LAPORAN PRAKTIK KERJA LAPANGAN (PKL)

PERUSAHAAN

PT. BIZNET GIO NUSANTARA

PEMBANGUNAN PENGUJIAN PADA NEO-CLI (AGNOSTIC ORCHESTRATION TOOLS FOR CLOUD INFRASTRUCTURE)

Diajukan untuk memenuhi sebagian persyaratan Kurikulum Sarjana

Disusun oleh:

Azzam Syawqi Aziz NIM : 155150207111132

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2018

PENGESAHAN

LAPORAN PRAKTIK KERJA LAPANGAN (PKL)

PERUSAHAAAN

PT. BIZNET GIO NUSANTARA

PEMBANGUNAN PENGUJIAN PADA NEO-CLI (AGNOSTIC ORCHESTRATION TOOLS FOR CLOUD INFRASTRUCTURE)

Diajukan untuk memenuhi sebagian persyaratan Kurikulum Sarjana

Program Studi Teknik Informatika

Bidang Rekayasa Perangkat Lunak

Disusun oleh:

Azzam Syawqi Aziz NIM : 155150207111132

Praktik Kerja Lapangan ini dilaksanakan pada
1 Juli sampai dengan 31 Agustus 2018

Telah diperiksa dan disetujui oleh:

Dosen Pembimbing PKL

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

Mengetahui,
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T, M.T, Ph.D
NIP: 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam laporan PKL ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain dalam kegiatan 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 laporan PKL ini terbukti terdapat unsur-unsur plagiasi, saya bersedia PKL ini digugurkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 31 Agustus 2018
Ketua Kelompok,

Azzam Syawqi Aziz
NIM : 155150207111132

KATA PENGANTAR

Puji syukur kehadirat Allah SWT yang telah melimpahkan rahmat, taufik dan hidayah-Nya sehingga laporan PKL yang berjudul "PEMBANGUNAN PENGUJIAN PADA NEO-CLI (AGNOSTIC ORCHESTRATION TOOLS FOR CLOUD INFRASTRUCTURE)" ini dapat terselesaikan.

1. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D selaku dosen pembimbing PKL yang telah dengan sabar membimbing dan mengarahkan penulis sehingga dapat menyelesaikan laporan ini.
2. Bapak Eka Tresna Irawan selalu pembimbing PKL lapangan yang telah memberikan banyak ilmu untuk penulis jadikan bekal pada penulisan laporan ini.
3. Bapak-bapak pengembang kakas-kakas bantu yang digunakan dalam laporan ini. Terimakasih telah menjawab pertanyaan-pertanyaan penulis. Hal ini secara tidak langsung mendukung kemudahan penyelesaian laporan ini.
4. Ayahanda dan Ibunda dan seluruh keluarga besar atas segala nasehat, kasih sayang, perhatian dan kesabarannya di dalam membesarkan dan mendidik penulis, serta yang senantiasa tiada henti-hentinya memberikan doa dan semangat demi terselesaikannya laporan ini.
5. Seluruh civitas akademika Teknik Informatika Universitas Brawijaya yang telah banyak memberi bantuan dan dukungan selama penyelesaian laporan PKL ini.

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

Malang, 31 Agustus 2018
Ketua Kelompok,

Azzam Syawqi Aziz
Email: azzamsa@student.ub.ac.id

ABSTRAK

Pengujian telah menjadi bagian penting dalam proses pembangunan perangkat lunak. Penggunaan perangkat lunak dalam aspek-aspek penting kehidupan mendorong para pengembang perangkat lunak untuk menghasilkan perangkat lunak dengan kemungkinan galat yang rendah. Oleh karena itu, pengujian menjadi proses penting untuk menghasilkan perangkat lunak dengan kriteria tersebut. Pada laporan ini, pengujian dilakukan pada tahapan *unit* dan *integration* dengan menggunakan metode pengujian *white-box* dan *black-box*. Pengujian *white-box* dilakukan dengan teknik *basis path testing*. *Basis path testing* dilakukan untuk mendapatkan kasus uji pada tahapan *unit* dan *integration*. Kasus uji pada proses pengujian *white-box* didapatkan mula-mula dengan menggambarkan *flow graph*, menghitung *cyclomatic complexity* lalu menentukan *independent path*. Dari *idependent path* yang didapatkan dibangun kasus uji dengan melewati setiap *path* yang ditemukan. Pada proses pengujian *black-box* kasus uji didapatkan dengan teknik *equivalence partitioning* dan *boundary value analysis*. *Equivalence partitioning* akan membagi data masukan yang digunakan untuk pengujian menjadi *valid* dan *invalid class*. Dari kedua kelas tersebut, diambil tiga data masukan. Sedangkan *boundary value analysis* menentukan data pengujian dari nilai-nilai yang berada pada daerah pinggir. Laporan ini telah membahas proses pembangunan pengujian pada perangkat lunak *neo-cli* yang berperan penting dalam operasional perusahaan. Oleh karena itu, pengujian dilakukan agar *neo-cli* memiliki kemungkinan galat yang rendah.

Kata kunci: pengujian, *basis path testing*, *equation partitioning*, *boundary value analysis*

ABSTRACT

Testing has become an important part of software development process. Software usage in imprtant life aspect encourages software developers to produce software with low possible errors. Therefore, testing is an important process to produce software with these criteria. In this report, testing is carried out at the stages of unit and integration by using white-box testing method and black-box testing method. White-box testing performed with basis path testing. Basis path testing performed at unit and integration level. Test case in the white-box testing process was obtained initially by draw the flow graph, calculates the cyclomatic complexity then specify independent path. Test case then built to pass each resulted path from previous step. In the black-box testing process, test case obtained with the equivalence partitioning and boundary value analysis technique. Equivalence partitioning will divide the input data for testing usage into valid andinvalid class. Of the two classes, taken three input data. Whereas boundary value analysis technique determine test data from values in the edge area. This report has discussed the testing development process on neo-cli software which plays an important role in company operations. Therefore, testing performed on neo-cli to lower its possibility of error.

Keyword: *testing*, *basis path testing*, *equation partitioning*, *boundary value analysis*

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

BAB 1 PENDAHULUAN

1.1 Latar Belakang

Saat ini perangkat lunak telah digunakan secara masif oleh manusia. Semua bidang dalam setiap aspek kehidupan hampir seluruhnya membutuhkan perangkat lunak. Mulai dari transportasi, kesehatan, pendidikan, tenaga pembangkit listrik dan berbagai macam bidang-bidang lainnya. Meningkatnya penggunaan perangkat lunak berpengaruh secara langsung pada kebutuhan infrastruktur teknologi informasi. Mahal dan sulitnya pemeliharaan infrastruktur teknologi informasi membuat penyedia jasa infrastruktur tekologi informasi kian menjamur. BiznetGio Nusantara merupakan salah satu diantaranya. BiznetGio Nusantara menyediakan jasa penyewaan infrastruktur teknologi informasi bernama *NeoCloud*. *NeoCloud* merupakan infrastruktur *cloud* berbasis *OpenStack*. *NeoCloud* melayani puluhan perusahaan besar di Indonesia sehingga kegagalan sistem pada *NeoCloud* merupakan hal yang sangat fatal. Kegagalan perangkat lunak dapat menyebabkan kerugian materi hingga kehilangan nyawa (Wong, Debroy, & Restrepo, 2009). Akan tetapi, kegagalan perangkat lunak dapat diantisipasi dengan melakukan prosedur pengujian pada proses pengembangan perangkat lunak. Pengujian dapat memprediksi kegagalan lebih dini dan menghasilkan perangkat lunak yang berkualitas (Burnstein, 2006).

Maka untuk mengantisipasi kegagalan dan menjadikan *NeoCloud* perangkat lunak yang berkualitas tinggi, dilakukanlah pengujian pada *NeoCloud*. Pada praktik kerja lapangan ini pengujian dilakukan pada *neo-cli*. *Neo-cli* merupakan perangkat lunak bagian dari *NeoCloud* yang digunakan oleh pengguna *NeoCloud* untuk memudahkan pengaturan infrastruktur *cloud* yang mereka miliki. Pengujian yang akan dibangun pada *neo-cli* dilakukan pada beberapa tahapan, yaitu pada tahapan *unit* dan *integration*. Pemilihan batas tahapan ini ditentukan oleh pembimbing lapangan dengan beberapa pertimbangan. Salah satunya adalah waktu praktik kerja lapangan yang terbatas. Sedangkan metode pengujian yang digunakan adalah *white-box testing* dan *black-box testing*. Metode *white-box testing* tidak dapat menemukan kebutuhan yang belum diimplementasikan (Dijkstra, 1970). Hal ini dapat diselesaikan dengan penggunaan metode *black-box testing*. Begitu juga dengan kekurangan metode *black-box testing* yang dapat menghasilkan *test-case* yang tidak tepat dan tidak dapat menemukan bagian yang belum diuji (Savenkov, 2008). Kelemahan pada metode *black-box testing* ini dapat diselesaikan dengan menggunakan metode *white-box testing*. Oleh karena itu, pada laporan ini kedua metode tersebut digunakan. Pengujian semua *path* atau *rigorous testing* pada proses *white-box*

testing tidak mungkin dilakukan sehingga digunakan teknik *basis path testing* untuk menentukan *path* yang akan diuji (Gregory, 2007). Begitu juga pada proses *black-box testing*. Pengujian semua kemungkinan *input* tidak dapat dilakukan sehingga digunakan teknik *equivalence partitioning* untuk menentukan *valid input* dan *invalid input* yang akan digunakan dan penggunaan teknik *boundary value analysis* untuk memilih *input* yang ada pada nilai-nilai *boundary*, karena nilai-nilai pada bagian tersebut memiliki kemungkinan galat yang besar (Presman, 2010). Proses pengujian menyita banyak waktu (Brooks Jr, 1995) sehingga *automated testing* juga digunakan pada laporan ini untuk mempercepat proses pengujian.

Pada laporan ini pengujian pada *neo-cli* akan dibangun pada tahapan *unit* dan *integration* dengan menggunakan kedua metode *white-box testing* beserta *black-box testing*. Pada proses *white-box testing* akan digunakan teknik *basis path testing*. Sedangkan pada proses *black-box testing* akan digunakan teknik *equivalence partitioning* dan *boundary value analysis*. *Automated testing* juga digunakan pada pengembangan pengujian *neo-cli* untuk mempercepat proses pengujian.

1.2 Rumusan Masalah

1. Bagaimana perancangan dan implementasi pengujian dengan teknik *basis path testing* pada perangkat lunak *neo-cli* ?
2. Bagaimana perancangan dan implementasi pengujian dengan teknik *equivalence partitioning* dan *boundary value analysis* pada perangkat lunak *neo-cli* ?

1.3 Tujuan

1. Merancang dan mengimplementasikan pengujian dengan teknik *basis path testing* pada perangkat lunak *neo-cli*.
2. Merancang dan mengimplementasikan pengujian dengan teknik *equivalence partitioning* dan *boundary value analysis* pada perangangkat lunak *neo-cli*.

1.4 Manfaat

Pembangunan pengujian pada perangkat lunak *neo-cli* diharapkan dapat memberikan kemudahan bagi pengembang untuk menemukan *bug*. Baik *bug* yang tersebunyi maupun yang muncul karena adanya penambahan fitur. Selain itu, laporan ini diharapkan memberikan informasi yang menjelaskan tentang proses pe-

ngujian dalam perangkat lunak. Baik dengan pendekatan *black-box* atau pun *black-box* dan teknik-teknik yang digunakan di dalamnya.

1.5 Batasan Masalah

1. Beberapa modul tidak memiliki dukungan untuk proses pengujian. Seperti modul *ncurses*
2. *Unit* yang berinteraksi dengan dunia luar tidak di uji secara *end-to-end* pada proses *integration testing*.

1.6 Sistematika Pembahasan

BAB I : PENDAHULUAN

Bab ini menjelaskan tentang latar belakang masalah, rumusan masalah, tujuan, manfaat, batasan masalah, dan sistematika pembahasan.

BAB II : PROFIL OBYEK PKL

Bab ini menjelaskan tentang sejarah perusahaan, visi dan misi perusahaan serta struktur organisasi perusahaan.

BAB III : TINJAUAN PUSTAKA

Bab ini menjelaskan dasar teori-teori yang digunakan untuk melakukan pengujian pada suatu perangkat lunak. Di dalamnya juga terdapat penjelasan teknologi yang digunakan untuk membangun pengujian pada perangkat lunak.

BAB IV : METODOLOGI

Bab ini menjelaskan metode-metode yang digunakan selama pengujian. Di dalamnya terdapat alur seperti proses analisis kebutuhan pengujian, perancangan dan implementasi kasus uji, dan penarikan kesimpulan.

BAB V : HASIL DAN PEMBAHASAN

Bab ini menjelaskan hasil dan pembahasan pengujian. Hasil dan pembahasan tersebut mencangkup langkah-langkah seperti analisis kebutuhan pengujian, dan perencanaan dan implementasi kasus uji

BAB VI : PENUTUP

Bab ini menjelaskan kesimpulan selama proses pengujian, saran untuk pengujian selanjutnya dan penjelasan terkait keberlanjutan pengujian yang dibangun.

BAB 2 PROFIL OBYEK PKL

2.1 Sejarah Perusahaan

Biznet Network merupakan perusahaan telekomunikasi yang di bantun pada tahun 2000. Awalnya Biznet hanya memiliki fokus pada dunia korporat dengan layanan internet, pusat data, serta layanan *hosting* dan *cloud computing*. Selanjutnya, pada tahun 2006 Biznet terus merebahkan sayapnya dengan membangun Biznet Metro yang merupakan *Carrier Grade Metro Ethernet Network* pertama di Indonesia. Setahun setelahnya diiringi munculnya Biznet Metro FTTH yang merupakan jaringan serat optik pertama di Asia Tenggara yang dapat melayani langsung hingga ke perumahan (BiznetGioNusantara, 2018).

BiznetGio merupakan hasil dari kerja sama (*joint venture*) antara Biznet Networks (www.biznetnetworks.com) dan Internet Initiative Japan (www.iij.ad.jp). BiznetGio dibangun pada tahun 2014 dengan fokus untuk menyediakan layanan *cloud computing*. Hal ini dapat mereka capai dengan memanfaatkan teknologi Biznet Networks seperti *data center* dan jarigan yang kemudian dikombinasikan dengan teknologi yang dimiliki oleh IIJ dalam bidang *cloud service*. Saat ini BiznetGio memiliki dua produk utama. Yaitu GIO Cloud dan NEO Cloud (BiznetGioNusantara, 2018).

2.2 Visi dan Misi Perusahaan

2.2.1 Visi

Indonesia dimana setiap individu dan bisnis dapat terhubung dengan lancar untuk menggapai potensi mereka secara individu dan kolektif.

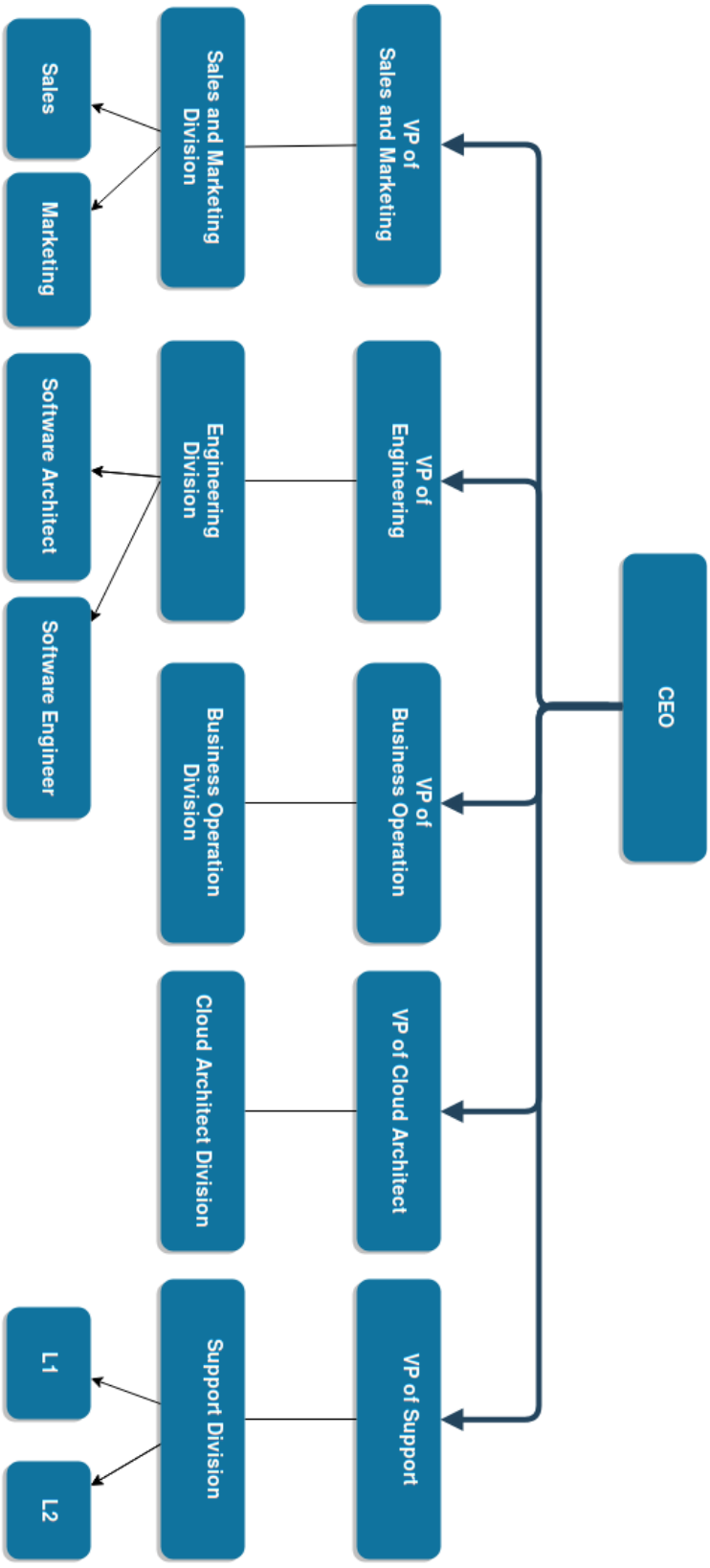
2.2.2 Misi

Menjadi perusahaan solusi jaringan dan multimedia melalui komitmen kami untuk inovasi kelas dunia, infrastruktur dan jasa.

2.3 Struktur Organisasi Perusahaan

Struktur organisasi pada perusahaan Biznet Gio Nusantara pada Gambar 2.1 terdiri dari *chief executive officer CEO* yang memimpin perushaaan. *CEO* membawahi beberapa *vice president VP*. *Vice president* dalam perusahan Biznet Gio Nusantara terdiri dari *vp of sales and marketing*, *vp if engineering*, *vp of bussines ope-*

ration, *vp of cloud architect*, dan *vp of support*. *vp of sales and marketing* bertugas untuk mengatur divisi *sales and marketing* yang mana divisi tersebut bertanggung jawab terhadap pemasaran dan penjulan. *Vp of engineering* bertugas untuk mengatur divisi *engineering*. Dalam divisi *engineering* terdapat staf *software architect* yang bertugas mendesain perangkat lunak dan staff *software engineer* bertugas mengimplementasikan perangkat lunak dari desain yang dihasilkan oleh staff *software architect*. *Vp of bussines operation* membawahi divisi *bussines operartion* yang bertugas mengatur operasional perusahaan sehari-hari. *Vp of cloud architect* membawahi divisi *cloud architect* yang bertugas mendesain arsitektur *cloud* Biznet Gio Nusantara dan *Vp of support* membawahi divisi *support* yang terdiri dari staf *L1* dan *L2*. Keduanya bertugas mengurusi keluhan pelanggan.



Gambar 2.1: Struktur Perusahaan Biznet Gio (BiznetGioNusantara, 2018)

BAB 3 LANDASAN KEPUSTAKAAN

3.1 Pengujian

Penggunaan perangkat lunak yang masif tanpa adanya standarisasi membuat *IEEE* merilis sebuah *issue* pada November-December 1999. *IEEE* dan *ACM* kemudian membentuk tim gabungan untuk mendefinisikan standardisasi pada proses pengembangan perangkat lunak. Standardisasi tersebut memuat tentang *scientific principles, engineering processes, standards, methods, tools, measurement and best practices*. Dimana pengujian atau *testing* termasuk di dalamnya (Burnstein, 2006).

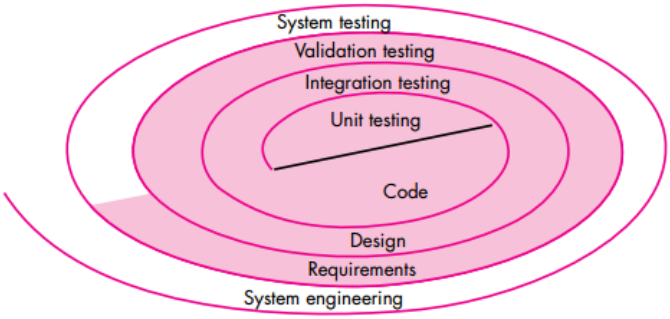
Pengujian bertujuan untuk menemukan cacat dan menguji kualitas suatu perangkat lunak. Dalam proses pengujian terdapat beberapa metode, tingkatan dan teknik. Setiap komponen pada bagian-bagian tersebut memiliki kelemahan dan kekurangan tertentu. Proses testing juga akan menghasilkan artefak-aretafak seperti *test-scenario, test-case, test-data* dan *test scripts* (Burnstein, 2006).

3.1.1 Tahapan Pengujian

Terdapat beberapa tahapan dalam pengujian. Tahapan-tahapan tersebut seperti *unit testing, integration testing, validation testing* dan *system testing* (Presman, 2010). Pada laporan ini tahapan yang akan dilakukan adalah *unit* dan *integration*. Batasan tahapan pengujian tersebut ditentukan oleh pembimbing lapangan dengan beberapa pertimbagan. Salah satunya adalah waktu praktik kerja lapangan yang terbatas.

Tahapan awal dalam pengujian adalah pengujian *unit*. Terdapat beberapa pendapat berbeda mengenai *'unit'* dalam *unit testing*. Fowler (2018b) berpendapat bahwa definisi *'unit'* dapat berarti *single method, single class* maupun kumpulan dari beberapa *class*. Osherove (2015) tidak mengikat definsi *'unit'* pada *class* maupun *method*. Melainkan mengikatnya dengan definsi atribut seperti *performance, reliability* dan *consistency*. *Google* tidak menggunakan istilah *unit, integration* ataupun *system* untuk merujuk pada tingkatan tertentu. Melainkan menggunakan *small, medium, large* (Whittaker, Arbon, & Carollo, 2012). Sedangkan Presman (2010) berpendapat bahwa fokus pengujian *unit* pada perangkat lunak konvensional adalah sebuah modul dan fokus pengujian *unit* pada perangkat lunak berorientasi objek adalah *class* dengan *testable unit* terkecil adalah *operation* atau *method*. Laporan ini mengikuti pendapat Presman (2010) dalam pemahaman pengujian *unit*. Tahapan selanjutnya adalah *Integration testing*. *Integrartion testing*

merupakan pengujian yang dilakukan pada kumpulan beberapa *unit* yang telah diuji. Hal ini dilakukan untuk memastikan suatu *unit* akan berjalan dengan baik jika dijalankan bersamaan dengan *unit* yang lain. Meskipun setiap *unit* telah diuji secara individu, terdapat beberapa kemungkinan galat yang terjadi ketika *unit-unit* tersebut diuji secara bersamaan. Beberapa kemungkinan galat yang akan terjadi di antaranya adalah hilangnya data ketika melintasi *interface* dan komponen yang tidak berjalan semestinya ketika digabungkan (Presman, 2010). Pada proses pengujian *unit* maupun *integrartion*, komponen tidak berupa *stand-alone program*. Maka dibutuhkan pembuatan *stub* ataupun *driver* selama proses pengujian. *Driver* hanyalah *main program* tiruan yang menjalankan kasus uji. Sedangkan *stub* merupakan sebuah *dummy subprogram* yang bekerja menggantikan komponen-kompenen pengujuian yang asli. Penggunaan *stub* dapat digantikan dengan *mock*. Pembuatan *mock* memiliki tujuan yang sama dengan *stub* (Fowler, 2018a) . Terlihat tahapan pengujian pada Gambar 3.2.



Gambar 3.2: Tahapan Pengujian (Presman, 2010)

3.1.2 Metode Pengujian

Metode atau yang juga dapat disebut *point of view* merupakan sudut pandang seorang *tester* tatkala mendesain sebuah *test case*. Terdapat di antaranya *white-box testing* dan *black-box testing*. Kedua metode tersebut memiliki kekurangan masing-masing. Metode *white-box testing* tidak dapat menemukan kebutuhan yang belum diimplementasikan (Dijkstra, 1970). Hal ini dapat diselesaikan dengan penggunaan metode *black-box testing*. Begitu juga dengan kekurangan metode *black-box testing* yang dapat menghasilkan *test-case* yang tidak tepat dan tidak dapat menemukan bagian yang belum diuji (Savenkov, 2008). Kelemahan pada metode *black-box testing* ini dapat diselesaikan dengan menggunakan metode *white-box testing*. Oleh karena itu, pada laporan ini kedua metode tersebut digunakan.

White-box testing merupakan suatu medote pengujian dimana proses pengujiannya dilakukan dengan melihat internal perangkat lunak sehingga pada tahapan ini seorang penguji wajib memiliki kemampuan pemrograman. Pengujian ini dilakukan untuk menguji apakah *logic* dan *data* pada perangkat lunak berfungsi sebagaimana mestinya (Myers, Sandler, & Badgett, 2011). Di sisi lain *black-box testing* hanya menguji bagian luar suatu perangkat lunak. Pada tahapan ini fungsionalitas suatu perangkat lunak diuji tanpa harus mengetahui internalnya. Oleh karena itu, tidak dibutuhkan kemampuan pemrograman saat membuat *test case* pada pengujian *black-box*. Penguji hanya mengetahui bagaimana seharusnya perangkat lunak berjalan (*what*), bukan bagaimana perangkat lunak itu melakukan sesuatu (*how*). Pada tahapan ini penguji hanya mempertimbangkan masukan (*input*) dan keluaran (*output*) perangkat lunak selama mendesain *test-case* (Myers, Sandler, & Badgett, 2011).

3.1.3 Teknik Pengujian

Terdapat beberapa teknik dalam pengujian. Pengujian semua *path* atau *rigorous testing* pada proses *white-box testing* tidak mungkin dilakukan sehingga digunakan teknik *basis path testing* untuk menentukan *path* yang akan diuji (Gregory, 2007). *Rigorous testing* atau *exhaustive testing* ($C\infty$) atau menguji semua kemungkinan juga tidak mungkin dilakukan pada saat pengujian *black-box* sehingga penggunaan teknik *equivalence partitioning* untuk menentukan masukan yang *valid* dan *invalid* diperlukan. Teknik *boundary value analysis* menjadi pelengkap teknik *equivalence partitioning*. *Boundary value analysis* digunakan untuk mendapatkan nilai masukan yang berada pada *boundary* atau *corner*, karena nilai-nilai pada bagian tersebut memiliki kemungkinan galat yang besar (Presman, 2010).

3.1.3.1 *Basis Path Testing*

Basis path testing merupakan suatu teknik pengujian pada metode *white-box* yang diajukan oleh McCabe pada tahun 1996. Teknik ini menggunakan penelitian McCabe yang sebelumnya yaitu *cyclomatic complexity* yang ia temukan pada 1976. Meski saat ini *cyclomatic complexity* banyak dikaitkan secara erat dengan rumus untuk menentukan jumlah *independent path*. *Cyclomatic complexity* awalnya ditemukan pada tahun 1976 diajukan untuk mengukur tingkat kompleksitas suatu modul dalam sebuah program. Modul yang melebihi nilai *cyclomatic complexity* 10 direkomendasikan untuk dipisah atau dilakukan *refactoring*. Tujuan kedua dari rumus tersebut adalah untuk melihat tingkat terstrukturnya (*structuredness*) suatu

modul. Hal yang membuat *cyclomatic complexity* dikaitkan secara erat dengan proses *basis path testing* karena kemdian McCabe sadar bahwa *cyclomatic complexity* dapat menemukan *independent path* pada suatu modul. *Independent path* adalah suatu jalur dalam sebuah program yang memperkenalkan setidaknya suatu rangkaian pemrosesan baru atau kondisi baru. Hal tersebut yang mengantarnya pada pengajuan *basis path testing* pada 1996. *Basis path testing* memiliki keunggulan karena V(G) atau *vector space* memiliki nilai yang menjadi batas atas *upper bound* untuk membuat *test-case* pada suatu program. Hal ini menguntungkan penguji karena dapat menentukan ukuran batas selesainya suatu pengujian. Selain itu, *basis path testing* memiliki nilai yang lebih baik dari pada *branch coverage*, yaitu *branch coverage ≤ cyclomatic complexity ≤ all of paths* (Gregory, 2007).

Langkah-langkah pengujian *basis path* adalah sebagai berikut (Presman, 2010):

1. Membuat *flow graph* dari desain atau *code*.

   Flow graph memiliki beberapa notasi seperti *sequence, if, while, until* dan *case*. Gambar notasi-notasi tersebut terlihat pada Gambar 3.3. Terlihat *flow graph* pada Gambar 3.4 yang dihasilkan dari *pseudocode* 3.1

2. Menentukan *cyclomatic complexity* dari *flow graph* yang dihasilkan.

   Cyclomatic complexity adalah suatu metrik perangkat lunak yang memberikan ukuran kuantitatif kompleksitas logis dari suatu program. *Cyclomatic complexity* dapat ditentukan dengan menggunakan Rumus 3.1:

$$
\begin{aligned}
V(G) &= jumlah\, region \\
V(G) &= E – N + 2 \\
V(G) &= P + 1,\ dimana\, P – predicate\, node
\end{aligned}
\tag{3.1}
$$

Maka Perhitungan *cyclomatic complexity* yang dilakukan pada *flow graph* 3.4 menghasilkan nilai sebagai berikut:

$$
\begin{aligned}
V(G) &= 2\, regions \\
V(G) &= 4\, edges – 4\, nodes\, +\, 2\, =\, 2 \\
V(G) &= 1\, predicate\, node\, +\, 1\, =\, 2
\end{aligned}
\tag{3.2}
$$

Jadi, *flow graph* pada Gambar 3.4 memiliki nilai *cyclomatic complexity* = 2.

3. Menentukan *independent path*

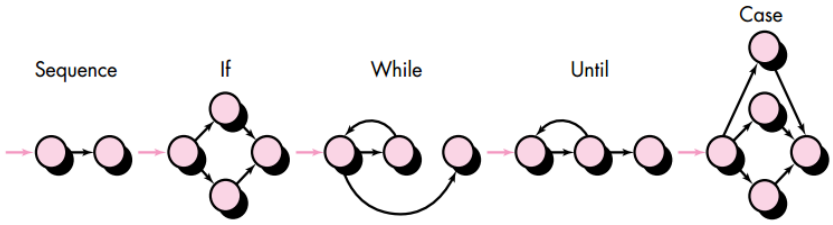
Independent path adalah suatu jalur dalam sebuah program yang memperkenalkan setidaknya suatu rangkaian pemrosesan baru atau kondisi baru. Nilai dari V(G) memberikan batas atas dari banyaknya *independent path* dari sebuah program. Dari contoh *pseudocode procedure hallo* kita mendapatkan 2 jalur *independent* sebagai berikut :

Jalur 1: 1 - 2 - 4

Jalur 2: 1 - 3 - 4

4. Membuat *test case* yang akan menjalankan setiap jalur pada *basis path*

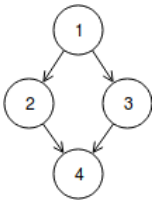
Terdapat dua *test case* yang akan dihasilkan. Pertama *test case* yang harus melewati jalur 1 sehingga pada *test case* pertama nilai *variable nama* harus bernilai *true*. Pada *test case* kedua nilai *variable nama* harus bernilai *false* agar jalur 2 dijalankan. *Test case* yang dihasilkan tampak seperti pada Tabel 3.1



Gambar 3.3: Notasi *flow graph* (Presman, 2010)

No	hallo
1	`procedure hallo(nama)`
2	`   IF nama == "Budi"  (1)`
3	`      RETURN "Hai" + nama  (2)`
4	`   ELSE`
5	`      RETURN "Nama kosong"  (3)`
6	`   ENDIF  (4)`
7	`end`

Tabel Kode 3.1: Contoh *pseudocode*



Gambar 3.4: Representasi *control flow graph* dari *pseudocode* 3.1

Tabel 3.1: *Test case* dari contoh *pseudocode* 3.1

Jalur	**Prosedur Uji**	***Expected Result***
1	Memberikan nilai *"Budi"* pada variable *nama*	Program menampilkan "Hai Budi"
2	Tidak memberikan nilai apapun pada variable *nama*	Program menampilkan "Nama kosong"

3.1.3.2 Equivalence Partitioning

Equivalence partitioning merupakan suatu teknik dalam metode pengujian *black-box* dimana prosesnya adalah membangi masukan kepada kelas-kelas data. Dari kelas-kelas tersebut *test case* nantinya didapatkan. *Equivalence class* merepresentasikan keadaan valid tau tidak validnya suatu masukan. Beberapa kondisi masukan di antaranya *numeric value, range of values, set of related values,* atau *boolean condition*. *Equivalence partitioning* dapat di definisikan sesuai dengan panduan berikut (Presman, 2010):

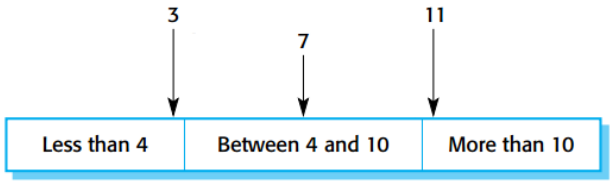
1. Jika kondisi masukan adalah *range*. Makan satu valid dan dua invalid *equivalence class* di definisikan.

2. Jika kondisi masukan adalah *specific value*. Makan satu valid dan dua invalid *equivalence class* di definisikan.

3. Jika kondisi masukan adalah *member of a set*. Makan satu valid dan satu invalid *equivalence class* di definisikan.

4. Jika kondisi masukan adalah *boolean*. Makan satu valid dan satu invalid *equivalence class* di definisikan.

Jika data masukan bertipe *range* dan masukan yang valid adalah *range* bernilai 4 hingga 10. Maka masukan yang tidak valid adalah semua angka yang kurang dari 4 dan semua angka yang lebih dari 10 seperti terlihat pada Gambar 3.5. Data yang digunakan untuk *test case* pada tipe masukan *range* berjumlah 3. Satu bernilai valid dan dua bernilai invalid. Oleh karena itu, data masukan *valid* yang kita gunakan adalah 11 dan 7. Sedangkan data *invalid* yang digunakan adalah 3.



Gambar 3.5: Contoh *equivalence partitioning* (Sommerville, 2014)

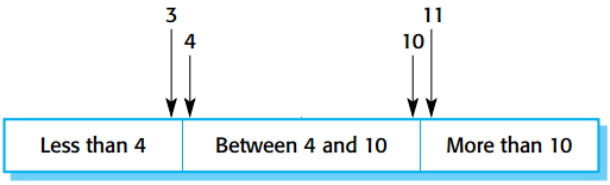
3.1.3.3 *Boundary Value Analysis*

Boundary value analysis merupakan suatu teknik yang melengkapi *equivalence partitioning*. *Boundary value analysis* dikembangkan untuk menganalisis nilai yang berada pada *boundary* masukan, karena nilai pada *boundary* atau *corner* memiliki peluang galat yang lebih tinggi. *Boundary value analysis* tidak berdiri sendiri, melainkan merupakan teknik yang melengkapi *equivalence partitioning*. Maka *boundary value analysis* mengambil nilai yang bersifat "pojok" dari nilai hasil *equivalence partitioning*. Panduan pemilihan nilai *boundary value analysis* di antaranya sebagai berikut (Presman, 2010):

1. Jika masukan merupakan sebuah *range* dari a hingga b. Maka nilai yang digunakan untuk *test case* adalah satu nilai di atas a dan satu nilai di bawah b.

2. Jika masukan merupakan sebuah *number of values*. Maka nilai yang digunakan untuk *test case* adalah nilai maksimum dan minimum serta satu nilai di atas maksimum dan satu nilai di bawah minimum.

Jika data masukan bertipe *range* dari 4 hingga 10. Maka nilai yang didapatkan untuk pengujian adalah nilai maksimum dan minimum serta satu nilai di atas maksimum dan satu nilai di atas minimum. Jadi nilai yang didapatkan adalah 4, 10, 3, dan 11 seperti terlihat pada Gambar 3.6.



Gambar 3.6: Contoh *boundary value analysis* (Sommerville, 2014)

3.1.4 *Automated Testing*

Menjalankan semua *test case* secara manual sangat menyita waktu. Bahkan mereka dapat menyita 50% dari pada waktu pengembangan (Brooks Jr, 1995). *Automated testing* dapat digunakan untuk mempercepat proses pengujian. *Automated testing* adalah proses menjalakan *test case* secara otomatis. *Automated testing* berjalan dengan adanya *trigger*. Baik dalam bentuk *event* maupun *time*. Prosesnya adalah dengan menjalankan *test case* dan membandingkannya dengan *output* yang sudah ditentukan. Hasil perbandingan tersebut akan dilaporkan kepada penguji secara otomatis setelah *automated testing* selesai dijalankan (Nshimiyiman, 2018). *Automated testing* dibangun dengan menggunakan *script* sesuai dengan *test case* yang didapatkan dari tahapan pengujian *white-box* maupun *black-box*. Tampak pada Tabel Kode 3.2 merupakan *script* yang dibangun untuk melakukan pengujian secara otomatis. *Script* tersebut didapatkan dari *test case* untuk pengujian *pseudo-code hallo* pada Tabel Kode 3.1. *Script* ini nantinya dijalankan dengan kakas bantu *pytest* untuk melakukan pengecekan berhasil atau gagalnya suatu pengujian secara otomatis. Sedangkan *trigger* untuk menjalankan *pytest* secara otomatis dilakukan dengan bantuan kakas bantu *travis-ci*.

No	test_hallo
1	`def test_case_1():`
2	`    assert hallo("Budi") == "Hai Budi"`
3	
4	`def test_case_2():`
5	`    assert hallo("Ani") == "Nama Kosong"`

Tabel Kode 3.2: *Contoh script atomated testing*

3.2 NEO-CLI

Neo-cli merupakan sebuah perangkat lunak orkestrasi untuk infrastruktur *cloud*. Sifatanya yang *agnostic* dapat melakukan orkestrasi kepada berbagai *cloud*

platform seperti *OpenStack* maupun *Amazon Web Services*. *Neo-cli* dikembangkan dengan *Python* dan beberapa *depedency* lainnya seperti *GitPython, ncurses* dan lainnya (*neo-cli* 2018). Kedepannya *neo-cli* akan mendukung penggunakan *query languange* bernama *BQL* (Irawan, et al., 2018).

3.3 Teknologi Pengujian Perangkat Lunak

3.3.1 Pytest

Python memiliki *built-in library* untuk melakukan pengujian bernama *python unittest*. Tetapi *python unittest* memiliki banyak kekurangan, salah satunya adalah kurangnya fungsionalitas yang dimiliki untuk melakukan pengujian. Oleh karena itu, penggunaan *pytest* diutamakan *knupp*. *Pytest* dibangun pada tahun 2014 oleh Holger Krekel. Kerkel membangun seluruh *code base pytest* dengan *python* (Krekel, 2018). Pada laporan ini, *pytest* digunakan untuk menjalankan *test case* dan mengecek apakah suatu pengujian berhasil atau gagal. Contoh pengujian yang dilakukan dari Tabel Kode 3.2 akan terlihat seperti Gambar 3.7 jika berhasil dan akan terlihat seperti Gambar 3.8 jika gagal. Pengujian gagal karena *variable nama* diganti nilainya menjadi selain “budi”.

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.5.3, pytest-3.10.1, py-1.7.0, pluggy-0.8.0
rootdir: /home/azzamsya/code-coba-home/hallo-pkl, inifile:
collected 1 item

test_hallo.py .                                                          [100%]

=========================== 1 passed in 0.12 seconds ===========================
```

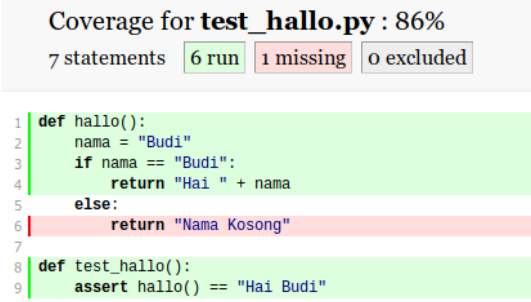
Gambar 3.7: Pengujian berhasil

```
$ pytest
============================= test session starts ==============================
platform linux -- Python 3.5.3, pytest-3.10.1, py-1.7.0, pluggy-0.8.0
rootdir: /home/azzamsya/code-coba-home/hallo-pkl, inifile:
collected 1 item

test_hallo.py F                                                          [100%]

=================================== FAILURES ===================================
_____________________________ test_hallo ______________________________

    def test_hallo():
>       assert hallo() == "Hai Budi"
E       AssertionError: assert 'Nama Kosong' == 'Hai Budi'
E         - Nama Kosong
E         + Hai Budi

test_hallo.py:9: AssertionError
=========================== 1 failed in 0.10 seconds ===========================
```

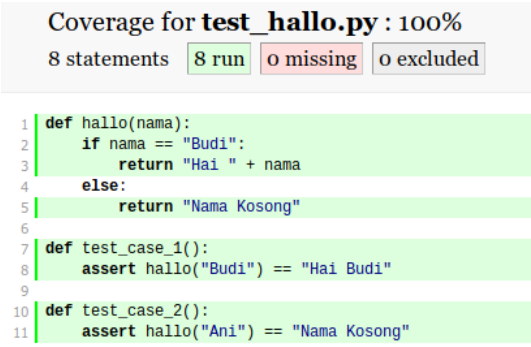
Gambar 3.8: Pengujian gagal

3.3.2 Coverage.py

Coverage.py merupakan kakas bantu yang digunakan untuk menghitung tingkat cakupan (*coverage*) suatu pengujian. *Coverage.py* memiliki fitur untuk mengekspor hasil *coverage* dalam bentuk *Text, HTML, PDF* maupun *XML*. Ned Batchelder menciptakan *coverage.py* pada 2004 dan merilisnya dengan lisensi *Apache 2.0* (Batchelder, 2018). Pada laporan ini *coverage.py* digunakan untuk menghitung seberapa besar cakupan pengujian yang telah dilakukan. Terlihat pada Gambar 3.9 cakupan pengujian hanya mencapai 80%, jika pengujian hanya menjalankan *test case* pertama. Cakupan akan mencapai 100% ketika menjalankan semua *test case* pada *test script* 3.2. Terlihat pada Gambar 3.10 cakupan mencapai 100% tatkala semua *test case* dijalankan.

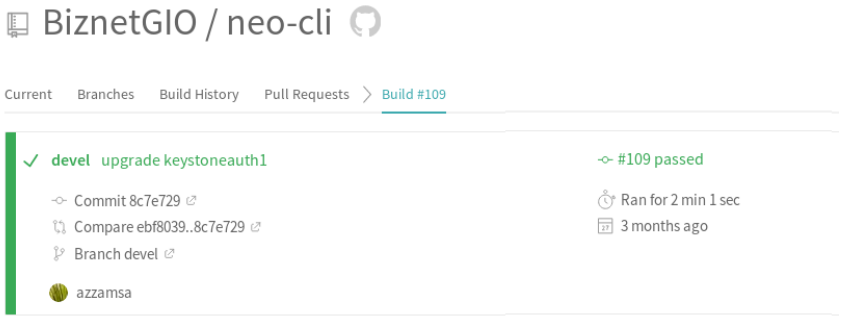


Gambar 3.9: 86% *coverage*



Gambar 3.10: 100% *coverage*

3.3.3 Travis CI



Gambar 3.11: *Travis-ci* melaporan status pengujian

Travis CI merupakan sebuah kakas bantu yang dapat mengatur *trigger* dalam proses *automated test*. *Travis CI* mendukung banyak bahasa pemrograman seperti *Python, C++, Ruby* dan lainnya. Semua pengaturan pada *travis ci* diatur dengan format *YAML* dalam berkas '*.travis.yml*'. Pengecekan juga dapat dilakukan secara otomatis pada *branch* maupun *pull request*. Hasilnya dapat dikirimkan melalui *web notification* maupun surel. *Travis ci* dibangun di Jerman pada 2011 oleh Travis CI, GmbH. *Travis ci* dirilis dengan lisensi *MIT* (*Travis CI Project* 2018). Di dalam laporan ini, *travis-ci* digunakan untuk mengatur *trigger* dalam proses *automated testing*. Pengujian nantinya akan dijalankan secara otomatis oleh *travis-ci* pada *trigger-trigger* tertentu yang telah ditentukan. *Trigger* dapat berbentuk waktu seperti hari atau jam. Maupun berbentuk *event* seperti dijalankan ketika terdapat perubahan *code*. Tampak pada Gambar 3.11 *travis-ci* melaporkan bahwa pengujian yang dijalankan secara otomatis pada *event commit upgrade keystoneauth1* berjalan dengan sukses dan memakan waktu dua menit.

BAB 4 METODOLOGI

Bab ini menjelaskan tentang langkah-langkah yang dilakukan selama pengembangan pengujian. Setelah studi literatur selesai dilakukan, dilaksanakan proses analisis kebutuhan pengujian, perencanaan dan implementasi kasus uji, dan penarikan kesimpulan.

Gambar 4.12: Alur metodologi

4.1 Studi Literatur

Studi Literatur dibutuhkan untuk mendalami teori-teori tentang pengujian lebih dalam. Sumber dari literatur tersebut berasal dari jurnal, buku, situs resmi maupun buku panduan perangkat lunak yang digunakan. Daftar literatur yang didalami terkait dengan:

1. Penelitian-penelitian terkait pengujian
2. Metode-metode pengujian yaitu *white-box* dan *black-box*.
3. Tahapan pengujian *unit* dan *integration*.
4. Teknik pengujian *basis path testing, boundary value analysis* dan *equivalence partitioning*.
5. Teknologi yang digunakan dalam penelitian

4.2 Analisis Kebutuhan Pengujian

Pada proses analis kebutuhan pengujian akan ditentukan tahapan yang akan di lakukan dan bagian yang akan diuji. Pengujian pada *neo-cli* hanya dilakukan

hanya pada tahapan *unit* dan *integration*. Pada fase ini ditentukan juga *method* apa saja yang akan diuji. Penentuan tahapan pengujian dan penetuan *method* yang akan diuji dilakukan oleh pembimbing lapangan dengan beberapa pertimbangan. Di antara pertimbangan tersebut adalah waktu praktik kerja lapangan yang terbatas.

4.3 Perancangan dan Implementasi Kasus Uji

Pada proses ini dilakukan perancangan dan implementasi (eksekusi) kasus uji. Tahapan yang dilakukan diawali dengan pengujian *unit* kemudian pengujian *integration*. Metode pengujian yang digunakan adalah metode pengujian *white-box* dan metode pengujian *black-box*. Teknik yang digunakan selama proses pengujian dengan methode *white-box* adalah teknik *basis path testing*. Teknik *equivalence partitioning* dan *boundary value analysis* adalah teknik yang digunakan selama proses pengujian dengan metode *black-box*. Perancangan kasus uji dalam metode *white-box* diawali dengan pembuatan *flow graph* dari sebuah *pseudocode*, kemudian menghitung *cyclomatic complexity*, menentukan *independent path* dan merancang kasus uji sesuai dengan *idependent path* yang didapatkan. Sedangkan perancangan kasus uji pada metode *black-box* didapatkan dari data pengujian yang dihasilkan dari penggunaan teknik *equivalence partitioning* dan *boundary value analysis*. Implementasi pengujian atau eksekusi pengujian dilakukan secara langsung setelah perancangan pada tahapan *unit* dan *integration* selesai. Hasil dari implementasi tersebut juga langsung dipaparkan. Hasil dari implementasi atau eksekusi pengujian diletakkan pada kolom *result* dan *status* pada tabel kasus uji. Metode pengujian *white-box* dilakukan pada seluruh tahapan *unit* dan *integration*. Sedangkan metode pengujian *black-box* hanya dilakukan pada bagian integrasi sistem yang meminta masukan kepada pengguna, seperti yang terjadi pada *method do_login*. Setelah pengujian secara manual pada tahap *unit* dan *integration* selesai dilakukan, maka dibangun *test script* untuk *automated testing* sehingga proses pengujian selanjutnya dapat dilakukan secara otomatis. *Automated testing* dilakukan menggunakan bantuan kakas bantu *travis-ci* yang konfigurasinya dijelaskan pada subbab "Pengaturan Lingkungan Pengujian untuk *Automated Testing*". Perhitungan cakupan pengujian dilakukan dengan bantuan kakas bantu *coverage.py* yang hasil cakupannya dijelaskan pada subbab "Hasil Cakupan Pengujian".

4.4 Penarikan Kesimpulan

Pengujian ditutup setelah semua *method* pada analisis kebutuhan pengujian selesai diuji. Penarikan kesimpulan dilakukan setelah pengujian ditutup. Ke-

simpulan diambil berdasarkan hasil dari seluruh proses yang dilakukan selama pengujian. Kesalahan-kesalahan akan dicat dan dijadikan saran agar tidak terulang kembali pada proses pengujian selanjutnya. Kesimpulan pengujian nantinya dapat digunakan untuk perbaikan dan peningkatan pengujian-pengujian selanjutnya.

BAB 5 HASIL DAN PEMBAHASAN

5.1 Analisis Kebutuhan Pengujian

Hasil pada tahapan ini berupa daftar *method* yang akan diuji. Daftar tersebut ditentukan oleh pembimbing lapangan dengan beberapa pertimbangan, antra lain terbatasnya waktu praktik kerja lapangan. Daftar *method* yang akan diuji terlihat pada Tabel 5.2

Tabel 5.2: Daftar *method* yang akan diuji

Kode	**Modul**	***Method***	**Deskripsi**
NC1	attach	attach	Sistem menjalankan perintah pada *remote virtual machine*
NC2	create	do_create	Sistem membuat *virtual machine*
NC3	auth	do_login	Sistem mengautentikasi pengguna
NC4	auth	do_logout	Sistem menghapus data autentikasi *user*
NC5	ls	get_stack_list	Sistem menampilkan daftar *stack*
NC6	ls	list_network	Sistem menampilkan daftar *networks*
NC7	vm	do_delete	Sistem menghapus *virtual machine*
NC8	orch	do_delete	Sistem menghapus *stack*
NC9	orch	do_update	Sistem memperbarui *virtual machine*
NC10	cli	show_help	Sistem menampilkan pesan bantuan
NC11	cli	show_version	Sistem dapat menampilkan versi perangkat lunak

5.2 Perancangan dan Implementasi Kasus Uji

5.2.1 Kasus Uji NC1

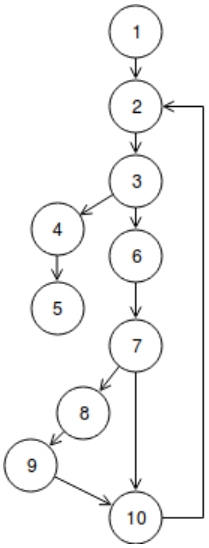
5.2.1.1 Pengujian Unit *ssh_out_stream*

Pengujian *unit* dilakukan pada *method ssh_out_stream* secara terisolasi dan sebagai *stand-alone program*. Maka dilakukan *mocking* pada *method exec_command* dan *exit_status_ready*.

No	ssh_out_stream	
1	`procedure ssh_out_stream()`	
2	`    exec_command(commands)`	`(1)`
3	`    # infinite while`	
4	`    WHILE True:`	`(2)`
5	`        IF exit_status_ready():`	`(3)`
6	`            break`	`(4)`
7	`        ENDIF`	`(5)`
8	`        length = channel`	`(6)`
9	`        IF length > 0:`	`(7)`
10	`            print length`	`(8)`
11	`        ENDIF`	`(9)`
12	`    ENDWHILE`	`(10)`
13	`end`	

Tabel Kode 5.3: *Pseudocode ssh_out_stream*

Flow graph yang dihasilkan dari *pseudocode* ssh_out_stream terlihat pada Gambar 5.13. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 3 *regions* = 3.



Gambar 5.13: *Flow graph* dari *pseudocode ssh_out_stream*

Independent Path

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

Tabel 5.3: Pengujian *unit ssh_out_stream*

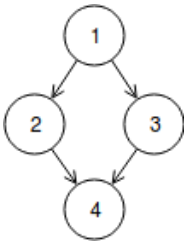
Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Memberikan nilai *true* pada *variable exit_status_ready()*	*ssh_out_stream* berhenti berjalan	*As expe-cted*	Valid
2	Memberikan nilai lebih kecil dari 0 pada *variable length*	*ssh_out_stream* tidak menampilkan nilai *variable length*	*As expe-cted*	Valid
3	Memberikan nilai lebih besar dari 0 pada *variable length*	*ssh_out_stream* menampilkan nilai *variable length*	*As expe-cted*	Valid

5.2.1.2 Pengujian Integrasi *attach*

No	attach
1	`procedure attach`
2	`TRY` (1)
3	`    ssh__out_stream(commands)` (2)
4	`EXCEPT`
5	`    exit` (3)
6	`ENDTRY` (4)
7	`end`

Tabel Kode 5.4: *Pseudocode attach*

Flow graph yang dihasilkan dari *pseudocode* attach terlihat pada Gambar 5.14. *Flow graph* tersebut Memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.14: *Flow graph* dari *pseudocode attach*

Independent Path

1. Jalur 1 : 1 - 2 - 4
2. Jalur 2 : 1 - 3 - 4

Tabel 5.4: Pengujian *integration attach*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Melakukan *raise exception*	Sistem menjalankan perintah pada *remote virtual machine*	*As expected*	Valid
2	Tidak Melakukan *raise exception*	Sistem berhenti	*As expected*	Valid

5.2.1.3 *Test Script* Untuk *Automated Testing*

No	test_attach_command
1	`@pytest.mark.run(order=3)`
2	`def test_attach_command(self):`
3	`    # neo.yml located inside tests dir`
4	`    os.chdir("tests")`
5	
6	`    # wait until vm fully resized`
7	`    vm_status = ''`
8	`    while vm_status != 'ACTIVE':`
9	`        # get 'unittest-vm' id`
10	`        vm_data = vm_lib.get_list()`
11	`        for vm in vm_data:`
12	`            if vm.name == 'unittest-vm':`
13	`                vm_status = vm.status`
14	`                time.sleep(4)`
15	`                print('vm still updating ...')`
16	
17	`    f = StringIO()`
18	`    with redirect_stdout(f):`
19	`        a = Attach({'<args>': ['-c', 'ls -a'],`
20	`                    '<command>': 'attach'}, '-c', 'ls -a')`
21	`        a.execute()`
22	`        out = f.getvalue()`
23	
24	`    os.chdir(os.pardir)`
25	`    assert 'Success' in out`

Tabel Kode 5.5: *Test script* untuk *attach*

5.2.2 Kasus Uji NC2

5.2.2.1 Pengujian Unit *get_heat_client*

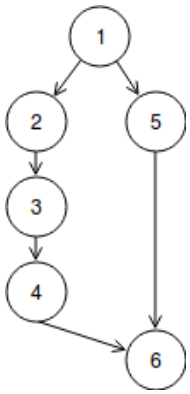
Pengujian *unit* dilakukan pada *method get_heat_client* secara terisolasi dan sebagai *stand-alone program*. Maka dilakukan *mocking* pada *method load_session* dan *heat_client*.

No	get_heat_client
1	`procedure get_heat_client()`
2	`    TRY                                (1)`
3	`        IF not session:                (2)`

```
 4              session = load_session()
 5              heat = heat_client()
 6              return heat                    (3)
 7          ENDIF                              (4)
 8      EXCEPT
 9         show_error_log                      (5)
10      ENDTRY                                 (6)
11  end
```

Tabel Kode 5.6: *Pseudocode get_heat_client*

Flow graph yang dihasilkan dari *pseudocode* get_heat_client terlihat pada Gambar 5.15. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.15: *Flow graph* dari *pseudocode get_heat_client*

Independent Path

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

Tabel 5.5: Pengujian *unit get_heat_client*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Memberikan nilai *raise exception*	Sistem menampilkan *error log* dan berhenti	*As expe-cted*	Valid

2	Memberikan nilai *false* pada *variable session*	Sistem mengambil nilai *session* dan mengembalikan nilai *variable heat*	*As expe-cted*	Valid

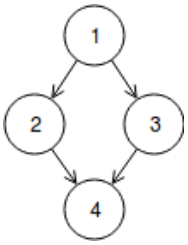
5.2.2.2 Pengujian Unit *yaml_parser*

Pengujian *unit* pada *method yaml_parser* dilakukan secara terisolasi. Maka dilakukan *mocking* pada *method load*

No	yaml_parser
1	`procedure yaml_parser()`
2	`    TRY                              (1)`
3	`        data = load_file             (2)`
4	`        return data`
5	`    EXCEPT`
6	`        show_error_log               (3)`
7	`    ENDTRY`
8	`end`

Tabel Kode 5.7: *Pseudocode yaml_parser*

Flow graph yang dihasilkan dari *pseudocode* yaml_parser terlihat pada Gambar 5.16. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.16: *Flow graph* dari *pseudocode yaml_parser*

Independent Path

1. Jalur 1 : 1 - 2 - 4
2. Jalur 2 : 1 - 3 - 4

Tabel 5.6: Pengujian *unit yaml_parser*

Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Tidak memberikan nilai *raise exception*	Sistem mengembalikan nilai *variable data*	*As expe-cted*	Valid
2	Memberikan nilai *raise exception*	Sistem menampilkan *error log* dan berhenti	*As expe-cted*	Valid

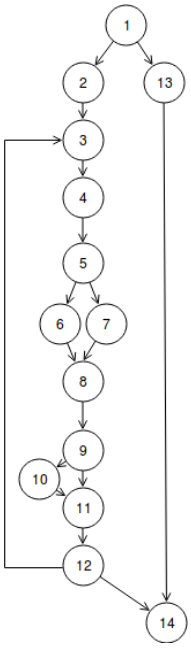
5.2.2.3 Pengujian Integrasi *do_create*

No	do_create
1	`procedure do_create(initialize)`
2	`    TRY` (1)
3	`        heat = get_heat_client()` (2)
4	`        FOR deploy in initialize:` (3)
5	`            deploy_init_file = deploy_dir` (4)
6	`            deploy_file = yaml_parser(deploy_init_file)`
7	`            deploy_template = deploy_file`
8	`            deploy_name = deploy_project`
9	`            IF not deploy_env_file` (5)
10	`              create_stack_without_env` (6)
11	`            ELSE`
12	`              create_stack_with_env` (7)
13	`            ENDIF` (8)
14	`            IF initialize > 0` (9)
15	`              sleep operation` (10)
16	`            ENDIF` (11)
17	`        ENDFOR` (12)
18	`    EXCEPT`
19	`       show_log` (13)
20	`    ENDTRY` (14)
21	`end`

Tabel Kode 5.8: *Pseudocode do_create*

Flow graph yang dihasilkan dari *pseudocode* do_create terlihat pada Gambar 5.17.

Flow graph tersebut memiliki nilai *cyclomatic complexity* V(G) = 5 *regions* = 5.



Gambar 5.17: *Flow graph* dari *pseudocode do_create*

Independent Path

1. Jalur 1 : 1 - 13 - 14
2. Jalur 2 : 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 11 - 12 - 14
3. Jalur 3 : 1 - 2 - 3 - 4 - 5 - 7 - 8 - 9 - 11 - 12 - 14
4. Jalur 4 : 1 - 2 - 3 - 4 - 5 - 7 - 8 - 9 - 10 - 11 - 12 - 14
5. Jalur 5 : 1 - 2 - 3 - 4 - 5 - 7 - 8 - 9 - 10 - 11 - 12 - (for) - 12 - 14

Tabel 5.7: Pengujian *integration do_create*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Memberikan nilai *raise exception*	Sistem menampilkan *error log* dan berhenti	*As expe-cted*	Valid

2	1)Memberikan nilai *false* pada *variable deploy_env_file*. 2)Memberikan nilai lebih kecil dari 0 pada *variable initialize*. 3)Memberikan nilai *false* pada kondisi *deploy in initialize* bernilai *false*	Sistem membuat *virtual machine* tanpa *env* dan tidak menjalankan *sleep operation*	*As expe-cted*	Valid
3	1)Memberikan nilai *true* pada *variable deploy_env_file true*. 2)Memberikan nilai lebih kecil dari 0 pada *variable initialize*. 3)Memberikan nilai *false* pada kondisi *deploy in initialize*	Sistem membuat *virtual machine* dengan *env* dan tidak menjalankan *sleep operation*	*As expe-cted*	Valid
4	1)Memberikan nilai *true* pada *variable deploy_env_file true*. 2)Memberikan nilai lebih besar dari 0 pada *variable initialize*. 3)Memberikan nilai *false* pada kondisi *deploy in initialize*	Sistem membuat *virtual machine* dengan *env* dan menjalankan *sleep operation*	*As expe-cted*	Valid

5	1)Memberikan nilai *true* pada *variable deploy_env_file*. 2)Memberikan nilai lebih besar dari 0 pada *variable initialize*. 3)Memberikan nilai *true* pada kondisi *deploy in initialize*	Sistem membuat *virtual machine* dengan *env* dan menjalankan *sleep operation*	*As expe-cted*	Valid

5.2.2.4 *Test Script* Untuk *Automated Testing*

No	test_do_create

```
1  @pytest.mark.run(order=1)
2  def test_do_create(self):
3      cwd = os.getcwd()
4      deploy_init = orch.initialize(cwd + "/tests/neo.yml")
5      orch.do_create(deploy_init)
6
7      # check deployed vm
8      vm_data = vm_lib.get_list()
9      for vm in vm_data:
10         if vm.name == 'unittest-vm':
11             for network_name, network in
                ↪  vm.networks.items():
12                 assert network_name == 'unittest-network'
```

Tabel Kode 5.9: *Test script do_create*

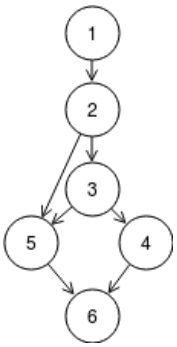
5.2.3 Kasus Uji NC3

5.2.3.1 Pengujian Unit *get_username*

No	get_username
1	`procedure get_username()`
2	`    prompt username_input          (1)`
3	`    IF username_input <= 5 and     (2)`
4	`       username_input >= 255       (3)`
5	`       return username_input       (4)`
6	`    ELSE`
7	`       return false                (5)`
8	`    ENDIF                          (6)`
9	`end`

Tabel Kode 5.10: *Pseudocode get_username()*

Flow graph yang dihasilkan dari *pseudocode* get_username terlihat pada Gambar 5.18. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 3 *regions* = 3.



Gambar 5.18: *Flow graph* dari *pseudocode get_username()*

Independent Path

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

Tabel 5.8: Pengujian *unit get_username*

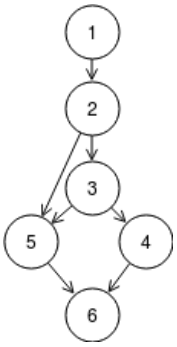
Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Memberikan nilai *false* pada kondisi "username_input <= 5"	Sistem mengembalikan nilai *false*	*As expe-cted*	Valid
2	1)Memberikan nilai *true* pada kondisi "username_input <= 5" 2)Memberikan nilai *false* pada kondisi "username_input >= 255"	Sistem mengembalikan nilai *false*	*As expe-cted*	Valid
3	1)Memberikan nilai *true* pada kondisi "username_input <= 5" 2)Memberikan nilai *true* pada kondisi "username_input >= 255"	Sistem mengembalikan nilai *variable username_input*	*As expe-cted*	Valid

5.2.3.2 Pengujian Unit *get_password*

No	get_password
1	`procedure get_password()`
2	`    prompt password_input          (1)`
3	`    IF password_input <= 5 and     (2)`
4	`       password_input >= 255       (3)`
5	`       return password_input       (4)`
6	`    ELSE`
7	`       return false                (5)`
8	`    ENDIF                          (6)`
9	`end`

Tabel Kode 5.11: *Pseudocode get_password*

Flow graph yang dihasilkan dari *pseudocode* get_password terlihat pada Gambar 5.19. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 3 *regions* = 3.



Gambar 5.19: *Flow graph* dari *pseudocode get_password*

Independent Path

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

Tabel 5.9: Pengujian *unit get_password*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Memberikan nilai *false* pada kondisi “password_input <= 5”	Sistem mengembalikan nilai *false*	*As expe-cted*	Valid
2	1)Memberikan nilai *true* pada kondisi “password_input <= 5” 2)Memberikan nilai *false* pada kondisi “password_input >= 255”	Sistem mengembalikan nilai *false*	*As expe-cted*	Valid

3	1)Memberikan nilai *true* pada kondisi "password_input <= 5" 2)Memberikan nilai *true* pada kondisi "password_input >= 255"	Sistem mengembalikan nilai *variable password_input*	*As expe-cted*	Valid

5.2.3.3 Pengujian Unit *generate_session*

No	generate_session

1
2
3
	`procedure generate_session(password, password)`
	`    sess = session with username and password   (1)`
	`return sess`

Tabel Kode 5.12: *Pseudocode generate_session*

Flow graph yang dihasilkan dari *pseudocode* generate_session terlihat pada Gambar 5.20. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.

Gambar 5.20: *Flow graph* dari *pseudocode generate_session*

Independent Path

1. 1 - 2 - 4
2. 1 - 3 - 4

Tabel 5.10: Pengujian *unit generate_session*

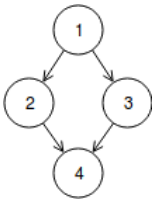
Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Menjalalankan *generate_session*	Sistem mengembalikan nilai dari *variable sess*	*As expected*	Valid

5.2.3.4 Pengujian Integrasi *do_login*

No	do_login
1	`procedure do_login()`
2	`    TRY` (1)
3	`        username = get_username()` (2)
4	`        password = get_password()`
5	`        generate_session(username, password)`
6	`        return true`
7	`    EXCEPT`
8	`        return false` (3)
9	`    ENDTRY` (4)
10	`end`

Tabel Kode 5.13: *Pseudocode do_login*

Flow graph yang dihasilkan dari *pseudocode* do_login terlihat pada Gambar 5.21. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.21: *Flow graph* dari *pseudocode do_login*

Independent Path

1. Jalur 1 : 1 - 2 - 4
2. Jalur 2 : 1 - 3 - 4

Tabel 5.11: Pengujian *integration do_login*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Memberikan nilai *raise exception*	Sistem mengautentikasi pengguna	*As expe-cted*	Valid

2	Tidak memberikan nilai *raise exception*	Sistem mengembalikan nilai *false*	*As expe-cted*	Valid

5.2.3.5 Pengujian *input do_login* menggunakan *equivalence partitioning*

Method do_login meminta masukan kepada pengguna. Masukan tersebut bertipe *range*. Maka dilakukan teknik *equivalence partitioing* untuk memilih *valid input* dan *invalid input* dalam metode pengujian *black-box*.
Equivalent Partitioning test data

Tabel 5.12: *Equivalent Partitioning do_login*

Input	***Valid Class***	***Invalid Class***
Nilai *username*	Segala karakter dengan jumlah batas minimum 5 dan maksimum 255	Segala karakter dengan jumlah kurang dari 5 dan lebih dari 255
Nilai *password*	Segala karakter dengan jumlah batas minimum 5 dan maksimum 255	Segala karakter dengan jumlah kurang dari 5 dan lebih dari 255

Pada teknik *equivalence partitioning argument* nilai *username* dan *password* merupakan tipe *range*. Maka didaptkan satu data *valid* dan dua data *invalid*.

- Nilai *username*
  - Kasus uji *valid* : 'azzamsa'
  - Kasus uji *invalid* : [karakter dengan jumlah 2], [karakter dengan jumlah 260]
- Nilai *password*
  - Kasus uji *valid* : 'foobar'
  - Kasus uji *invalid* : [karakter dengan jumlah 2], [karakter dengan jumlah 260]

Tabel 5.13: Pengujian *input do_login* dengan teknik *equivalence partitioning*

No	Prosedur Uji	*Expected Result*	*Result*	Status
1	1)Memberikan nilai *valid* 'azzamsa' pada *username* 2)Memberikan nilai *valid* 'foobar' pada *password*	Autentikasi pengguna berhasil	*As expe-cted*	Valid
2	1)Memberikan nilai *invalid* [karakter dengan jumlah 2] pada *username* 2)Memberikan nilai *invalid* [karakter dengan jumlah 2] pada *password*	Autentikasi pengguna gagal	*As expe-cted*	Valid
3	1)Memberikan nilai *invalid* [karakter dengan jumlah 260] pada *username* 2)Memberikan nilai *invalid* [karakter dengan jumlah 260] pada *password*	Autentikasi pengguna gagal	*As expe-cted*	Valid

5.2.3.6 Pengujian *Input do_login* Menggunakan *Boundary Value Analysis*

Method do_login meminta masukan kepada pengguna. Masukan tersebut bertipe *range*. Maka dilakukan teknik *boundary value analysis* untuk memilih *input* yang berada pada daerah *boundary* dalam metode pengujian *black-box*.

Boundary Value Analysis test data

Tabel 5.14: *Boundary Value login*

Input	***Boundary Value***
Nilai *username*	Jumlah karakter satu tingkat dibawah batas minimum dan satu tingkat diatas maksimum
Nilai *password*	Jumlah karakter satu tingkat dibawah batas minimum dan satu tingkat diatas maksimum

Pada teknik *boundary value analysis*. Nilai *username* dan *password* merupakan tipe *range*. Maka kasus uji didapatkan dari satu nilai dibawah batas minimum dan satu nilai diatas batas maksimum.

- Nilai *boundary* untuk *username*
  - Satu nilai dibawah minimum : [karakter dengan jumlah 4]
  - Satu nilai diatas maksimum : [karakter dengan jumlah 6]
- Nilai *boundary* untuk *password*
  - Satu nilai dibawah minimum : [karakter dengan jumlah 254]
  - Satu nilai diatas maksimum : [karakter dengan jumlah 256]

Tabel 5.15: Pengujian *input do_login* dengan teknik *boundary value analysis*

No	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	1)Memberikan nilai *invalid* [karakter dengan jumlah 4] pada *username* 2)Memberikan nilai *invalid* [karakter dengan jumlah 254] pada *password*	Autentikasi pengguna gagal	*As expected*	Valid

2	1)Memberikan nilai *invalid* [karakter dengan jumlah 6] pada *username* 2)Memberikan nilai *invalid* [karakter dengan jumlah 256] pada *password*	Autentikasi pengguna gagal	*As expe-cted*	Valid

5.2.3.7 *Test Script* Untuk *Automated Testing*

No	test_do_login

1
2
3
4
5
6
7
 	
8
 	
9
10
11
	`@pytest.mark.run(order=0)`
	`def test_do_login(self, monkeypatch):`
	`  login.load_env_file()`
	`  username = os.environ.get('OS_USERNAME')`
	`  passwd = os.environ.get('OS_PASSWORD')`
	`  # give value to input() prompt`
	`  monkeypatch.setattr('builtins.input', lambda x:`
	`  ↪ username)`
	`  monkeypatch.setattr('getpass.getpass', lambda x:`
	`  ↪ passwd)`
	`  # return True is login succeed`
	`  output = login.do_login()`
	`  assert output == True`

Tabel Kode 5.14: *Test script* utnuk *do_login*

5.2.4 Kasus Uji NC4

5.2.4.1 Pengujian Unit *check_session*

No	check_session

1
2
3
	`procedure check_session()`
	`  return is_session_pkl_exist (1)`
	`end`

Tabel Kode 5.15: *Pseudocode check_session*

Flow graph yang dihasilkan dari *pseudocode* check_session terlihat pada Gambar 5.22. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.22: *Flow graph* dari *pseudocode check_session*

Independent Path

1. Jalur 1 : 1

Tabel 5.16: Pengujian *unit check_session*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Menjalankan *check_session*	Sistem akan mengembalikan nilai dari fungsi *is_session_pkl_exist*	*As expe-cted*	Valid

5.2.4.2 Pengujian integrasi *do_logout*

No	do_logout

1
2
3
4
5
	`procedure do_logout()`
	`    IF check_session()`
	`        remove login_data     (1)`
	`    ENDIF`
	`end`

Tabel Kode 5.16: *Pseudocode do_logout*

Flow graph yang dihasilkan dari *pseudocode* do_logout terlihat pada Gambar 5.23. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.23: *Flow graph* dari *pseudocode do_logout*

Independent Path

1. Jalur 1 : 1

Tabel 5.17: Pengujian *integration do_logout*

Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Memberikan nilai *true* pada nilai *return method check_session*	Sistem menghapus data autentikasi	*As expe-cted*	Valid

5.2.4.3 *Test Script* Untuk *Automated Testing*

No	test_do_logout

1
2
3
4
5
6
	`@pytest.mark.run(order=-1)`
	`def test_do_logout(self):`
	`    login.do_logout()`
	`    # session removed if logout succeed`
	`    output = login.check_session()`
	`    assert output == False`

Tabel Kode 5.17: *test script logout*

5.2.5 Kasus Uji NC5

5.2.5.1 Pengujian Unit *get_heat_client*

Pengujian *unit* pada *method get_heat_client* telah dilakukan sebelumnya pada Tabel 5.15

5.2.5.2 Pengujian Integrasi *get_stack_list*

No	get_stack_list

1
2
3
4
5
	`procedure get_stack_list()`
	`    h_client = get_heat_client()`
	`    stack_list = h_client CALL heat_stack_value  (1)`
	`    return stack_list`
	`end`

Tabel Kode 5.18: *Pseudocode get_list*

Flow graph yang dihasilkan dari *pseudocode* get_list terlihat pada Gambar 5.24. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.24: *Flow graph* dari *pseudocode get_list*

Independent Path

1. Jalur 1 : 1

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Menjalankan *get_list*	Sistem akan melempar nilai *variable data_stack*	*As expe-cted*	Valid

Tabel 5.18: Pengujian *unit get_list*

5.2.5.3 *Test Script* Untuk *Automated Testing*

No	test_ls_stack

1
2
3
4
	`def test_ls_stack(self):`
	`  with pytest.raises(SystemExit):`
	`  a = Ls({'<command>': 'ls'}, 'stack')`
	`  a.execute()`

Tabel Kode 5.19: *Test script get_stack_list*

5.2.6 Kasus Uji NC6

5.2.6.1 Pengujian Unit *get_neutron_client*

Pengujian unit dilakukan pada *method get_neutron_client* secara terisolasi dan sebagai *stand-alone program*. Maka dilakukan *mocking* pada *method load_dumped_session* dan *neutron_client*.

No	get_neutron_client

1
2
3
4
5
6
7
	`procedure get_neutron_client()`
	`    IF not session:`
	`        session = load_dumped_session()    (1)`
	`        neutron = neutron_client()`
	`        return neutron`
	`    ENDIF`
	`end`

Tabel Kode 5.20: *Pseudocode get_neutron_client*

Flow graph yang dihasilkan dari *pseudocode* get_neutron_client terlihat pada Gambar 5.25. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *regions* = 1.

Gambar 5.25: *Flow graph* dari *pseudocode get_neutron_client*

Independent Path

1. Jalur 1 : 1

Tabel 5.19: Pengujian *unit get_neutron_client*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Memberikan nilai *true* pada *variable session*	Sistem mengembalikan nilai *variable neutron*	*As expected*	Valid

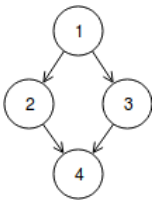
5.2.6.2 Pengujian Unit *get_network_list*

Pengujian unit dilakukan pada *method get_list* secara terisolasi dan sebagai *stand-alone program*. Maka dilakukan *mocking* pada *method list_networks*.

No	get_network_list
1	`procedure get_network_list()`
2	`    TRY (1)`
3	`        n_client = get_neutron_client()`
4	`        networks = n_client CALL list_networks() (2)`
5	`        return networks`
6	`    EXCEPT`
7	`        show_error_log (3)`
8	`    ENDTRY (4)`
9	`end`

Tabel Kode 5.21: *Pseudocode get_network_list*

Flow graph yang dihasilkan dari *pseudocode* get_network_list terlihat pada Gambar 5.26. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.26: *Flow graph* dari *pseudocode get_network_list*

Independent Path 5.26

1. Jalur 1 : 1 - 2 - 4
2. Jalur 2 : 1 - 3 - 4

Tabel 5.20: Pengujian *unit get_network_list*

Ja-lur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Melakukan *raise exception*	Sistem mengembalikan nilai *variable network*	*As expe-cted*	Valid

2	Tidak melakukan *raise exception*	Sistem menampilkan *error log* dan berhenti	*As expe-cted*	Valid

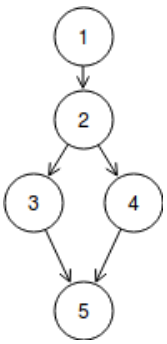
5.2.6.3 Pengujian Integrasi *list_network*

Pengujian *unit* dilakukan pada *method list_network* secara terisolasi dan sebagai *stand-alone program*. Maka dilakukan *mocking* pada *method get_network_list*.

No	list_network
1	`procedure list_network()`
2	`    network_list = get_network_list()  (1)`
3	`    IF network_list == 0  (2)`
4	`        print `no data' (3)`
5	`    ELSE`
6	`        print network_list  (4)`
7	`    ENDIF  (5)`
8	`end`

Tabel Kode 5.22: *Pseudocode list_network*

Flow graph yang dihasilkan dari *pseudocode* list_network terlihat pada Gambar 5.27. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.27: *Flow graph* dari *pseudocode ist_network*

Independent Path

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

Tabel 5.21: Pengujian *integration list_network*

Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Memberikan nilai 0 pada *variable network_list*	Sistem akan menampilan pesan 'no data'	*As expe-cted*	Valid
2	Memberikan nilai 1 pada *variable network_list*	Sistem akan menampilkan daftar *network*	*As expe-cted*	Valid

5.2.6.4 *Test Script* Untuk *Automated Testing*

No	test_ls_net

1
2
3
4
	`def test_ls_net(self):`
	`  with pytest.raises(SystemExit):`
	`  a = Ls({'<command>': 'ls'}, 'network')`
	`  a.execute()`

Tabel Kode 5.23: *Test script ls network*

5.2.7 Kasus Uji NC7

5.2.7.1 Pengujian Unit *get_nova_client*

Pengujian *unit* dilakukan pada *method get_nova_client* secara terisolasi dan sebagai *stand-alone program*. Maka dilakukan *mocking* pada *method nova_client*.

No	get_nova_client

1
2
3
4
5
6
	`procedure get_nova_client(session)`
	`  IF not session:`
	`  compute = nova_client()  (1)`
	`  return compute`
	`  ENDIF`
	`end`

Tabel Kode 5.24: *Pseudocode get_nova_client*

Flow graph yang dihasilkan dari *pseudocode* get_nova_client terlihat pada Gambar 5.28. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.28: *Flow graph* dari *pseudocode get_nova_client*

Independent Path

1. Jalur 1 : 1

Tabel 5.22: Pengujian *unit get_nova_client*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Menjalankan *get_nova_client*	Sistem mengembalikan nilai *variable compute*	*As expe-cted*	Valid

5.2.7.2 Pengujian Integrasi *do_delete*

No	do_delete

1
2
3
4
	`procedure do_delete(instance_id, session)  (1)`
	`  initialize compute = get_nova_client()`
	`  compute_delete(instnce_id)`
	`end`

Tabel Kode 5.25: *Pseudocode do_delete*

Flow graph yang dihasilkan dari *pseudocode* do_delete terlihat pada Gambar 5.29. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.29: *Flow graph* dari *pseudocode do_delete*

Independent Path

1. Jalur 1 : 1

Tabel 5.23: Pengujian *integration do_delete*

Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Menjalankan *do_delete*	Sistem menghapus *virtual machine*	*As expe-cted*	Valid

5.2.7.3 *Test Script* Untuk *Automated Testing*

No	test_do_delete_vm

```
1   @pytest.mark.run(order=-2)
2       def test_do_delete_vm(self):
3           # wait until successfully created
4           vm_status = ''
5           while vm_status != 'ACTIVE':
6               # get 'unittest-vm' id
7               vm_data = vm_lib.get_list()
8               for vm in vm_data:
9                   if vm.name == 'unittest-vm':
10                      vm_status = vm.status
11                      instance_id = vm.id
12                      vm_name = vm.name
13              time.sleep(2)
14              print('waiting until vm activated ...')
15
16          vm_lib.do_delete(instance_id)
17          print(vm_name + ' with id ' + instance_id +
            ↪  ' deleted')
18
19          # wait until successfully deleted
20          while 'unittest' in vm_data:
21              vm_data = vm_lib.get_list()
22              time.sleep(2)
23              print('waiting until vm fully deleted ...')
24
25          assert 'unittest-vm' not in vm_data
```

Tabel Kode 5.26: *Test script do_delete*

5.2.8 Kasus Uji NC8

5.2.8.1 Pengujian Unit *get_heat_client*

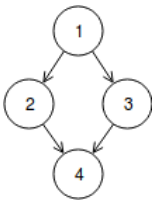
Pengujian *unit* untuk *get_heat_client* telah dilakukan sebelumnya pada Tabel 5.5.

5.2.8.2 Pengujian Integrasi *do_delete*

No	do_delete
1	`procudure do_delete(stack_name)`
2	`    TRY` (1)
3	`        delete stack_name` (2)
4	`        return true`
5	`    EXCEPT`
6	`        return false` (3)
7	`    ENDTRY` (4)
8	`end`

Tabel Kode 5.27: *Pseudocode do_delete*

Flow graph yang dihasilkan dari *pseudocode* do_delete terlihat pada Gambar 5.30. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 2 *regions* = 2.



Gambar 5.30: *Flow graph* dari *pseudocode do_delete*

Independent Path

1. Jalur 1 : 1 - 2 - 4
2. Jalur 2 : 1 - 3 - 4

Tabel 5.24: Pengujian *integration do_delete*

Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Tidak melakukan *raise exception*	*Stack* berhasil dihapus *true*	*As expe-cted*	Valid
2	Melakukan *raise exception*	*Stack* tidak terhapus	*As expe-cted*	Valid

5.2.8.3 *Test Script* Untuk *Automated Testing*

No	test_do_delete_stack

```
1  @pytest.mark.run(order=-2)
2  def test_do_delete_stack(self):
3      # wait until successfully created
4      vm_status = ''
5      while vm_status != 'ACTIVE':
6          # get 'unittest-vm' id
7          vm_data = vm_lib.get_list()
8          for vm in vm_data:
9              if vm.name == 'unittest-vm':
10                 vm_status = vm.status
11                 instance_id = vm.id
12                 vm_name = vm.name
13         time.sleep(2)
14         print('waiting until vm activated ...')
15
16     orch.do_delete('unittest-network')
17     orch.do_delete('unittest-key')
18     print(vm_name + ' with id ' + instance_id +
       ↪  ' deleted')
19
20     # wait until successfully deleted
21     while 'unittest' in vm_data:
22         vm_data = vm_lib.get_list()
23         time.sleep(2)
24         print('waiting until vm fully deleted ...')
```

```
25
26      assert 'unittest-vm' not in vm_data
```

Tabel Kode 5.28: *Test script do_delete*

5.2.9 Kasus Uji NC9

5.2.9.1 Pengujian Unit *get_heat_client*

Pengujian *unit* untuk *method get_heat_client* telah dilakukan sebelumnya pada Tabel 5.5

5.2.9.2 Pengujian Unit *yaml_parser*

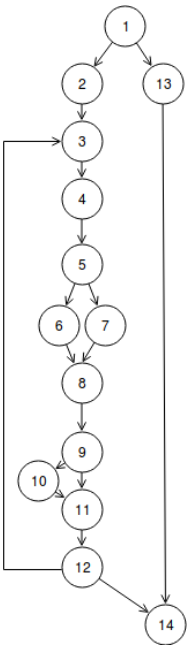
Pengujian *unit* untuk *method yaml_parser* telah dilakukan sebelumnya pada Tabel 5.6.

5.2.9.3 Pengujian Integrasi *do_update*

No	do_update
1	`procedure do_update(initialize)`
2	`    TRY` (1)
3	`        initialize heat = get_heat_client()` (2)
4	`        FOR deploy in initialize:` (3)
5	`            deploy_init_file = deploy_dir` (4)
6	`            deploy_file = yaml_parser(deploy_init_file)`
7	`            deploy_template = deploy_file`
8	`            deploy_name = deploy_project`
9	`            IF not deploy_env_file` (5)
10	`              update_stack_without_env` (6)
11	`            ELSE`
12	`              update_stack_with_env` (7)
13	`            ENDIF` (8)
14	`            IF initialize > 0` (9)
15	`              sleep operation` (10)
16	`            ENDIF` (11)
17	`        ENDFOR` (12)
18	`    EXCEPT`
19	`       show_error_log` (13)
20	`    ENDTRY` (14)
21	`end`

Tabel Kode 5.29: *Pseudocode do_update*

Flow graph yang dihasilkan dari *pseudocode* do_update terlihat pada Gambar 5.31. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 5 *regions* = 5.



Gambar 5.31: *Flow graph* dari *pseudocode do_update*

Independent Path

1. Jalur 1 : 1 - 13 - 14
2. Jalur 2 : 1 - 2 - 3 - 4 - 5 - 6 - 8 - 9 - 11 - 12 - 14
3. Jalur 3 : 1 - 2 - 3 - 4 - 5 - 7 - 8 - 9 - 11 - 12 - 14
4. Jalur 4 : 1 - 2 - 3 - 4 - 5 - 7 - 8 - 9 - 10 - 11 - 12 - 14
5. Jalur 5 : 1 - 2 - 3 - 4 - 5 - 7 - 8 - 9 - 10 - 11 - 12 - (for) - 12 - 14

Tabel 5.25: Pengujian *integration do_update*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Melakukan*raise exception*	Sistem menampilkan *error log* dan berhenti	*As expe-cted*	Valid

2	1)Memberikan nilai *false* pada *variable deploy_env_file* 2)Memberikan nilai lebih kecil dari 0 pada *variable initialize* 3)Memberikan nilai *false* pada kondisi *deploy in initialize*	Sistem memperbarui *virtual machine* tanpa *env* dan tidak menjalankan *sleep operation*	*As expe-cted*	Valid
3	1)Memberikan nilai *true* pada *variable deploy_env_file* 2)Memberikan nilai lebih kecil dari 0 pada *variable initialize* 3)Memberikan nilai *false* pada kondisi *deploy in initialize*	Sistem memperbarui *virtual machine* dengan *env* dan tidak menjalankan *sleep operation*	*As expe-cted*	Valid
4	1)Memberikan nilai *true* pada *variable deploy_env_file* 2)Memberikan nilai lebih besar dari 0 pada *variable initialize* 3)Memberikan nilai *false* pada kondisi *deploy in initialize*	Sistem memperbarui *virtual machine* dengan *env* dan menjalankan *sleep operation*	*As expe-cted*	Valid

5	1)Memberikan nilai *true* pada *variable deploy_env_file* 2)Memberikan nilai lebih besar dari 0 pada *variable initialize* 3)Memberikan nilai *true* pada kondisi *deploy in initialize*	Sistem memperbarui *virtual machine* dengan *env* dan menjalankan *sleep operation*	*As expe-cted*	Valid

5.2.9.4 *Test Script* Untuk *Automated Testing*

No	test_do_update

```
1   @pytest.mark.run(order=2)
2   def test_do_update(self):
3       cwd = os.getcwd()
4
5       # wait until last vm successfully created
6       vm_status = ''
7       while vm_status != 'ACTIVE':
8           # get 'unittest-vm' id
9           vm_data = vm_lib.get_list()
10          for vm in vm_data:
11              if vm.name == 'unittest-vm':
12                  vm_status = vm.status
13                  vm_name = vm.name
14          time.sleep(2)
15          print('waiting until vm activated ...')
16
17      a = Update({'<args>': ['-f', 'tests/neo2.yml'],
18                  '<command>': 'update'}, '-f',
                    ↪  'tests/neo2.yml')
19      a.execute()
20      print(vm_name + ' updated')
21
22      # wait until successfully updated
23      updated_status = None
```

```
24      while updated_status == None:
25          vm_data = orch.get_list()
26          for vm in vm_data:
27              if "unittest-vm" in vm:
28                  updated_status = vm[4]
29          time.sleep(2)
30          print('waiting until vm fully updated ...')
31
32      assert updated_status != None
```

Tabel Kode 5.30: *Test script do_update*

5.2.10 Kasus Uji NC10

5.2.10.1 Pengujian Integrasi *show_help*

No	show_help

1
2
3
	`procudure show_help()`
	`    print help       (1)`
	`end`

Tabel Kode 5.31: *Pseudocode show_help*

Flow graph yang dihasilkan dari *pseudocode* show_help terlihat pada Gambar 5.32. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.32: *Flow graph* dari *pseudocode show_help*

Independent Path

1. Jalur 1 : 1

Tabel 5.26: Pengujian *integration show_help*

Jalur	**Prosedur Uji**	***Expected Result***	***Result***	**Status**
1	Menjalankan *show_help*	Sistem menampilkan pesan bantuan	*As expected*	Valid

5.2.10.2 *Test Script* Untuk *Automated Testing*

No	test_returns_usage_information
1	`def test_returns_usage_information(self):`
2	`        # take output from 'neo -h'. Then take the first word`
3	`        output = Popen(['neo', '-h'], stdout=PIPE).communicate()[0]`
4	`        assert 'Usage:' in str(output)`
5	
6	`        output = Popen(['neo', '--help'], stdout=PIPE).communicate()[0]`
7	`        assert 'Usage:' in str(output)`

Tabel Kode 5.32: *Test script show_help*

5.2.11 Kasus Uji NC11

5.2.11.1 Pengujian Integrasi *show_version*

No	show_version
1	`procudure show_version()`
2	`    print version          (1)`
3	`end`

Tabel Kode 5.33: *Pseudocode show_version*

Flow graph yang dihasilkan dari *pseudocode* show_version terlihat pada Gambar 5.33. *Flow graph* tersebut memiliki nilai *cyclomatic complexity* V(G) = 1 *region* = 1.

Gambar 5.33: *Flow graph* dari *pseudocode show_version*

Independent Path

1. Jalur 1 : 1

Tabel 5.27: Pengujian *integration show_version*

Jalur	Prosedur Uji	*Expected Result*	*Result*	Status
1	Menjalankan *show_version*	Sistem menampilkan versi perangkat lunak	*As expe-cted*	Valid

5.2.11.2 *Test Script* Untuk *Automated Testing*

No	test_returns_version_information
1	`def test_returns_version_information(self):`
2	`    output = Popen(['neo', '--version'], stdout=PIPE).communicate()[0]`
3	`    assert "{}\n".format(VERSION).encode() == output`

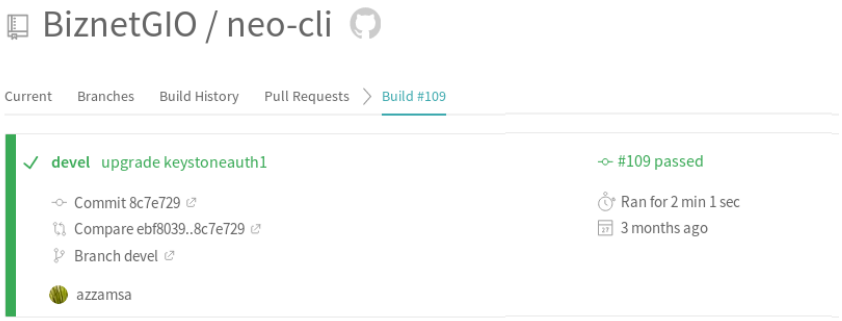
Tabel Kode 5.34: *Test script show_version*

5.2.12 Pengaturan Lingkungan Pengujian untuk *Automated Testing*

Pengaturan lingkuan pengujian untuk *automated testing* diatur dengan konfigurasi seperti terlihat pada Tabel Kode 5.35. *Automated testing* dilakukan dengan bantuan perangkat lunak *travis-ci*. Terlihat pada Gambar 5.34 pengujian otomatis yang dilakukan oleh *travis-ci* berhasil.

No	config.yml
1	`language: python`
2	`python:`
3	`- '3.6'`
4	`install:`
5	`- pip install -r requirements.txt`
6	`- pip install -e .`
7	`- pip install coverage pytest pytest-cov pytest-ordering testfixtures`
8	`- pip freeze --local`
9	`script:`
10	`- pytest --cov=neo -vv -s`
11	`before_install:`
12	`- mv tests/.test.env $HOME/.neo.env`
13	`- rm -rfv tests/.deploy`

Tabel Kode 5.35: *Test Environment Configuration*



Gambar 5.34: *Travis-ci* melaporan status pengujian

5.2.13 Hasil Cakupan Pengujian

Perhitugan cakupan pengujian menggunakan kakas bantu *coverage.py*. Terlihat presentase cakupan dalam Gambar 5.35. Status hasil pengujian juga dapat dilihat pada gambar 5.35. *PASSED* menandakan pengujian yang dilakukan berhasil dan *FAILED* menyatakan pengujian yang dilakukan gagal.

```
tests/test_auth.py::TestAuth::test_do_login PASSED                    [ 10%]
tests/test_create.py::TestCreate::test_do_create PASSED               [ 20%]
tests/test_update.py::TestUpdate::test_do_update PASSED               [ 30%]
tests/test_attach.py::TestAttach::test_attach_vm PASSED               [ 40%]
tests/test_help.py::TestHelp::test_returns_usage_information PASSED   [ 50%]
tests/test_help.py::TestVersion::test_returns_version_information PASSED [ 60%]
tests/test_ls.py::TestLs::test_ls_stack PASSED                        [ 70%]
tests/test_ls.py::TestLs::test_ls_net PASSED                          [ 80%]
tests/test_remove.py::TestRemove::test_do_delete_vm PASSED            [ 90%]
tests/test_auth.py::TestAuth::test_do_logout PASSED                   [100%]
```

Gambar 5.35: Hasil cakupan pengujian

BAB 6 PENUTUP

6.1 Kesimpulan

Berdasarkan proses pembangunan pengujian yang di dalamnya terdapat tahapan seperti analisis kebutuhan pengujian, dan perancangan dan implementasi kasus uji. Dapat disimpulkan bahwa:

1. Nilai cakupan pengujian tidak dipatok dengan nilai umum 80%-85%. Hal ini disebabkan karena nilai tersebut tidak akan mungkin dicapai dengan keberadaan modul-modul yang tidak mendukung proses pengujian.

2. Beberapa modul dengan *cyclomatic complexity* yang tinggi menyulitkan proses pengujian. Sudah selayaknya dilakukan *refactoring* pada modul-modul tersebut.

3. Pengujian harus dilakukan secara terisolasi. Oleh karena itu, dilakukan *mock* pada bagian lain di luar sesi pengujian tersebut.

4. *Test script* untuk *automated testing* dapat melakukan pengujian secara otomatis. Hal ini dapat mempercepat proses pengujian.

5. Penggunaan teknik *basis path testing* memberikan batas *path* yang akan diuji. *Path* yang terbatas memberikan tolak ukur selesainya pengujian. Berbeda dengan *rigorous testing* yang berusaha menguji banyak *path* dan tidak memiliki tolak ukur jelas tentang ukuran selesainya sebuah pengujian.

6. Penggunaan teknik *equivalence partitioning* dan *boundary value analysis* memberikan batasan masukan untuk pengujian. Dengan teknik tersebut pengujian dapat dilakukan dengan data masukan yang sangkil (efisien). Tidak dengan data masukan yang sembarang.

6.2 Saran

Banyak modul yang memiliki nilai *cyclomatic complexity* yang tinggi sehingga membutuhkan *refactoring*. Modul-modul lain yang berhubungan dengan dunia luar (*outside world*) selayaknya diproses dengan *mock* dan tidak menggunakan *resource* yang sebenarnya. Penggunaan *resource* asli (tanpa *mock*) hanya akan menyulitkan proses pengujian. Oleh karena itu, diharapkan agar semua modul yang berhubungan dengan dunia luar untuk diuji menggunakan *mock*.

6.3 Keberlanjutan

Proses pengujian ini akan dilanjutkan dengan membuat *mock* pada modul-modul lain yang berhubungan dengan dunia luar (*outside world*). *Refactoring* juga akan dilakukan pada modul-modul dengan *cyclomatic complexity* yang tinggi. *Automated testing* nantinya hanya akan dilakukan pada pengujian *unit* dan *integration*. Hal-hal yang memudahkan proses pengujian juga akan ditambahkan, seperti proses *code review* menggunakan *gerrit*. Tata pelaksanaan *pull request* akan diketatkan pada bagian *code style guide*, dalam hal ini yaitu penggunaan *PEP8*.

DAFTAR PUSTAKA

Batchelder, N., 2018. *Coverage.py Documentation*. Available at: <https://coverage.readthedocs.io/en/coverage-4.5.1a/> [Accessed Sept. 23, 2018].

BiznetGioNusantara, 2018. *Tentang Kami*. Available at: <https://www.biznetgio.com/about> [Accessed Nov. 8, 2018].

Brooks Jr, F. P., 1995. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E*. Pearson Education India.

Burnstein, I., 2006. *Practical software testing: a process-oriented approach*. Springer Science & Business Media.

Dijkstra, E. W., 1970. *Notes on structured programming*.

Fowler, M., 2018a. *Mocks Aren't Stubs*. Available at: <https://martinfowler.com/articles/mocksArentStubs.html> [Accessed Sept. 23, 2018].

— 2018b. *UnitTest*. Available at: <https://martinfowler.com/bliki/UnitTest.html> [Accessed Sept. 23, 2018].

Gregory, L., 2007. *Path Testing*.

Irawan, E. T., Bappedyanto, D., Hariyadi, D., & Aziz, A. S., 2018. BIZNET QUERY LANGUAGE PADA INFRASTRUCTURE AS CODE. *TEKNOMATIKA*, 11(1).

Krekel, H., 2018. *Pytest Documentation*. Available at: <https://docs.pytest.org/en/latest/contents.html> [Accessed Sept. 23, 2018].

Myers, G. J., Sandler, C., & Badgett, T., 2011. *The art of software testing*. John Wiley & Sons.

neo-cli 2018. Available at: <https://github.com/BiznetGIO/neo-cli> [Accessed Sept. 23, 2018].

Nshimiyiman, M., 2018. *Travis-ci - Send notification with custom message*. Available at: <http://www.marcellin.me/blog/travis-ci-notification-custom-message/> [Accessed Sept. 23, 2018].

Osherove, R., 2015. *The art of unit testing*. MITP-Verlags GmbH & Co. KG.

Presman, R. S., 2010. *Software Enginering a Practioner's Aproach*. New York: McGraw-Hill Companies Inc.

Savenkov, R., 2008. *How to become a software tester*. Roman Savenkov.

Sommerville, I., 2014. *Software Engineering*. Pearson Education.

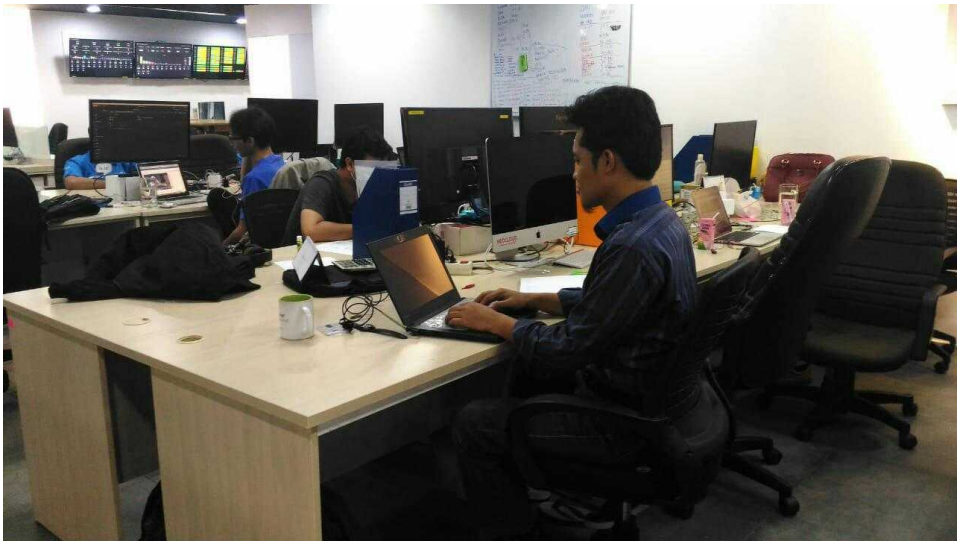
Travis CI Project 2018. Available at: <https://docs.travis-ci.com/> [Accessed Sept. 23, 2018].

Whittaker, J. A., Arbon, J., & Carollo, J., 2012. *How Google tests software*. Addison-Wesley.

Wong, W. E., Debroy, V., & Restrepo, A., 2009. The role of software in recent catastrophic accidents. *Ieee reliability society 2009 annual technology report*, 59(3).

Lampiran Foto Kegiatan

Gambar 7.0.1: Dokumentasi bersama pembimbing lapangan Bapak Eka Tresna Irawan



Gambar 7.0.2: Dokumentasi saat di ruangan kerja Biznet Gio Nusantara

Gambar 7.0.3: Dokumentasi saat di ruangan rapat Biznet Gio Nusantara