DSL UNTUK PENGUJIAN KEAMANAN BERBASIS BDD

Seminar Tugas Akhir II

Disusun sebagai syarat kelulusan tingkat sarjana

Oleh

Ridho Pratama

NIM 13516032

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

Agustus 2020

DSL UNTUK PENGUJIAN KEAMANAN BERBASIS BDD

Seminar TA2

~~**Laporan Tugas Akhir II**~~

Oleh

Ridho Pratama

NIM 13516032

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir di Bandung, pada tanggal 14 Agustus 2020.

Pembimbing

Yudistira Dwi Wardhana Asnar ST, Ph.D.

NIP. 19800827 201504 1 002

DSL UNTUK PENGUJIAN KEAMANAN BERBASIS BDD

Seminar Tugas Akhir II

Oleh

Ridho Pratama

NIM 13516032

Program Studi Teknik Informatika

Sekolah Teknik Elektro dan Informatika

Institut Teknologi Bandung

Telah disetujui dan disahkan sebagai Laporan Tugas Akhir di Bandung, pada tanggal 14 Agustus 2020.

Pembimbing

Yudistira Dwi Wardhana Asnar ST, Ph.D.

NIP. 19800827 201504 1 002

LEMBAR PERNYATAAN

Dengan ini saya menyatakan bahwa:

1. Pengerjaan dan penulisan Laporan Tugas Akhir ini dilakukan tanpa menggunakan bantuan yang tidak dibenarkan.
2. Segala bentuk kutipan dan acuan terhadap tulisan orang lain yang digunakan di dalam penyusunan laporan tugas akhir ini telah dituliskan dengan baik dan benar.
3. Laporan Tugas Akhir ini belum pernah diajukan pada program pendidikan di perguruan tinggi mana pun.

Jika terbukti melanggar hal-hal di atas, saya bersedia dikenakan sanksi sesuai dengan Peraturan Akademik dan Kemahasiswaan Institut Teknologi Bandung bagian Penegakan Norma Akademik dan Kemahasiswaan khususnya Pasal 2.1 dan Pasal 2.2.

Bandung, 14 Agustus 2020
Ridho Pratama
NIM 13516032

KATA PENGANTAR

Gunakan bagian ini untuk memberikan ucapan terima kasih kepada semua pihak yang secara langsung atau tidak langsung membantu penyelesaian tugas akhir, termasuk pemberi beasiswa jika ada. Utamakan untuk memberikan ucapan terima kasih kepada tim pembimbing tugas akhir dan staf pengajar atau pihak program studi, bahkan sebelum mengucapkan terima kasih kepada keluarga. Ucapan terima kasih sebaiknya bukan hanya menyebutkan nama orang saja, tetapi juga memberikan penjelasan bagaimana bentuk bantuan/dukungan yang diberikan. Gunakan bahasa yang baik dan sopan serta memberikan kesan yang enak untuk dibaca. Sebagai contoh: "Tidak lupa saya ucapkan terima kasih kepada teman dekat saya, Tito, yang sejak satu tahun terakhir ini selalu memberikan semangat dan mengingatkan saya apabila lengah dalam mengerjakan Tugas Akhir ini. Tito juga banyak membantu mengoreksi format dan layout tulisan. Apresiasi saya sampaikan kepada pemberi beasiswa, Yayasan Beasiswa, yang telah memberikan bantuan dana kuliah dan biaya hidup selama dua tahun. Bantuan dana tersebut sangat membantu saya untuk dapat lebih fokus dalam menyelesaikan pendidikan saya. ....". Ucapan permintaan maaf karena kekurangsempurnaan hasil Tugas Akhir tidak perlu ditulis.

Daftar Isi

Daftar Gambar

BAB I

PENDAHULUAN

I.1 Latar Belakang

Aplikasi web umum pada saat ini, seperti *e-commerce*, umumnya berfokus pada mekanisme keamanan seperti *secure transfer protocol*, *paramerter sanitization*, dan menggunakan bermacam skema kriptografi. Para pengembang aplikasi tersebut lalu beranggapan dengan memberikan fitur keamanan seperti yang disebutkan sudah cukup, padahal masih banyak kelemahan keamanan aplikasi terjadi pada tingkat logika bisnis.

Business Logic Error (CWE-840) adalah salah satu kelemahan keamanan program yang disebabkan oleh kesalahan pada tingkat implementasi logika bisnis. Pengujian kelemahan ini tidak dapat diautomasi oleh kakas otomatis seperti *scanner* karena bergantung kepada domain dan bisnis aplikasi. Kelemahan ini juga biasanya terlupakan atau tidak dilakukan karena pada siklus pengembangan aplikasi biasa, yang diuji hanyalah kebutuhan fungsionalitas aplikasi apakah telah terimplementasi dengan baik, sementara kelemahan-kelemahan yang mungkin ada tidak teruji.

Pada saat ini, pengujian keamanan biasanya dilakukan setelah pengembangan aplikasi selesai, dan pengujian dilakukan dengan cara *blackbox* yaitu aplikasi yang telah selesai diberikan bermacam-macam masukan. Tetapi pengujian dengan cara *blackbox* masih memiliki kelemahan dimana walaupun mungkin bisa menemukan kelemahan keamanan yang sudah umum diketahui seperti *injection*, cara ini masih jarang menemukan kelemahan keamanan yang terjadi karena *business logic error*

yang biasanya membutuhkan langkah-langkah yang sangat spesifik dan berbeda tiap aplikasinya.

Pengujian keamanan akan menjadi lebih efektif jika diintegrasikan ke dalam siklus pengembangan aplikasi, seperti pengujian fungsionalitas. Pada fungsionalitas, pengujian diintegrasikan ke dalam pengembangan dengan menggunakan kakas TDD dan BDD. Kakas yang ada ini jika diikuti dengan baik bisa memberi jaminan bahwa fungsionalitas telah berjalan dengan baik. Namun kakas ini belum bisa digunakan bersamaan untuk pengujian keamanan karena sifat dari pengujian keamanan itu sendiri. Pengujian keamanan mengharuskan kita mencoba semua kemungkinan input yang pada normalnya tidak boleh diterima oleh aplikasi.

I.2 Rumusan Masalah

Dari latar belakang tersebut, penulis kemudian merumuskan masalah yaitu:

1. Apa yang menyebabkan pengujian keamanan berbeda dengan pengujian fungsionalitas?
2. Kenapa pengujian sulit dilakukan dengan baik walaupun telah menggunakan kerangka pengujian?
3. Apa saja hal yang dibutuhkan pada kerangka pengujian agar dapat mendukung pengujian keamanan?

I.3 Tujuan

Subbab sebelum ini telah menjelaskan latar belakang dan rumusan masalah tugas akhir ini. Karena itu, tujuan dari tugas akhir ini adalah membangun kakas pengujian aplikasi dengan kerangka BDD atau TDD, dimana kakas dapat juga dapat melakukan pengujian keamanan dari *business logic error* dengan mudah.

I.4 Batasan Masalah

Untuk mencapai tujuan yang telah dijelaskan pada subbab sebelumnya, kakas pengujian difokuskan pada hal berikut:

1. Untuk menguji ancaman keamanan yang disebabkan oleh *business logic error*
2. Kerangka pengujian yang digunakan sebagai acuan adalah kerangka pengujian BDD Gherkin

I.5 Metodologi

Metodologi yang digunakan pada pengerjaan tugas akhir ini antara lain:

1. Studi Literatur
2. Eksplorasi kebutuhan

   Pada tahap ini dilakukan analisa penyebab terjadinya *business logic error*. Lalu dilakukan penentuan fitur yang dibutuhkan untuk bahasa pengujian.
3. Desain
4. Implementasi
5. Evaluasi

BAB II

TINJAUAN PUSTAKA

Pada bab ini berisi hasil tinjauan pustaka yang menjadi dasar analisa dan perancangan pada BAB III. Bab ini secara garis besar berisi keamanan perangkat lunak dan pengujiannya, tantangan pada pengujian perangkat lunak berbasis web, *Domain-Specific Language*, serta BDD dan Gherkin.

II.1 Keamanan Perangkat Lunak

Penggunaan komputer yang semakin hari semakin luas membuat perangkat lunak yang ada semakin besar dan rumit, yang berarti juga bertambahnya masalah keamanan yang ada pada perangkat lunak tersebut. Hal ini menyebabkan keamanan perangkat lunak menjadi hal yang semakin penting.

Keamanan perangkat lunak (*software security*) adalah kriteria dimana perangkat lunak tetap bekerja dengan benar walaupun diserang dengan niat jahat. *Security* berbeda dengan *safety* dimana security fokus terhadap kebenaran perangkat lunak saat sedang dalam serangan yang dilakukan dengan sengaja, sedangkan *safety* fokus terhadap kebenaran perangkat lunak saat terjadi kegagalan baik pada tingkat perangkat lunak maupun perangkat keras.

Masalah keamanan perangkat lunak terjadi karena adanya celah atau kecacatan pada perangkat lunak yang dapat dimanfaatkan oleh penyerang. Celah ini dapat berbentuk kekurangan bawaan pada bahasa pemrograman yang digunakan, seperti penggunaan `gets()` pada bahasa C/C++ yang memiliki resiko *buffer overflow*, hingga

celah yang terjadi karena kesalahan pada desain perangkat lunak tersebut. Skala pembuatan perangkat lunak yang semakin besar dengan proses pengembangan yang melibatkan banyak orang menyebabkan tidak ada satu orang yang paham cara kerja perangkat lunak secara keseluruhan.

Ada beberapa cara yang dapat dilakukan untuk menanggulangi masalah keamanan perangkat lunak, namun pada saat ini perlindungan keamanan perangkat lunak dilakukan secara *de facto*, yaitu dengan perlindungan yang diimplementasi setelah aplikasi selesai dikembangkan. Perlindungan ini biasanya melindungi aplikasi dengan cara memperhatikan data yang masuk ke dalam aplikasi tidak menimbulkan bahaya atau dapat menyebabkan masalah, pada dasarnya, perlindungan jenis ini berdasar terhadap pencarian dan mengatasi celah pada aplikasi setelah ditemukan. Namun, perlindungan perangkat lunak seharusnya mengidentifikasi dan mengatasi masalah dari dalam perangkat lunak tersebut, sebagai contoh, walaupun ada baiknya mencoba menghadapi serangan *buffer overflow* dengan membaca *traffic* yang masuk ke dalam aplikasi, cara yang lebih bagus tentu saja memperbaiki perangkat lunak dari kodenya sehingga tidak ada kemungkinan *buffer overflow* (McGraw 2004).

II.2 CWE dan *Business Logic Error*

Common Weakness Enumeration (CWE) adalah daftar dan kategorisasi dari kelemahan keamanan yang umum ditemukan pada perangkat lunak dan keras (CWE 2019a). CWE dikelola oleh MITRE yang merupakan organisasi non profit. CWE memungkinkan *security engineer* untuk memiliki bahasa umum untuk menyampaikan kelemahan keamanan yang ada. CWE juga mengeluarkan daftar 25 kelemahan keamanan yang paling banyak digunakan, daftar ini dapat digunakan oleh *programmer* dan *tester* untuk membantu dalam pengembangan aplikasi.

Business Logic Error(BLE) adalah kategori celah keamanan berasal dari kesalahan yang biasanya memudahkan penyerang untuk memanipulasi logika bisnis aplikasi (CWE 2019b). BLE biasanya sulit untuk ditemukan secara otomatis, karena mereka biasanya melibatkan penggunaan fungsionalitas aplikasi dengan sah. Namun,

banyak BLE dapat memiliki pola-pola yang mirip dengan kelemahan implementasi dan detail yang sudah banyak dimengerti.

Klasifikasi dari BLE masih kurang dipelajari, walaupun eksploitasi dari BLE sering terjadi di sistem nyata. Masih banyak perdebatan apakah BLE merepresentasikan sebuah konsep baru, atau variasi dari konsep yang sudah dipahami.

Beberapa kategori dari BLE adalah:

1. Melewati authentikasi dengan alur berbeda (CWE-288)
2. Incorrect Behavior Order: Early Amplification (CWE-408)
3. Melewati authorisasi dengan parameter dari user (CWE-408)
4. Mekanisme pengembalian password yang lemah (CWE-640)

Banyak BLE berorientasi terhadap proses bisnis, alur aplikasi, dan urutan perilaku, yang dimana kelemahan-kelemahannya tidak banyak dipelajari di CWE.

II.3 Pengujian Keamanan Perangkat Lunak

Celah-celah keamanan yang ada pada perangkat lunak selalu menjadi risiko keamanan (*security risk*). Mengelola risiko keamanan ini menjadi seminimal mungkin adalah salah satu tugas praktisi keamanan perangkat lunak. Dalam mengelola risiko ini dilakukan beberapa hal (Potter dan McGraw 2004), diantaranya:

- Membuat kasus penyalahgunaan
- Membuat daftar kebutuhan keamanan
- Melakukan analisis risiko arsitektur
- Membuat perencanaan pengujian keamanan berbasis risiko
- Melakukan pengujian keamanan
- Melakukan pembersihan setelah terjadinya pelanggaran keamanan

Sistem keamanan bukanlah keamanan sistem. Walaupun fitur keamanan seperti

cryptography, *access control*, dan lain lain memiliki peran penting dalam keamanan perangkat lunak, keamanan itu sendiri adalah sifat dari sistem secara keseluruhan, bukan hanya dari mekanisme dan fitur keamanannya. Sebuah *buffer overflow* adalah masalah keamanan, baik itu terletak di dalam fitur keamanan ataupun di dalam sebuah tampilan non-kritikal. Karena itu dalam menguji keamanan perangkat lunak memiliki dua macam pendekatan (McGraw 2004):

1. Menguji mekanisme keamanan untuk memastikan bahwa fungsionalitasnya telah diterapkan dengan baik
2. Melakukan pengujian keamanan berbasis risiko berdasarkan pemahaman dan menyimulasikan pendekatan si penyerang sistem

Banyak *programmer* yang dengan salah mengira bahwa keamanan cukup hanya dengan mengimplementasikan dan menggunakan fitur-fitur keamanan. Banyak penguji perangkat lunak yang ditugaskan untuk melakukan pengujian keamanan melakukan kesalahan ini.

Seperti dalam pengujian lainnya, pengujian keamanan perangkat lunak terdiri dari memilih siapa orang yang akan melakukan pengujian dan apa yang akan dilakukannya. Dalam memilih orang ada dua kasus tergantung approach yang telah disebutkan, pada kasus pertama dapat dilakukan oleh staff QA dengan cara pengujian perangkat lunak seperti biasa untuk melakukan pengujian fungsional fitur-fitur keamanan sesuai spesifikasi. Namun pada kasus kedua, staff QA biasa akan kesulitan melaksanakan pengujian berbasis risiko karena membutuhkan bidang keahlian tertentu. Pertama, penguji harus dapat berpikir seperti penyerang sistem, kedua, pengujian keamanan kadang tidak memberikan hasil yang berhubungan langsung dengan celah keamanan yang ada, sehingga butuh keahlian untuk menginterpretasi dan memahami hasil pengujian (Potter dan McGraw 2004).

Kedua, dalam memilih metode pengujian, ada dua metode yang dapat dilakukan. Pertama dengan cara *White-box* yang dilakukan dengan menganalisis dan memahami kode serta desain dari program. Cara ini cukup efektif dalam menemukan kesalahan pemrograman, dalam beberapa kasus, pengujian ini dapat dilakukan oleh

static analyzer. Cara kedua adalah pengujian *Black-box* yang dilakukan dengan cara menguji program yang sedang berjalan dengan berbagai macam masukan tanpa harus mengetahui masukan program. Dalam pengujian keamanan, masukan buruk dapat dimasukkan dalam usaha untuk merusak program. Kedua cara pengujian dapat mengungkapkan adanya risiko keamanan dan kemungkinan eksploitasi. Masalah yang biasa terjadi dengan pengujian keamanan adalah terkadang organisasi atau perusahaan tidak memiliki waktu dan sumberdaya untuk melakukan pengujian yang cukup.

Dalam melakukan pengujian keamanan, ada beberapa tantangan yang mungkin dihadapi (Thompson 2003):

1. Adanya efek samping

   Dalam melakukan pengujian keamanan dengan pendekatan menguji fungsionalitas perangkat lunak, biasanya diberikan sebuah masukan A dan diperiksa apakah perangkat lunak mengembalikan hasil B sesuai dengan spesifikasi. Namun yang kadang terlupakan bahwa aplikasi dapat memiliki efek samping yang dapat dimanfaatkan penyerang sebagai celah keamanan. Salah satu contohnya adalah perangkat utilitas RDISK pada Windows NT 4.0, yang berfungsi untuk membuat *Emergency Repair Disk*. Program ini pada umumnya berjalan baik sesuai spesifikasi, namun saat program berjalan, ia membuat sebuah file sementara yang dapat dibaca oleh siapa saja. Hal ini berarti pengguna tamu (*guest*) dapat membaca isi file tersebut yang termasuk *registry Windows* yang berisi pengaturan tentang sistem yang dapat dimanfaatkan penyerang.

2. Keadaan Pengujian Keamanan Saat Ini

   Perusahaan yang menyediakan jasa pengujian keamanan biasanya memiliki daftar-daftar celah yang umum ada. Mereka biasanya hanya menggunakan daftar tersebut untuk membuat rencana pengujian. Cara seperti ini biasanya tidak akan dapat menemukan celah-celah keamanan yang baru.

3. Ketidakamanan dan kegagalan aplikasi penunjang

Perangkat lunak modern berjalan pada sistem yang saling bergantung satu sama lain, dimana satu aplikasi menggunakan puluhan *library* dan berkomunikasi dengan beberapa komponen lainnya. Hal ini dapat menimbulkan dua masalah. Pertama, aplikasi dapat memiliki celah dari salah satu komponen yang ia gunakan. Kedua, sebuah komponen yang digunakan untuk menyediakan fungsionalitas keamanan dapat saja pada suatu saat rusak dan berhenti bekerja.

4. Masukan tidak terkira dari pengguna

   Masukan dari pengguna adalah salah satu sumber celah yang paling umum dan paling mudah dieksploitasi. Beberapa contoh yang umum digunakan adalah masukan yang panjang, karakter spesial, dan nilai-nilai khusus. Salah satu contoh celah yang terjadi dari masukan pengguna ini adalah *buffer overflow*, yang memungkinkan penyerang menyisipkan kode pada masukan yang sangat panjang, hingga tidak bisa ditampung *buffer* dan dijalankan oleh komputer.

5. Ketidakamanan desain

   Banyak celah keamanan terjadi sejak perangkat lunak masih dalam tahap desain. Kadang celah tersebut tidak bisa langsung diketahui karena terjadi setelah semua bagian sistem selesai dirancang namun gabungan dari keseluruhan sistem tersebut menyebabkan adanya celah. Kadang celah juga terjadi pada test interface, yaitu bagian program yang sengaja disisipkan dan memberi celah untuk pengujian, namun tidak dihilangkan saat program akan dirilis.

6. Ketidakamanan implementasi

   Walaupun spesifikasi perangkat lunak telah didesain sebaik mungkin dengan mempertimbangkan berbagai macam aspek keamanan, celah tetap dapat terjadi karena implementasi perangkat lunak yang tidak sempurna.

II.4 Tantangan Dalam Pengujian Aplikasi *Web*

Perangkat lunak berbasis web adalah salah satu jenis perangkat lunak paling umum pada saat ini. Perangkat lunak ini menjadi tulang belakang dari komunikasi di dunia dan banyak hal-hal yang membutuhkan keamanan tinggi menggunakan perangkat lunak berbasis web seperti perbankan. Hal seperti menyebabkan perangkat lunak berbasis web menjadi salah satu target yang empuk untuk dimanfaatkan celah dan kekurangannya. Sifat dari aplikasi web yang dinamis, kompleks, dan selalu berubah-ubah membuat semakin mudahnya muncul celah baru pada aplikasi web jika tidak diperhatikan (Jaiswal, Raj, dan Singh 2014).

Beberapa masalah umum yang ada pada perangkat lunak berbasis web adalah:

1. Autentikasi: memastikan pengguna yang meminta data adalah benar pengguna tersebut
2. Autorisasi: memastikan pengguna boleh melakukan hal yang dilakukannya.
3. *Cross-site scripting*: celah dimana penyerang dapat memasukkan kode jahat ke halaman web yang dijalankan di browser pengguna lain.
4. *SQL injection*: celah dimana disisipkannya kode jahat di dalam perintah SQL yang kemudian dijalankan oleh *database*.
5. *Cross-site request forgery*: celah dimana sebuah *website* dapat dieksploitasi untuk mengirimkan perintah palsu dari sebuah user.
6. *Malicious file execution*: aplikasi web menjalankan kode jahat yang berada di sebuah file bebas

Beberapa tantangan dalam melakukan pengujian keamanan terhadap aplikasi web adalah (Jaiswal, Raj, dan Singh 2014):

1. Butuhnya pengembangan kakas yang dapat mengotomatisasi pengujian aplikasi web.
2. Pengembangan aplikasi web yang dinamis dan *Rich Content* seperti *Single-Page Application* mempersulit *crawling* halaman web sehingga bisa saja ada

state halaman yang tidak bisa dicapai oleh kakas pengujian.

3. Bahasa pemrograman yang digunakan pada implementasi tidak memiliki fitur yang dapat memaksa penggunaan aturan keamanan yang dapat menyebabkan bahaya terhadap keamanan dan integritas data pengguna.

II.5 *Behavior-Driven Development*

Behavior-Driven Development(BDD) adalah kerangka pengembangan dan pengujian perangkat lunak yang mendorong percakapan dan contoh konkret untuk memberikan pemahaman bersama atas tingkah laku perangkat (North 2017). BDD adalah ekstensi dari kerangka TDD, dimana yang didefinisikan adalah tingkah laku(*behavior*) dari perangkat lunak, bukan kasus-kasus uji eksplisit.

Menurut (Solis dan Wang 2011), BDD memiliki 6 karakteristik utama yaitu:

1. *Ubiquitous Language*

   Ubiquitous Language (Bahasa Umum) adalah sebuah bahasa yang strukturnya berasal dari model domain dan mengandung istilah-istilah yang akan digunakan untuk mendeskripsikan perilaku suatu perangkat lunak. Bahasa umum yang didasari dari domain bisnis memungkinkan customer, bisnis, dan *developer* saling berkomunikasi dengan jelas dan tanpa ambigu.

   BDD sendiri juga memiliki bahasa umumnya yang digunakan untuk mendeskripsikan fitur dan skenario perilaku perangkat lunak. Bahasa ini *domain independent*.

2. Proses Dekomposisi Iteratif

   Pada BDD analisis dimulai dengan identifikasi perilaku yang diharapkan dari sistem, yang lebih konkret dan mudah ditentukan. Lalu perilaku sistem akan diturunkan dari hasil bisnis yang seharusnya terjadi. Hasil bisnis itu kemudian diubah menjadi kumpulan fitur yang menyatakan apa saja yang harus ada agar hasil bisnis dihasilkan. Proses dekomposisi ini dilakukan secara iteratif, yang berarti tidak harus melakukan banyak analisi pada awalnya.

3. Penjelasan *User Story* dan Skenario dengan Simpel

   Pada BDD, biasanya deskripsi skenario, fitur, dan *user story* ditulis dalam sebuah template tertentu dengan menggunakan bahasa simpel. Berbagai macam kakas BDD seperti JBehave, NBehave, SpecFlow, dan Cucumber menggunakan cara ini walaupun memiliki kata kunci berbeda, tetapi masih memiliki arti semantik yang sama.

4. Pengujian Penerimaan Otomatis

   Pada BDD, skenario-skenario dari fitur yang telah sebelumnya dideskripsikan digunakan sebagai acuan untuk melakukan uji penerimaan (*acceptance testing*) secara otomatis. *Programmer* akan mulai dari salah satu skenario yang telah didefinisikan, yang kemudian dijadikan kode pengujian yang akan mengarahkan implementasi. Sebuah skenario terdiri dari langkah-langkah yang menggambarkan elemen-elemen yang ada dalam sebuah skenario.

5. Kode Spesifikasi Berorientasi Perilaku yang Mudah Dibaca

   BDD menganjurkan bahwa kode seharusnya menjadi bagian dari dokumentasi sistem. Kode seharusnya mudah dibaca dan spesifikasi seharusnya menjadi bagian dari kode.

   StoryQ dan JSpec menyediakan API yang memungkinkan *programmer* untuk mendeskripsikan *user story* dan skenario sebagai kode. JBehave dan NBehave juga membantu untuk menulis skenario sebagai kode dengan menggunakan *annotation*. Kebalikannya, Cucumber tidak berfokus kepada tingkat implementasi sehingga tidak memiliki karakteristik ini.

6. *Behaviour Driven* pada Fase Berbeda

   Karakteristik-karakteristik BDD yang telah dideskripsikan sebelumnya terjadi pada fase-fase yang berbeda selama dalam siklus pengembangan perangkat lunak. Pada fase rencana awal, perilaku perangkat lunak berhubungan dengan hasil bisnis. Pada fase analisis, hasil bisnis dipecah menjadi sekumpulan fitur yang melingkupi perilaku sistem. Pada fase implementasi, spesifikasi

tersebut digunakan untuk mengarahkan implementasi dan pengujian otomatis. *Programmer* dianjurkan untuk memikirkan perilaku dari komponen yang sedang mereka kembangkan dan interaksinya dengan komponen lain.

II.6 *Domain-Specific Language*

Domain-Specific Language (DSL) adalah bahasa pemrograman berkemampuan terbatas yang berfokus pada suatu domain tertentu (Fowler dan Parsons 2011).

Dari definisi diatas, ada empat poin penting:

1. DSL dapat digunakan untuk memerintahkan komputer. Seperti bahasa pemrograman lainnya, DSL haruslah bisa untuk dipahami manusia, tetapi masih mungkin diolah oleh komputer.
2. DSL adalah bahasa pemrograman komputer. Hal ini berarti DSL harus terasa seperti bahasa yang dimana kemampuannya tidak hanya muncul dari masing-masing ekspresinya, tetapi juga saat ekspresi-ekspresi tersebut digabungkan.
3. Bahasa pemrograman umum (*General Purpose Programming Language*) memiliki banyak fitur. Hal ini membuatnya sangat berguna, namun menjadi susah untuk dipelajari dan digunakan. DSL dengan kemampuannya yang terbatas hanya memiliki fitur-fitur minimum yang dibutuhkan untuk domainnya.
4. Sebuah bahasa dengan kemampuan terbatas hanya akan berguna jika ia memiliki fokus yang jelas terhadap sebuah domain kecil.

DSL terbagi menjadi dua kategori, yaitu:

1. *External DSL*

   DSL eksternal adalah bahasa yang terpisah dari bahasa utama aplikasi. Biasanya, DSL eksternal memiliki syntaxnya sendiri, namun kadang dapat menggunakan bahasa lain seperti XML. Sebuah kode pada DSL eksternal biasanya akan diproses oleh aplikasi utama. Beberapa contoh DSL eksternal adalah Regex, SQL, awk, sed.

2. *Internal DSL*

   DSL internal adalah sebuah cara tertentu untuk menggunakan sebuah bahasa. Sebuah DSL internal ditulis dalam bahasa yang sama dengan bahasa utama aplikasi, namun hanya menggunakan sebagian fitur bahasa untuk mengurusi bagian kecil dari keseluruhan sistem. Salah satu bahasa yang memiliki banyak DSL internal adalah Ruby, karena struktur Ruby yang ekspresif memudahkan dibuatnya DSL. Web framework Rails yang ditulis dengan Ruby adalah salah satu contoh DSL.

DSL adalah sebuah alat yang memiliki fokus yang jelas dan hanya mengurusi suatu aspek kecil tertentu. Sebuah aplikasi bisa saja menggunakan banyak DSL untuk mengurusi berbagai aspek sistemnya. Beberapa kelebihan menggunakan DSL adalah:

1. Meningkatkan produktivitas

   Salah satu daya tarik utama dari DSL adalah ia menyediakan cara untuk menyampaikan sebuah maksud sebuah sistem dengan lebih jelas. Hal ini menyebabkan programmer lebih mudah memahami maksud dan tujuan sebuah kode dan sistem.

2. Merepresentasikan pengetahuan domain dengan lebih baik

   DSL dapat didesain sedemikian mungkin untuk merepresentasikan dan mengabstraksikan suatu domain tertentu, sehingga bisa digunakan bukan hanya oleh *programmer* saja, tetapi juga oleh ahli domain tersebut.

Sementara kekurangan menggunakan DSL adalah:

1. *Language cacophony*

   Beberapa komplain yang sering didengar saat menggunakan DSL adalah *language cacophony*, dimana bahasa biasanya sulit untuk dipelajari, sehingga menggunakan banyak bahasa akan lebih sulit dari pada menggunakan satu bahasa. Kebutuhuan untuk mempelajari banyak bahasa menyebabkan sulit untuk mengerjakan proyek dan menambah orang baru kedalam proyek.

Namun dari komplain ini, banyak orang yang berpikiran bahwa mempelajari sebuah DSL akan sesulit mempelajari bahasa pemrograman general biasa. Tetapi, DSL sebenarnya lebih mudah dipelajari karena keterbatasannya.

2. Biaya pembuatan

   Seperti semua bagian dari program, DSL juga merupakan program yang harus dibuat dan dipelihara. Tentu saja hal ini dapat menambah biaya yang harus dikeluarkan. Biaya pembuatan DSL juga dapat lebih tinggi karena tim yang ada tidak terbiasa membuat DSL sehingga harus belajar lagi, yang juga dapat menambah biaya.

II.7 Cucumber dan Gherkin

Cucumber adalah salah satu kakas yang mendukung kerangka BDD (Wynne dan Hellesøy 2012). Cucumber memungkinkan *programmer* untuk menulis spesifikasi perilaku perangkat lunak dengan mudah dan kemudian dijalankan dalam pengujian, spesifikasi ini ditulis dengan bahasa Gherkin.

Gherkin adalah sebuah DSL yang digunakan untuk mendeskripsikan fitur dan skenario yang digunakan sebagai spesifikasi perilaku perangkat lunak. Gherkin ditulis dengan bahasa manusia sehingga dapat dipahami oleh seluruh pihak, namun walaupun ditulis dengan bahasa manusia, Gherkin dapat diolah dan digunakan oleh komputer sebagai garis besar penjalanan pengujian otomatis.

```
1  Feature: Guess the word
2    # The first example has two steps
3    Scenario: Maker starts a game
4      When the Maker starts a game
5      Then the Maker waits for a Breaker to join
6
7    # The second example has three steps
8    Scenario: Breaker joins a game
9      Given the Maker has started a game with the word "silky"
10     When the Breaker joins the Maker game
```

```
11      Then the Breaker must guess a word with 5 characters
```

Diatas adalah salah satu contoh kode Gherkin. Test dalam Gherkin dibagi menjadi fitur-fitur. Tiap fitur tersebut akan dibagi menjadi skenario, yang terdiri dari langkah-langkah.

II.7.1 *Step*

Step merupakan langkah kerja yang konkret. Setiap *step* dimulai dengan `Given`, `When`, `Then`, `And`, dan `But`. Cucumber akan menjalankan tiap *step* dalam sebuah *scenario* secara berurutan. Saat Cucumber akan menjalankan sebuah *step*, ia akan mencari kode definisi step yang sesuai. Misal jika sebuah *step* ditulis sebagai `When the maker starts a game`, maka Cucumber akan mencari *step definition* yang bernama `the maker starts a game`.

II.7.2 *Step Definition*

Cucumber memungkinkan terhubungnya antara definisi langkah pada Gherkin dengan kode test nyata melalui nama pada *step*. Cucumber dapat digunakan dengan beberapa bahasa pemrograman berbeda. *Step definition* pada Cucumber didefinisikan menggunakan fungsi-fungsi simpel, seperti:

```
1 package com.example;
2 import io.cucumber.java.en.Given;
3
4 public class StepDefinitions {
5     @Given("I have {int} cukes in my belly")
6     public void i_have_n_cukes_in_my_belly(int cukes) {
7         System.out.format("Cukes: %n\n", cukes);
8     }
9 }
```

BAB III

ANALISIS DAN PERANCANGAN

III.1 Analisis Permasalahan

Menurut kemudahan pengujiannya, kelemahan kemanan terbagi menjadi yang mudah diuji dan yang sulit diuji. Kelemakan keamanan yang mudah diuji biasanya memiliki penyebab yang jelas, telah banyak dipelajari, dan memiliki langkah yang mudah untuk mengujinya. Salah beberapa contoh dari kelemahan ini adalah *Cross Site Scripting* (XSS) dan *SQL injection*. Kedua kelemahan ini telah banyak dipelajari dan dipahami mekanismenya sehingga untuk mengujinya telah ada kumpulan *payload* yang bisa digunakan untuk menguji kelemahan ini. Selanjutnya adalah kelemahan yang sulit diuji. Kelemahan keamanan yang sulit diuji biasanya belum banyak dipelajari dan membutuhkan penggunaan program secara normal sehingga tidak bisa langsung diketahui dari awal. Salah satu dari kelas kelemahan yang termasuk kedalam kelemahan yang sulit diuji adalah *Business Logic Errors* (BLE).

BLE (CWE-840) merupakan jenis kelemahan yang terjadi karena kekurangan dan cacat dalam logika bisnis program. Kelemahan BLE biasanya muncul dalam proses bisnis, alur aplikasi dan urutan langkah-langkah. Jenis kelemahan ini sulit diidentifikasi karena terjadi sebagai efek samping dari logika bisnis program, bukan karena kelemahan teknologi yang digunakan, dan sulit diuji karena bentuk nyata dari jenis kelemahan yang terjadi pada program sebenarnya dapat memiliki banyak bentuk dan untuk menguji kelemahan tersebut perlu dilakukan urutan langkah-langkah tertentu.

Dalam kebanyakan *Software Development Life Cycle* (SDLC), setelah dilakukan penggalian kebutuhan, pemodelan arsitektur, dan desain detail akan didapatkan skenario yang berisi deskripsi-deskripsi fitur yang akan diimplementasikan, kemudian *programmer* akan mulai melakukan implementasi. Sementara *tester* akan melakukan pengumpulan *security requirement* dan *threat modelling* sehingga didapatkan *misuse case, abuse case* dan *threat model*. Setelah hasil pengumpulan ini didapatkan, kasus-kasus ini dapat diubah menjadi skenario-skenario yang berisi langkah-langkah dalam terjadinya suatu kasus ancaman keamanan.

Dalam BDD, fitur-fitur yang ada dalam sebuah program dideskripsikan dengan skenario berisi langkah yang menjelaskan alur terjadinya sebuah skenario. Pada kakas BDD yang ada pada saat ini, biasanya kasus test ini ditulis dengan bahasa dan *syntax* yang mudah dimengerti semua pihak mulai dari *programmer* yang akan melakukan implementasi hingga *stakeholder* yang memiliki aplikasi. Penggunaan bahasa tentu saja memiliki *tradeoff* yang berbentuk dalam sulitnya untuk mendeksripsikan keadaan yang lebih kompleks.

Hasil dari pemodelan ancaman keamanan yang berbentuk skenario yang terdiri dari langkah-langkah cocok dengan metodologi BDD yang juga mendeskripsikan skenario dari fitur program dalam bentuk langkah-langkah. *File* fitur BDD berisi pengetahuan tentang kejadian apa saja yang mungkin terjadi terhadap suatu fitur program. Kumpulan pengetahuan ini dapat digunakan untuk membantu menemukan kelemahan keamanan yang termasuk kedalam BLE, karena kelemahan BLE umumnya terjadi setelah langkah-langkat tertentu dijalankan secara berurutan.

Pada saat ini, ada beberapa kakas yang dapat digunakan untuk melakukan metodologi BDD. Beberapa diantaranya adalah Cucumber dan Rspec. Cucumber adalah kakas metodologi BDD yang menggunakan bahasa Gherkin. Kakas ini dapat digunakan dengan beberapa bahasa pemrograman seperti Java, Javascript, Ruby, dan Python. Sementara Rspec adalah kakas metodologi BDD yang menggunakan DSL Ruby, dan hanya bisa digunakan dengan bahasa pemrograman Ruby. Cucumber memiliki kelebihan dimana *file* pengujian ditulis dengan bahasa Gherkin yang berbentuk bahasa manusia, sehingga dapat dimengerti oleh banyak pihak. Namun keku-

rangannya adalah *syntax* Gherkin yang simpel menyebabkan sulitnya untuk mengekspresikan konstruk yang lebih kompleks karena hanya didesain untuk pengujian fungsionalitas, bukan untuk pengujian keamanan.

III.2 Kebutuhan dan Rancangan Solusi

Pada bagian ini akan dibahas tiga poin pengembangan terhadap bahasa Gherkin dan kakas Cucumber yang dapat diambil untuk membuat Gherkin dan Cucumber menjadi lebih cocok untuk pengujian keamanan. Tiga poin tersebut adalah kemampuan untuk menyatakan kegagalan, kemampuan untuk menyatakan variasi, dan pengacakan skenario. Dua poin pertama merupakan pengembangan terhadap bahasa Gherkin, dan poin terakhir merupakan pengembangan terhadap kakas Cucumber sebagai *runtime* bahasa Gherkin.

III.2.1 Bisa menyatakan kegagalan

Pada pengujian fungsionalitas, kita hanya menuliskan kasus-kasus positif dimana skenario berjalan dengan benar. Namun untuk pengujian BLE, menuliskan dan melakukan pengujian dimana skenario tidak berjalan dengan benar haruslah sama mudahnya dengan skenario yang benar.

Pada saat ini, penulisan *step* pada Gherkin lebih berorientasi terhadap kasus sukses. Misalkan untuk step `Then the basket should have items in it`, dapat memiliki step gagal berupa `Then the basket should not have items in it`. Secara semantik hal dua tadi adalah kebalikan, dimana logika pengujian sama, berbeda namun di akhir kode *step definition*. Hal ini menyebabkan banyaknya duplikasi kode *step definition* dan membuat *programmer* malas untuk melakukannya. Duplikasi ini dikarenakan setiap kode *step definition* dari Gherkin dianggap sukses/*passing* jika tidak ada exception yang terjadi. Kita ingin mengurangi duplikasi dan mengingkatkan *code reuse*. Kita ingin agar kode *step definition* dari step `Then the basket should have items in it` dapat digunakan untuk skenario sukses ataupun gagal.

Pada saat ini Gherkin hanya menyatakan *step* yang positif seperti ”barang sukses ditambahkan ke dalam keranjang”, namun butuh mendeskripsikan *step function* lagi untuk negatif *step* tersebut seperti ”barang gagal ditambahkan ke dalam keranjang” atau ”barang tidak sukses ditambahkan ke dalam keranjang”.

Untuk pembahasan fitur ini, kita akan mengacu pada kode Gherkin dibawah:

```
Feature: keranjang
  Scenario: menambahkan barang ke dalam keranjang
    Given user telah login
    When user memasukkan 1 barang
    Then ada 1 barang di dalam keranjang
```

Untuk desain fitur Failure dapat kita lakukan beberapa hal.

III.2.1.1 Scenario Outline

Screnario di atas memiliki initial state dimana user telah login. Programmer dapat menambahkan satu skenario lagi untuk keadaan dimana user gagal login, seperti:

```
Feature: keranjang
  Scenario: menambahkan barang ke dalam keranjang
    Given user telah login
    When user memasukkan 1 barang
    Then sukses ada 1 barang di dalam keranjang
  Scenario: user belum login menambahkan barang
    Given user belum login
    When user memasukkan 1 barang
    Then gagal ada 1 barang di dalam keranjang
```

Fitur *Scenario Outline* dari Gherkin dapat dimanfaatkan untuk memperpendek skenario ini menjadi

```
Feature: keranjang
  Scenario Outline: menambahkan barang ke dalam keranjang
    Given user <login state> login
    When user memasukkan 1 barang
    Then <result state> ada 1 barang di dalam keranjang
    Examples:
```

```
      | login state | result state |
      | sudah       | sukses       |
      | belum       | gagal        |
```

Dengan menggabungkan fitur *scenario outline* dengan *fail scenario* kita dapat membuat bagian baru dari *scenario outline* yang menyatakan contoh yang harusnya gagal, sehingga tidak membutuhkan variabel `result state`, dengan menggunakan fitur tersebut kode pengujian kita dapat menjadi seperi

```
Feature: keranjang
  Scenario Outline: menambahkan barang ke dalam keranjang
    Given user <login state> login
    When user memasukkan 1 barang
    Then ada 1 barang di dalam keranjang
    Examples:
    | login state   |
    | telah         |
    Fail Examples:
    | login state   |
    | belum         |
```

III.2.2 Bisa menyatakan variansi

Gherkin pada saat ini memiliki fitur *scenario outline* yang dapat menyatakan banyak skenario yang mirip hanya dalam satu skenario saja dengan menggunakan template dan tabel. *Scenario outline* dapat memenuhi kebutuhan bahasa untuk menyatakan variansi.

Namun fitur ini dapat dikembangkan dengan menambah kemampuan untuk menyatakan domain/tipe data suatu variabel, misalkan domain `integer`, `positive`, `string`, *enum*, dan lain lainnya. Hal ini membuat deklarasi variansi lebih singkat dan padat. Kemampuan ini juga dapat digabungkan dengan poin sebelumnya. Untuk *Scenario outline*, kita dapat menyatakan tabel *example* dengan kombinasi-kombinasi varian yang harus gagal. Untuk domain variabel, kita dapat menyatakan nilai mana saja yang harus gagal.

Sebagai contoh dalam aplikasi *e-commerce*, seorang user dapat memiliki status

telah atau belum *login*. Seperti yang telah dibahas pada poin sebelumnya, kita dapat menggunakan fitur *scenario outline* dan *fail scenario* untuk menyatakan pengujian fitur tersebut. Namun dengan bertambahnya fitur dan skenario yang akan diuji, mengulang penggunaan *scenario outline* yang berulang-ulang menjadi sulit. Dengan contoh diatas tentu saja *state* seorang user simpel saja, hanya sudah atau belum login, namun jika attribut yang dimiliki user memiliki banyak nilai, misalnya jenis user yang dapat berupa admin, pembeli dan penjual, jika ada perubahan dari spesifikasi maka semua tempat yang terkait dengan attribut ini harus diubah.

Untuk mengatasi hal ini, kita dapat menambahkan fitur dimana kita dapat membuat deklarasi nilai-nilai yang dapat dimiliki suatu variabel, seperti enum. Sebagai contoh, fitur admin dashboard suatu aplikasi *e-commerce* dapat diuji dengan menggunakan kode berikut

```
1  Variables:
2    user role: enum buyer, seller, admin
3  Feature: Admin
4    Scenario: admin mengubah setting website
5      Given user dengan role <user role> telah login
6      When user merubah pengaturan website
7      Then pengaturan website berubah
8      Variables Accepted:
9      | user role    |
10     | admin        |
```

Bagian pada `Variables Accepted` mirip dengan bagian *Examples* pada scenario outline, namun *tester* hanya perlu menulis kombinasi nilai variabel-variabel yang diterima, *runtime* gherkin akan secara otomatis menguji semua kombinasi variabel.

Deklarasi ini bersifat global sehingga dapat digunakan di file fitur lainnya. Dengan fitur ini duplikasi didalam kode pengujian menjadi berkurang dan kode memiliki satu sumber fakta sehingga jika terjadi perubahan pada spesifikai tidak harus mengubah kode pada banyak tempat.

III.2.3 Pengacakan Skenario

Salah satu penyebab kesulitan dari pengujian kelemahan keamanan jenis BLE adalah karena kelemahan ini memiliki langkah-langkah yang harus dijalankan di dalam aplikasi, namun teknologi pada saat ini masih belum bisa melakukan eksplorasi langkah-langkah yang bisa dilakukan dalam aplikasi dari satu *state* ke *state* lainnya, terutama dalam aplikasi berbasis web.

Dalam *file* pengujian BDD telah terdapat sekumpulan pengetahuan tentang langkah-langkah yang dapat terjadi dalam suatu aplikasi dalam bentuk *step* given, when dan then. Pengetahuan ini dapat dimanfaatkan oleh kakas pengujian untuk membangkitkan skenario-skenario baru secara acak dengah harapan skenario acak ini dapat membantu menemukan kelemahan-kelemahan baru. Cara ini mirip dengan *fuzzy testing* dimana kasus pengujian dibangkitkan secara acak.

III.3 Struktur Bahasa

Pada bagian ini akan dipaparkan grammar bahasa yang akan dibuat dalam EBNF. Gherkin yang menggunakan bahasa manusia memiliki struktur yang cukup bebas namun kita masih dapat memperkirakan grammarnya sebisa mungkin dalam EBNF.

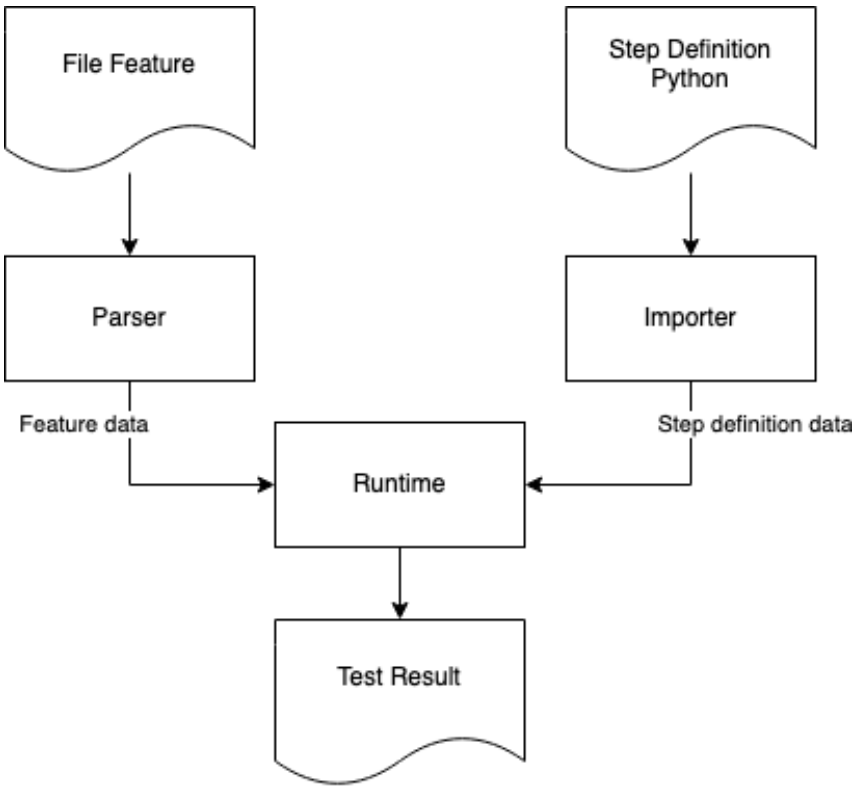
```
string
  : .+
token
  : \w+
newline
  : \n ;
stepKeyword
  : 'given' | 'when' | 'then' ;
stepLine
  : stepKeyword string newline ;
stepLines
  : stepLine+ ;
scenarioKeyword
  : 'scenario' | 'fail scenario' ;
```

```
scenario
  : scenarioKeyword ':' string newline
    stepLines variableAcceptedTable?;
backgroundScenario
  : 'background' ':' string newline stepLines ;

tableLine
  : '|' ( string '|' )+ ;
dataTable
  : (tableLine newline)+ ;
dataTableBody
  : ':' newline dataTable ;
exampleDataTable
  : ('example' dataTableBody)? ('fail example' dataTableBody)?
   ;
scenarioOutline
  : 'scenario outline' ':' newline stepLines
    exampleDataTable? variableAcceptedTable?;

variableDeclarationType
  : 'enum' string (',' string)*
  | 'bool' ;
variableDeclarationEntry
  : string ':' variableDeclarationEntry newline ;
variableDeclaration
  : 'variable' ':' newline variableDeclarationEntry+ ;
variableAcceptedTable
  : 'variable accepted' dataTableBody ;

featureChild
  : scenario | backgroundScenario
  | variableDeclaration | scenarioOutline ;
feature
  : 'feature' ':' string newline featureChild+;

topLevelEntry
  : feature | variableDeclaration ;
```

```
51
52 featureFile
53   : topLevelEntry+ ;
```

III.4 Desain Arsitektur

Dalam pembuatan kakas ini, ada tiga komponen yang saling berkerja sama. Pertama adalah *parser* yang berfungsi untuk mengubah *file* fitur menjadi struktur data yang lebih mudah diolah. Kedua adalah *importer* yang berfungsi untuk membaca *file* python dan mengumpulkan seluruh definisi step. Ketiga adalah *runtime* yang menerima hasil dari *parser* dan *importer* dan kemudian menjalankan pengujian. Diagram arsitektur seperti pada gambar III.4.1.



Gambar III.4.1: Desain Arsitektur Kakas

BAB IV

IMPLEMENTASI

IV.1 Detail Implementasi

Seluruh komponen dibuat dengan menggunakan bahasa python. Kakas dibuat dengan bentuk library python sehingga dapat diinstall dengan mudah menggunakan *pip*. Kakas didesain untuk menguji program yang juga ditulis dengan bahasa python.

IV.1.1 Parser

Komponen *parser* berfungsi untuk membaca *file* fitur dan menguraikan isinya menjadi struktur data yang dapat diolah. Komponen ini hanya menghasilkan skenario dalam bentuk dasar yang hanya berisi step-step, sehingga semua bentuk skenario yang lebih kompleks seperti *scenario outline* akan diuraikan terlebih dahulu menjadi semua kemungkinan kombinasi skenario dasarnya. Sebagai contohnya adalah

```
1 Scenario Outline: menambahkan barang ke dalam keranjang
2   Given user <login state> login
3   When user memasukkan 1 barang
4   Then <result state> ada 1 barang di dalam keranjang
5   Examples:
6     | login state | result state |
7     | sudah       | sukses       |
8     | belum       | gagal        |
```

Kode diatas akan diuraikan menjadi skenario dasar dengan bentuk

```
Scenario: menambahkan barang ke dalam keranjang (1)
  Given user sudah login
  When user memasukkan 1 barang
  Then sukses ada 1 barang di dalam keranjang
Scenario: menambahkan barang ke dalam keranjang (2)
  Given user belum login
  When user memasukkan 1 barang
  Then gagal ada 1 barang di dalam keranjang
```

Perubahan ini berfungsi agar komponen *runtime* hanya perlu tahu cara menjalankan skenario dasar saja sehingga lebih simpel, dan juga memperbanyak jumlah step yang diketahui oleh kakas sehingga dapat membangkitkan skenario acak yang lebih beragam.

Kode penguraian menggunakan arsitektur *Parsing Expression Grammar* (PEG), dimana pada arsitektur ini setiap *non-terminal* seperti `scenario`, `scenarioOutline`, `feature` yang ada dalam *grammar* pada III.3 dibentuk menjadi fungsi-fungsi yang dapat disusun, sehingga kode penguraian mirip dengan *grammar*. Fungsi-fungsi ini memiliki tipe parameter dan kembalian yang sama sehingga dapat dikomposisikan menghasilkan fungsi yang lebih kompleks. Pada psudeocode dibawah akan diilustrasikan kode penguraian untuk grammar `scenario` simpel.

```
# fungsi kombinator
def optional(func)       # func?
def zero_or_more(func)   # func*
def one_or_more(func)    # func+
def or(func1, func2,..) # func1 | func2 | ..
def literal(string)      # "string"
def parseString(input)   # string
def newline(input):
  return literal("\n")(input)

def parseStepKeyword:
  return or(literal("given"), literal("when"), literal("then"))

def parseStepLine(input):
  keyword, input = parseStepKeyword(input)
```

```
16    line, input = parseString(input)
17    _, input = newline(input)
18    return Step(keyword, line), input
19
20  def parseScenarioKeyword:
21    return or(
22      literal("scenario"),
23      literal("fail scenario")
24    )
25
26  def parseStepLines(input):
27    return one_or_more(parseStepLine)(input)
28
29  def parseScenario(input):
30    keyword, input = parseScenarioKeyword(input)
31    _, input = literal(":")(input)
32    desc, input = parseString(input)
33    _, input = newline(input)
34    steps, input = parseStepLines(input)
35
36    return Scenario(keyword, desc, steps), input
```

IV.1.2 Importer

Importer berfungsi untuk membaca *file* step descriptor yang ditulis dalam python. Importer menghasilkan kumpulan fungsi *step descriptor* yang akan digunakan untuk menjalankan step-step yang telah didefinisikan dari skenario.

Importer akan membaca semua file python yang ada dalam folder fitur, lalu mencoba untuk mencari semua fungsi yang merupakan step descriptor yang memiliki *decorator* python. Semua fungsi ini lalu dikumpulkan dan diteruskan ke komponen *runtime*. Cara kerja importer secara simpel digambarkan oleh *psudeocode* berikut:

```
1  def feature_folder
2  def step_descriptors = []
3
4  for file in feature_folder:
```

```
    file_objects = import(file)
    for function in file_objects.functions:
      if function is step descriptor:
        step_descriptors.add(function)

return step_descriptors
```

IV.1.2.1 API

Komponen ini adalah bagian dari kakas *library* yang digunakan oleh user untuk menulis *step descriptor*. Bagian ini meng-*export* decorator-decorator yang disediakan. Decorator ini berfungsi untuk menandai fungsi sebagai *step descriptor* sehingga dapat dibedakan dari fungsi biasa oleh *importer*. Cara kerja descriptor dan contoh penggunaannya digambarkan oleh *psudeocode* berikut:

```
def step_decorator(keyword, function):
  mark function as step decorator
  return function
def given(func):
  return step_decorator("given", func)
def when(func):
  return step_decorator("when", func)
def then(func):
  return step_decorator("then", func)

# contoh penggunaan
@given("user awalnya punya {num} kue")
def step1(num):
  user.kue = num

@when("user memakan {num} kue")
def step2(num):
  user.kue -= num

@then("user sisa kue {num}")
def step3(num):
```

```
22    assert user.kue == num
```

IV.1.3 Runtime

Runtime berfungsi untuk menjalankan pengujian. Runtime menerima hasil dari parser dan importer, mencocokkan step-step dalam skenario dengan fungsi *step descriptor* yang cocok, dan kemudian menjalankan skenario pengujian.

Runtime menghasilkan laporan penjalanan pengujian. Laporan ini berisi skenario apa saja yang berhasil dan gagal. Laporan ini juga berisi penyebab kegagalan skenario dalam bentuk catatan exception.

Bagian runtime juga berfungsi untuk membangkitkan skenario acak dari *step* yang telah ada dan menjalankannya.

IV.2 Skenario Pengujian

Pengujian dilakukan dengan tujuan untuk menguji apakah kakas yang dibuat berjalan dengan baik dan bisa memenuhi fungsionalitas yang dibutuhkan. Pengujian dilakukan dengan cara membuat file fitur untuk suatu proyek tertentu secara perlahan hingga semua fitur telah diuji secara BDD. Proyek yang akan diuji adalah proyek python yang dibuat sendiri atau proyek sampel aplikasi dalam python yang bersifat *open-source*.

Daftar Pustaka

[1] CWE. *Common Weakness Enumeration*. 2019. URL: https://cwe.mitre.org/.

[2] CWE. *CWE-840: Business Logic Error*. Jun. 2019. URL: https://cwe.mitre.org/data/definitions/840.html.

[3] Martin Fowler dan Rebecca Parsons. *Domain-specific languages*. Addison-Wesley, 2011.

[4] Arunima Jaiswal, Gaurav Raj, dan Dheerendra Singh. "Security Testing of Web Applications: Issues and Challenges". Dalam: *International Journal of Computer Applications* 88 (jan. 2014). DOI: 10.5120/15334-3667.

[5] G. McGraw. "Software security". Dalam: *IEEE Security Privacy* 2.2 (mar. 2004), hal. 80–83. ISSN: 1558-4046. DOI: 10.1109/MSECP.2004.1281254.

[6] Dan North. *Introducing BDD*. Feb. 2017. URL: https://dannorth.net/introducing-bdd/.

[7] B. Potter dan G. McGraw. "Software security testing". Dalam: *IEEE Security Privacy* 2.5 (sept. 2004), hal. 81–85. ISSN: 1558-4046. DOI: 10.1109/MSP.2004.84.

[8] C. Solis dan X. Wang. "A Study of the Characteristics of Behaviour Driven Development". Dalam: *2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications*. Aug. 2011, hal. 383–387. DOI: 10.1109/SEAA.2011.76.

[9] H. H. Thompson. “Why security testing is hard”. Dalam: *IEEE Security Privacy* 1.4 (jul. 2003), hal. 83–86. ISSN: 1558-4046. DOI: 10.1109/MSECP.2003.1219078.

[10] Matt Wynne dan Aslak Hellesøy. *The cucumber book: behaviour-driven development for testers and developers*. Pragmatic Bookshelf, 2012.