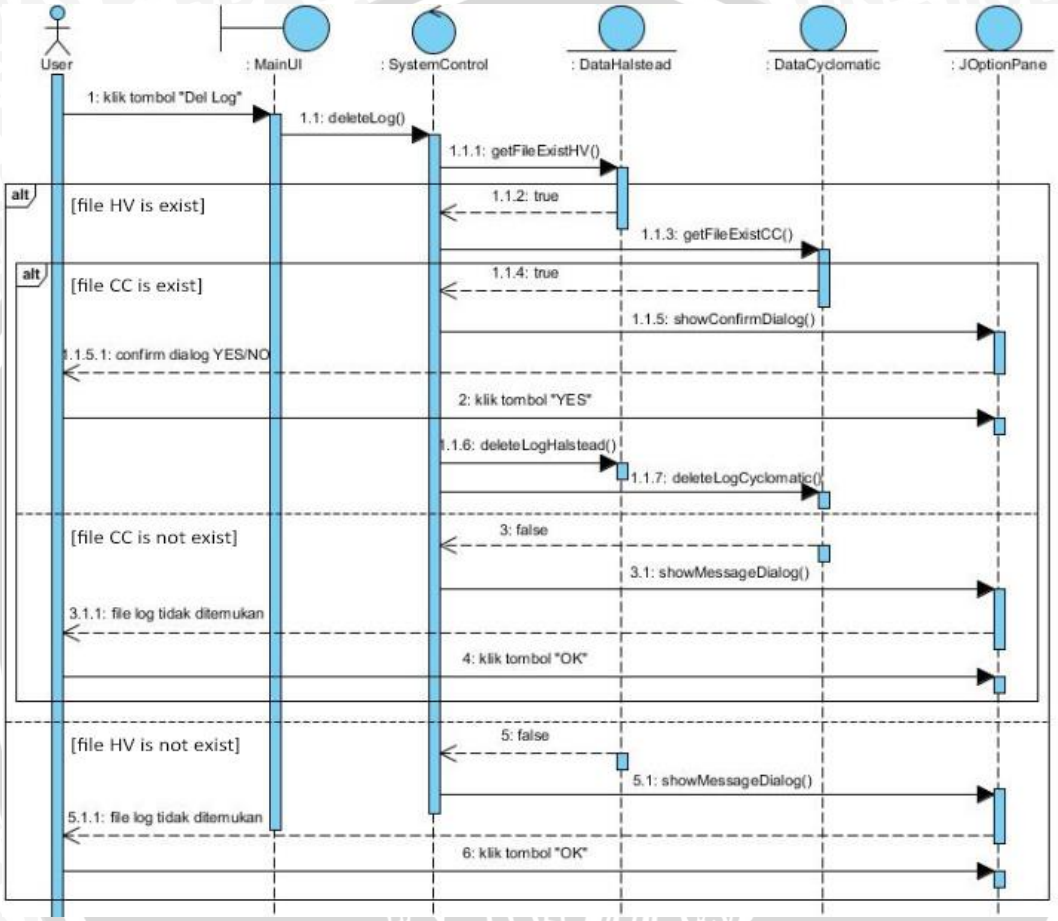
Sequence diagram Lihat *Log Cyclomatic Complexity* menjelaskan model interaksi *user* dan sistem dalam melakukan proses *read* data dari hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*. Beberapa objek yang terkait dalam proses diantaranya adalah *User*, *Main UI*, *System Control*, *Data Cyclomatic* dan *JOption Pane* yang ditunjukan pada Gambar 5.3.



Gambar 5.3 *Sequence Diagram* Lihat *Log Cyclomatic Complexity*

5.1.1.4 Pemodelan *Sequence Diagram Delete Log*

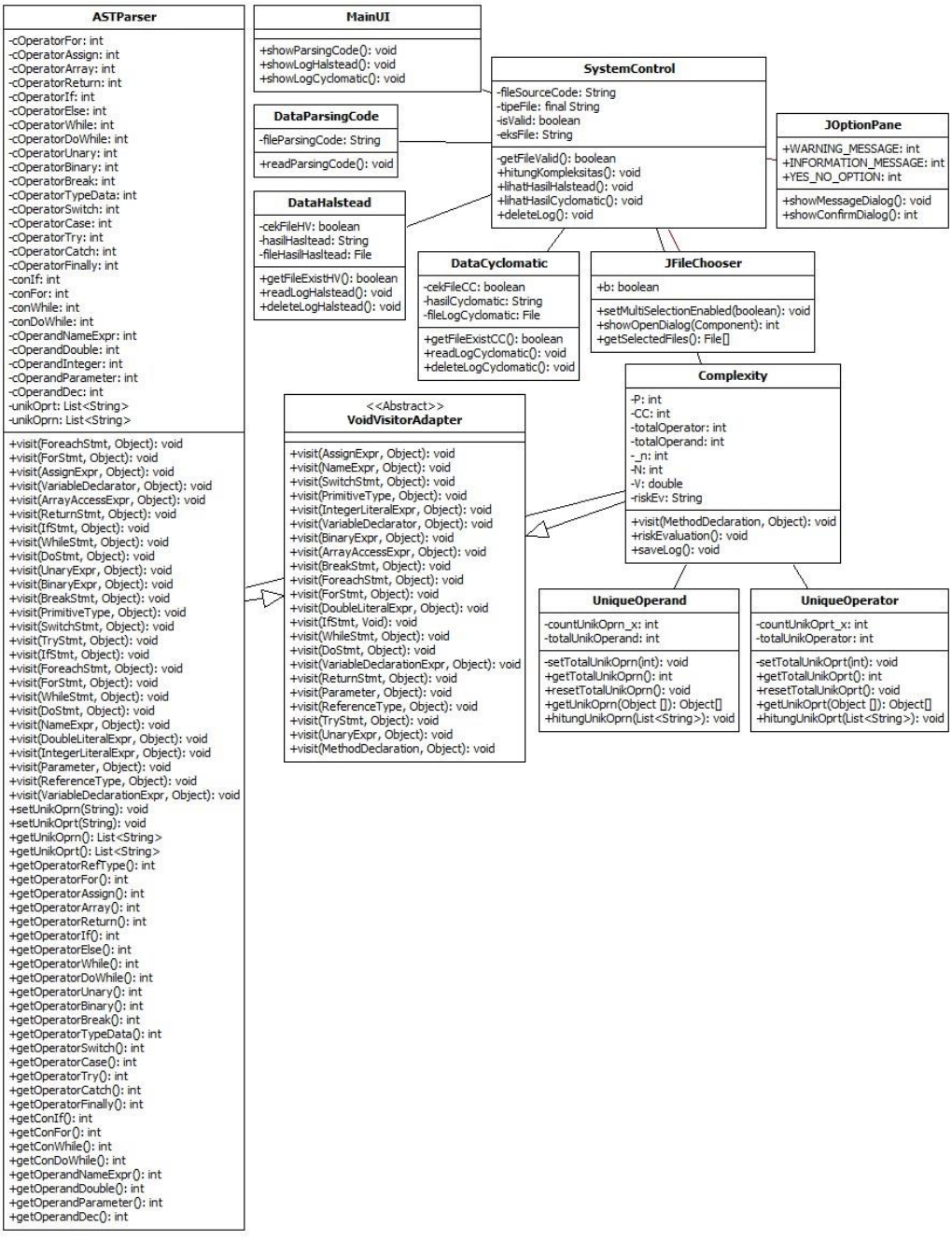
Sequence diagram Delete Log menjelaskan model interaksi *user* dan sistem dalam melakukan proses hapus data *file* hasil perhitungan kompleksitas kode sumber metode *Halstead's Volume* dan *Cyclomatic Complexity*. Beberapa objek yang terkait dalam proses diantaranya adalah *User*, *Main UI*, *System Control*, *Data Halstead*, *Data Cyclomatic* dan *JOption Pane* yang ditunjukan pada Gambar 5.4.



Gambar 5.4 *Sequence Diagram Delete Log*

5.1.1.5 Pemodelan *Class Diagram*

Class diagram yang digunakan dalam perancangan sistem ini ditunjukan pada Gambar 5.5.



Gambar 5.5 *Class Diagram*

Dalam pemodelan *class diagram*, sistem ini menggunakan 2 *class* dari *package Swing* java yaitu *class* JFileChooser dan *class* JOptionPane. Khusus untuk *class* VoidVisitorAdapter merupakan *class abstract* yang berasal dari *library Java Parser and Abstract Syntax Tree*. *Sub class* ASTParser mengimplementasikan

method abstract yang ada pada *super class* VoidVisitorAdapter untuk mendapatkan data *predicates node*, *operator* dan *operand* dari kode sumber.

5.1.2 Perancangan Komponen

Perancangan komponen menjelaskan *atribut* dan algoritma *method* dalam sebuah *class* yang telah dimodelkan sebelumnya pada pemodelan *class diagram*. Dalam perancangan komponen pada penelitian ini mengambil *sample* dari 3 *class* yaitu, *class System Control*, *class Complexity* dan *class Data Halstead*.

5.1.2.1 *Class System Control*

Terdapat 4 atribut dalam *class System Control*, antara lain seperti pada tabel 5.1.

Tabel 5.1 Atribut *Class System Control*

No	Nama Atribut	Tipe Data	*Modifier*
1	fileSourceCode	String	Private
2	eksFile	String	Private
3	tipeFile	String	Private
4	isValid	Boolean	Private

Algoritma *Method* getFileValid

```
FUNCTION getFileValid
IF eksFile.contains(tipeFile) THEN
  return isValid = true;
ELSE
  return isValid = false;
ENDIF
END getFileValid
```

Algoritma *Method* hitungKompleksitas

```
PROCEDURE hitungKompleksitas(MainUI ui, ASTParser astParser,
DataParsingCode dataPC, UniqueOperator uOprt, UniqueOperand
uOprn)
  JFileChooser pilihFile = JFileChooser("E:/Semester
  8/DataTesting");
  pilihFile.SetMultiSelectionEnable = true;
  pilihFile.showOpenDialog;
  File[] files = pilihFile.getSelectedFiles;
FOR File file: files
  String fileSourceCode = file.getAbsolutePath;
  FileInputStream in = FileInputStream(fileSourceCode);
  int i = fileSourceCode.lastIndexOf(".");
  eksFile = fileSourceCode.substring(i);
  CompilationUnit cu = null;
IF getValid THEN
  cu = JavaParser.parse(in);
  Complexity(astParser, uOprt, uOprn).visit(cu, null);
  ui.showParsingCode(dataPC);
ELSE
```

```
  JOptionPane.showMessageDialog ("File tidak valid, silahkan
  pilih file bertipe java.");
  hitungKompleksitas(ui, astParser, dataPC, uOprt, uOprn);
ENDIF
ENDFOR
END hitungKompleksitas
```

Algoritma *Method* lihatHasilHalstead

```
PROCEDURE lihatHasilHalstead(MainUI ui, DataHalstead dataHV)
IF dataHV.getFileExistHV == true THEN
  ui.showLogHalstead(dataHV);
ELSE
  JOptionPane.showMessageDialog ("File log Halstead's Volume
  tidak ditemukan.");
ENDIF
END lihatHasilHalstead
```

Algoritma *Method* lihatHasilCyclomatic

```
PROCEDURE lihatHasilCyclomatic(MainUI ui, DataCyclomatic dataCC)
IF dataCC.getFileExistCC == true THEN
  ui.showLogCyclomatic(dataCC);
ELSE
  JOptionPane.showMessageDialog ("File log Cyclomatic Complexity
  tidak ditemukan.");
ENDIF
END lihatHasilCyclomatic
```

Algoritma *Method* deleteLog

```
PROCEDURE deleteLog(MainUI ui, DataHalstead dataHV,
DataCyclomatic dataCC)
IF dataHV.getFileExistHV == true
AND dataCC.getFileExistCC == true THEN
  int opsiPilihan = JOptionPane.showConfirmDialog ("Anda yakin
  untuk menghapus log?");
IF opsiPilihan == JOptionPane.YES_OPTION THEN
  dataHV.deleteLogHalstead;
  dataCC.deleteLogCyclomatic;
  ui.logArea.setText("");
ENDIF
ELSE
  JOptionPane.showMessageDialog ("File log tidak ditemukan.");
ENDIF
END deleteLog
```

5.1.2.2 *Class Complexity*

Terdapat 8 atribut dalam *class Complexity*, antara lain seperti pada tabel 5.2.

Tabel 5.2 Atribut *Class Complexity*

No	Nama Atribut	Tipe Data	*Modifier*
1	P	Integer	Private
2	CC	Integer	Private

Tabel 5.2 Atribut *Class Complexity* (lanjutan)

3	totalOperator	Integer	Private
4	totalOperand	Integer	Private
5	N	Integer	Private
6	N	Integer	Private
7	V	Double	Private
8	riskEv	String	Private

Algoritma *Method* visit

```
PROCEDURE visit(MethodDeclaration n, Object arg)
IF n.getBody != null THEN
  astParser.reset;
  astParser.visit(n, arg);
  totalOperator=0;
  totalOperand=0;
  uOprt.hitungUnikOprt;
  uOprn.hitungUnikOprn;
  totalOperator      =      astParser.getOperatorBinary      +
astParser.getOperatorUnary   +   astParser.getOperatorReturn   +
astParser.getOperatorTypeData  +   astParser.getOperatorArray   +
astParser.getOperatorAssign    +    astParser.getOperatorFor    +
astParser.getOperatorWhile + astParser.getOperatorDoWhile +
astParser.getOperatorIf     +     astParser.getOperatorElse     +
astParser.getOperatorSwitch    +    astParser.getOperatorCase    +
astParser.getOperatorTry    +    astParser.getOperatorCatch    +
astParser.getOperatorFinally   +   astParser.getOperatorBreak   +
astParser.getOperatorMethodType + astParser.getOperatorRefType;
  totalOperand      =      astParser.getOperandNameExpr       +
astParser.getOperandDec     +     astParser.getOperandInteger    +
astParser.getOperandDouble   +   astParser.getOperandParameter   +
astParser.getOperandMethodName;
  N = totalOperator + totalOperand;
  _n = uOprt.getTotalUnikOprt + uOprn.getTotalUnikOprn;
  V = N * (log(_n) / log(2));
  uOprn.hitungUnikOprn;
  unik.getUnikOprn.clear;
  CC    =    astParser.getConIf    +    astParser.getConFor    +
astParser.getConWhile     +     astParser.getConDoWhile      +
astParser.getOperatorSwitch + 1;
riskEvaluation;
saveLog;
ENDIF
END visit
```

Algoritma *Method* riskEvaluation

```
PROCEDURE riskEvaluation
IF 1<= CC AND CC <= 10 THEN
  riskEv = "Bebas dari resiko.";
ELSE IF 11 <= CC AND CC <= 20 THEN
  riskEv = "Memiliki resiko sedang.";
ELSE IF 21 <= CC AND CC <= 50 THEN
  riskEv = "Memiliki resiko tinggi.";
ELSE
  riskEv = "Sangat beresiko.";
```

```
ENDIF
ENDIF
ENDIF
END riskEvaluation
```

5.1.2.3 *Class Data Halstead*

Terdapat 3 atribut dalam *class Data Halstead*, antara lain seperti pada tabel 5.3.

Tabel 5.3 Atribut *Class Data Halstead*

No	Nama Atribut	Tipe Data	*Modifier*
1	cekFileHV	Boolean	Private
2	hasilHasltead	String	Private
3	fileLogHalstead	File	Private

Algoritma *Method* getFileExistHV

```
FUNCTION getFileExistHV
IF fileLogHalstead.exists THEN
  return cekFileHV = true;
ELSE
  return cekFileHV = false;
ENDIF
END getFileExistHV
```

Algoritma *Method* readLogHalstead

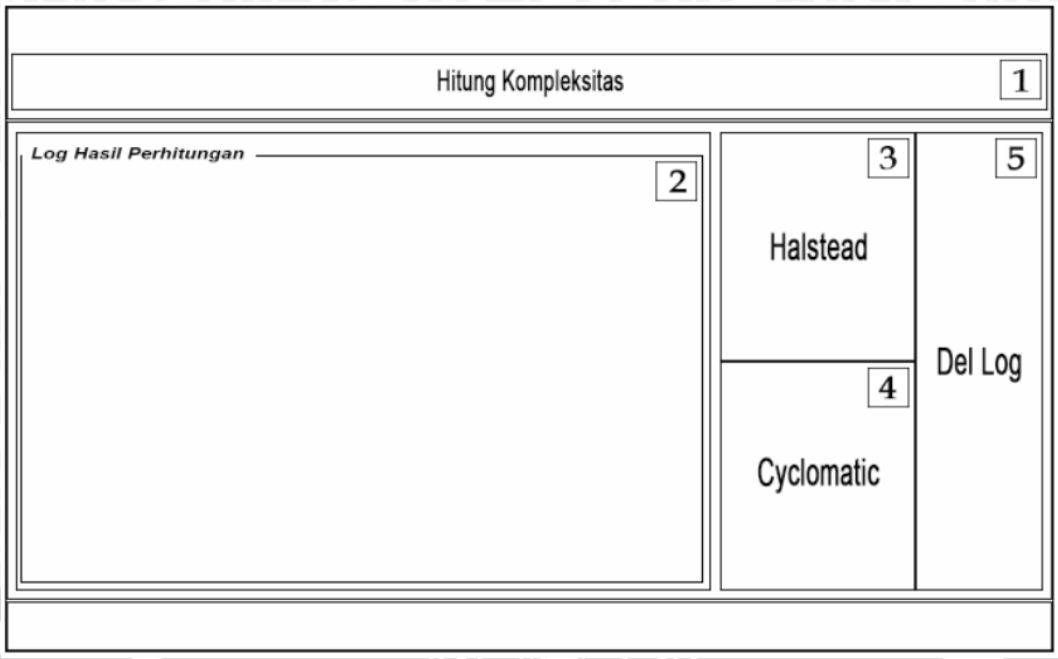
```
PROCEDURE readLogHalstead(MainUI ui)
  FileReader hasilHV = FileReader(hasilHalstead);
  BufferedReader br = BufferedReader(hasilHV);
  ui.logArea.read(br, null);
  ui.LogArea.requestFocus;
END readLogHalstead
```

Algoritma *Method* deleteLogHalstead

```
PROCEDURE deleteLogHalstead
  fileLogHalstead.delete;
END deleteLogHalstead
```

5.1.3 Perancangan *User Interface*

Gambar 4.7 merupakan desain *user interface* sistem perhitungan kompleksitas kode sumber berdasarkan metrik *Halstead* dan Cyclomatic *Complexity*.



Gambar 5.6 Desain *User Interface*

Keterangan Gambar 5.6:

1. Tombol "Hitung Kompleksitas", tombol untuk melakukan proses perhitungan kompleksitas kode sumber.
2. *Text Area*, tempat untuk menampilkan hasil *parsing code* dan *log* kompleksitas kode sumber metode *Halstead's Volume* maupun *Cyclomatic Complexity*.
3. Tombol "Halstead", tombol untuk menampilkan *log* hasil kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.
4. Tombol "Cyclomatic" tombol untuk menampilkan *log* hasil kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.
5. Tombol "Del Log", tombol untuk menghapus semua *file log* hasil perhitungan kompleksitas dari direktori penyimpanan *file*.

5.2 Implementasi

Tahapan implementasi perangkat lunak yang dilakukan terdiri dari penjelasan tentang spesifikasi lingkungan sistem, implementasi *class* dan implementasi *user interface*.

5.2.1 Spesifikasi Lingkungan Sistem

Spesifikasi lingkungan sistem terdiri dari spesifikasi perangkat keras dan spesifikasi perangkat lunak.

5.2.1.1 Spesifikasi Perangkat Keras

Tabel 5.5 merupakan spesifikasi perangkat keras yang digunakan dalam proses pengembangan perangkat lunak Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity*.

Tabel 5.5 Spesifikasi Perangkat Keras

Nama Komponen	Spesifikasi
System Model	Asus A450CC-WX147D
Processor	Intel® Core™ i7-3537U
Hard Disk Drive	750 GB
Memory RAM	4 GB DDR3
Display	14.0” 16:9 HD (1366x768)

5.2.1.2 Spesifikasi Perangkat Lunak

Tabel 5.6 merupakan spesifikasi perangkat lunak yang digunakan dalam proses pengembangan perangkat lunak Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity*.

Tabel 5.6 Spesifikasi Perangkat Lunak

Nama Komponen	Spesifikasi
Operating System	Windows 8.1 Enterprise 64-bit
Programming Language	Java
Text Editor / IDE	Netbeans IDE 8.0.2 dengan JDK 1.7

5.2.2 Implementasi *Class*

Implementasi *class* berdasarkan pada pemodelan *class diagram* yang telah dibuat sebelumnya. Adapun *file-file* implementasi dari *class* tersebut ditunjukan pada Tabel 5.7.

Tabel 5.7 Implementasi *Class*

No	*Package*	Nama *Class*	Nama *File*
1	mvc.controller	SystemControl	SystemControl.java
2	mvc.model	ASTParser	ASTParser.java
3	mvc.model	Complexity	Complexity.java
4	mvc.model	DataCyclomatic	DataCyclomatic.java
5	mvc.model	DataHalstead	DataHalstead.java
6	mvc.model	DataParsingCode	DataParsingCode.java
7	mvc.model	UniqueOperand	UniqueOperand.java
8	mvc.model	UniqueOperator	UniqueOperator.java
9	mvc.view	MainUI	MainUI.java

5.2.2.1 Implementasi *Method*

Implementasi *method* mengambil *sample* masing-masing 1 *method* dari 3 *class* yaitu, *method* hitungKompleksitas pada *class System Control*, *method* riskEvaluation pada *class Complexity* dan *method* getFileExistHV pada *class Data Halstead* yang ditunjukan pada Tabel 5.8, Tabel 5.9 dan Tabel 5.10.

Tabel 5.8 Implementasi *Method* hitungKompleksitas

```
public void hitungKompleksitas(MainUI ui, ASTParser astParser,
DataParsingCode dataPC, UniqueOperator uOprt, UniqueOperand
```

Tabel 5.8 Implementasi *Method* hitungKompleksitas (lanjutan)

```
throws FileNotFoundException, IOException, ParseException {
JFileChooser pilihFile = new JFileChooser("E:/Semester
8/DataTesting ");
        pilihFile.setMultiSelectionEnabled(true);
        pilihFile.showOpenDialog(null);
        File[] files = pilihFile.getSelectedFiles();
        for (File file : files) {
            String fileSourceCode = file.getAbsolutePath();
        FileInputStream in = new FileInputStream(fileSourceCode);
            int i = fileSourceCode.lastIndexOf(".");
            eksFile = fileSourceCode.substring(i);
            CompilationUnit cu = null;
            try {
                if (getFileValid()) {
                    cu = JavaParser.parse(in);
         new Complexity(astParser, uOprt, uOprn).visit(cu, null);
                    ui.showParsingCode(dataPC);
                } else {
JOptionPane.showMessageDialog(null, "File tidak valid, silahkan
pilih file bertipe java.", "Peringatan!",
JOptionPane.WARNING_MESSAGE);
         hitungKompleksitas(ui, astParser, dataPC, uOprt, uOprn);
                }
            } finally {
                in.close();
            }
                    }
    }
```

Tabel 5.9 Implementasi *Method* riskEvaluation

```
public void riskEvaluation() {
        if ((1 <= CC) && (CC <= 10)) {
            riskEv = "Bebas dari resiko.";
        } else if ((11 <= CC) && (CC <= 20)) {
            riskEv = "Memiliki resiko sedang.";
        } else if ((21 <= CC) && (CC <= 50)) {
            riskEv = "Memilik resiko tinggi.";
        } else {
            riskEv = "Sangat beresiko.";
        }
}
```

Tabel 5.10 Implementasi *Method* getFileExistHV

```
public boolean getFileExistHV() {
        if(fileLogHalstead.exists()) {
            return cekFileHV = true;
        }
        else {
            return cekFileHV = false;
        }
    }
```

5.2.3 Implementasi *User Interface*

Implementasi *user interface* sistem hanya terdiri dari 1 halaman utama yang digunakan untuk menampilkan hasil *parsing code*, *log Halstead's Volume* dan *log Cyclomatic Complexity*.

5.2.3.1 *User Interface* Halaman Utama Sistem

Berikut ini implementasi *user interface* halaman utama sistem ditunjukan pada Gambar 5.7.

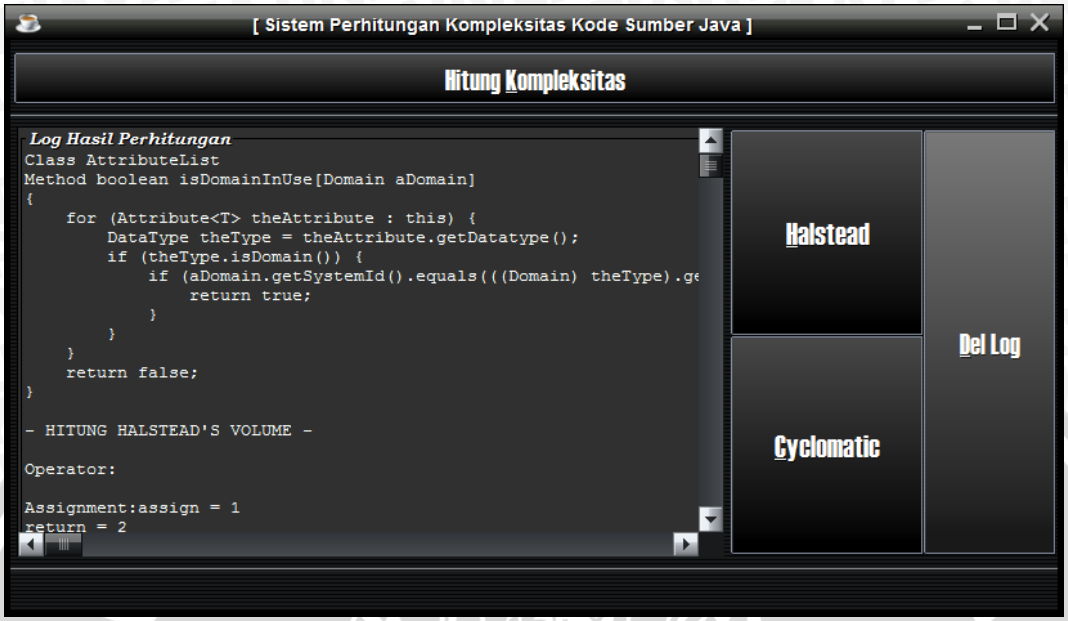
Gambar 5.7 *User Interface* Halaman Utama Sistem

Dalam halaman utama sistem yang ditunjukan pada Gambar 5.7, terdapat 4 tombol menu yang dapat digunakan oleh *user* untuk menjalankan fungsi sistem antara lain: tombol "Hitung Kompleksitas", tombol "Halstead", tombol "Cyclomatic" dan tombol "Del Log". *User* dapat menggunakan tombol-tombol menu dengan cara melakukan klik pada tombol atau menjalankan perintah *shortcut* pada setiap tombol.

5.2.3.2 *User Interface* Hasil *Parsing Code*

User interface hasil *parsing code* berfungsi untuk menampilkan hasil *parsing* kode sumber kepada *user*. Gambar 5.8 berikut ini merupakan

implementasi *user interface* hasil *parsing code* setelah proses hitung kompleksitas berhasil dilakukan.



Gambar 5.8 *User Interface* Hasil *Parsing Code*

Dalam *user interface* Gambar 5.8 ditampilkan hasil *parsing code* dan pengukuran kompleksitas kode sumber pada setiap *method* yang ada dalam masing-masing *class*. Data yang ditampilkan meliputi kode sumber *body method*, jumlah *predicate node*, jumlah *operator* dan *operand*, jumlah unik *operator* dan *operand* serta nilai kompleksitas *Halstead's Volume* dan *Cyclomatic Complexity*.

5.2.3.3 *User Interface Log Halstead's Volume*

User interface log Halstead's Volume berfungsi untuk menampilkan *log* hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*. Gambar 5.9 berikut ini merupakan implementasi *user interface* saat tombol "Halstead" ditekan.

Gambar 5.9 *User Interface Log Halstead's Volume*

Dalam *user interface* Gambar 5.9 ditampilkan nama *class*, nama *method*, hasil *Halstead's Volume* yang merupakan isi dari *file log Halstead* yang telah disimpan secara otomatis dalam proses hitung kompleksitas.

5.2.3.4 *User Interface Log Cyclomatic Complexity*

User interface log Cyclomatic Complexity berfungsi untuk menampilkan *log* hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*. Gambar 5.10 berikut ini merupakan implementasi *user interface* saat tombol "Cyclomatic" ditekan.

Gambar 5.10 *User Interface Log Cyclomatic Complexity*

Dalam *user interface* Gambar 5.10 ditampilkan nama *class*, nama *method*, hasil *Cyclomatic Complexity* dan evaluasi resiko yang merupakan isi dari *file log Cyclomatic* yang telah disimpan secara otomatis dalam proses hitung kompleksitas.

BAB 6 PENGUJIAN

6.1 Pengujian Unit

Pengujian unit yang dilakukan meliputi, pengujian unit *Data Halstead* dan pengujian unit *Data Cyclomatic*. Pengujian unit pada 2 *class* tersebut dilakukan dengan menguji secara mandiri salah satu *method* pada setiap *class* yaitu, *method* getFileExistHV pada *class Data Halstead* dan *method* getFileExistCC pada class *Data Cyclomatic*.

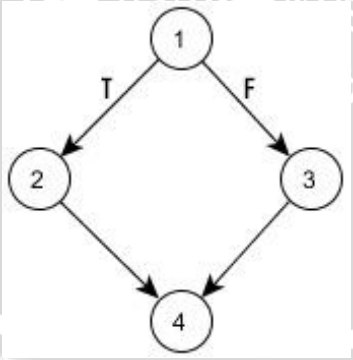
6.1.1 Pengujian Unit *Data Halstead*

Berikut ini merupakan pengujian unit yang dilakukan pada *class Data Halstead method* getFileExistHV. Algoritma *method* getFileExistHV ditunjukan pada Tabel 6.1.

Tabel 6.1 Algoritma *Method* getFileExistHV

```
FUNCTION getFileExistHV                    1
IF fileLogHalstead.exist THEN
  return cekFileHV = true;                 2
ELSE
  return cekFileHV = false;                3
ENDIF    4
END getFileExistHV
```

Berdasarkan algoritma pada Tabel 6.1, maka diperoleh hasil *flow graph* yang ditunjukan pada Gambar 6.1 berikut ini.



Gambar 6.1 *Flow Graph Method* getFileExistHV

Berdasarkan hasil *flow graph* pada Gambar 6.1, maka dapat dihitung jumlah *Cyclomatic Complexity* dengan persamaan sebagai berikut.

V(G) = E – N + 2

V(G) = 4 – 4 + 2

V(G) = 2

Sehingga, didapatkan 2 jalur independen yaitu:

Jalur 1: 1 – 2 – 4

Jalur 2: 1 – 3 – 4

Berdasarkan jalur independen yang telah ditentukan, maka dapat didefinisikan kasus uji yang dijelaskan pada Tabel 6.2.

Tabel 6.2 Kasus Uji *Method* getFileExistHV

Jalur	Kasus Uji	Prosedur Uji	*Expected Result*	*Test Result*
1	*File log Halstead's Volume* ditemukan.	1. Set nama *file log Halstead* sesuai dengan nama *file* di direktori penyimpanan *file*.	Mengembalikan nilai *true*.	Mengembalikan nilai *true*.
2	*File log Halstead's Volume* tidak ditemukan.	1. Set nama *file log Hasltead* yang berbeda dengan nama *file* di direktori penyimpanan *file*.	Mengembalikan nilai *false*.	Mengembalikan nilai *false*.

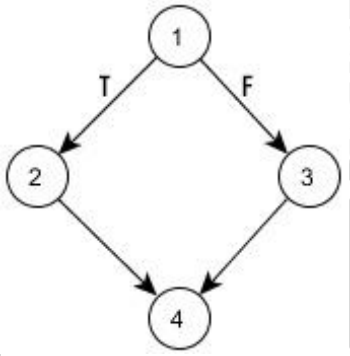
6.1.2 Pengujian Unit *Data Cyclomatic*

Berikut ini merupakan pengujian unit yang dilakukan pada *class Data Cyclomatic method* getFileExistCC. Algoritma *method* getFileExistCC ditunjukan pada Tabel 6.3.

Tabel 6.3 Algoritma *Method* getFileExistCC

```
FUNCTION getFileExistCC
IF fileLogCyclomatic.exist THEN          (1)
  return cekFileCC = true;               (2)
ELSE
  return cekFileCC = false;              (3)
ENDIF                                    (4)
END getFileExistCC
```

Berdasarkan algoritma pada Tabel 6.3, maka diperoleh hasil *flow graph* yang ditunjukan pada Gambar 6.2 berikut ini.



Gambar 6.2 *Flow Graph Method* getFileExistCC

Berdasarkan hasil *flow graph* pada Gambar 6.2, maka dapat dihitung jumlah *Cyclomatic Complexity* dengan persamaan sebagai berikut.

V(G) = E – N + 2

V(G) = 4 – 4 + 2

V(G) = 2

Sehingga, didapatkan 2 jalur independen yaitu:

Jalur 1: 1 – 2 – 4

Jalur 2: 1 – 3 – 4

Berdasarkan jalur independen yang telah ditentukan, maka dapat didefinisikan kasus uji yang dijelaskan pada Tabel 6.4.

Tabel 6.4 Kasus Uji *Method* getFileExistCC

Jalur	Kasus Uji	Prosedur Uji	*Expected Result*	*Test Result*
1	*File log Cyclomatic* ditemukan.	1. Set nama *file log Cyclomatic* sesuai dengan nama *file* di direktori penyimpanan *file*.	Mengembalikan nilai *true*.	Mengembalikan nilai *true*.
2	*File log Cyclomatic* tidak ditemukan.	1. Set nama *file log Cyclomatic* yang berbeda dengan nama *file* di direktori penyimpanan *file*.	Mengembalikan nilai *false*.	Mengembalikan nilai *false*.

Berdasarkan *test result* yang ditunjukan pada Tabel 6.2 dan Tabel 6.4 terhadap masing-masing 2 kasus uji pada *class Data Halstead method* getFileExistHV dan *class Data Cyclomatic method* getFileExistCC maka dapat disimpulkan bahwa hasil pengujian tersebut adalah valid.

6.2 Pengujian Integrasi

Pengujian integrasi yang dilakukan dengan menguji relasi antar *class* yang meliputi, *class System Control*, *class Data Halstead* dan *class Data Cyclomatic* dalam proses *delete log*. Relasi *class System Control* dengan *class Data Halstead* dan *class Data Cyclomatic* terdapat pada *method* getFileExistHV, deleteLogHalstead dan *method* getFileExistCC, deleteLogCyclomatic.

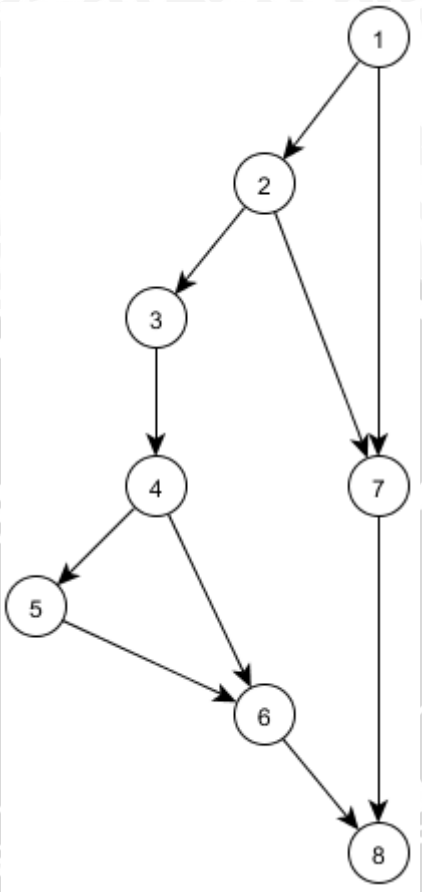
6.2.1 Pengujian Integrasi *Delete Log*

Berikut ini merupakan pengujian integrasi *delete log* dengan menguji *method* deleteLog pada class *System Control*. Algoritma *method* deleteLog ditunjukan pada Tabel 6.5.

Tabel 6.5 Algoritma *Method* deleteLog

```
PROCEDURE deleteLog(MainUI ui, DataHalstead dataHV,
DataCyclomatic dataCC)
IF dataHV.getFileExistHV == true                    (1)
AND dataCC.getFileExistCC == true                   (2)
THEN JOptionPane.showConfirmDialog "Anda yakin untuk   (3)
menghapus log?";
IF opsiPilihan == YES_OPTION THEN                   (4)
  dataHV.deleteLogHalstead;
  dataCC.deleteLogCyclomatic;                       (5)
  ui.logArea.setText("");
ENDIF                                               (6)
ELSE
  JOptionPane.showMessageDialog "File log
  tidak ditemukan.";                                (7)
ENDIF                                               (8)
END deleteLog
```

Berdasarkan algoritma pada Tabel 6.5, maka diperoleh hasil *flow graph* yang ditunjukan pada Gambar 6.3 berikut ini.



Gambar 6.3 *Flow Graph Method* deleteLog

Berdasarkan hasil *flow graph* pada Gambar 6.3, maka dapat dihitung jumlah *Cyclomatic Complexity* dengan persamaan sebagai berikut.

V(G) = E – N + 2

V(G) = 10 – 8 + 2

V(G) = 4

Sehingga, didapatkan 4 jalur independen yaitu:

Jalur 1: 1 – 7 – 8

Jalur 2: 1 – 2 – 7 – 8

Jalur 3: 1 – 2 – 3 – 4 – 6 – 8

Jalur 4: 1 – 2 – 3 – 4 – 5 – 6 – 8

Berdasarkan jalur independen yang telah ditentukan, maka dapat didefinisikan kasus uji yang dijelaskan pada Tabel 6.6.

Tabel 6.6 Kasus Uji *Method* deleteLog

Jalur	Kasus Uji	Prosedur Uji	*Expected Result*	*Test Result*
1	*File log Halstead* tidak ditemukan.	1. Hapus *file* logHalstead.txt dari direktori penyimpanan *file*.	Tampilkan *message dialog* "*file log* tidak ditemukan".	*Message dialog* "*file log* tidak ditemukan" ditampilkan.
2	*File log Halstead* ditemukan, file log *Cyclomatic* tidak ditemukan.	1. Buat *file* logHalstead.txt di direktori penyimpanan *file*. 2. Hapus *file* logCyclomatic.txt dari direktori penyimpanan *file*.	Tampilkan *message dialog* "*file log* tidak ditemukan".	*Message dialog* "*file log* tidak ditemukan" ditampilkan.
3	*File log Hasltead* dan *Cyclomatic* ditemukan, dipilih opsi *NO*.	1. Buat *file* logHalstead.txt dan logCyclomatic.txt di direktori penyimpanan *file*.	*File log Halstead* dan *Cyclomatic* tidak dihapus dari direktori penyimpanan *file*.	*File log Halstead* dan *Cyclomatic* tidak dihapus dari direktori penyimpanan *file*.
4	*File log Hasltead* dan *Cyclomatic* ditemukan, dipilih opsi *YES*.	1. Buat *file* logHalstead.txt dan logCyclomatic.txt di direktori penyimpanan *file*.	*File log Halstead* dan *Cyclomatic* dihapus dari direktori penyimpanan *file*.	*File log Halstead* dan *Cyclomatic* dihapus dari direktori penyimpanan *file*.

Berdasarkan *test result* yang ditunjukan pada Tabel 6.6 terhadap 4 kasus uji pada *class System Contol method* deleteLog maka dapat disimpulkan bahwa hasil pengujian tersebut adalah valid.

6.3 Pengujian Validasi

Berikut ini merupakan pengujian validasi berdasarkan skenario *use case* yang telah dibuat pada analisis kebutuhan.

6.3.1 Hitung Kompleksitas

Berikut ini adalah pengujian validasi Hitung Kompleksitas yang dijelaskan pada Tabel 6.7 dan Hitung Kompleksitas alur alternatif 1 yang ditunjukan pada Tabel 6.8.

Tabel 6.7 Hitung Kompleksitas

Nama Kasus Uji	Hitung Kompleksitas Normal (TC_F001)
Objek Uji	Kebutuhan Fungsional SRS_F001
Tujuan Pengujian	Untuk memastikan sistem mampu memenuhi kebutuhan fungsional melakukan proses hitung kompleksitas.
Data Masukan	*File* kode sumber bertipe java.
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Klik tombol "Hitung Kompleksitas". 3. Pilih *file* kode sumber bertipe java. 4. Klik tombol "Open"
Hasil yang Diharapkan	Sistem menampilkan hasil *parsing code* dan perhitungan kompleksitas setiap *method* dalam *class*.

Tabel 6.8 Hitung Kompleksitas Alur Alternatif 1

Nama Kasus Uji	Hitung Kompleksitas Alternatif 1 (TC_F002)
Objek Uji	Kebutuhan Fungsional SRS_F001
Tujuan Pengujian	Untuk memastikan sistem mampu menangani kondisi pemilihan *file* yang tidak sesuai dengan ketentuan.
Data Masukan	*File* kode sumber bukan bertipe java.
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Klik tombol "Hitung Kompleksitas". 3. Pilih *file* kode sumber selain tipe java. 4. Klik tombol "Open"
Hasil yang Diharapkan	Sistem mampu menampilkan pesan kesalahan bahwa "*file* tidak *valid*, silahkan pilih *file* bertipe java".

6.3.2 Lihat *Log Halstead's Volume*

Berikut ini adalah pengujian validasi Lihat *Log Halstead's Volume* yang dijelaskan pada Tabel 6.9 dan lihat *log Halstead's Volume* alur alternatif 1 yang ditunjukan pada Tabel 6.10.

Tabel 6.9 Lihat *Log Halstead's Volume*

Nama Kasus Uji	Lihat *Log Halstead's Volume* Normal (TC_F003)
Objek Uji	Kebutuhan Fungsional SRS_F002
Tujuan Pengujian	Untuk memastikan sistem mampu memenuhi kebutuhan fungsional membaca *file log* hasil kompleksitas *Halstead's Volume*.
Data Masukan	-
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Lakukan proses hitung kompleksitas. 3. Klik tombol "Halstead"
Hasil yang Diharapkan	Sistem mampu menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.

Tabel 6.10 Lihat *Log Halstead's Volume* Alur Alternatif 1

Nama Kasus Uji	Lihat *Log Halstead's Volume* Alternatif 1 (TC_F004)
Objek Uji	Kebutuhan Fungsional SRS_F002
Tujuan Pengujian	Untuk memastikan sistem mampu menangani kondisi ketika *file log Halstead's Volume* tidak ditemukan.
Data Masukan	-
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Klik tombol "Halstead"
Hasil yang Diharapkan	Sistem mampu menampilkan informasi bahwa "*file log Halstead's Volume* tidak ditemukan".

6.3.3 Lihat *Log Cyclomatic Complexity*

Berikut ini adalah pengujian validasi Lihat *Log Cyclomatic Complexity* yang dijelaskan pada Tabel 6.11 dan lihat *log Cyclomatic Complexity* alur alternatif 1 yang ditunjukan pada Tabel 6.12.

Tabel 6.11 Lihat *Log Cyclomatic Complexity*

Nama Kasus Uji	Lihat *Log Cyclomatic Complexity* Normal (TC_F005)
Objek Uji	Kebutuhan Fungsional SRS_F003
Tujuan Pengujian	Untuk memastikan sistem mampu memenuhi kebutuhan fungsional membaca *file log* hasil kompleksitas *Cyclomatic Complexity*.
Data Masukan	-
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Lakukan proses hitung kompleksitas. 3. Klik tombol "Cyclomatic"
Hasil yang Diharapkan	Sistem mampu menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.

Tabel 6.12 Lihat *Log Cyclomatic Complexity* Alur Alternatif 1

Nama Kasus Uji	Lihat *Log Cyclomatic Complexity* Alternatif 1 (TC_F006)
Objek Uji	Kebutuhan Fungsional SRS_F003
Tujuan Pengujian	Untuk memastikan sistem mampu menangani kondisi ketika *file log Cyclomatic Complexity* tidak ditemukan.
Data Masukan	-
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Klik tombol "Cyclomatic"
Hasil yang Diharapkan	Sistem mampu menampilkan informasi bahwa "*file log Cyclomatic Complexity* tidak ditemukan".

6.3.4 *Delete Log*

Berikut ini adalah pengujian validasi D*elete Log* yang dijelaskan pada Tabel 6.13, *delete log* alur alternatif 1, alternatif 2 dan alternatif 3 yang ditunjukan pada Tabel 6.14, Tabel 6.15 dan Tabel 6.16.

Tabel 6.13 *Delete Log*

Nama Kasus Uji	*Delete Log* Normal (TC_F007)
Objek Uji	Kebutuhan Fungsional SRS_F004
Tujuan Pengujian	Untuk memastikan sistem mampu memenuhi kebutuhan fungsional menghapus *file log* hasil kompleksitas *Halstead's Volume* dan *Cyclomatic Complexity*.
Data Masukan	-
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Lakukan proses hitung kompleksitas. 3. Klik tombol "Del Log". 4. Pilih opsi *YES*.
Hasil yang Diharapkan	Sistem mampu menghapus *file log* hasil perhitungan kompleksitas metode *Halstead's Volume* dan *Cyclomatic Complexity*.

Tabel 6.14 *Delete Log* Alur Alternatif 1

Nama Kasus Uji	*Delete Log* Alternatif 1 (TC_F008)
Objek Uji	Kebutuhan Fungsional SRS_F004
Tujuan Pengujian	Untuk memastikan sistem mampu menangani kondisi ketika *file log Halstead's Volume* tidak ditemukan.
Data Masukan	-
Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Klik tombol "Del Log".
Hasil yang Diharapkan	Sistem menampilkan informasi bahwa "*file log* tidak ditemukan".

Tabel 6.15 *Delete Log* Alur Alternatif 2

Nama Kasus Uji	*Delete Log* Alternatif 2 (TC_F009)
Objek Uji	Kebutuhan Fungsional SRS_F004
Tujuan Pengujian	Untuk memastikan sistem mampu menangani kondisi ketika *file log Halstead's Volume* ditemukan tetapi *file log Cyclomatic Complexity* tidak ditemukan.
Data Masukan	-
Prosedur Uji	3. Jalankan sistem perhitungan kompleksitas. 4. Lakukan proses hitung kompleksitas. 5. Hapus *file* logCyclomatic.txt dari direktori penyimpanan *file*. 6. Klik tombol "Del Log".
Hasil yang Diharapkan	Sistem menampilkan informasi bahwa "*file log* tidak ditemukan".

Tabel 6.16 *Delete Log* Alur Alternatif 3

Nama Kasus Uji	*Delete Log* Alternatif 3 (TC_F0010)
Objek Uji	Kebutuhan Fungsional SRS_F004
Tujuan Pengujian	Untuk memastikan sistem mampu menangani kondisi ketika *user* memilih opsi *NO*.
Data Masukan	-

Tabel 6.16 *Delete Log* Alur Alternatif 3 (lanjutan)

Prosedur Uji	1. Jalankan sistem perhitungan kompleksitas. 2. Lakukan proses perhitungan kompleksitas. 3. Klik tombol "Del Log" 4. Pilih opsi *NO*.
Hasil yang Diharapkan	Sistem tidak menghapus *file log* hasil perhitungan kompleksitas kode sumber kedua metode.

6.3.5 Akurasi Sistem

Berikut ini adalah pengujian validasi untuk kebutuhan non fungsional akurasi sistem yang dijelaskan pada Tabel 6.17.

Tabel 6.17 Akurasi Sistem

Nama Kasus Uji	Akurasi (TC_NF001)
Objek Uji	Kebutuhan Non Fungsional SRS_NF001
Tujuan Pengujian	Untuk memastikan sistem memiliki tingkat akurasi perhitungan kompleksitas minimal 50% pada setiap metode *Halstead's Volume* dan *Cyclomatic Complexity*.
Data Masukan	*File* kode sumber bertipe java.
Prosedur Uji	1. Lakukan proses perhitungan kompleksitas menggunakan sistem dan secara manual terhadap kode sumber dengan 30 java *class* data uji dari 3 aplikasi yaitu, ERDesigner, JXplorer – A Java Ldap Browser dan Java PasswordSafe yang diperoleh dari *website* sourceforge.net. 2. Bandingkan hasil perhitungan dari sistem dan manual, jika hasil keduanya sama maka hasil perhitungan valid. 3. Hitung jumlah data *file* kode sumber yang *valid* dan data yang tidak *valid* dari keseluruhan data uji *file* kode sumber. 4. Hitung akurasi berdasarkan total data *file* kode sumber yang *valid* dibagi dengan total seluruh data uji *file* kode sumber dikalikan 100%.
Hasil yang Diharapkan	Sistem memiliki tingkat akurasi perhitungan kompleksitas minimal 50% pada setiap metode *Halstead's Volume* dan *Cyclomatic Complexity*.

6.3.6 Analisis Hasil Pengujian Validasi

Analisis hasil pengujian validasi terdiri dari analisis hasil pengujian kebutuhan fungsional dan kebutuhan non fungsional.

6.3.6.1 Fungsional

Berdasarkan pengujian yang telah dilakukan pada setiap kasus uji kebutuhan fungsional, maka diperoleh hasil pengujian yang ditunjukan pada Tabel 6.18.

Tabel 6.18 Hasil Pengujian Fungsional

No	Kebutuhan	Kasus Uji	*Test Result*	Status Validitas
1	SRS_F001	TC_F001	Sistem menampilkan hasil *parsing code* dan perhitungan kompleksitas setiap *method*.	Valid
2	SRS_F001	TC_F002	Sistem menampilkan pesan kesalahan bahwa "*file* tidak *valid*, silahkan pilih *file* bertipe java".	Valid
3	SRS_F002	TC_F003	Sistem menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Halstead's Volume*.	Valid
4	SRS_F002	TC_F004	Sistem menampilkan informasi bahwa "*file log Halstead's Volume* tidak ditemukan".	Valid
5	SRS_F003	TC_F005	Sistem menampilkan rincian informasi hasil perhitungan kompleksitas kode sumber berdasarkan metode *Cyclomatic Complexity*.	Valid
6	SRS_F003	TC_F006	Sistem menampilkan informasi bahwa "*file log Cyclomatic Complexity* tidak ditemukan".	Valid
7	SRS_F004	TC_F007	Sistem menghapus *file log* hasil perhitungan kompleksitas metode *Halstead's Volume* dan *Cyclomatic Complexity*.	Valid
8	SRS_F004	TC_F008	Sistem menampilkan informasi bahwa "*file log* tidak ditemukan".	Valid
9	SRS_F004	TC_F009	Sistem menampilkan informasi bahwa "*file log* tidak ditemukan".	Valid
10	SRS_F004	TC_F0010	Sistem tidak menghapus *file log* hasil perhitungan kompleksitas kode sumber kedua metode.	Valid

6.3.6.2 Non Fungsional

Berdasarkan perhitungan yang telah dilakukan pada seluruh data uji 96 *method* dalam 30 java *class*, maka diperoleh hasil akurasi metode *Halstead's Volume* dan *Cyclomatic Complexity* yang ditunjukan pada Tabel 6.19 dan Tabel 6.20.

Tabel 6.19 Hasil Perhitungan Akurasi Metode *Halstead's Volume*

No	Nama *Class*	Nama *Method*	Hasil	
			Sistem	Manual
1	AttributeList	isDomainInUse	47.548	47.548
2	BaseRendererComponent	getPreferredSize	2.0	2.0
		Update	50.718	50.718
3	ClasspathCommand	Execute	36.541	36.541
4	DataTypeIO	getInstance	20.679	20.679
		jdbcTypeToString	64.529	64.529
		stringToJdbcType	58.810	65.729
		loadUserTypes	254.187	276.484
		deserializeDataTypesFrom	601.377	606.699
		serializeDataTypesFor	238.418	238.418
5	DataTypeList	findByName	30.880	30.880
6	DefaultEditorPane	getScrollPane	4.754	4.754
		Initialize	2.0	2.0
7	DialectFactory	getInstance	53.339	53.339
		registerDialect	12.0	12.0
		getDialect	10.0	10.0
		getSupportedDialects	28.073	28.073
8	JDBCUtils	closeQuietly	24.0	24.0
9	DomainList	findBySystemId	25.849	25.849
		findByName	30.880	30.880
10	EditorFactory	createEditorFor	247.362	247.362
11	GeneratorUtils	findClosestJavaTypeFor	10.0	10.0
		findClosestJavaTypeFor	30.0	30.0
		findClosestJavaTypeFor	245.822	245.822

Tabel 6.19 Hasil Perhitungan Akurasi Metode *Halstead's Volume* (lanjutan)

12	GenericConnectionProvider	createConnection	4.754	4.754
		createScriptStatementSeparator	8.0	8.0
		generatesManagedConnection	4.754	4.754
13	GenericFileFilter	Accept	23.264	23.264
		getDescription	8.0	8.0
		getCompletedFile	33.688	33.688
14	Index	getExpressions	4.754	4.754
		getIndexType	4.754	4.754
		setIndexType	13.93	13.93
		clone	42.110	42.110
		restoreFrom	33.604	33.604
		isModified	302.534	302.534
15	IndexExpressionList	findByAttributeName	56.472	56.472
		findByAttribute	25.849	25.849
		addExpressionFor	39.302	39.302
		removeAttribute	28.073	28.073
		containsAllExpressions	170.966	170.966
16	Cluster	computeHierarchy	164.233	164.233
		buildTree	175.136	167.586
		performClusterLayout	1886.542	1865.377
17	Util	centreWithin	86.370	86.370
		centreWithin	14.0	14.0
18	FileOpenDialogue	Accept	111.013	111.013
		getDescription	8.0	8.0
		setOptions	2.0	2.0

Tabel 6.19 Hasil Perhitungan Akurasi Metode *Halstead's Volume* (lanjutan)

19	PasswordSafeJ	Main	34.869	34.869
		addEntry	10.0	10.0
		addWindowEvents	18.575	18.575
		Exit	2.0	2.0
		leafSelected	20.679	20.679
		openFile	272.323	263.637
20	SpringUtilities	makeGrid	910.783	910.783
		getConstraintsForCell	78.869	78.869
		makeCompactGrid	780.195	780.195
21	TreeHandler	getRootNode	4.754	4.754
		getScrollPane	25.266	25.266
		getTree	4.754	4.754
		setOptions	42.0	42.0
		valueChanged	62.269	62.269
22	ButtonBar	actionPerformed	379.979	379.979
		setConnected	4.754	4.754
		setDisconnected	4.754	4.754
23	ButtonRegister	registerItem	36.0	36.0
		setItemEnabled	157.173	157.173
		setCommonState	118.536	118.536
		setDisconnectState	87.569	87.569
		setConnectedState	69.760	69.760
		Put	63.398	63.398
		Get	6.339	6.339
24	ConfirmDialog	**getUserResponse**	82.454	72.648
		actionPerformed	102.798	102.798
		startModal	198.808	198.808
		stopModal	2.01	2.0

Tabel 6.19 Hasil Perhitungan Akurasi Metode *Halstead's Volume* (lanjutan)

25	JXResourceLoader	addResource	194.486	194.486
		getInputStream	47.548	47.548
		getImage	56.472	56.472
		getResource	53.150	53.150
		getJarContainingResource	113.299	113.299
		getPrefixedResources	133.437	145.947
		getWildCardResources	311.777	343.872
26	KeystoreOptions	**setupKeyList**	352.837	361.885
		DoOK	50.718	50.718
27	TextFilterPanel	getFilter	8.0	8.0
		displayFilter	13.931	13.931
		isFilterValid	39.863	39.863
28	Audioaccessory	**actionPerformed**	55.350	49.828
		propertyChange	134.917	126.233
		setCurrentClip	206.438	206.438
		stopPlay	13.931	18.094
29	Largestringeditor	**setStringValue**	70.308	82.045
		doOK	22.458	22.458
30	usercertificateeditor	setValue	207.452	207.452

Berdasarkan hasil perbandingan perhitungan sistem dan manual, didapatkan bahwa dari total data uji 96 *method* dalam 30 *class* terdapat 84 *method* yang *valid* dan 12 *method* yang tidak valid. Sehingga, diperoleh nilai akurasi dibawah ini.

Akurasi sistem = $\frac{84}{96}$ x 100% = 87.5%

Tabel 6.20 Hasil Perhitungan Akurasi Metode *Cyclomatic Complexity*

No	Nama *Class*	Nama *Method*	Hasil	
			Sistem	Manual
1	AttributeList	isDomainInUse	4	4

Tabel 6.20 Hasil Perhitungan Akurasi Metode *Cyclomatic Complexity* (lanjutan)

2	BaseRendererComponent	getPreferredSize	1	1
		Update	1	1
3	ClasspathCommand	Execute	2	2
4	DataTypeIO	getInstance	2	2
		jdbcTypeToString	3	3
		stringToJdbcType	3	3
		loadUserTypes	6	6
		deserializeDataTypesFrom	5	5
		serializeDataTypesFor	3	3
5	DataTypeList	findByName	3	3
6	DefaultEditorPane	getScrollPane	1	1
		Initialize	1	1
7	DialectFactory	getInstance	3	3
		registerDialect	1	1
		getDialect	1	1
		getSupportedDialects	1	1
8	JDBCUtils	closeQuietly	2	2
9	DomainList	findBySystemId	3	3
		findByName	3	3
10	EditorFactory	createEditorFor	6	6
11	GeneratorUtils	findClosestJavaTypeFor	1	1
		findClosestJavaTypeFor	1	1
		findClosestJavaTypeFor	2	2
12	GenericConnectionProvider	createConnection	1	1
		createScriptStatementSeparator	1	1
		generatesManagedConnection	1	1

Tabel 6.20 Hasil Perhitungan Akurasi Metode *Cyclomatic Complexity* (lanjutan)

13	GenericFileFilter	Accept	2	2
		getDescription	1	1
		getCompletedFile	2	2
14	Index	getExpressions	1	1
		getIndexType	1	1
		setIndexType	1	1
		clone	2	2
		restoreFrom	2	2
		isModified	9	9
15	IndexExpressionList	findByAttributeName	4	4
		findByAttribute	3	3
		addExpressionFor	3	3
		removeAttribute	2	2
		containsAllExpressions	7	7
16	Cluster	computeHierarchy	5	5
		buildTree	5	5
		performClusterLayout	31	31
17	Util	centreWithin	1	1
		centreWithin	1	1
18	FileOpenDialogue	Accept	4	4
		getDescription	1	1
		setOptions	1	1
19	PasswordSafeJ	Main	1	1
		addEntry	1	1
		addWindowEvents	1	1
		Exit	1	1
		leafSelected	1	1
		openFile	3	3

Tabel 6.20 Hasil Perhitungan Akurasi Metode *Cyclomatic Complexity* (lanjutan)

20	SpringUtilities	makeGrid	6	6
		getConstraintsForCell	1	1
		makeCompactGrid	7	7
21	TreeHandler	getRootNode	1	1
		getScrollPane	2	2
		getTree	1	1
		setOptions	1	1
		valueChanged	2	2
22	ButtonBar	actionPerformed	14	14
		setConnected	1	1
		setDisconnected	1	1
23	ButtonRegister	registerItem	2	2
		setItemEnabled	7	7
		setCommonState	2	2
		setDisconnectState	3	3
		setConnectedState	5	5
		Put	3	3
		Get	1	1
24	ConfirmDialog	getUserResponse	3	3
		actionPerformed	4	4
		startModal	9	9
		stopModal	1	1
25	JXResourceLoader	addResource	3	3
		getInputStream	2	2
		getImage	2	2
		getResource	2	2
		getJarContainingResource	4	4
		getPrefixedResources	3	3
		getWildCardResources	5	5

Tabel 6.20 Hasil Perhitungan Akurasi Metode *Cyclomatic Complexity* (lanjutan)

26	KeystoreOptions	setupKeyList	9	9
		DoOK	1	1
27	TextFilterPanel	getFilter	1	1
		displayFilter	1	1
		isFilterValid	2	2
28	Audioaccessory	actionPerformed	3	3
		propertyChange	5	5
		setCurrentClip	4	4
		stopPlay	2	2
29	Largestringeditor	setStringValue	2	2
		doOK	1	1
30	usercertificateeditor	setValue	4	4

Berdasarkan hasil perbandingan perhitungan sistem dan manual, didapatkan bahwa dari total data uji 96 *method* dalam 30 *class* semuanya menunjukan hasil valid. Sehingga, diperoleh nilai akurasi dibawah ini.

$$\text{Akurasi sistem} = \frac{96}{96} \times 100\% = 100\%$$

Dari perhitungan akurasi yang telah dilakukan menggunakan metode *Halstead's Volume* dan *Cyclomatic Complexity* dilakukan analisis sebagai berikut.

1. Berdasarkan analisis hasil kompleksitas *Halstead's Volume* terhadap data uji 96 *method* dalam 30 *class* terdapat anomali pada 12 data uji, salah satunya pada data uji nomor 29 *method* setStringValue dimana hasil kompleksitas dari sistem tidak sesuai dengan hasil kompleksitas perhitungan manual. Hal tersebut disebabkan oleh adanya *operator-operator* yang berasal dari data *import library java* yang tidak dapat dideteksi oleh sistem.
2. Berdasarkan analisis hasil kompleksitas metode *Cyclomatic Complexity* didapatkan kompleksitas terbesar mencapai nilai 31 pada data uji nomor 16 *method* performClusterLayout, nilai kompleksitas sedang sebesar 14 pada data uji nomor 22 *method* actionPerformed dan nilai kompleksitas yang terkecil adalah 1 seperti pada data uji nomor 6 *method* Initialize. Besar kecil nilai kompleksitas tersebut berbanding lurus dengan banyaknya jumlah kondisi percabangan maupun kondisi perulangan dalam kode sumber setiap *method* dalam *class*.

BAB 7 PENUTUP

7.1 Kesimpulan

Kesimpulan yang diperoleh dalam penelitian Pengembangan Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity*, antara lain:

1. Perhitungan kompleksitas kode sumber java dapat dilakukan oleh sistem dengan metode *Halstead's Volume* dan *Cyclomatic* Complexity, melalui proses *parsing code* terhadap kode sumber menggunakan *library Java Parser and AST*.
2. Berdasarkan perhitungan akurasi didapatkan bahwa akurasi sistem untuk metode *Halstead's Volume* sebesar 87.5% dan metode *Cyclomatic Complexity* sebesar 100% menunjukan bahwa validitas algoritma kedua metode adalah valid.
3. Keterbatasan dari sistem ini adalah tidak dapat mendeteksi *operator* dalam kode sumber yang berasal dari data *import library* java.

7.2 Saran

Saran untuk penelitian Pengembangan Sistem Perhitungan Kompleksitas Kode Sumber Berdasarkan Metrik *Halstead* dan *Cyclomatic Complexity*, antara lain:

1. Akurasi metode *Halstead's Volume* dapat ditingkatkan dengan memberikan kemampuan kepada sistem untuk mendeteksi *operator* khusus dari data *import library* java.
2. Sebaiknya ditambahkan fitur untuk menampilkan *flow graph* dari setiap *method* dalam *class* yang telah berhasil dihitung kompleksitasnya agar memudahkan *user* memahami jalur independen program.
3. Dikembangkan secara lebih luas untuk dapat mengukur atau menghitung kompleksitas kode sumber bahasa pemrograman yang lain selain bahasa java.

DAFTAR PUSTAKA

Sommerville, I., 2011. *Software engineering*. 9th ed. London: Addison-Wesley.

Sommerville, I., 2010. *Software engineering*. 8th ed. London: Addison-Wesley.

Rosa, A.S. & Shalahudin, M., 2014. *Rekayasa perangkat lunak* (*terstruktur dan berorientasi objek*). Bandung: Modula.

Priyambadha, B. & Rochimah, S., 2013. *Sistem Deteksi Cacat Perangkat Lunak Berbasis Aturan Menggunakan Decision Tree*, [online] Tersedia di: <http://cybermatika.stei.itb.ac.id/ojs/index.php/cybermatika/article/download/42/15> [Diakses 23 Maret 2015]

Chhabra. P. & Bansal, L., 2014. *An Effective Implementation of Improved Halstead Metrics for Software Parameters Analysis*, [online] Tersedia di: <www.ijert.org/> [Diakses 23 Maret 2015]

Tomas, P., Escalona, M.J. & Mejias., 2013. Computer Standards & Interfaces. *Open source tools for measuring the Internal Quality of Java Software products*, [e-journal]. Tersedia melalui: Science Direct <www.sciencedirect.com> [Diakses 27 Maret 2015]

McCabe, J.T., 1976. Transactions on Software Engineering. *A Complexity Measure*, [e-journal]. Tersedia melalui: IEEE <ieeexplore.ieee.org> [Diakses 27 Maret 2015]

Fathoni., 2009. *Pengukuran Kualitas Perangkat Lunak Berdasarkan Kompleksitas Menggunakan Metode Function Point*, [online] Tersedia di: <http://download.portalgaruda.org/> [Diakses 15 April 2015]

Fahrurrozi, I. & Azhari, S.N., 2010. *Proses Pemodelan Software dengan Metode Waterfall dan Extreme Programming*: *Studi Perbandingan*, [online] Tersedia di: <http://jurnal.stmikelrahma.ac.id/> [Diakses 15 April 2015]

Zhang, H., Zhang, X. & Gu, M., 2007. International Symposium on Pacific Rim Dependable Computing. *Predicting Defective Software Components from Code Complexity Measures*, [e-journal]. Tersedia melalui: IEEE <ieeexplore.ieee.org> [Diakses 20 April 2015]

Grant, S., Cordy, R.J. & Skillicorn B.D., 2013. Science of Computer Programming. *Using heuristics to estimate an appropriate number of latent topics in source code analysis*, [e-journal]. Tersedia melalui: Science Direct <www.sciencedirect.com> [Diakses 20 April 2015]

Kearney. K.J., Sedlmeyer, L.R., Thompson, B.W., Gray, A.M. & Adler, A.M., 1986. *Software Complexity Measurement*, [online] Tersedia di: <www.lsmod.de/> [Diakses 15 April 2015]

Harman, M., 2010. *Why Source Code Analysis and Manipulation Will Always Be Importand*, [online] Tersedia di: <http://www0.cs.ucl.ac.uk> [Diakses 11 Mei 2015]

Dalal, S. & Chhillar, S.R., 2012. *Case Studies of Most Common and Severe Types of Software System Failure*, [online] Tersedia di: <http://www.ijarcsse.com/> [Diakses 11 Mei 2015]

Agarwal, K. A. & Katiyar, V., 2013. *Implementation of Cyclomatic Complexity Matrix*, [online] Tersedia di: <worldsciencepublisher.org> [Diakses 02 Juni 2015]

Sharma, A & Kushwaha, D.S., 2010. *A Complexity measure based on Requirement Engineering Document*, [online] Tersedia di: <arxiv.org/> [Diakses 02 Juni 2015]

Ankita., 2014. *A Framework for Improving the Concept of Cyclomatic Complexity in Object-Oriented Programming*, [online] Tersedia di: <http://dspace.thapar.edu/> [Diakses 19 Agustus 2015]

Saini, D. & Kaur, P., 2014. *Swarm Intelligence Technique and Artificial Intelligence Technique in Software Testing* (*Integer Type Test Data Generation*), [online] Tersedia di: <www.ijesrt.com/> [Diakses 19 Agustus 2015]

Madhavji, H.N., 1991. Software Engineering Journal. *The Process Cycle*, [e-journal]. Tersedia melalui: IEEE <ieeexplore.ieee.org> [Diakses 20 September 2015]

_____. *IEEE Standard Glossary of Software Engineering Terminologi*. IEEE std 610.12-1990.

Cahyono, S., 2006. *Panduan Praktis Pemrograman Database Menggunakan MySQL dan Java*. Bandung: Informatika.

Hosseini, R., 2013. *Fine-Grained Concept Indexing of Java Programming Contents Using JavaParser*, [online] Tersedia di: <http://www.pitt.edu/> [Diakses 25 Januari 2016]