Evaluasi Deteksi *Smell Code* dan *Anti Pattern* pada Aplikasi Berbasis Java

http://dx.doi.org/10.28932/jutisi.v5i3.1981

Sendy Ferdian Sujadi ✉ #1

#S1-Sistem Informasi Bisnis, Universitas Kristen Maranatha
Jl. Prof. drg. Surya Sumantri No. 65, Bandung
1sendy.fs@it.maranatha.edu

Abstract — **This paper presents an evaluation result of smell code and anti-pattern detection in Java based application development. The main objective to be achieved in this research is to determine the proper way in the detection of smell code and anti-pattern in the development of Java based software, and to evaluate the impact of using code inspection tools and software metrics to refactoring code in Java based software development. Smell code to be detected in this research is Long Parameter List, Large Class, Lazy Class, Feature Envy, Long Method, and Dead Code. Anti-pattern that will be detected is The Blob / God Class and Lava Flow. The selection of smell code and anti-pattern is based on the definition, characteristics, detection factor, and software metrics. To support the research process is done through the evaluation stage of a case study Java based application as a sample for inspection of code for the detection of smell code and anti-pattern and calculation software metrics. Case studies of selected applications as sample applications are E-Commerce applications with functional master data management of goods and customers as well as management of sales and payment transactions. The detection of the smell code and anti-pattern on the case study is done in stages so it can be determined whether or not to refactor. As well as ensuring the technique of making the program better fit the characteristics and rules of object-oriented programming.**

Keywords— **Java; anti-pattern; smell code; software metrics; refactoring**

I. PENDAHULUAN

Seiring bertambahnya kompleksitas kelas, semakin sulit bagi pengembang baru untuk beradaptasi dengan pengembangan perangkat lunak, sehingga biaya perangkat lunak meningkat. Oleh karena itu, penting untuk menghasilkan kode yang sempurna dan mudah dimengerti [1]. Pengembang sering kali tidak memperhatikan kaidah bahasa pemrograman yang baik yang dapat menghasilkan *smell code* ataupun sering sekali menggunakan konsep *anti pattern* dikarenakan terfokus pada hasil fungsionalitas yang harus dicapai pada waktu yang sempit. Hal seperti ini yang menyebabkan kemungkinan implementasi yang buruk dan ketidaksempurnaan kode. Oleh karena itu perlu dilakukan inspeksi terhadap kode yang dirancang dengan buruk secara otomatis agar dapat dilakukan *refactoring* agar menghasilkan kode dengan rancangan yang baik.

Pemeriksaan kode hanyalah pendekatan di tingkat kode namun menjanjikan untuk mendeteksi kemungkinan kesalahan dalam perangkat lunak [2]. Alat untuk pemeriksaan kode [3] untuk berbagai bahasa pemrograman telah dikembangkan dan banyak digunakan untuk *real project*. Sebagian besar alat inspeksi kode berguna untuk mendeteksi kesalahan dalam pola yang telah ditentukan seperti CheckStyle [4], PMD [5], Jdeodorant [6], dan Dead Code Detector.

Fowler [7] menyarankan konsep *smell code* adalah struktur yang perlu dikeluarkan dari kode sumber oleh *refactoring* untuk meningkatkan kemampuan pemeliharaan perangkat lunak. *Smell code* didefinisikan dan diatur secara informal. *Smell code* itu sendiri bukanlah masalah tapi merupakan pertanda adanya masalah. Ini menunjukkan struktur dan kualitas produk perangkat lunak yang buruk. *Smell code* tidak sama dengan kesalahan sintaks atau peringatan kompilator. *Smell code* adalah indikasi buruknya disain program atau praktik pemrograman yang buruk. *Smell code* bukanlah kesalahan, tapi bisa membuat proyek perangkat lunak sulit dikembangkan dan dipelihara saat program membutuhkan modifikasi. *Smell code* bisa dilepas dengan menerapkan metode *refactoring*. *Smell code* adalah struktur program anomali yang mungkin menunjukkan masalah pemeliharaan perangkat lunak. *Smell code* mempengaruhi unit kode yang berbeda, seperti kelas dan method.

Anti-pattern [8] adalah solusi yang buruk untuk masalah desain yang berulang. *Anti-pattern* terjadi pada sistem berorientasi objek saat pengembang perangkat lunak tidak menggunakan saat merancang dan menerapkan kelas sistem mereka. Beberapa studi empiris telah menyoroti bahwa anti-pattern memiliki dampak negatif pada pemahaman dan maintenance sistem perangkat lunak. Akibatnya, identifikasi anti-pattern baru-baru ini mendapat perhatian dari para peneliti dan praktisi yang telah mengajukan berbagai pendekatan untuk mendeteksi *anti-pattern*.

Manajemen proses yang berhasil memerlukan perencanaan, pengukuran, dan kontrol. Dalam pengembangan program, persyaratan ini diterjemahkan ke dalam mendefinisikan proses pemrograman dalam serangkaian operasi, setiap operasi memiliki kriteria keluaran sendiri. Selanjutnya harus ada beberapa alat untuk mengukur kelengkapan produk pada setiap titik perkembangannya dengan inspeksi atau pengujian. Dan akhirnya, data terukur harus digunakan untuk mengendalikan proses. Pendekatan ini tidak hanya secara konseptual, namun telah berhasil diterapkan dalam beberapa proyek pemrograman yang mencakup sistem dan aplikasi pemrograman, baik besar maupun kecil. Hal ini mengaktifkan prediktabilitas yang lebih tinggi daripada cara lain dan penggunaan inspeksi telah meningkatkan produktivitas dan kualitas produk [9].

Disiplin *iron-clad* pada semua peraturan, yang dapat menghambat kerja pemrograman, tidak diperlukan, namun harus ada pemahaman yang jelas tentang fleksibilitas (atau ketidak lengkapan) dari masing-masing peraturan yang diterapkan pada berbagai aspek proyek. Contoh fleksibilitas mungkin menghapuskan aturan bahwa semua jalur utama akan diuji untuk kasus di mana pengujian berulang terhadap jalur yang diberikan secara logis tidak lebih dari menambahkan biaya. Contoh ketidaksempurnaan yang perlu adalah bahwa semua kode harus diperiksa. Prasyarat manajemen proses adalah serangkaian operasi yang jelas dalam prosesnya [9].

Analisis dan desain perangkat lunak berorientasi objek (OOAD) memberikan banyak manfaat seperti reusabilitas, penguraian masalah menjadi objek yang mudah dipahami dan membantu modifikasi masa depan. Namun siklus pengembangan perangkat lunak OOAD tidak lebih mudah daripada pendekatan prosedural yang khas. Oleh karena itu, perlu memberikan panduan yang dapat diandalkan yang dapat diikuti untuk membantu memastikan praktik pemrograman berorientasi objek yang baik dan menulis kode yang dapat diandalkan. Metrik pemrograman berorientasi objek adalah aspek yang harus dipertimbangkan. Metrik menjadi seperangkat standar yang dengannya seseorang dapat mengukur keefektifan teknik Analisis berorientasi objek dalam perancangan suatu sistem. Banyak metrik yang berbeda telah diusulkan untuk sistem berorientasi objek. Metrik berorientasi objek mengukur struktur prinsip yang, jika tidak dirancang dengan benar, berdampak negatif pada atribut kualitas desain dan kode. Metrik berorientasi objek yang ada terutama diterapkan pada konsep kelas, kopling, dan pewarisan [10].

Refactorings [7] adalah perubahan struktural yang dilakukan pada program yang melestarikan semantik program. Sementara banyak karya [7] [11] melihat bagaimana *refactoring* dapat memperbaiki struktur dan desain sebuah program, karya ini berfokus pada edisi kedua, yaitu bagaimana memeriksa bahwa perubahan tersebut adalah pelestarian semantik. Pelestarian perilaku penting karena jika tidak terjamin, program bisa menghasilkan hasil yang berbeda setelah terjadi perubahan. *Refactoring*, jika diterapkan dengan benar, sangat ideal untuk evolusi perangkat lunak, karena sudah diketahui bahwa tidak ada bug baru yang diperkenalkan.

Adapun rumusan masalah yang akan dibahas, yaitu: Bagaimana menentukan cara yang tepat untuk pendeteksian *smell code* dan *anti-pattern* pada pengembangan perangkat lunak berbasis Java dan Seberapa besar dampak penggunaan *code inspection tools* dan *software metrics* terhadap nilai kualitas perangkat lunak pada pengembangan perangkat lunak berbasis Java. Tujuan utama penelitian adalah menentukan cara yang tepat dalam pemeriksaan kode untuk mencari *smell code* dan *anti-pattern* pada pengembangan perangkat lunak berbasis Java, serta melakukan evaluasi dampak penggunaan *code inspection tools* dan *software metrics* terhadap nilai kualitas perangkat lunak.

Memproduksi kode yang dirancang dengan baik selama tahap pengembangan memiliki nilai yang signifikan karena proses ini membuat proyek perangkat lunak lebih mudah dipahami dan menghasilkan kualitas kode yang lebih tinggi. Akibatnya, biaya pemeliharaan proyek perangkat lunak akan menurun [1].

Pengembangan perangkat lunak dengan menggunakan konsep *Object Oriented Programming* (OOP) dan tidak menggunakan kaidah bahasa pemrograman yang baik yang akan menghasilkan *smell code* serta penggunaan *anti pattern* dapat memberikan dampak sulitnya proses pemeliharaan [12]. Inspeksi terhadap kode pada tahap pembuatan program dinilai dapat meningkatkan nilai kualitas pada pengembangan perangkat lunak tersebut serta mengurangi biaya pemeliharaan. Penggunaan *code inspection tools* untuk inspeksi terhadap kode pada pengembangan perangkat lunak dinilai perlu untuk dievaluasi. Hal ini dikarenakan banyaknya pengembang akan berpartisipasi dalam pengembangan perangkat lunak yang diwariskan daripada mengerjakan proyek baru maka sangat penting untuk menciptakan proyek yang dapat dimengerti. Memperbaiki *smell code* atau menghindari *anti pattern* pada pembuatan perangkat lunak selama tahap pengembangan akan membuat aplikasi lebih mudah dipahami. Akibatnya, pengembang akan menghasilkan lebih banyak pekerjaan dalam waktu yang lebih singkat dan pendekatan ini akan mengurangi biaya pemeliharaan perangkat lunak [1].

Hal ini membuat *code inspection tools* yang akan digunakan pada pengembangan perangkat lunak perlu untuk dievaluasi kemampuan inspeksi terhadap kode untuk mendeteksi adanya *smell code* atau *anti pattern* untuk selanjutnya di-*refactoring*. Penggunaan *code inspection tools* dapat berpengaruh baik atau buruk terhadap cara memprogram pengembang untuk menyelesaikan persoalan yang hendak diselesaikannya [1].

II. HIPOTESIS

Hipotesis yang ingin dievaluasi, yaitu:

1. *Smell code* dan *anti pattern* dapat dideteksi dengan tepat oleh inspeksi terhadap kode menggunakan *code inspection tools* dan perhitungan *software metrics*.
2. *Smell code* dan *anti pattern* akan dapat ditemukan dengan tepat dengan cara inspeksi terhadap kode oleh *code inspection tools* dan perhitungan software metrics agar teknik *refactoring* yang dilakukan terhadap perangkat lunak akan menghasilkan kode dengan rancangan yang baik.

III. PERANGKAT LUNAK PENGUKURAN METRIK PEMROGRAMAN BERORIENTASI OBJEK

Metrik pemrograman berorientasi objek adalah aspek yang harus dipertimbangkan. Metrik menjadi seperangkat standar yang dengannya seseorang dapat mengukur keefektifan teknik Analisis berorientasi objek dalam perancangan suatu sistem. Banyak metrik yang berbeda telah diusulkan untuk sistem berorientasi objek. Metrik berorientasi objek mengukur struktur prinsip yang, jika tidak dirancang dengan benar, berdampak negatif pada atribut kualitas desain dan kode. Metrik berorientasi objek yang ada terutama diterapkan pada konsep kelas, kopling, dan pewarisan [10].

A. Weighted Methods per Class (WMC)

WMC adalah penghitungan method yang diterapkan di dalam kelas atau jumlah kompleksitas method (kompleksitas method diukur dengan kompleksitas cyclomatic). Pengukuran kedua sulit dilaksanakan karena tidak semua method dapat dinilai dalam hierarki kelas karena warisan [10].

Definisi: Pertimbangkan Kelas C1, dengan methods M1...Mn yang didefinisikan di kelas. Biarkan c1...cn menjadi kompleksitas methods. Kemudian:

$$\text{WMC} = \sum_{i=1}^{n} ci \quad \text{.....(1)}$$

Jika semua kompleksitas method dianggap kesatuan, maka WMC = n, jumlah method. Dasar teoritis: WMC berhubungan langsung dengan definisi kompleksitas Bunge tentang kompleksitas, karena method adalah sifat kelas objek dan kompleksitas ditentukan oleh kardinalitas rangkaian sifatnya. Oleh karena itu, jumlah method adalah ukuran definisi kelas dan juga atribut kelas, karena atribut sesuai dengan properti [13].

Sudut pandang: [10][13]

1. Jumlah method dan kompleksitas method yang terlibat adalah prediksi berapa banyak waktu dan usaha yang dibutuhkan untuk mengembangkan dan memelihara kelas.
2. Semakin besar jumlah method di kelas, semakin besar dampak potensial pada subkelas, karena subkelas akan mewarisi semua method yang didefinisikan di kelas.
3. Kelas dengan sejumlah besar method cenderung lebih spesifik aplikasi, sehingga membatasi kemungkinan penggunaan ulang.

B. Depth of inheritance tree (DIT)

Sudut kelas dengan hierarki warisan adalah jumlah langkah maksimum dari simpul kelas ke akar dan diukur dengan jumlah kelas leluhur. Metrik pendukung untuk DIT adalah jumlah method yang diwarisi (NMI) [10].

Definisi: Kedalaman pewarisan kelas adalah metrik DIT untuk kelas. Dalam kasus yang melibatkan multiple inheritance, DIT akan menjadi panjang maksimum dari node ke akar subkelas [13].

Dasar teoritis: DIT berhubungan dengan gagasan Bunge tentang ruang lingkup properti. DIT adalah ukuran berapa banyak kelas leluhur yang berpotensi mempengaruhi kelas ini [13].

Sudut pandang: [13][10]

1. Semakin dalam sebuah kelas berada dalam hirarki, semakin besar jumlah method yang cenderung diwarisi, membuatnya lebih kompleks untuk memprediksi perilakunya.
2. Subkelas yang lebih dalam merupakan kompleksitas desain yang lebih besar, karena lebih banyak method dan kelas terlibat.
3. Semakin dalam kelas tertentu berada dalam hirarki, semakin besar kemungkinan penggunaan kembali method warisan.

C. Coupling Between Object Classes (CBO)

Coupling Between Object Classes (CBO) adalah hitungan dari jumlah kelas lain dimana sebuah kelas digabungkan. Hal ini diukur dengan menghitung jumlah hierarki kelas non-inheritance yang berbeda dimana kelas bergantung. Kopling yang kuat mempersulit sistem karena kelas lebih sulit dipahami, diubah atau diperbaiki dengan sendirinya jika saling terkait dengan kelas lainnya. Merancang sistem dengan kopling terlemah di antara kelas dapat mengurangi kompleksitas. Hal ini meningkatkan modularitas dan meningkatkan enkapsulasi [10]. CBO untuk kelas adalah hitungan jumlah kelas lain yang digabungkan [13].

Dasar teoritis: [13] CBO berkaitan dengan gagasan bahwa suatu objek digabungkan ke objek lain jika salah satunya bertindak di sisi lain, yaitu method penggunaan atau variabel contoh lainnya. Seperti yang dinyatakan sebelumnya, karena objek dari kelas yang sama memiliki sifat yang sama, dua kelas digabungkan bila method dideklarasikan dalam satu method penggunaan kelas atau variabel contoh yang didefinisikan oleh kelas lainnya.

Sudut pandang: [10][13]

1. Kopling yang berlebihan antara kelas objek sangat merugikan desain modular dan mencegah penggunaan kembali. Kelas yang lebih independen

adalah, semakin mudah untuk menggunakannya kembali di aplikasi lain.
2. Untuk meningkatkan modularitas dan meningkatkan enkapsulasi, pasangan kelas antar-objek harus dijaga seminimal mungkin. Semakin besar jumlah pasangan, semakin tinggi sensitivitas terhadap perubahan pada bagian desain lainnya, dan oleh karena itu perawatan lebih sulit.
3. Ukuran kopling berguna untuk menentukan seberapa rumit pengujian berbagai bagian dari suatu desain. Semakin tinggi kopling kelas antar objek, pengujian yang lebih ketat perlu dilakukan.

D. Lack of Cohesion in Methods (LCOM)

Lack of Cohesion (LCOM) mengukur ketidaksamaan method di kelas dengan variabel atau atribut. Modul yang sangat kohesif harus berdiri sendiri. Kohesi tinggi menunjukkan pembagian kelas yang baik. Kohesi tinggi menyiratkan kesederhanaan dan reusabilitas tinggi. (Cho, Kim, & Kim, 2001)

Definisi: Pertimbangkan Kelas C1 dengan n methods M1, M2..., Mn. Misalkan {Ij} = set of instance variables yang digunakan dengan method Mi. Ada n set seperti {I1}, ..., {In}. Misalkan P = { (Ii,Ij) | Ii ∩ Ij = Ø} dan Q = { (Ii,Ij) | Ii ∩ Ij _ Ø }. (Chidamber & Kemerer, 1994)

Jika semua n set {I_1},... {In} adalah Ø lalu misalkan P = Ø.

$$\text{LCOM} = |P| - |Q|, \text{if } |P| > |Q| \\ = 0 \text{ otherwise} 28 \quad \ldots\ldots (2)$$

Dasar teoritis ini menggunakan konsep tingkat kemiripan method. Tingkat kesamaan untuk dua method M1 dan M2 pada kelas C1 diberikan oleh [13]. σ() = {I1} ∩ {I2} di mana {I1} dan {I2} adalah himpunan variabel contoh yang digunakan oleh M1 dan M2.

LCOM adalah penghitungan jumlah pasangan method yang kesamaannya adalah 0 (yaitu σ() adalah himpunan nihil) dikurangi jumlah pasangan method yang kesamaannya tidak nol. Semakin besar jumlah method serupa, semakin kohesif kelas, yang konsisten dengan konsep tradisional tentang kohesi yang mengukur keterkaitan antara bagian program. Jika tidak ada method kelas yang menampilkan perilaku apa pun, yaitu tidak menggunakan variabel instan, mereka tidak memiliki kesamaan dan nilai LCOM untuk kelas akan menjadi nol [13].

Nilai LCOM memberikan ukuran sifat method yang relatif berbeda di kelas. Sejumlah kecil pasangan yang terputus-putus (unsur himpunan P) menyiratkan kesamaan method yang lebih besar. LCOM sangat terkait dengan variabel instance dan method kelas, dan oleh karena itu merupakan ukuran atribut dari kelas objek [13].

Sudut pandang: [13]

1. Kohesi method dalam kelas sangat diinginkan, karena mendorong enkapsulasi.
2. Kurangnya kohesi menyiratkan kelas mungkin harus dibagi menjadi dua atau lebih sub kelas.
3. Setiap ukuran ketidakteraturan method membantu mengidentifikasi kekurangan dalam desain kelas.
4. Kohesi rendah meningkatkan kompleksitas, sehingga meningkatkan kemungkinan kesalahan selama proses pembangunan.

E. Cyclomatic Complexity (CC)

Cyclomatic complexity adalah suatu software metric yang memberikan pengukuran logical complexity dari suatu program. Method ini menggunakan teori graph untuk menghitung kompleksitas program. Nilai kompleksitas dihitung dengan tiga cara, yaitu: [14][15]

1. Jumlah wilayah dari *flow graph* yang berhubungan dengan *cyclomatic complexity*.
2. *Cyclomatic complexity* V(G) didefiniskan dengan:
$$V(G) = E - N + 2 \quad \ldots\ldots (3)$$
E adalah jumlah *Edge*, dan N adalah jumlah *Node* dalam *flow graph*.
3. *Cyclomatic complexity* V(G) juga didefinisikan dengan rumus:
$$V(G) = P + 1 \quad \ldots\ldots (4)$$
P adalah jumlah predicated node pada flow graph G.

[16] mengatakan bahwa Myers mengusulkan metrik extended *cyclomatic complexity* V(G) turut memperhitungkan operator *boolean* (*AND* dan *OR*) dimasukkan dalam perhitungan nilai *cyclomatic complexity*.

F. Lines of Code (LOC)

Seperti namanya menunjukkan metrik LOC adalah ukuran modul dan ini mungkin merupakan metrik perangkat lunak tertua. Isu metrik LOC yang paling banyak dibahas adalah yang harus disertakan dalam ukuran [17] memeriksa empat aspek LOC yang harus dipertimbangkan, yaitu :garis kosong, baris komentar, deklarasi data, baris yang berisi beberapa instruksi terpisah.

Penggunaan praktis LOC dalam rekayasa perangkat lunak dapat diringkas sebagai berikut [18]:

1. Sebagai predikator pengembangan atau upaya pemeliharaan.
2. Sebagai kovarian untuk metrik lain, "menormalkan" mereka ke kepadatan kode yang sama.

Poin terpenting dari kedua penggunaan LOC ini, adalah fakta bahwa ia dapat digunakan sebagai penyesuaian kovarian untuk ukuran saat menggunakan metrik lainnya. Pertimbangkan dua kelas A dan B yang berbeda, di mana A memiliki nilai LOC jauh lebih tinggi daripada B. Jika seseorang ingin menghitung metrik WMC untuk kedua kelas, kemungkinan besar A mengembalikan nilai WMC tertinggi karena nilainya ukuran lebih besar Hasil ini terlalu bergantung pada ukuran kelas dan tidak memberikan representasi yang baik dari nilai WMC. Untuk memperbaiki ini, LOC dapat digunakan untuk menormalisasi kedua nilai WMC sehingga tidak bergantung pada ukuran tetapi hanya pada seberapa tinggi WMC yang dimiliki kedua kelas.

G. Source Line Of Code (SLOC)

SLOC digunakan untuk menghitung estimasi banyaknya resource yang akan dialokasikan dalam pembuatan sebuah perangkat lunak. SLOC menjadi standar pengukuran pada *metric* yang berbasis *size oriented* [15]. Besarnya nilai SLOC dapat mempengaruhi nilai *maintainability*. SLOC didapatkan dengan cara menghitung jumlah kode baris pada suatu kode sumber. Komen atau spasi pada kode program tidak dihitung saat perhitungan [14].

Alasan pemilihan metrik NCLoC, *halstead volume* dan *extended cyclomatic complexity* dalam perhitungan nilai *maintainability* adalah agar terciptanya pengukuran kuantitatif. Pengukuran kuantitatif tersebut dilakukan dengan menghitung banyaknya variabel yang digunakan, banyaknya alur eksekusi dalam kode program, dan banyaknya baris kode program [14].

H. Number of Parameters (NP)

Metrik ini menghitung jumlah parameter dalam *signature method* untuk *method* atau *konstruktor*. *Method* dengan sejumlah besar parameter seringkali lebih sulit untuk digunakan kembali karena cenderung khusus dan ini menyiratkan bahwa kelas hilang dari model. Batas *default* untuk NP adalah 0 sampai dengan 4. Jika jumlah parameter melebihi 4, disarankan untuk memindahkan *method* yang dipanggil atau melewati sebuah objek. Secara umum, *method* dan *konstruktor* dengan lebih dari tiga parameter lebih sulit dipahami dan digunakan. Kode sumber dengan *method* yang memiliki daftar parameter panjang dan *method invoking* dengan *signature* serupa cenderung rawan kesalahan.

I. Number of Attributes on Class (NOF)

Total jumlah atribut dalam lingkup yang dipilih. Digunakan untuk menyusun bagaimana kompleks datanya. Cara menghitungnya yaitu hitung atribut yang terdapat di kelas. Pengembang cenderung berpikir bahwa kelas dengan jumlah atribut yang lebih tinggi mengandung informasi yang lebih kompleks: ini sangat salah! Misalnya kelas dengan 10 atribut integer lebih mudah dipahami daripada kelas dengan 4 linked list.

J. Tight Class Cohesion (TCC)

Definisi: Penentuan ukuran kohesi kelas ini berdasarkan pada koneksi langsung pasangan method. Misalkan NP (C) adalah jumlah total pasangan *method abstrak* di kelas C. Jika ada n method di kelas C, maka: [19]

$$\mathrm{NP(C)} = (\mathrm{n} * (\mathrm{n} - 1))/2 \quad \ldots\ldots(5)$$

Dua method terhubung secara langsung jika mereka mengakses variabel contoh umum. Biarkan NDC (C) menjadi jumlah koneksi langsung untuk kelas C. *Metric Tight Class Cohesion* (TCC) adalah jumlah relatif *method* yang terhubung secara langsung. Secara formal hal ini dapat dinyatakan sebagai: [19]

$$\mathrm{TCC(C)} = (\mathrm{NDC(C)})/(\mathrm{NP(C)}) \quad \ldots\ldots(6)$$

Pengamatan: Nilai TCC dinormalisasi dalam kisaran antara [0,1]. Semakin tinggi nilai TCC untuk kelas, semakin kuat kohesi kelas tersebut, dan akibatnya semakin baik desainnya [19].

Kesimpulan: Kelas yang memiliki nilai TCC lebih rendah dari 0,5 (kadang-kadang 0,3) adalah kelas kandidat untuk proses perancangan ulang. Desain ulang terdiri dari kemungkinan pemisahan kelas dalam dua atau lebih kelas yang lebih kecil dan lebih kohesif. Dari perspektif lain subjek dengan TCC rendah mungkin menunjukkan kelas yang merangkum lebih dari satu fungsionalitas. Dengan kata lain, cacat desain yang dapat dideteksi dengan menggunakan TCC adalah kurangnya kohesi, dan satu cara konkret untuk mengurangi atau menghilangkannya mungkin merupakan pemecahan kelas [19].

K. Access to Foreign Data (ATFD)

Access to Foreign Data (ATFD) mewakili jumlah kelas eksternal dari mana kelas tertentu mengakses atribut, secara langsung atau melalui method aksesor. Method menggunakan secara langsung lebih dari beberapa atribut kelas lainnya [20].

IV. METODE PENELITIAN

Penelitian dilakukan melalui tahapan mempelajari teori terkait kaidah bahas pemrograman Java yang baik, *smell code*, *anti-pattern*, *object oriented programming*, *code inspection tools* yang akan digunakan, *refactoring*, dan *software metrics*; melakukan pendefinisian dan klasifikasi properti dari *smell code*, *anti-pattern*, *software metrics*, dan teknik *refactoring* untuk dianalisis lebih lanjut pada kode program dengan melakukan kode inspeksi oleh *code inspection tools* dan dibandingkan dengan *software metrics* lalu dilakukan *refactoring* agar dapat membandingkan nilai dari kualitas kode program tersebut; melakukan analisis dan perancangan karakteristik pengembangan aplikasi Java dan penggunaan *code inspection tools* untuk pemeriksaan kode yang akan digunakan dalam penelitian; evaluasi terhadap sebuah studi kasus aplikasi berbasis Java sebagai *sample* untuk di inspeksi kode-nya untuk pendeteksian *smell code* dan *anti pattern* serta perhitungan *software metrics*.

Studi kasus aplikasi yang dipilih sebagai aplikasi *sample* adalah aplikasi E-Commerce dengan fungsionalitas pengelolaan data master barang dan customer serta pengelolaan data transaksi penjualan dan pembayaran. Tahapan evaluasi ini dimulai dari pengembangan aplikasi *sample*, inspeksi kode dengan *code inspection tools* serta menghitung *software metrics* setelah itu dilakukan analisis dan *refactoring* untuk selanjutnya di inspeksi kembali dan seterusnya hingga didapat pola aplikasi sesuai kaidah bahasa pemrograman berorientasi objek yang baik dan arsitektur pemrograman Java yang baik.

Metrik produk perangkat lunak mengukur produk perangkat lunak pada tahap pengembangan yang berbeda, mulai dari mengukur kompleksitas perancangan perangkat

lunak hingga ukuran kode sumber akhir. Terukurnya *smell code* dan *anti-pattern* tergantung pada ukuran, kompleksitas, dan struktur *smell code* dan *anti-pattern*. beberapa *smell code* dan *anti-pattern* seperti "*Long Method*" dapat dengan mudah dideteksi oleh metrik perangkat lunak seperti kompleksitas *Cyclomatic* dan langkah-langkah *Halstead*. Beberapa *smell code* dan *anti-pattern* seperti "*Dead Code*" dan "*Middle Man*" sulit dideteksi oleh metrik perangkat lunak. Banyak *smell code* dan *anti-pattern* tampaknya tidak terdeteksi oleh metrik perangkat lunak. Properti *refactoring* menggambarkan solusi untuk setiap *smell code* dan *anti-pattern*. Beberapa *refactoring* bisa memecahkan beberapa *smell code* dan *anti-pattern*, dan beberapa *smell code* dan *anti-pattern* mungkin memerlukan beberapa metode *refactoring* untuk dihapus.

Untuk meningkatkan kualitas produk perangkat lunak, skenario terbaik adalah mengidentifikasi dan menghilangkan semua *smell code* dan *anti-pattern* dari kode sumber. Namun, pada kenyataannya, tidak semua *code inspection tools* dapat mengenali semua *smell code* dan *anti-pattern*. Misalnya, PMD dapat mengenali *smell code Dead Code* tetapi Checkstyle belum tentu dapat mengenali *smell code Dead Code*. Di sisi lain, tidak semua *smell code* dan *anti-pattern* memiliki tingkat kepentingan yang sama dengan kode sumbernya. Ada *tradeoff* antara biaya identifikasi / pemindahan *smell code* dan *anti-pattern* dan peningkatan kualitas perangkat lunak.

Kesimpulan ditarik dari hasil evaluasi studi kasus aplikasi Java dengan melihat nilai kualitas perangkat lunak dan membandingkan hasil *code metrics* pada saat aplikasi sebelum pengecekan kode dan sesudah *refactoring* kode program yang diminta setelah pengecekan kode.

Tabel I, II, III, IV, V dan VI berisikan definisi, cara mendeteksi dengan metrik perangkat lunak, cara melakukan *refactoring* dan *code inspection tools* yang dapat mendeteksi *smell code* dan *anti pattern*.

TABEL I
DEFINISI DARI *SMELL CODE YANG AKAN DI EVALUASI*

No.	*Smell Code*	Definisi
1	*Long Parameter List*	Terjadi saat penggabungan beberapa algoritma dalam 1 metode, dimana setiap algoritma butuh beberapa parameter yang berbeda-beda. Berarti sejumlah besar parameter dilanjutkan menjadi satu konstruktor atau statement dalam method yang membuat kode tidak konstan.
2	*Large Class*	Kelas berisi banyak variabel, method dan statement. Lebih banyak method, variabel atau kondisi pengambilan keputusan dalam satu kelas dapat mengurangi kohesi.
3	*Lazy Class*	Kelas yang hanya memiliki sedikit baris kode dan tidak memiliki method.
4	*Feature Envy*	Method mengakses objek data lain lebih banyak dari pada datanya sendiri. Menunjukkan method satu kelas yang nampaknya lebih ditujukan pada atribut kelas lain dari kelasnya sendiri.
5	*Long Method*	*Smell code* ini menandakan jumlah statement, variabel, kondisi pengambilan keputusan dan kondisi pengulangan yang banyak dalam satu method yang menyebabkan method terlalu panjang, sulit dimengerti dan digunakan kembali.
6	*Dead Code*	Variabel, parameter, atribut, method atau kelas tidak lagi digunakan dikarena sudah usang. Kode semacam itu juga dapat ditemukan dalam kondisi kompleks, ketika salah satu cabang tidak terjangkau karena kesalahan atau keadaan lainnya.

TABEL II
KLASIFIKASI DARI *SMELL CODE YANG AKAN DI EVALUASI*

No.	*Smell Code*	*SW Metrics*	*Refactoring*
1	*Long Parameter List*	NOP > 7	**Preserve Whole Object* **Replace Parameter with Method Object* **Introduce Parameter Object*
2	*Large Class*	a. If LOC > 300 & long methods > 5 b. If DIT > 5 c. If CBO > 10 **If there is 1 rule above then this smell code is detected*	**Extract Class* **Extract SubClass* **Extract Interface* **Duplicate Observed Data*
3	*Lazy Class*	a. If NOM = 0 b. If LOC < 100 & WMC <= 2 **If there is 1 rule above then this smell code is detected*	**Collapse Hierarchy* **Inline class*
4	*Feature Envy*	a. If CBO > 5 b. If LCOM > 2 **If there is a rule above then this smell code is detected*	**Move Method.* **Extract Method*

No.	*Smell Code*	*SW Metrics*	*Refactoring*
5	*Long Method*	a. NOP > 7 b. NLOC > 20 c. VG > 5 d. NBD > 6	**Extract Method* **Replace Temp with Query* **Preserve Whole Object* **Introduce Parameter Object* **Replace Method with Method Object* **Extract Method* **Decompose Conditional*
6	*Dead Code*	-	**Collapse Hierarchy* **Inline Class* **Rename Method* **Remove Parameter* **Subtitute Algorithm*

TABEL III
TOOLS DARI *SMELL CODE YANG AKAN DI EVALUASI*

No.	*Smell Code*	*Tools*
1	*Long Parameter List*	Checkstyle
2	*Large Class*	Tidak ada
3	*Lazy Class*	Tidak ada
4	*Feature Envy*	JDeodorant
5	*Long Method*	JDeodorant
6	*Dead Code*	Dead Code Detector, PMD, Elimination by IDE

TABEL IV
DEFINISI DARI *ANTI PATTERN YANG AKAN DI EVALUASI*

No.	Anti Pattern	Definisi
1	*The Blob / God Class*	Kelas tunggal yang memiliki banyak atribut dan method yang tidak terkait yang tercakup dalam satu kelas. Kurangnya arsitektur berorientasi objek dan terlalu terbatas intervensi.
2	*Lava Flow*	Method, kelas, atau atribut yang tidak terdokumentasi dan tidak jelas berhubungan dengan arsitektur sistem. Terdapat kode yang tidak terpakai atau usang. Seiring arus dan waktu, dengan cepat menjadi tidak mungkin untuk mendokumentasikan kode atau memahami arsitekturnya cukup untuk melakukan perbaikan.

TABEL V
KLASIFIKASI DARI *ANTI PATTERN YANG AKAN DI EVALUASI*

No.	*Anti Pattern*	*SW Metrics*	*Refactoring*
1	The Blob / God Class	a. WMC > 47 b. NOF > 60 c. ATFD > 5 d. TCC > 0.33	**Extract Class* **MVC Design Pattern*
2	Lava Flow	-	**Collapse Hierarchy* **Inline Class* **Rename Method* **Remove Parameter* **Subtitute Algorithm*

TABEL VI
TOOLS DARI *ANTI PATTERN YANG AKAN DI EVALUASI*

No.	*Anti Pattern*	*Tools*
1	The Blob / God Class	PMD
2	Lava Flow	Dead Code Detector, PMD, Elimination by IDE

V. PROPERTI *SMELL CODE*, *ANTI PATTERN*, *CODE INSPECTION TOOLS* DAN *SOFTWARE METRICS*

Banyak sifat *anti pattern* yang ada ditentukan oleh Brown, dkk [21]. Dalam penelitian ini properti yang dipilih adalah nama, penyebab, konsekuensi, gejala, dan *refactoring*. Properti lain seperti akar penyebab, variasi, latar belakang, dan bentuk umum tidak disertakan karena bergantung pada pengalaman pribadi pengembang perangkat lunak [21]. Properti *smell code* yang dipilih adalah nama, gejala, metrik, dan solusi atau refactoring. Gejala properti menggambarkan bagaimana menemukan *smell code*.

Mendefinisikan properti *refactoring* sebagai nama, skenario, dan mekanika. Nama refactoring biasanya terdiri dari operasi dan objek. Misalnya, untuk *refactoring* "*Remove Middle Man*", "*Remove*" adalah operasi sementara "*Middle Man*" adalah sebuah objek. Properti skenario memberikan deskripsi untuk setiap *refactoring* tentang kapan akan diterapkan. Properti mekanik menjelaskan bagaimana menerapkan metode langkah demi langkah untuk setiap *refactoring* untuk memecahkan masalah terkait. Penelitian ini berfokus pada refactoring yang dibutuhkan [7].

Metode pengujian tradisional tidak dapat secara efektif mengidentifikasi *smell code* atau *anti pattern*. Dan praktik *review* membaca baris demi baris untuk memverifikasi kualitas, kinerja, ukuran, dan produktivitas tidak efektif dalam hal waktu dan biaya. Proses ini membutuhkan waktu dari proses bisnis yang penting, memperlambat proses pengembangan dan menyebabkan cacat yang mampu menghasilkan masalah sistem secara keseluruhan di beberapa lapisan aplikasi. Cacat yang tidak diketahui termasuk beberapa lapisan aplikasi yang menyebabkan malapetaka dalam infrastruktur. Masalah yang tidak terdeteksi ini rata-rata 52% dari upaya yang diperlukan untuk memperbaiki masalah sistem [22]. Saat ini ada

sejumlah besar alat yang tersedia, yang secara otomatis memeriksa kode pengembang dan menyarankan perubahan pada kode untuk pemeliharaan atau kebenaran. Alat analisis kode statis bertujuan menyoroti kemungkinan kesalahan sebagai peringatan, yang dapat diperiksa oleh pengembang. Jika peringatan itu mungkin dan alat ini mampu melakukannya, maka bisa langsung memperbaikinya, jika diinginkan oleh pengguna [23].

Smell code dan *anti pattern* bisa dideteksi secara heuristik tergantung dari pengalaman seorang programmer atau melalui aplikasi metrik perangkat lunak [13][18]. Metrik properti adalah bagaimana metrik perangkat lunak digunakan untuk mengidentifikasi *smell code* dan *anti pattern*.

VI. Evaluasi Penggunaan *Code Inspection Tools* dan *Software Metrics* pada Pengembangan Perangkat Lunak

Evaluasi dilakukan dengan mendapatkan hasil kode inspeksi dan nilai perhitungan metrik yang dibutuhkan berdasarkan pencarian *smell code* dan *anti pattern*, setelah itu dilakukan *refactoring* untuk menghilangkan smell code dan anti pattern. Netbeans IDE digunakan untuk proses pembuatan studi kasus. Selain itu, IDE Netbeans ditambahkan dengan plugin Checstyle and Dead Code Detector untuk pemeriksaan kode dan Source Code Metrics untuk menghitung metrik perangkat lunak, yaitu WMC, TCC, NOM, NOF, LCOM, NOP, LOC, VG, dan NBD. Selain IDE Netbeans, IDE Eclipse juga ditambahkan dengan plugin PMD dan JDeodorant untuk pemeriksaan kode. Checkstyle dapat mendeteksi *Long Parameter Lists*, Dead Code Detector dapat mendeteksi *Dead Code / Lava Flow*, *PMD* dapat mendeteksi *God Class*, dan JDeodorant dapat mendeteksi *God Class, Long Method*, dan *Feature Envy*. Selain menghitung metrik perangkat lunak yang digunakan juga CKJM (Chidamber dan Kemerer Java Metrics) untuk menghitung metrik perangkat lunak DIT dan CBO. Deteksi kode dalam studi kasus dilakukan secara bertahap mulai dari PMD, JDeodorant, Checkstyle, dan Dead Code Detector terakhir.

Pola desain (MC)V, lapisan model dan conttoller disatukan. Tujuan penggunaan pola (MC)V adalah menjaga lapisan back-end dan front-end pada studi kasus. *Package* model berisikan objek data dan operasi untuk pengelolaan objek data tersebut dan *package view* yang terpisah serta dibangun dengan *God Class* dan *Large Class* untuk menerapkan *smell code* dan *anti pattern*.

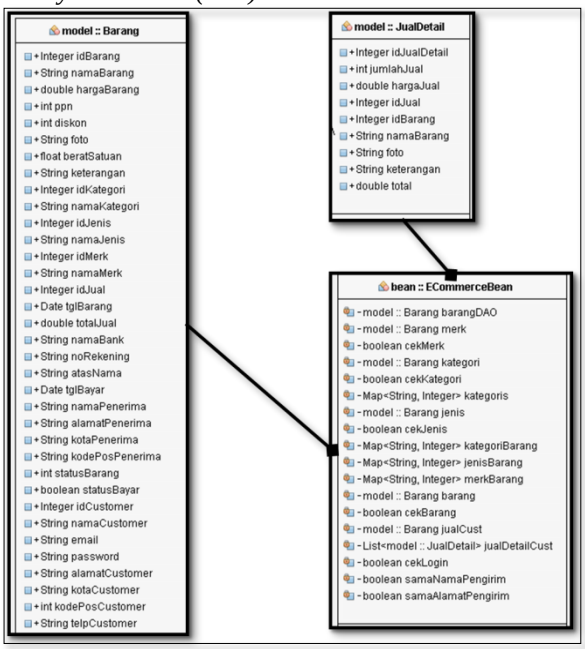
Evaluasi dan *refactoring* dilakukan bertahap pada studi kasus, dimulai dari inspeksi kode oleh *code inspection tools* dan perhitungan software metrics lalu dianalisis dan dilakukan perbaikan kode dengan teknik refactoring agar sesuai dengan kaidah bahasa pemrograman berorientasi objek yang baik hingga kode tidak terdeteksi *smell code* dan *anti pattern* dari kode *baseline*. Tabel VII adalah jenis *code inspection* dan jenis dari evaluasi yang dilakukan.

TABEL VII
Referensi Nomor, Code Inspection, Jenis Hasil Inspeksi Kode dan Perhitungan Software Metric Sub Bab A-E

No	*Code Inspection*	Jenis
1	The Blob / God Class	*Anti-Pattern*
2	Lava Flow	*Anti-Pattern*
3	Long Parameter List	*Smell Code*
4	Large Class	*Smell Code*
5	Lazy Class	*Smell Code*
6	Feature Envy	*Smell Code*
7	Long Method	*Smell Code*
8	Dead Code	*Smell Code*

A. *Hasil Inspeksi Kode dan Perhitungan Software Metric pada Code Baseline*

Gambar 1. *Class Diagram* Kode Sumber *Baseline* terdapat menggambarkan dua kelas yakni Barang dan JualDetail yang terdapat pada paket model yang berfungsi sebagai model data dan logika proses bisnis aplikasi serta satu kelas ECommerceBean yang terdapat pada paket bean yang berfungsi sebagai ManageBean yaitu kelas yang mengelola *passing* data ke XHTML dan pengelola alur data ke *Entity Java Bean (*EJB*)*.



Gambar 1. *Class Diagram* Kode Sumber *Baseline*

p-ISSN : 2443-2210
e-ISSN : 2443-2229

TABEL VIII
HASIL INSPEKSI KODE PADA SUMBER KODE *BASELINE E-COMMERCE*

No	Code Inspection Tools	
	Tools	**Hasil**
1	PMD	Kelas ECommerceBean dan Barang terdeteksi *God Class*
	JDeodorant	Kelas ECommerceBean dan Barang terdeteksi *God Class* dengan *refactoring Extract Class*
2	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi
3	Checkstyle	Kelas Barang terdeteksi *Long Parameter List*
4	-	Tidak terdeteksi
5	-	Tidak terdeteksi
6	JDeodorant	Kelas ECommerceBean terdeteksi *Feature Envy* dan *Refactoring Move Method*
7	JDeodorant	Kelas ECommerceBean dan Barang terdeteksi *Long Method*
8	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi

TABEL IX
HASIL PERHITUNGAN *SOFTWARE METRIC* PADA SUMBER KODE *BASELINE E-COMMERCE*

No	Software Metrics	
	Nilai Deteksi	**Nilai Hasil Pengukuran**
1	WMC > 47 NOF > 15 ATFD > 5 TCC < 0.33	ECommerceBean: terdeteksi *God Class* a. WMC = 60 b. NOF = 18 c. ATFD = 66 d. TCC = 0.15078901 Barang: terdeteksi *God Class* a. WMC = 109 b. NOF = 35 c. ATFD = 241 d. TCC = 0.07504004
2	-	Tidak terdeteksi
3	NOP > 7	Kelas Barang terdeteksi *Long Parameter List*
4	a. LOC > 300 & long methods > 5. b. DIT > 5 c. CBO > 10	Kelas Barang terdeteksi *Large Class*
5	a. NOM = 0 b. LOC < 100 & WMC <= 2	Kelas JualDetail terdeteksi *Lazy Class*
6	CBO > 5 LCOM > 2	Tidak terdeteksi
7	NOP > 7 NLOC > 20 VG > 5 NBD > 6	Tidak terdeteksi
8	-	Tidak terdeteksi

Tabel VIII dan IX menampilkan hasil dari inspeksi kode dan perhitungan software metric yang didapat dari kode sumber hasil Gambar 1. *Class Diagram* Kode Sumber *Baseline*. Hasil dari Tabel VIII dan IX dirangkun dalam Tabel X dengan menandakan terdeteksi atau tidak terdeteksi suatu *code inspection* pada sumber kode kelas yang dibuat.

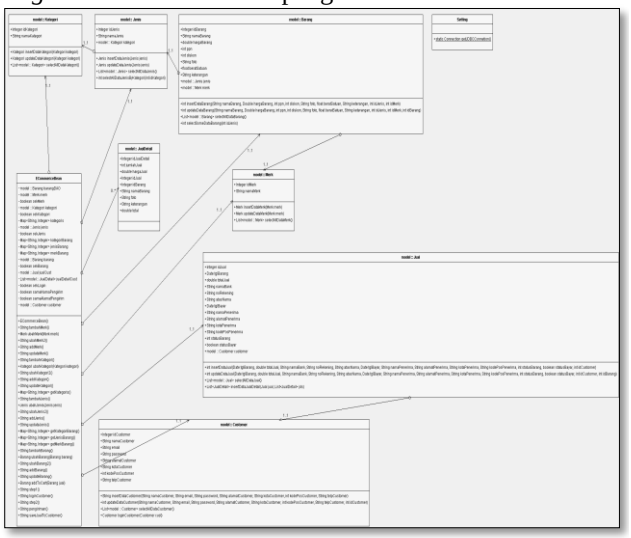
TABEL X
HASIL DARI SUMBER KODE *BASELINE E-COMMERCE*

No	Jenis	ECB	B	JD	Hasil Deteksi
1	T	1	1	0	2
	SW	1	1	0	2
2	T	0	0	0	0
	SW	0	0	0	0
3	T	0	1	0	1
	SW	0	1	0	1
4	T	0	0	0	0
	SW	0	1	0	1
5	T	0	0	0	0
	SW	0	0	1	1
6	T	1	0	0	1
	SW	0	0	0	0
7	T	1	1	0	2
	SW	0	0	0	0
8	T	0	0	0	0
	SW	0	0	0	0
				Total	11
				Rata-Rata	0.229166667

Keterangan:
1: Terdeteksi
0: Tidak Terdeteksi
T: Tools
SW: Software Metric
ECB: kelas ECommerceBean
B: kelas Barang
JD: kelas JualDetail

B. *Hasil Inspeksi Kode dan Perhitungan Software Metric pada Code Versi - 1*

Gambar 2. *Class Diagram* Kode Versi–1 ini menggambarkan hasil *Extract Class* untuk menanggulangi *God Class* dan *Replace Parameter with Method Object* untuk menanggulangi *Long Parameter Lists* pada *code baseline* bagian paket model. Dari *Extract Class* ini menjadi tujuh buah kelas yang terdapat pada paket model yang berfungsi sebagai model data dan logika proses bisnis aplikasi dan satu kelas ECommerceBean yang terdapat pada paket bean yang berfungsi sebagai kelas yang mengelola *passing* data ke XHTML dan pengelola alur data ke EJB.



Gambar 2. *Class Diagram* Kode Versi–1

TABEL XI
HASIL INSPKESI KODE PADA SUMBER KODE *VERSI-1 E-COMMERCE*

No	Code Inspection Tools	
	Tools	**Hasil**
1	PMD	Kelas ECommerceBean terdeteksi *God Class*
	JDeodorant	Kelas ECommerceBean terdeteksi *God Class* dengan *refactoring Extract Class*;
2	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi
3	Checkstyle	1. Kelas Barang, Customer, Jual terdeteksi *Long Parameter List*
4	-	Tidak terdeteksi
5	-	Tidak terdeteksi
6	JDeodorant	Kelas ECommerceBean terdeteksi *Feature Envy* dan *Refactoring Move Method*
7	JDeodorant	Kelas ECommerceBean dan Barang terdeteksi *Long Method*
8	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi

TABEL XII
HASIL PERHITUNGAN *SOFTWARE METRIC* PADA SUMBER KODE *VERSI-1 E-COMMERCE*

No	Software Metrics	
	Nilai Deteksi	**Nilai Hasil Pengukuran**
1	WMC > 47 NOF > 15 ATFD > 5 TCC < 0.33	Kelas ECommerceBean: terdeteksi *God Class* a. WMC = 60 b. NOF = 19 c. ATFD = 66 d. TCC = 0.1
2	-	Tidak terdeteksi
3	NOP > 7	Kelas Barang, Customer, Jual terdeteksi *Long Parameter List*
4	LOC > 300 & long methods > 5 DIT > 5 CBO > 10	Tidak terdeteksi
5	NOM = 0 LOC < 100 & WMC <= 2	Kelas JualDetail terdeteksi *Lazy Class*
6	CBO > 5 LCOM > 2	Tidak terdeteksi
7	NOP > 7 NLOC > 20 VG > 5 NBD > 6	Tidak terdeteksi
8	-	Tidak terdeteksi

Tabel XI dan XII menampilkan hasil dari inspeksi kode dan perhitungan software metric yang didapat dari kode sumber hasil Gambar 2. *Class Diagram* Kode Versi–1. Hasil dari Tabel XI dan XII dirangkun dalam Tabel XIII dengan menandakan terdeteksi atau tidak terdeteksi suatu *code inspection* pada sumber kode kelas yang dibuat.

TABEL XIII
HASIL DARI SUMBER KODE *VERSI-1 E-COMMERCE*

No	Jns	ECB	K	J	M	B	C	Jl	JD	S	Hasil Deteksi
1	T	1	0	0	0	0	0	0	0	0	1
	SW	1	0	0	0	0	0	0	0	0	1
2	T	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0
3	T	0	0	0	0	1	1	1	0	0	3
	SW	0	0	0	0	1	1	1	0	0	3
4	T	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0
5	T	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	1	0	1
6	T	1	0	0	0	0	0	0	0	0	1

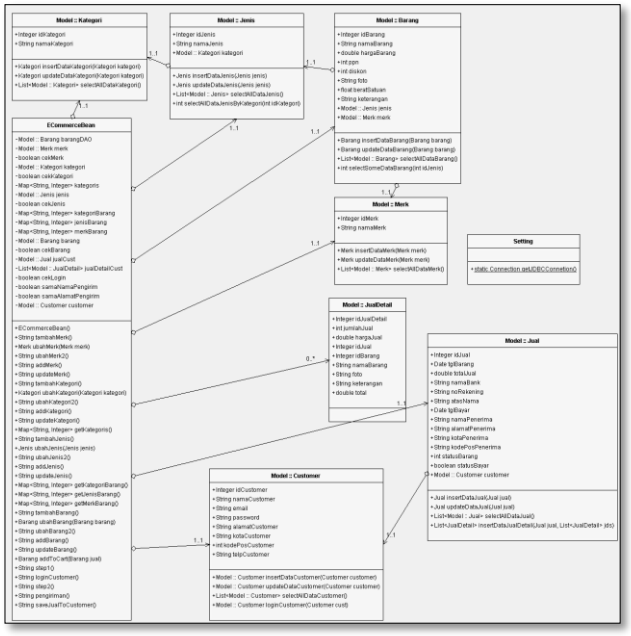
p-ISSN : 2443-2210
e-ISSN : 2443-2229

No	Jns	ECB	K	J	M	B	C	Jl	JD	S	Hasil Deteksi
	SW	0	0	0	0	0	0	0	0	0	0
7	T	1	0	0	0	1	0	0	0	0	2
	SW	0	0	0	0	0	0	0	0	0	0
8	T	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0
									Total		12
									Rata-Rata		0.083333333

Keterangan:	**Keterangan:**
1: Terdeteksi	M: kelas Merk
0: Tidak Terdeteksi	B: kelas Barang
T: Tools	C: kelas Customer
SW: Software Metric	Jl: kelas Jual
ECB: kelas ECommerceBean	JD: kelas JualDetail
K: kelas Kategori	S: kelas Setting
J: kelas Jenis	

C. Hasil Inspeksi Kode dan Perhitungan Software Metric pada Code Versi - 2

Gambar 3. *Class Diagram* Kode Versi–2 ini menggambarkan hasil penerapan *framework Java Persistence API* (JPA) untuk pengelolaan antara Java dan basis data serta untuk *refactoring Long Method* pada kelas *controller*. Dari *refactoring* ini terdapat dua kelas *controller* yang terdapat pada paket control yang berfungsi sebagai kelas yang mengelola *passing* data ke XHTML dan pengelola alur data ke EJB.



Gambar 3. *Class Diagram* Kode Versi–2

TABEL XIV
HASIL INSPKESI KODE PADA SUMBER KODE *VERSI-2 E-COMMERCE*

No	Code Inspection Tools	
	Nama Tools	**Hasil**
1	PMD	Kelas ECommerceBean terdeteksi *God Class*
	JDeodorant	Kelas ECommerceBean terdeteksi *God Class* dengan refactoring *Extract Class*;
2	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi
3	Checkstyle	Tidak terdeteksi
4	-	Tidak terdeteksi
5	-	Tidak terdeteksi
6	JDeodorant	Kelas ECommerceBean terdeteksi *Feature Envy* dan *Refactoring Move Method*;
7	JDeodorant	Kelas ECommerceBean dan Barang terdeteksi *Long Method*
8	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi

TABEL XV
HASIL PERHITUNGAN *SOFTWARE METRIC* PADA SUMBER KODE *VERSI-2 E-COMMERCE*

No	Software Metrics	
	Nilai Deteksi	**Nilai Hasil Pengukuran**
1	WMC > 47 NOF > 15 ATFD > 5 TCC < 0.33	Kelas ECommerceBean: terdeteksi *God Class* a. WMC = 60 b. NOF = 19 c. ATFD = 66 d. TCC = 0.1
2	-	Tidak terdeteksi
3	NOP > 7	Tidak terdeteksi
4	LOC > 300 & long methods > 5 DIT > 5 CBO > 10	Tidak terdeteksi
5	NOM = 0 LOC < 100 & WMC <= 2	Kelas JualDetail: terdeteksi *Lazy Class* NOM = 0 LOC = 19 & WMC = 0
6	CBO > 5 LCOM > 2	Tidak terdeteksi
7	NOP > 7 NLOC > 20 VG > 5 NBD > 6	Tidak terdeteksi
8	-	Tidak terdeteksi

Tabel XIV dan XV menampilkan hasil dari inspeksi kode dan perhitungan software metric yang didapat dari kode sumber hasil Gambar 3. *Class Diagram* Kode Versi–2. Hasil dari Tabel XIV dan XV dirangkun dalam Tabel XVI

dengan menandakan terdeteksi atau tidak terdeteksi suatu *code inspection* pada sumber kode kelas yang dibuat.

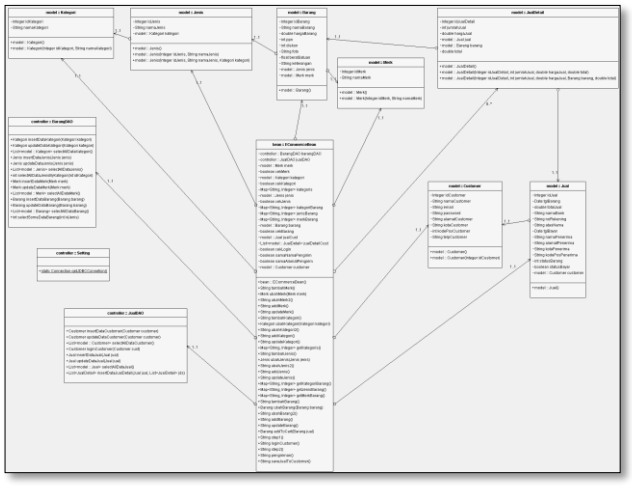
TABEL XVI
HASIL DARI SUMBER KODE *VERSI-2 E-COMMERCE*

No	Jns	ECB	K	J	M	B	C	Jl	JD	S	BD	JlD	Hasil Deteksi
1	T	1	0	0	0	0	0	0	0	0	0	0	1
	SW	1	0	0	0	0	0	0	0	0	0	0	1
2	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
3	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
4	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
5	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
6	T	1	0	0	0	0	0	0	0	0	1	0	2
	SW	0	0	0	0	0	0	0	0	0	0	0	0
7	T	1	0	0	0	0	0	0	0	0	0	0	1
	SW	0	0	0	0	0	0	0	0	0	0	0	0
8	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
											Total		5
											Rata-Rata		0.028409091

Keterangan:	**Keterangan:**
1: Terdeteksi	M: kelas Merk
0: Tidak Terdeteksi	B: kelas Barang
T: Tools	C: kelas Customer
SW: Software Metric	Jl: kelas Jual
ECB: kelas ECommerceBean	JD: kelas JualDetail
KB: kelas KategoriBean	S: kelas Setting
K: kelas Kategori	BD: kelas BarangDAO
J: kelas Jenis	JlD: kelas JualDAO

D. Hasil Inspeksi Kode dan Perhitungan Software Metric pada Code Versi - 3

Gambar 3. *Class Diagram* Kode ini menggambarkan hasil pemisahan *ManagedBean* sesuai dengan fungsi *passing data* antar XHTML serta untuk *refactoring Feature Envy* dan *God Class* pada kelas *ManageBean*. Dari refactoring ini terdapat dua kelas *ManageBean* yang terdapat pada paket bean yang berfungsi sebagai kelas yang mengelola *passing* data ke XHTML dan pengelola alur data ke EJB.



Gambar 4. *Class Diagram* Kode Versi–3

TABEL XVII
HASIL INSPKESI KODE PADA SUMBER KODE *VERSI-3 E-COMMERCE*

No	Code Inspection Tools	
	Nama Tools	**Hasil**
1	PMD	Kelas ECommerceBean terdeteksi *God Class*
	JDeodorant	Kelas ECommerceBean terdeteksi *God Class* dengan *refactoring Extract Class*;
2	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi
3	Checkstyle	Tidak terdeteksi
4	-	Tidak terdeteksi
5	-	Tidak terdeteksi
6	JDeodorant	Kelas BarangDAO terdeteksi *Feature Envy*
7	JDeodorant	Kelas ECommerceBean dan BarangDAO terdeteksi *Long Method*
8	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi

TABEL XVIII
HASIL PERHITUNGAN *SOFTWARE METRIC* PADA SUMBER KODE *VERSI-3 E-COMMERCE*

No	Software Metrics	
	Nilai Deteksi	**Nilai Hasil Pengukuran**
1	a. WMC > 47 b. NOF > 15	Kelas ECommerceBean: terdeteksi *God Class*

p-ISSN : 2443-2210
e-ISSN : 2443-2229

No	Software Metrics	
	Nilai Deteksi	**Nilai Hasil Pengukuran**
	c. ATFD > 5 d. TCC < 0.33	a. WMC = 60 b. NOF = 19 c. ATFD = 66 d. TCC = 0.1
2	-	Tidak terdeteksi
3	NOP > 7	Tidak terdeteksi
4	a. LOC > 300 & long methods > 5. b. DIT > 5 c. CBO > 10	Tidak terdeteksi
5	a. NOM = 0 b. LOC < 100 & WMC <= 2	Tidak terdeteksi
6	a. CBO > 5 b. LCOM > 2	Tidak terdeteksi
7	a. NOP > 7 b. NLOC > 20 c. VG > 5 d. NBD > 6	Tidak terdeteksi
8	-	Tidak terdeteksi

Tabel XVII dan XVIII menampilkan hasil dari inspeksi kode dan perhitungan software metric yang didapat dari kode sumber hasil Gambar 4. *Class Diagram* Kode Versi–3. Hasil dari Tabel XVII dan XVIII dirangkun dalam Tabel XIX dengan menandakan terdeteksi atau tidak terdeteksi suatu *code inspection* pada sumber kode kelas yang dibuat.

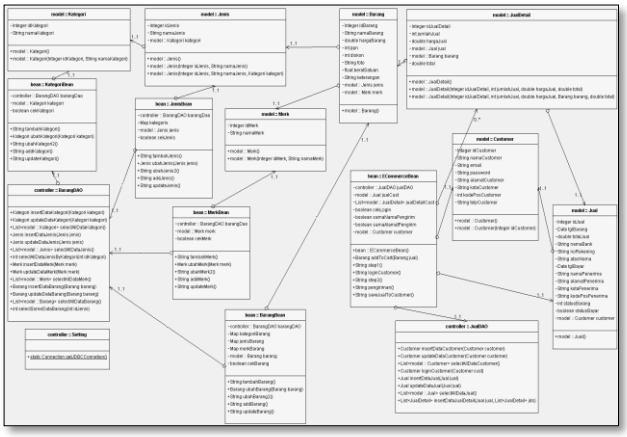
TABEL XIX
HASIL DARI SUMBER KODE KODE *VERSI-3 E-COMMERCE*

No	Jns	ECB	K	J	M	B	C	Jl	JD	S	BD	JlD	Hasil Deteksi
1	T	1	0	0	0	0	0	0	0	0	0	0	1
	SW	1	0	0	0	0	0	0	0	0	0	0	1
2	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
3	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
4	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
5	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
6	T	0	0	0	0	0	0	0	0	0	1	0	1
	SW	0	0	0	0	0	0	0	0	0	0	0	0
7	T	1	0	0	0	0	0	0	0	0	0	0	1
8	T	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0
												Total	4
												Rata-Rata	0.022727273

Keterangan:	**Keterangan:**
1: Terdeteksi	B: kelas Barang
0: Tidak Terdeteksi	C: kelas Customer
T: Tools	Jl: kelas Jual
SW: Software Metric	JD: kelas JualDetail
ECB: kelas ECommerceBean	S: kelas Setting
K: kelas Kategori	BD: kelas BarangDAO
J: kelas Jenis	JlD: kelas JualDAO
M: kelas Merk	

E. Hasil Inspeksi Kode dan Perhitungan Software Metric pada Code Versi - 4

Gambar 5. *Class Diagram* Kode Versi–4 ini menggambarkan hasil penerapan *framework Java Persistence API* (JPA) untuk pengelolaan antara Java dan basis data serta untuk *refactoring Long Method* pada kelas *controller*. Dari *refactoring* ini terdapat dua kelas *controller* yang terdapat pada paket control yang berfungsi sebagai kelas yang mengelola *passing* data ke XHTML dan pengelola alur data ke EJB.



Gambar 5. *Class Diagram* Kode Versi–4

TABEL XX
HASIL INSPKESI KODE PADA SUMBER KODE *VERSI-4 E-COMMERCE*

No	*Code Inspection Tools*	
	Nama *Tools*	**Hasil**
1	PMD	Tidak terdeteksi
	JDeodorant	Kelas BarangBean terdeteksi *God Class* dengan *refactoring Extract Class*
2	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi
3	Checkstyle	Tidak terdeteksi
4	-	Tidak terdeteksi
5	-	Tidak terdeteksi
6	JDeodorant	Kelas BarangDAO terdeteksi *Feature Envy*
7	JDeodorant	Kelas BarangDAO terdeteksi *Long Method*
8	Dead Code Detector, PMD, Elimination by IDE	Tidak terdeteksi

TABEL XXI
HASIL PERHITUNGAN *SOFTWARE METRIC* PADA SUMBER KODE *VERSI-4 E-COMMERCE*

No	*Software Metrics*	
	Nilai Deteksi	**Nilai Hasil Pengukuran**
1	WMC > 47 NOF > 15 ATFD > 5 TCC < 0.33	Tidak terdeteksi
2	-	Tidak terdeteksi
3	NOP > 7	Tidak terdeteksi
4	LOC > 300 & long methods > 5 DIT > 5 CBO > 10	Tidak terdeteksi
5	NOM = 0 LOC < 100 & WMC <= 2	Tidak terdeteksi
6	CBO > 5 LCOM > 2	Tidak terdeteksi
7	NOP > 7 NLOC > 20 VG > 5 NBD > 6	Tidak terdeteksi
8	-	Tidak terdeteksi

Tabel XX dan XXI menampilkan hasil dari inspeksi kode dan perhitungan software metric yang didapat dari kode sumber hasil Gambar 5. *Class Diagram* Kode Versi–4. Hasil dari Tabel XX dan XXI dirangkun dalam Tabel XXII dengan menandakan terdeteksi atau tidak terdeteksi suatu *code inspection* pada sumber kode kelas yang dibuat.

TABEL XXII
HASIL DARI SUMBER KODE *VERSI-4 E-COMMERCE*

No	Jns	ECB	KB	JB	MB	BB	K	J	M	B	C	Jl	JD	S	BD	JlD	Hasil Deteksi
1	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
2	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
3	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
4	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	T	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
7	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
8	T	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SW	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
														Total			1
														Rata-Rata			0.00416666667

Keterangan:	**Keterangan:**
1: Terdeteksi	J: kelas Jenis
0: Tidak Terdeteksi	M: kelas Merk
T: Tools	B: kelas Barang
SW: Software Metric	C: kelas Customer
ECB: kelas ECommerceBean	Jl: kelas Jual
KB: kelas KategoriBean	JD: kelas JualDetail
JB: kelas JenisBean	S: kelas Setting
MB: kelas MerkBean	BD: kelas BarangDAO
BB: kelas BarangBean	JlD: kelas JualDAO
K: kelas Kategori	

Dikarenakan menggunakan teknik MVC *design pattern* maka pada kelas BarangDAO terdeteksi *feature envy* oleh JDeodorant. *Feature envy* menunjukkan *method* satu kelas yang nampaknya lebih ditujukan pada atribut kelas lain dari

posisi sekarang atau disebut juga *method* mengakses data objek lain lebih banyak dari pada datanya sendiri [12] [24]. Dikarenakan teknik MVC mengharuskan memisahkan Java kelas untuk model atau kelas yang berisi atribut dan *method setter-getter* dengan Java kelas yang mengelola proses bisnis dari kelas model yang berarti kelas memiliki *method* yang mengakses data objek lain yaitu Java kelas model maka *feature envy* yang di deteksi ini tidak ada dilakukan *refactoring* sebab sesuai dengan penggunaan MVC *design pattern* dan secara hasil perhitungan *software metrics* tidak terdeteksi adanya *feature envy*.

VII. KESIMPULAN DAN SARAN UNTUK MASA DEPAN

Berdasarkan hasil evaluasi, didapat kesimpulan *code inspection tools* PMD, Checkstyle, JDeodorant, dan Dead Code Detector dapat digunakan untuk inspeksi secara otomatis dengan kemapuan yang berbeda untuk setiap pendeteksiannya. Meskipun demikian dalam pendeteksian terhadap kode dalam mencari *smell code* dan *anti pattern* dari kode yang diinspeksinya ada baiknya penggunaan *code inspection tools* digunakan bersamaan dengan perhitungan *software metric* untuk mencari *smell code* dan *anti pattern* agar hasil lebih akurat dan dapat menentukan apakah perlu di-*refactoring* saat ini atau nanti atau memang tidak diperlukan *refactoring*. Setelah pendeteksian terhadap kode untuk menemukan *smell code* dan *anti pattern*, tidak semua hasil *smell code* dan *anti pattern* yang terdeteksi harus dilakukan *refactoring* sebab melalui hasil perhitungan nilai *software metric*-nya tidak menunjukkan adanya gejala terdapatnya *smell code* dan *anti pattern* serta penggunaan arsitektur atau pola design perangkat lunak yang digunakan diharuskan mengabaikan *refactoring* untuk *smell code* atau *anti pattern* yang terdeteksi. Contoh kasus pada evaluasi penelitian ini adalah MVC *design pattern* yang menngharuskan memisahkan fungsi *model* dan *controller* maka pada *controller* pasti akan menghasilkan *smell code feature envy*. Dari hasil deteksi kode sumber oleh *code inspection tools* PMD, Checkstyle, JDeodorant, dan Dead Code Detector juga didapat hasil:

1. Deteksi *God Class* dapat dilakukan oleh PMD dan JDeodorant dimana JDeodorant menampilkan hasil dengan memberikan saran untuk melakukan *refactoring Extract Class*.
2. Deteksi *Lava Flow*, *Large Class*, *Lazy Class* dan *Dead Code* tidak dapat dilakukan oleh *code inspection tools* yang digunakan.
3. Deteksi *Long Parameter List* dapat dilakukan oleh *Checkstyle*.
4. Deteksi *Feature Envy* dan *Long Method* dapat dilakukan oleh JDeodorant dimana JDeodorant menampilkan hasil deteksi *Feature Envy* dengan memberikan saran untuk melakukan *Refactoring Move Method*.

Pemilihan *smell code* dan *anti pattern* lebih lengkap lagi, seperti *smell code* yang terdiri dari 5 kategori, yaitu: Bloaters, Object-Orientation Abusers, Change Preventers, Dispensables, dan Couplers serta mencakup semua *anti pattern* yang terdapat pada buku AntiPatterns-Refactoring Software, Architectures, and Projects in Crisis [21] sangat disarankan untuk pengembangan penelitian kedepannya.

DAFTAR PUSTAKA

[1] S. Velioglu and Y. E. Selçuk, "An Automated Code Smell and Anti-Pattern Detection Approach," *Software Engineering Research and Applications*, pp. 271-275, 2017.

[2] M. Noro and A. Sawada, "Software Architecture and Specification Model for Customizable Code Inspection Tools," *Asia-Pacific Software Engineering Conference*, pp. 230-237, 2015.

[3] N. Rutar, C. B. Almazan dan J. S. Foster, "A Comparison of Bug Finding Tools for Java," *International Symposium on Software Reliability Engineering*, 2004.

[4] "Checkstyle," [Online]. Available: http://checkstyle.sourceforge.net/.

[5] pmd, "PMD/Java.," 2002. [Online]. Available: http://pmd.sourceforge.net/.

[6] nikolaos, "JDeodorant," [Online]. Available: https://users.encs.concordia.ca.

[7] M. Fowler, K. Beck, J. Brant, W. Opdyke and D. Roberts, Refactoring: improving the design of existing code, Addison-Wesley, 1999.

[8] F. Palomba, G. Bavota, R. Oliveto and A. D. Lucia, "Anti-Pattern Detection: Methods, Challenges, and Open Issues," 2014.

[9] E. M. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182 - 211, 1976.

[10] E. S. Cho, M. S. Kim and S. D. Kim, "Component Metrics to Measure Component Quality," *Software Engineering Conference, 2001. APSEC 2001. Eighth Asia-Pacific*, pp. 419-426, 2001.

[11] E. Tilevich and Y. Smaragdakis, "Binary refactoring: improving code behind the scenes," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference*, Saint Louis, 2005.

[12] S. Kaur and S. Singh, "Influence of Anti-Patterns on Software Maintenance: A Review," *ICAET*, vol. 8887, p. 975, 2015.

[13] S. R. Chidamber and C. F. Kemerer, "A metrics Suite for Object Oriented Design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476 - 493, 1994.

[14] D. W. Kurt, "The Software Maintainability Index Revisited," in *Idaho National Engineering and Environmental Laboratory*, 2001.

[15] R. S. Pressman, Software Engineering A Practitioner_s Approach, 7ed, 2010.

[16] F. R. Oppedijk, "Comparison of the SIG Maintainability Model," University of Amsterdam, 2008.

[17] N. E. Fenton, "Software Measurement: A Necessary Scientific Basis," in *IEEE Transactions on Software Engineering*, 1994.

[18] J. Rosenberg, "Some Misconceptions About Lines of Code," *Fourth International Software Metrics Symposium*, pp. 137-142, 1997.

[19] J. M. Bieman and B.-K. Kang, "Cohesion and Reuse in an Object-Oriented System," *ACM Symposium on Software Reusability*, 1995.

[20] F. A. Fontana, P. Braione and M. Zanoni, "Automatic detection of bad smells in code: An experimental assessment," *Journal of Object Technology*, vol. 11, no. 2, pp. 1-38, 2011.

[21] W. J. Brown, R. C. Malveau, H. W. ". McCormick and T. J. Mowbray, AntiPatterns, Refactoring Software, Architectures, and Projects in Crisis, Canada: John Wiley & Sons, Inc., 1998.

[22] CAST, "Code Analysis Tools," 2017. [Online]. Available: http://www.castsoftware.com/products/code-analysis-tools.

[23] J.-P. Ostberg and S. Wagner, "Do We Stop Learning from Our Mistakes When Using Automatic Code Analysis Tools An Experiment Proposal," *User Evaluation for Software Engineering Researchers (USER), 2012*, pp. 21-24, 2012.

[24] sourcemaking.com, "Code Smells," sourcemaking.com, 2007. [Online]. Available: https://sourcemaking.com/refactoring/smells.

[25] S. Kaur and D. R. Maini, "Analysis of Various Software Metrics Used To Detect Bad Smells," *The International Journal Of Engineering And Science (IJES)*, pp. 15-19, 2016.