BAB 1

PENDAHULUAN

1.1 Latar Belakang

Pengujian perangkat lunak merupakan salah satu tahapan dari *software development life cycle*. *Software Development Life Cycle*(SDLC) merupakan metodologi dari pengembangan perangkat lunak, metode ini dibagi menjadi beberapa fase seperti *analysis, design, coding, testing, installation* dan *maintenance*[7]. *Testing*/Pengujian perangkat lunak adalah proses untuk mencari kesalahan pada setiap *item* perangkat lunak, mencatat hasilnya, mengevaluasi setiap aspek pada setiap komponen (sistem) dan mengevaluasi fasilitas-fasilitas dari perangkat lunak yang akan dikembangkan[1].Pengujian pada perangkat lunak merupakan tahapan yang wajib dilakukan sebelum perangkat lunak tersebut digunakan, agar memastikan sudah tidak ada *error/bug* pada perangkat lunak yang sedang dikembangkan. Pengujian perangkat lunak memakan *resource* yang berat, baik itu waktu maupun waktu maupun tenaga kerja, karena untuk melakukan pengujian perangkat lunak dibutuhkan *developer* ahli *Software Quality Assurance*(SQA). SQA yang bertanggung jawab untuk memastikan bahwa perangkat lunak yang sedang dikembangkan bebas dari *bug* agar siap untuk di-*release* dan digunakan oleh pengguna.

Unit testing merupakan tahapan pertama dari pengujian perangkat lunak. *Unit Testing* adalah proses pengujian bagian kode secara individu, suatu komponen, untuk menentukan apakah kode tersebut berfungsi dengan semestinya[5]. Salah satu teknik untuk membuat *unit testing* yaitu dengan code generation. *Code generation* adalah teknik membuat suatu program yang membuat program lain. Menggunakan *code generation* untuk *unit testing* membuatnya mudah untuk mempertahankan dan memperpanjang *unit testing*[6].

Ada beberapa metode untuk melakukan pengujian perangkat lunak, salah satunya yaitu *Test-Driven Development*(TDD). TDD adalah praktik pemrograman yang menginstruksikan pengembang perangkat lunak untuk menulis kode baru hanya ketika tes yang dilakukan secara otomatis gagal, dan menghilangkan duplikasi. Tujuan TDD adalah kode bersih yang berfungsi[2]. Tetapi, TDD memiliki masalah, bahwa TDD berfokus pada keadaan sistem daripada *behaviour* yang diinginkan oleh sistem, dan kode pada pengujian *highly coupled* dengan implementasi sistem yang ada[3]. Maka dari itu metode yang akan digunakan pada skripsi ini adalah *Behaviour-Driven Development*.

Behaviour-Driven Development(BDD) merupakan evolusi dari TDD untuk menyelesaikan masalah yang ada pada TDD. BDD adalah pendekatan pengembangan perangkat lunak yang *agile* untuk mendorong kolaborasi antara semua peserta dalam proses pengembangan perangkat lunak[4].

Prinsip inti dari BDD adalah "orang bisnis dan teknologi harus merujuk ke sistem yang sama dengan cara yang sama"[4]. Berbeda dengan TDD yang berpatokan pada test untuk *develope* lebih jauh program yang akan digunakan, BDD berpatokan pada keinginan *stakeholder/customer* untuk mengembangkan programnya. Untuk mencapai kesepakatan antara *developer* dan *stakeholder/customer*, maka dibutuhkan *language* untuk menspesifikasikan *behaviour* sebuah sistem agar kedua pihak paham, dan dapat mewujudkan hal berikut[4]:

- *Stakeholder/Customer* untuk menentukan persyaratan dari perspektif bisnis.
- Analis bisnis untuk melampirkan contoh konkret (skenario atau *acceptance tests*) yang menje-

laskan *behaviour* sistem.

- *Developer* untuk mengimplementasikan *behaviour* sistem yang diperlukan secara TDD.

Pengujian dengan basis *behaviour specification* akan berfokus kepada hal yang *stakeholder/customer* harapkan pada suatu sistem dengan *behaviour* yang sudah di spesifikasikan. Pengujian akan berawal dengan membuat *scenario* yang berisi [4]:

1. *Context/Starting state* : Posisi awal sistem sebelum terjadinya suatu *event*.
2. *Event* : *Task* yang dilakukan oleh pengguna pada sistem.
3. *Outcome* : Hasil yang diharapkan dari sistem.

Developer akan menjadikan *scenario* sebagai patokan dasar bekerjanya suatu sistem, dan akan melakukan pengujian berbasis *scenario*. Pengujian dilakukan untuk memastikan bahwa sistem sudah bekerja sesuai apa yang *stakeholder/customer* harapkan, dan jika *scenario* berhasil dilakukan, maka sistem sudah siap untuk digunakan.

Pada skripsi ini akan dibangun sebuah perangkat pengujian yang mengimplementasi behaviour specification untuk menguji sebuah perangkat lunak.

1.2 Rumusan Masalah

- Bagaimana cara kerja pengerjaan Pengujian Berbasis Behaviour Specification?
- Bagaimana implementasi perangkat lunak yang dapat menguji program berbasis Behaviour Specification ?

1.3 Tujuan

- Mempelajari cara kerja Pengujian Berbasis Behaviour Specification
- Menghasilkan perangkat penguji yang dapat menguji sebuah perangkat lunak berbasiskan *behaviour specification*.

1.4 Batasan Masalah

Mengingat banyaknya perkembangan yang bisa ditemukan dalam permasalahan ini, maka perlu adanya batasan-batasan masalah yang jelas mengenai apa yang dibuat dan diselesaikan dalam program ini. Adapun batasan-batasan masalah pada penelitian ini sebagai berikut :

1.5 Metodologi

Bagian-bagian pekerjaan skripsi ini adalah sebagai berikut :

1. Melakukan Studi literatur mengenai pengujian perangkat lunak, *unit testing*, dan *behaviour specification*.
2. Melakukan eksplorasi perangkat lunak.
3. Menganalisis kebutuhan perangkat lunak.
4. Membuat rancangan desain perangkat lunak, basis data, antarmuka perangkat lunak.

5. Implementasi perangkat lunak yang menerima *input behaviour spesification*, mengolah, dan mengubah ke bentuk *unit testing*.

6. Melakukan pengujian terhadap perangkat lunak.

7. Melaporkan hasil penelitian dalam bentuk dokumen skripsi.

1.6 Sistematika Pembahasan

- BAB I. PENDAHULUAN
  Bab ini berisi tentang latar belakang masalah, rumusan masalah, tujuan, batasan masalah, dan metodologi.

- BAB II. LANDASAN TEORI
  Bab 2 memuat uraian tentang teori yang akan digunakan pada pembuatan perangkat lunak.

BAB 2

LANDASAN TEORI

2.1 Pengujian Perangkat Lunak

Pandangan umum pengujian perangkat lunak adalah bahwa kegiatan ini adalah untuk menemukan *bug*. Tujuan pengujian perangkat lunak adalah untuk memenuhi syarat kualitas program perangkat lunak dengan mengukur atribut dan kemampuannya terhadap ekspektasi dan standar yang berlaku. Pengujian perangkat lunak juga menyediakan informasi berharga untuk upaya pengembangan perangkat lunak.[9]

Kualitas perangkat lunak adalah sesuatu yang diinginkan semua orang. Manajer tahu bahwa mereka menginginkan kualitas tinggi, pengembangan perangkat lunak tahu mereka ingin menghasilkan produk yang berkualitas, dan pengguna bersikeras bahwa perangkat lunak bekerja secara konsisten dan dapat diandalkan.[9]

Banyak kelompok kualitas perangkat lunak mengembangkan *software quality assurance plan*, dimana hal itu sama dengan *test plans*. Rencana jaminan kualitas perangkat lunak dapat mencakup berbagai kegiatan di luar yang termasuk dalam *test plan*. *Quality assurance plan* mencakup keseluruhan kualitas, rencana pengujian adalah salah satu alat kontrol kualitas dari rencana jaminan kualitas.[9]

Pada pembahasan pengujian perangkat lunak, ada *term* yang biasa digunakan, yaitu[8]:

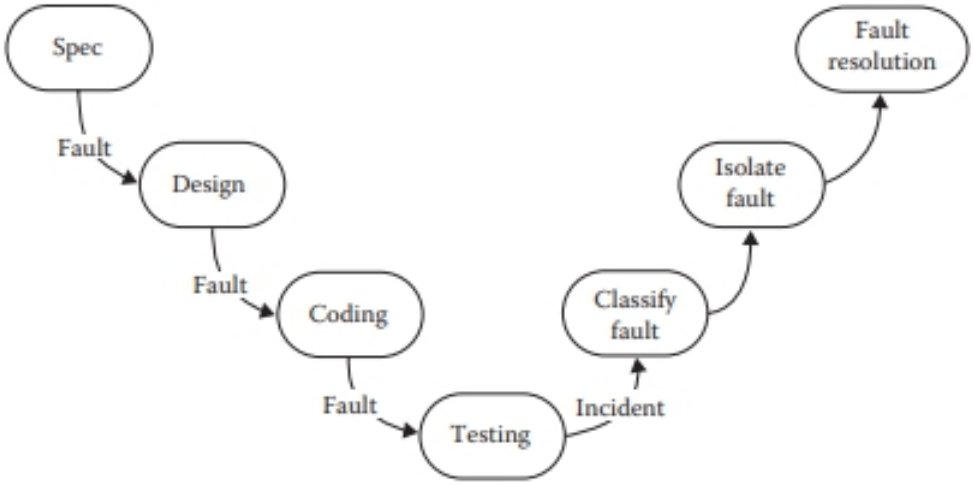
- **Error** — Orang membuat *error*. Sinonim yang baik adalah *mistake*. Ketika orang membuat *error* saat melakukan *coding*, kami menyebut *error* ini *bug*. *Error* cenderung menyebar; *requirements error* dapat diperbesar selama proses desain dan lebih diperkuat lagi selama pengkodean.

- **Fault** — *Fault* adalah hasil dari *error*. Lebih tepat untuk mengatakan bahwa *fault* adalah representasi dari *error*, di mana representasi adalah mode ekspresi, seperti teks naratif, diagram Bahasa Pemodelan Bersatu, diagram hierarki, dan kode sumber. *Defect* adalah sinonim yang baik untuk *fault*, sama juga seperti *bug*. *Fault* bisa sulit dipahami. *Error* yang disebakan oleh kelalaian menghasilkan *fault* di mana ada sesuatu yang hilang yang seharusnya ada di dalam representasi.

- ***Failure*** — *Failure* terjadi ketika kode yang sesuai dengan *fault* dijalankan. Dua kehalusan muncul di sini: satu adalah bahwa *failure* hanya terjadi dalam representasi yang dapat dieksekusi, yang biasanya dianggap sebagai kode sumber, atau lebih tepatnya, kode objek yang dimuat; kehalusan kedua adalah bahwa definisi ini hanya mengaitkan *failure* dengan *fault* komisi.

- **Incident** — Ketika *failure* terjadi, itu mungkin atau mungkin tidak mudah terlihat oleh pengguna (atau pelanggan atau penguji). Suatu *incident* adalah gejala yang terkait dengan *failure* yang memberi tahu pengguna tentang terjadinya *failure*.

- **Test** — *Testing* jelas berkaitan dengan *errors, faults, failures, and incidents*. *Test* adalah tindakan melatih perangkat lunak dengan *test case*. *Test* memiliki dua tujuan berbeda: untuk menemukan *failures* atau untuk menunjukkan eksekusi yang benar.



Gambar 2.1: Siklus hidup pengujian.

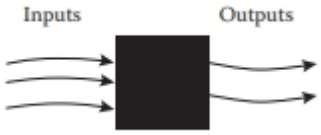
- **Test case** — *Test case* memiliki identitas dan dikaitkan dengan perilaku program. Ini juga memiliki serangkaian input dan output yang diharapkan.

Gambar 2.1 menggambarkan model siklus hidup untuk pengujian. Perhatikan bahwa, dalam fase pengembangan, tiga peluang muncul untuk membuat *error*, yang menghasilkan *fault* yang dapat menyebar melalui proses *development*. Langkah resolusi *fault* adalah kesempatan lain untuk *error* (dan *fault* baru). Ketika suatu perbaikan menyebabkan perangkat lunak yang sebelumnya benar untuk berperilaku salah, perbaikannya kurang[8].

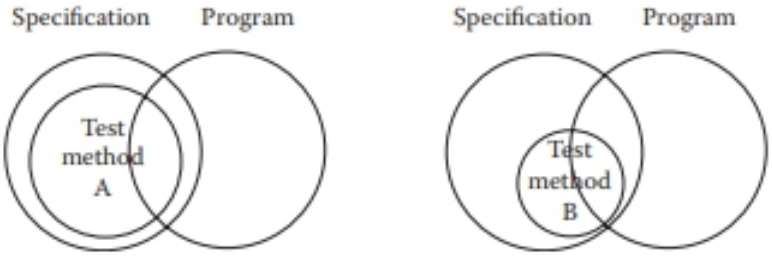
Dari urutan *term* ini, dapat dilihat bahwa *test case* menempati posisi sentral dalam pengujian. Proses pengujian dapat dibagi lagi menjadi langkah-langkah terpisah: *test planning, test case development*, menjalankan *test case*, dan mengevaluasi hasil pengujian.Untuk *test case* ada dua pendekatan mendasar digunakan untuk mengidentifikasi *test case*; secara tradisional, ini disebut pengujian fungsional dan struktural. *Specification-based* dan *code-based* adalah nama yang lebih deskriptif. Kedua pendekatan memiliki beberapa metode identifikasi *test case* yang berbeda; mereka umumnya hanya disebut metode pengujian.

2.1.1 *Specification-Based/Black-box Testing* [8]

Alasan bahwa pengujian berbasis spesifikasi pada awalnya disebut *functional testing* adalah bahwa setiap program dapat dianggap sebagai fungsi yang memetakan nilai dari domain input ke nilai dalam rentang outputnya. Gagasan ini umumnya digunakan dalam *engineering*, ketika suatu sistem dianggap sebagai *black box*. Ini mengarah pada istilah sinonim lainnya — pengujian *black-box*, di mana konten (implementasi) kotak hitam tidak diketahui, dan fungsi kotak hitam dipahami sepenuhnya dalam hal input dan outputnya (lihat Gambar 2.2). Sering kali, pengujian beroperasi sangat efektif dengan pengetahuan *black box*; pada kenyataannya, ini adalah pusat orientasi objek. Sebagai contoh, kebanyakan orang berhasil mengoperasikan mobil dengan hanya pengetahuan "*black box*".



Gambar 2.2: Engineer's black box.



Gambar 2.3: Comparing specification-based test case identification methods

Dengan pendekatan berbasis spesifikasi untuk menguji identifikasi kasus, satu-satunya informasi yang digunakan adalah spesifikasi perangkat lunak. Oleh karena itu, *test case* memiliki dua keunggulan berbeda:

1. Mereka tidak tergantung pada bagaimana perangkat lunak diimplementasikan, jadi jika implementasi berubah, *test case* masih berguna.

2. Pengembangan *test case* dapat terjadi secara paralel dengan implementasi, sehingga mengurangi keseluruhan interval pengembangan proyek.

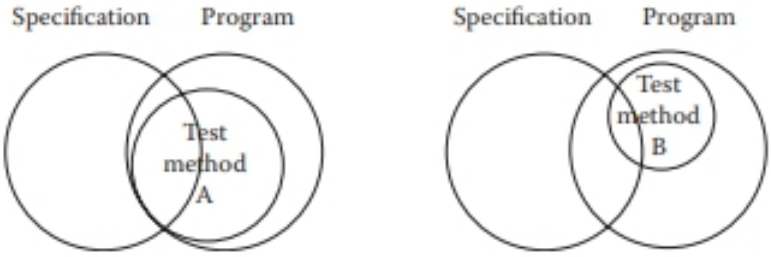
Di sisi negatif, kasus uji berbasis spesifikasi sering mengalami dua masalah, redudansi yang signifikan mungkin ada di antara kasus uji, dan diperparah oleh kemungkinan kesenjangan perangkat lunak yang tidak diuji.

Gambar 2.3 menunjukkan hasil *test case* yang diidentifikasi oleh dua metode berbasis spesifikasi. Metode A mengidentifikasi serangkaian kasus uji yang lebih besar daripada metode B. Perhatikan bahwa, untuk kedua metode, rangkaian kasus uji sepenuhnya terkandung dalam rangkaian perilaku tertentu. Karena metode berbasis spesifikasi didasarkan pada perilaku yang ditentukan, sulit untuk membayangkan metode ini mengidentifikasi perilaku yang tidak ditentukan.

2.1.2 *Code-Based/White-box Testing*[8]

Pengujian berbasis kode adalah pendekatan mendasar lainnya untuk menguji *test case*. Untuk membandingkannya dengan pengujian *black box*, kadang-kadang disebut pengujian *white box* (atau bahkan *clear box*). Metafora *clear box* mungkin lebih tepat karena perbedaan mendasar adalah bahwa implementasi (*black box*) diketahui dan digunakan untuk mengidentifikasi *test case*. Kemampuan untuk "melihat ke dalam" *black box* memungkinkan tester untuk mengidentifikasi *test case* berdasarkan bagaimana fungsi tersebut benar-benar dijalankan.

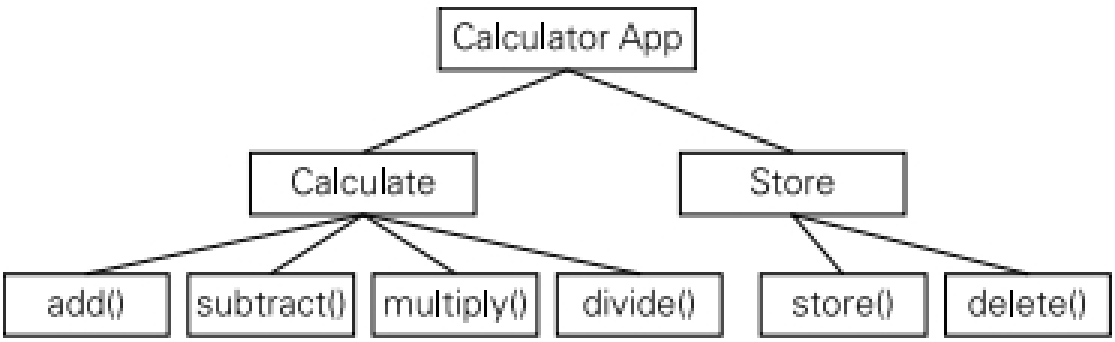
Pengujian berbasis kode telah menjadi subjek dari beberapa teori yang cukup kuat. Dengan konsep-konsep ini, tester dapat dengan ketat menggambarkan dengan tepat apa yang akan diuji. Karena dasar teorinya yang kuat, pengujian berbasis kode cocok untuk menjadi definisi dan penggunaan *test coverage metrics*. *Test coverage metrics* menyediakan cara untuk secara eksplisit menyatakan sejauh mana *item* perangkat lunak telah diuji, dan ini pada gilirannya membuat manajemen pengujian lebih jelas. Gambar 2.4 menunjukkan hasil *test case* yang diidentifikasi oleh dua metode berbasis kode. Seperti sebelumnya, metode A mengidentifikasi satu set kasus uji yang lebih besar daripada metode B. Apakah satu set kasus uji yang lebih besar tentu lebih baik? Ini adalah pertanyaan yang bagus, dan pengujian berbasis kode menyediakan cara-cara penting untuk mengembangkan jawaban. Perhatikan bahwa, untuk kedua metode, himpunan kasus uji sepenuhnya terkandung dalam himpunan perilaku yang diprogram. Karena metode berbasis kode didasarkan pada program, sulit membayangkan metode ini mengidentifikasi perilaku yang tidak diprogram. Sangat mudah untuk membayangkan, bagaimanapun, bahwa serangkaian kasus uji berbasis kode relatif kecil sehubungan dengan perilaku lengkap yang diprogram.



Gambar 2.4: Comparing code-based test case identification methods.

2.2 *Unit Testing*[10]

Tahap paling dalam pada pengujian perangkat lunak adalah *unit testing*.Dalam *unit testing*, kita menguji setiap unit kode secara individu, biasanya sebuah *method*, dalam isolasi untuk melihat apakah jika diberikan suatu kondisi tertentu, apakah respons yang didapat akan sama dengan yang diharapkan (lihat Gambar 2.5). Memecah pengujian ke tingkat dasar memberi keyakinan bahwa setiap bagian dari aplikasi akan berperilaku seperti yang diharapkan dan memungkinkan untuk menutupi kasus di mana hal yang tidak terduga dapat terjadi dan menanganinya dengan cara yang tepat.



Gambar 2.5: Contoh struktur aplikasi yang menunjukkan kelas dan method. Method adalah "unit" yang akan diuji.

Pada contoh diatas, *method* yang disorot adalah unit individual dari aplikasi ini yang perlu diuji. Jika tahu bahwa masing-masing *method* pada kelas *Calculator* berfungsi seperti yang diharapkan, maka kita yakin bahwa fitur-fitur aplikasi *Calculator* tersebut telah berjalan sesuai harapan.

Misalnya, kita ingin menguji apakah hasil dari *method* dengan dua angka benar-benar menambahkannya untuk menghasilkan jumlah yang benar. Memecah kode menjadi *unit-unit* ini membuat proses pengujian lebih mudah. Saat berurusan dengan *unit* kecil dari sebuah aplikasi, kita memiliki pemahaman yang jelas tentang cara kerja unit tersebut dan hal-hal yang memungkinkan untuk salah pada potongan kode tertentu, sehingga memungkinkan untuk menutupi *unit* dengan tes yang sesuai.

Selain itu, saat melakukan pengujian dengan cara ini biasanya jelas jika kita telah memecah kode menjadi unit-unit. Jika harus menulis banyak tes berbeda untuk mencakup semua kemungkinan berbeda yang dapat dilalui *method* ini, *method* tersebut mungkin terlalu besar dan harus dipertimbangkan untuk melakukan *refactoring* menjadi dua metode atau lebih dengan tanggung jawab yang sama. Sebaliknya, mungkin ada kasus di mana *method* terlalu sederhana dan dapat dikombinasikan dengan beberapa fungsi lain untuk membuat *method* yang lebih berguna. Sebagai

seorang *programmer* yang berpengalaman, kita harus mulai merasakan metode mana yang yang sudah "berukuran" baik dan mana yang tidak. Sepuluh baris sering merupakan aturan praktis yang baik untuk diikuti. Sebagai programmer yang baik, kita harus berusaha untuk memberikan kode yang paling mudah dibaca.

Tes yang ditulis adalah cerita yang menjelaskan kode dari sebuah program. Apa yang ingin dibaca atau dilihat ketika pertama kali membaca kode dan mencoba memahami apa fungsinya? Konvensi penamaan variabel yang jelas, ringkas, nama kelas, nama file, dan tes dapat membantu membuat kode lebih jelas dan mudah dipelihara untuk orang lain.

2.3 *Behavior-Driven Development*[11]

Behavior-Driven Development (BDD) adalah seperangkat praktik rekayasa perangkat lunak yang dirancang untuk membantu tim membangun dan memberikan perangkat lunak yang lebih bernilai dan berkualitas lebih cepat. Ini mengacu pada *agile* dan *lean practices* termasuk, khususnya, *Test-Driven Development* (TDD) dan *Domain-Driven Design* (DDD). Tetapi yang paling penting, BDD menyediakan bahasa umum berdasarkan kalimat-kalimat sederhana dan terstruktur yang diekspresikan dalam bahasa Inggris (atau dalam bahasa asli para pemangku kepentingan) yang memfasilitasi komunikasi antara anggota tim proyek dan pemangku kepentingan bisnis.

Untuk lebih memahami cara kerja BDD, dengan BDD yang berevolusi dari TDD, kita harus membahasa sedikit tentang TDD. Test-Driven Development (TDD) adalah praktik pengembangan yang menggunakan *unit test* untuk menentukan, merancang, dan memverifikasi kode yang akan ditulis. Sebelum menerapkan fungsionalitas, *developer* menulis *unit test* yang sengaja digagalkan untuk menunjukkan bagaimana fungsi ini seharusnya bekerja. Pada saat yang sama, pengujian gagal ini juga membuktikan bahwa implementasi saat ini belum mendukung fungsionalitas yang baru. Baru setelah itu *developer* menulis kode program. Setelah *unit testing* berlalu, *developer* tahu bahwa fungsionalitas telah berhasil diimplementasikan. Pada tahap ini, mereka dapat meninjau kode mereka untuk merapikan dan menyempurnakan desain.

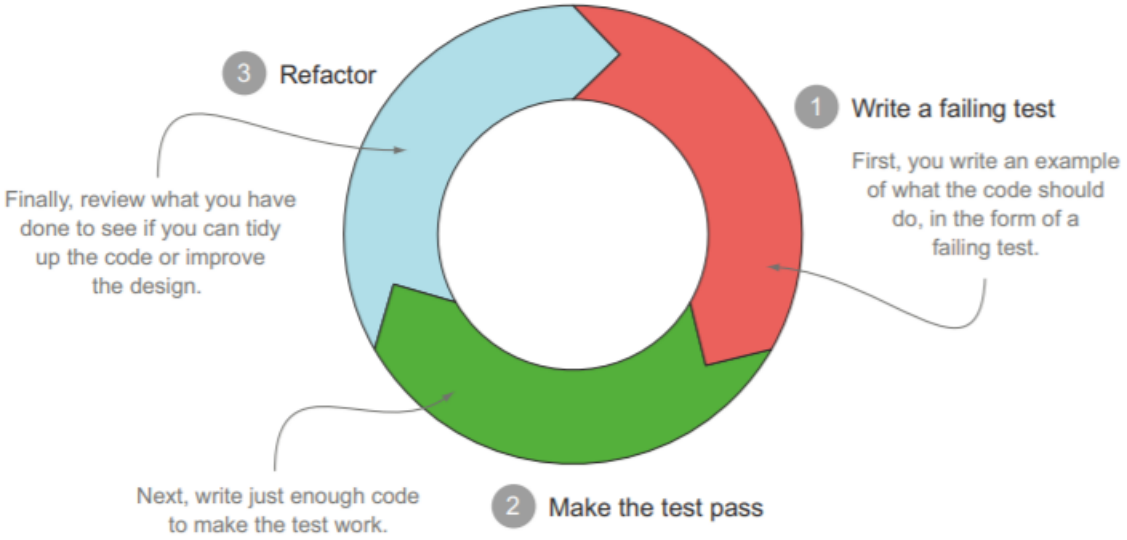
BDD pada awalnya dirancang sebagai versi perbaikan TDD. BDD awalnya ditemukan oleh Dan North pada awal hingga pertengahan 2000-an sebagai cara yang lebih mudah untuk mengajar dan mempraktikkan Test-Driven Development (TDD). TDD, ditemukan oleh Kent Beck pada awal munculnya *Agile Development*, itu adalah teknik efektif yang menggunakan *unit test* untuk menentukan, merancang, dan memverifikasi kode program.

Ketika pengguna TDD perlu mengimplementasikan fitur, mereka terlebih dahulu menulis tes gagal yang menjelaskan, atau menentukan fitur itu. Selanjutnya, mereka menulis kode yang cukup untuk lulus melakukan tes kode. Akhirnya, mereka memperbaiki kode untuk membantu memastikan bahwa kode itu akan mudah dirawat (lihat gambar 2.6). Teknik sederhana namun kuat ini mendorong pengembang untuk menulis kode yang lebih bersih, dirancang lebih baik, lebih mudah dirawat dan menghasilkan jumlah kecacatan kode yang jauh lebih rendah.

Terlepas dari kelebihannya, banyak orang masih kesulitan mengadopsi dan menggunakan TDD secara efektif. *Developer* sering mengalami kesulitan mengetahui di mana harus memulai atau tes apa yang harus mereka tulis selanjutnya. Terkadang TDD dapat menyebabkan *developer* menjadi terlalu fokus pada detail, kehilangan gambaran yang lebih luas tentang tujuan bisnis yang seharusnya mereka terapkan. Beberapa tim juga menemukan bahwa sejumlah besar *unit test* dapat menjadi sulit untuk dipertahankan karena ukuran proyek bertambah.

Faktanya, banyak *unit testing* tradisional, ditulis dengan atau tanpa TDD, secara erat digabungkan dengan implementasi kode tertentu. Mereka fokus pada metode atau fungsi yang mereka uji, bukan pada apa yang harus dilakukan kode dalam bisnis *term*.

BDD juga berfungsi dengan baik untuk *requirements analysis*. Bekerja dengan rekan analis bisnis Chris Matts, North mulai menerapkan apa yang telah ia pelajari ke ruang *requirements analysis*. Eric Evans memperkenalkan gagasan *Domain-Driven Design*, yang mempromosikan penggunaan bahasa yang dapat dimengerti dimanapun agar dapat dipahami oleh orang dari sisi



Gambar 2.6: Test-Driven Development three-phase cycle

bisnis untuk menggambarkan dan memodelkan suatu sistem. Visi North dan Matts adalah untuk menciptakan bahasa untuk dimengerti oleh siapapun yang dapat digunakan oleh analis bisnis untuk mendefinisikan persyaratan secara jelas, dan itu juga dapat dengan mudah diubah menjadi *acceptance tests* yang otomatis. Untuk mengimplementasikan visi ini, mereka mulai mengekspresikan *acceptance criteria* untuk *user stories* dalam bentuk contoh yang terstruktur, yang dikenal sebagai "skenario," seperti ini:

```
Given a customer has a current account
When the customer transfers funds from this account to an overseas account
Then the funds should be deposited in the overseas account
And the transaction fee should be deducted from the current account
```

Keyword dari skenario yaitu *Given,When* dan *Then*, maknda dari masing-masing *keyword* ialah:

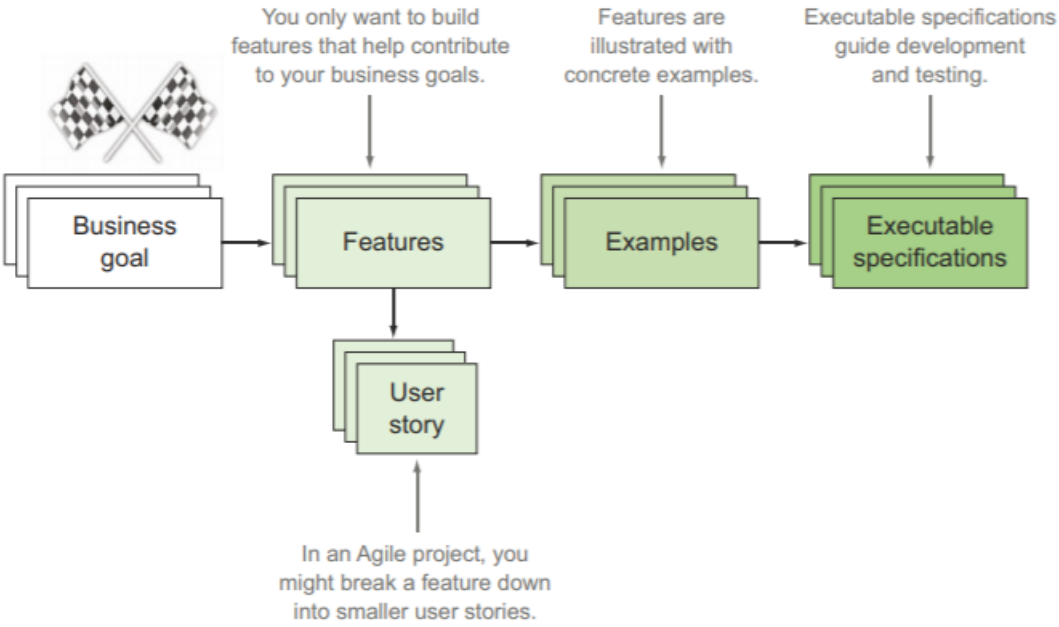
- *Given* menjelaskan prekondisi untuk skenario dan mempersiapkan *environment* untuk tes.
- *When* menjelaskan tindakan yang sedang dilakukan pada tes.
- *Then* menjelaskan hasil yang diharapkan.

Pemilik bisnis dapat dengan mudah memahami skenario yang ditulis seperti ini. Ini memberikan tujuan yang jelas dan obyektif untuk setiap cerita dalam hal apa yang perlu dikembangkan dan apa yang perlu diuji.

Notasi ini akhirnya berkembang menjadi bentuk yang umum digunakan, dan sering disebut sebagai Gherkin. Dengan *tools* yang tepat, skenario yang ditulis dalam formulir ini dapat diubah menjadi *acceptance criteria* yang otomatis dan dapat dieksekusi secara otomatis juga kapan pun diperlukan.

Ketika sebuah tim yang mempraktikkan BDD memutuskan untuk mengimplementasikan sebuah fitur, mereka bekerja bersama dengan pengguna dan *stakeholders* lainnya untuk mendefinisikan *user stories* dan skenario tentang apa yang diharapkan oleh pengguna untuk diberikan oleh fitur ini. Secara khusus, para pengguna membantu mendefinisikan sekumpulan contoh konkret yang menggambarkan *key outcome* dari fitur tersebut. (Lihat Gambar 2.7)

Examples ini menggunakan kosakata umum dan dapat dengan mudah dipahami oleh pengguna dan anggota tim *developer*. Mereka biasanya diekspresikan menggunakan Notasi *Given ... When ... Then* yang sama seperti contoh tadi di paragraf ke 7. Misalnya, contoh sederhana yang menggambarkan fitur "*Transfer funds between a client's accounts*" mungkin terlihat seperti ini:



Gambar 2.7: Examples play a primary role in BDD, helping everyone understand the requirements more clearly

```
Scenario:  Transferring money to a savings account
Given I have a current account with 1000.00
And I have a savings account with 2000.00
When I transfer 500.00 from my current account to my savings account
Then I should have 500.00 in my current account
And I should have 2500.00 in my savings account
```

Examples memainkan peran utama dalam BDD, hanya karena mereka adalah cara yang sangat efektif untuk mengkomunikasikan persyaratan yang jelas, tepat, dan tidak ambigu. Spesifikasi yang ditulis dalam bahasa alami, ternyata, merupakan cara komunikasi yang sangat buruk, karena ada begitu banyak ruang untuk ambiguitas, asumsi, dan kesalahpahaman. Contohnya adalah cara yang bagus untuk mengatasi keterbatasan ini dan mengklarifikasi persyaratan. *Examples* adalah cara yang bagus untuk mengatasi keterbatasan ini dan mengklarifikasi persyaratan.

2.4 Bahasa Pemrograman HTML [13]

HyperText Markup Language (HTML) adalah bahasa pengkodean komputer yang digunakan untuk mengubah teks biasa menjadi teks aktif untuk ditampilkan dan digunakan di web dan juga untuk memberikan teks biasa, tidak terstruktur, jenis struktur yang diandalkan manusia untuk membacanya. Tanpa semacam struktur yang digunakan pada html, teks biasa hanya akan berjalan bersama tanpa apapun untuk membedakan satu untaian kata dari yang lain.

HTML terdiri dari penanda yang dikodekan yang disebut *tag* yang mengelilingi dan membedakan bit teks, yang menunjukkan fungsi dan tujuan dari teks yang diberi tag. Tag tertanam secara langsung dalam dokumen teks biasa di mana mereka dapat diinterpretasikan oleh perangkat lunak komputer. Tag HTML menunjukkan sifat sebagian konten dan memberikan informasi penting tentangnya. Tag berdiri sendiri tidak ditampilkan dan berbeda dari konten yang sebenarnya mereka sampaikan.

2.5 Bahasa Pemrograman PHP [12]

PHP (PHP Hypertext Pre-processor) adalah bahasa scripting yang disematkan HyperText Markup Language (HTML). Tujuan dari bahasa ini adalah untuk memungkinkan pembangunan halaman Web dinamis dengan cepat dan mudah. PHP bekerja bersama dengan server web dan dapat digunakan dengan berbagai sistem operasi, termasuk Microsoft Windows dan UNIX.

PHP berbeda dari Common Gateway Interface (CGI) lainnya, yang ditulis dalam bahasa seperti Perl atau C, karena mengharuskan membuat program terpisah, yang menghasilkan HTML. PHP berbeda karena tertanam dalam dokumen HTML dengan tag awal dan akhir khusus yang memungkinkan untuk keluar dan masuk dari PHP. Ini memberikan waktu tampilan halaman yang cepat, keamanan tinggi dan transparansi bagi pengguna. Dengan PHP kita dapat memperoleh semua yang dapat dicapai dengan menulis aplikasi CGI terpisah, seperti membuat halaman web yang dinamis, pemrosesan formulir dan penanganan file.

Sintaks PHP mirip dengan bahasa pemrograman C, C ++ dan Java. Jika memiliki pengetahuan tentang bahasa-bahasa ini, maka akan menemukan bahwa bahasa PHP sangat akrab. PHP sederhana dan mudah untuk dipahami, jangan takut untuk mempelajari bahasa ini.

Salah satu fasilitas terpenting dan terkuat dari PHP adalah kemampuannya untuk berinteraksi dengan berbagai macam basis data. Lebih dari 20 basis data yang berbeda saat ini didukung, memungkinkan pengembang PHP untuk membuat halaman Web dengan basis data yang mudah.

2.6 Codeigniter [13]

CodeIgniter adalah *framework* aplikasi web yang bersifat *open source* untuk bahasa PHP. CodeIgniter memiliki banyak fitur yang membuatnya menonjol dari *framework* lainnya. Tidak seperti beberapa *framework* PHP lain yang, dokumentasinya sangat menyeluruh dan lengkap, mencakup setiap aspek *framework*.

Di sisi pemrograman, CodeIgniter kompatibel dengan PHP4 dan PHP5, sehingga akan berjalan di sebagian besar web host di luar sana. CodeIgniter juga menggunakan pola desain *Model View Controller* (MVC), yang merupakan cara untuk mengatur aplikasi Anda menjadi tiga bagian berbeda: *model* — lapisan abstraksi basis data, *view* — file tampilan depan, dan *control* — logika bisnis aplikasi. Pada intinya, CodeIgniter juga memanfaatkan pola desain *Singleton* secara ekstensif. Ini adalah cara untuk memuat kelas sehingga jika mereka dipanggil beberapa kali, instance kelas yang sama akan dikembalikan. Ini sangat berguna untuk koneksi basis data, karena hanya ingin satu koneksi setiap kali kelas digunakan.

2.7 Code Coverage

Code Coverage adalah metrik yang digunakan untuk mengkarakterisasi tingkat di mana beberapa kode aplikasi telah dieksekusi setelah dilakukannya tes. Ini dihitung dengan hasil berbentuk persentase, dihitung dengan pembagian $\frac{Codeexec}{Coderegion}$, menunjukkan jumlah kode sumber yang sudah dieksekusi ($\text{Code}_{\text{exec}}$) berkenaan dengan jumlah total kode sumber yang harus dilakukan ($\text{Code}_{\text{region}}$)[16].

Semakin tinggi *code coverage*-nya, semakin besar jumlah kode yang dikerjakan selama eksekusi pengujian sehubungan dengan *region* kode yang diuji. Meskipun telah ditunjukkan bahwa cakupan kode yang tinggi tidak selalu berarti deteksi *bug* yang tinggi. Namun, tidak mungkin untuk menyatakan apa pun tentang area kode yang tidak pernah dijalankan. Dengan demikian, *code coverage* yang digunakan sebagai relatif *measure* berfungsi sebagai indikator yang cocok untuk pengujian yang baik[16].

Secara umum, *code coverage* dilakukan dengan menginstruksikan kode aplikasi dan kemudian memperoleh data cakupan dari bagian instrumentasi. Ada beberapa alat yang menyediakan informasi seperti itu, misalnya, JaCoCo4. Namun, dalam beberapa konteks, peneliti tidak diizinkan untuk melakukan instrumentasi, karena pengujian tidak dilakukan di tingkat pengembangan. Dalam kasus

seperti itu, hanya mungkin untuk melakukan proses pemantauan yang tidak perlu memodifikasi aplikasi yang sudah di-*compile*. Untuk Android, kami menggunakan ddmlib5 *tracer* untuk melihat ketika metode dipanggil saat melakukan eksekusi *test case*[16].

Ada beberapa kriteria untuk melakukan *code coverage*, yaitu [17] :

- *Statemen Coverage* — Mengeksekusi pernyataan program secara individual dan mengamati semua hasilnya. *Statemen coverage* 100% berarti semua pernyataan telah dieksekusi setidaknya satu kali.
- *Branch Coverage* — Mengeksekusi *branch* yang ada pada *statement* program. *Branch coverage* 100% berarti semua branch yang ada di program sudah berhasil di eksekusi setidaknya satu kali.
- *Condition coverage* — Mengeksekusi semua kondisi yang ada pada statement kode program. *Condition coverage* 100% berarti semua kemungkinan kombinasi nilai dari kondisi yang mempengaruhi jalur telah dieksplorasi dalam pengujian.
- *Branch&Condition Coverage* — Gabungan dari *branch coverage* dan juga *condition coverage*.
- *Path Coverage* — Mengekesekusi semua kemungkinan kombinasi pada kode program, bisa dibilang ini gabungan semua dari kriteria sebelumnya. *Path coverage* 100% berarti semua kode pada program sudah berhasil dieksekusi dan semua kemungkinan kode program sudah berhasil dilewati setidaknya satu kali.

LAMPIRAN A

KODE PROGRAM

Listing A.1: MyCode.c

```c

// This does not make algorithmic sense,
// but it shows off significant programming characters.

#include<stdio.h>

void myFunction( int input, float* output ) {
    switch ( array[i] ) {
        case 1: // This is silly code
            if ( a >= 0 || b <= 3 && c != x )
                *output += 0.005 + 20050;
            char = 'g';
            b = 2^n + ~right_size - leftSize * MAX_SIZE;
            c = (--aaa + &daa) / (bbb++ - ccc % 2 );
            strcpy(a,"hello $@?");
    }
    count = ~mask | 0x00FF00AA;
}

// Fonts for Displaying Program Code in LATEX
// Adrian P. Robson, nepsweb.co.uk
// 8 October 2012
// http://nepsweb.co.uk/docs/progfonts.pdf
```

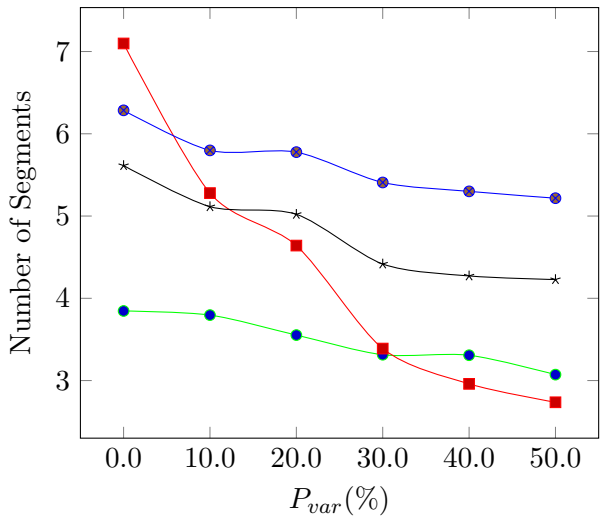
Listing A.2: MyCode.java

```java
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;

//class for set of vertices close to furthest edge
public class MyFurSet {
    protected int id;                                   //id of the set
    protected MyEdge FurthestEdge;                      //the furthest edge
    protected HashSet<MyVertex> set;                    //set of vertices close to furthest edge
    protected ArrayList<ArrayList<Integer>> ordered;    //list of all vertices in the set for each trajectory
    protected ArrayList<Integer> closeID;               //store the ID of all vertices
    protected ArrayList<Double> closeDist;              //store the distance of all vertices
    protected int totaltrj;                             //total trajectories in the set

    /*
     * Constructor
     * @param id : id of the set
     * @param totaltrj : total number of trajectories in the set
     * @param FurthestEdge : the furthest edge
     */
    public MyFurSet(int id,int totaltrj,MyEdge FurthestEdge) {
        this.id = id;
        this.totaltrj = totaltrj;
        this.FurthestEdge = FurthestEdge;
        set = new HashSet<MyVertex>();
        ordered = new ArrayList<ArrayList<Integer>>();
        for (int i=0;i<totaltrj;i++) ordered.add(new ArrayList<Integer>());
        closeID = new ArrayList<Integer>(totaltrj);
        closeDist = new ArrayList<Double>(totaltrj);
        for (int i = 0;i <totaltrj;i++) {
            closeID.add(-1);
            closeDist.add(Double.MAX_VALUE);
        }
    }

}
```

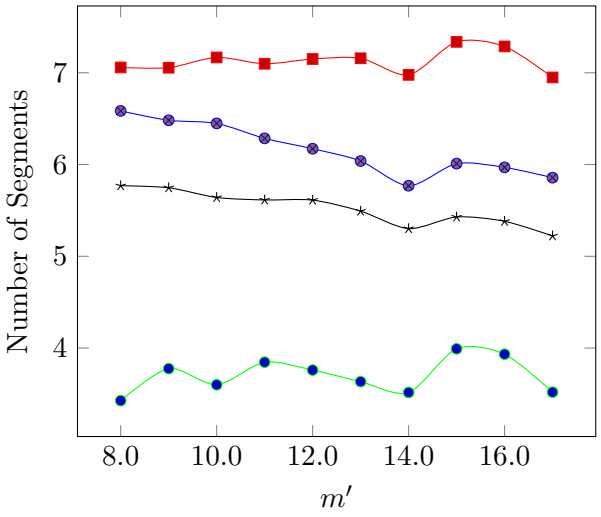
LAMPIRAN B

HASIL EKSPERIMEN

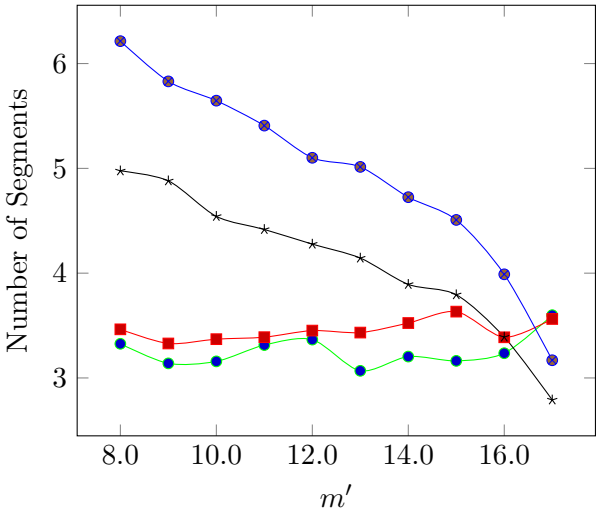
Hasil eksperimen berikut dibuat dengan menggunakan TIKZPICTURE (bukan hasil excel yg diubah ke file bitmap). Sangat berguna jika ingin menampilkan tabel (yang kuantitasnya sangat banyak) yang datanya dihasilkan dari program komputer.



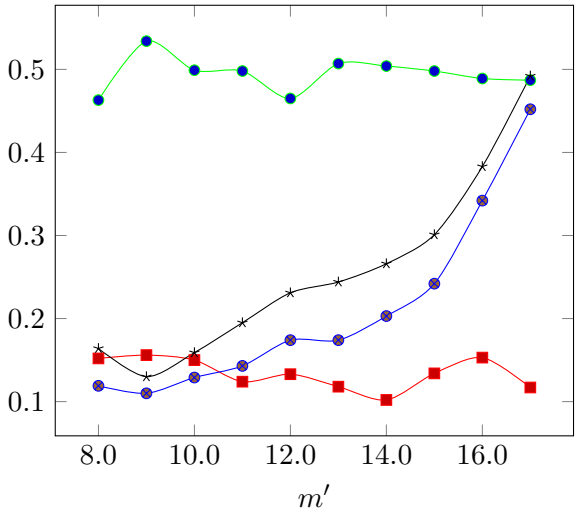
Gambar B.1: Hasil 1



Gambar B.2: Hasil 2



Gambar B.3: Hasil 3



Gambar B.4: Hasil 4