Progresif: Jurnal Ilmiah Komputer
Jl. Ahmad Yani, K.M. 33,5 - Kampus STMIK Banjarbaru
Loktabat – Banjarbaru (Tlp. 0511 4782881), e-mail: puslit.stmikbjb@gmail.com
e-ISSN: 2685-0877
p-ISSN: 0216-3284

Analisis *Dependency Injection* dan *Model-View-Presenter* Pada Aplikasi Berbasis *Android*

Kartarina[1*], Apriliansyah[2]
[1]Sistem Informasi/Teknik dan Perancangan, Universitas Bumigora
[2]Ilmu Komputer/Teknik dan Perancangan, Universitas Bumigora
[1,2]Jl. Ismail Marzuki, Mataram, Indonesia
**Email Corresponding Author*: kartarina@universitasbumigora.ac.id

Abstrak

Coupling merupakan salah satu indikator kualitas perancangan perangkat lunak terstruktur dan berorientasi objek. *Coupling* yang ketat akan sulit dirawat sehingga langkah awal mengendalikan *Coupling* yang baik adalah dengan menerapkan *Design Pattern*. Eksperimen penerapan *Design Pattern* membuktikan bahwa design pattern tidak menjamin pengkodean dapat lepas dari *Design Smell*. Masalah *Coupling* pada objek yang berkaitan dengan sumber data adalah ketika objek membutuhkan referensi objek *Android Context*. *Android* mengumumkan penggunaan *Hilt* sebagai standar dalam otomatisasi *Dependency Injection*. Penerapan *MVP* dapat membantu pengiriman referensi objek *Android Context* ke *Model*. Penelitian ini bertujuan menganalisis lebih lanjut untuk mengetahui hasil komparasi metrik kopling objek dari penerapan *Dependency Injection* secara manual dan otomatisasi pada aplikasi *Android*. Penelitian ini menghasilkan, otomatisasi penambahan jumlah kopling kelas pada *View* dan mengurangi kopling kelas pada Presenter. Selain itu, manfaat dari penerapan *Dependency Injection* secara otomatisasi mengurangi tanggung jawab *Class* pada *Presenter* dan menghilangkan *Design Smell* atau desain yang kompleks, walau begitu, penerapan *Dependency Injection* secara otomatisasi mempengaruhi pola *MVP* yang menggunakan *Dependency Injection* secara manual.

Kata kunci: *Coupling; Design Pattern; Dependency Injection; Model-View-Presenter*

Abstract

Coupling is one indicator of the quality for structured and object-oriented software design. Tight couplings will be difficult to maintain so the first step when controlling a good coupling is to apply a Design Pattern. Experiments on the application of Design Patterns prove that design patterns do not guarantee that coding can be separated from Design Smell. Coupling problem with objects related to data sources is that when the object requires an Android Context object reference. Android announced the use of Hilt as standard in Dependency Injection automation. Implementing MVP can help pass Android Context object references to Models. This aims to analyze further to find out the results of the comparison of object coupling metrics from the manual and automated application of Dependency Injection on Android applications. The results of the research, automatic increasing the number classes of coupling on the View and reduce class coupling on the Presenter. In addition, the benefits of implementing Dependency Injection automatically reduces class responsibility on Presenter and Design Smell or complex designs, however, automated Dependency Injection implementation affects the MVP pattern that uses Dependency Injection manually.

Keywords: *Coupling; Design Pattern; Dependency Injection; Model-View-Presenter*

1. Pendahuluan

Pada perancangan perangkat lunak, *Coupling* merupakan salah satu indikator kualitas perancangan perangkat lunak terstruktur dan berorientasi objek [1]. *Coupling* ketat menjadi indikasi awal sebuah perancangan perangkat lunak berorientasi objek sulit dirawat. Penelitian oleh Melton & Tempero [2], menunjukkan keberadaan *Coupling* ketat telah umum dalam perangkat lunak. *Coupling* ketat telah lama dianggap sebagai gejala kerusakan perangkat lunak

yang harus dikendalikan dengan baik dalam sistem perangkat lunak menurut (Parnas, Lakos, Fowler, Martin) dalam Oyetoyan [3]. Ini berarti bahwa *Coupling* tidak dapat dihilangkan melainkan dapat dikontrol dengan baik. Langkah awal dalam mengendalikan *Coupling* dengan baik adalah dengan menerapkan *Design Pattern* seperti yang dilakukan oleh Fowler [1]. Walau begitu, bukti eksperimental penerapan *Design Pattern* tidak menjamin basis kode pada perangkat lunak lepas dari *Design Smell* [4]. *Design Smell* berdampak negatif pada aspek pemeliharaan potongan kode dan pemahaman tentang program [5]. *Coupling* ketat dan *Design Smell* menjadi masalah dalam perangkat lunak yang harus dikendalikan dan dirawat dengan baik.

Fowler [1], mendeskripsikan bahwa *Coupling* sebagai hubungan antar komponen seperti *class, Object* dan *Method*, jika suatu komponen memerlukan perubahan maka komponen lain ikut berubah. Ini mengacu pada bagaimana komponen saling bergantung atau saling terkait. Dampak perubahan yang terjadi dapat diketahui dari ketat atau longgarnya suatu *Coupling* [6]. *Design Smell* menyebabkan kekakuan, kerapuhan, dan perancangan yang kompleks atau berlebihan [7], adalah dampak dari *Coupling* yang tidak dikendalikan dengan baik pada perangkat lunak. *Design Smell* berpengaruh dalam memahami bagian kecil dari sistem, modifikasi, dan penggunaan kembali dikemudian hari. Selain itu, distribusi perangkat lunak seperti kerangka kerja atau perpustakaan umum yang memiliki *Design Smell* berdampak negatif pada reputasi perusahaan atau organisasi.

Android menempati posisi pertama dengan 84.8% prakiraan pangsa pasar sistem operasi pengiriman *Smartphone* seluruh dunia [8], memberikan masalah pada kerangka kerja dasar yang digunakan dalam pengembangan aplikasi *Android.* Hal yang menjadi perhatian adalah *Coupling* antara *Android Context* sebagai objek independen dengan objek dependen lainya. Sebagai contoh, ketika sebuah objek yang berkaitan dengan sumber data membutuhkan objek *Android Context*, hal ini membuat setiap kelas saling mendukung untuk mengirim referensi objek (*Dependency Injection*) *Android Context* yang dibuat pada kelas *Activity (View)* ke sumber data *(Model)*. Penerapan *Model-View-Presenter* membantu pengiriman referensi objek *Android Context* ke *Model* secara runtunan, dimulai dari *View* kirim ke *Presenter* dan dari *Presenter* kirim ke *Model*. Walau begitu, kode yang ditulis menjadi kompleks dan berlebihan. Hingga pada bulan Mei 2021, *Android* mengumumkan penggunaan *Hilt* [9], sebagai standar untuk otomatisasi *Dependency Injection* [10], dalam menyelesaikan masalah *Coupling* dan *Design Smell*.

Penelitian tentang penerapan *Model-View-Presenter* sudah dilakukan oleh Baramuli et al [11], dengan hasil penelitian berupa kemudahan *Unit Test* pada basis kode antarmuka. Penelitian yang sama juga dilakukan oleh Prabowo et al [12], dengan hasil penelitian menunjukkan peningkatan pemeliharaan, modularitas, dan mereduksi kompleksitas. Penelitian lain yang dilakukan oleh Rizki et al [13], menunjukkan kinerja dari aplikasi yang menerapkan *Model-View-Presenter* lebih sedikit penggunaan memori daripada *Model-View-Controller* dan *Model-View-ViewModel*.

Dikarenakan *Coupling* dan *Design Smell* memiliki dampak pada kualitas perangkat lunak dan berangkat dari penelitian sebelumnya tentang *Model-View-Presenter* pada aplikasi *Android*, yang memberi dampak dalam *Unit Tes* [11], pemeliharaan [12], dan berdampak pada efisiensi penggunaan memori, maka penelitian ini melakukan analisis lebih lanjut pada tingkat komponen objek (*View, Model, Presenter*) dari penerapan *Dependency Injection* secara manual dan otomatisasi menggunakan *Hilt* pada *Model-View-Presenter*. Analisis ini fokus pada tingkat komponen karena penelitian sebelumnya yang dilakukan oleh Prabowo et al [12] berdasarkan metrik tingkat proyek yang melibatkan semua kode program. Analisis ini bertujuan untuk mengetahui hasil komparasi pada *coupling* pada desain aplikasi android yang menerapkan *Dependency Injection* secara Manual dan otomatisasi pada *Model-View-Presenter.*

2. Metodologi

Populasi dalam penelitian ini diambil dari hasil rancang bangun aplikasi. Sampel yang digunakan terdiri dari kelas dan paket. Banyaknya jumlah sampel ditentukan lewat hasil observasi komponen pada domain yang menerapkan *Dependency Injection* secara manual dan otomatisasi menggunakan *Hilt* pada *Model-View-Presenter*. Jumlah populasi sebanyak 197 kelas dan 84 paket diambil dari total kelas dan paket hasil rancang bangun aplikasi yang menerapkan *Dependency Injection* secara manual dan otomatisasi menggunakan *Hilt* pada *Model-View-Presenter*. Sampel yang digunakan dalam penelitian ini sebanyak 28 kelas dan 14

paket, diambil dari domain *Natural Language Processing task (NLP-task)* yaitu pada *View dan Presenter* dan *Repository* yang terdiri dari 14 kelas dan 7 paket sebelum dan sesudah penerapan *Hilt*. Domain *NLP Task* digunakan sebagai domain sampel karena merupakan fungsionalitas utama yang ada pada aplikasi dan tidak terjadi bias nilai sebelum dan sesudah menerapkan *Dependency Injection* secara otomatisasi menggunakan *Hilt*. *Repository* masuk dalam sampel dikarenakan sebagai sumber data yang berasosiasi dengan *NLP Task*. Hal ini, untuk memenuhi syarat dari polfa *Model-View-Presenter* yang terdiri dari *View*, *Presenter*, dan *Model*.

Jenis data yang digunakan dalam analisis data adalah data kuantitatif yang bersumber dari hasil metrik CK [14] dan Martin [15], pada basis kode aplikasi dengan bantuan alat metrik otomatisasi *MetricReloaded* dan *MetricsTree*. Jenis metrik yang digunakan dalam analisis data ditampilkan dalam Tabel 1 tentang rangkaian metrik yang digunakan dalam analisis *Coupling*.

Table 1 Rangkaian metrik dalam analisis *Coupling*

Nama Metrik	Maksud
Coupling between Object Classes (CBO)	Ukuran jumlah banyaknya kelas yang berasosiasi dengan kelas lain.
Response for a Class (RFC)	Ukuran jumlah metode yang digunakan dalam merespon pesan pada sebuah kelas.
Afferent Coupling (AC)	Ukuran jumlah total kelas eksternal yang digabungkan ke kelas paket karena *Coupling* masuk (independen).
Efferent Coupling (EC)	Ukuran jumlah total kelas eksternal yang digabungkan ke kelas paket karena *Coupling* keluar (dependen).
Instability (I)	Ukuran rasio ketidakstabilan antar paket antara jumlah total *Coupling* masuk dan keluar dari kelas di dalam paket dari/ke kelas di luar paket.

Metode penelitian yang digunakan dalam analisis data adalah komparatif kuantitatif, yaitu membandingkan dua buah data metrik sebelum dan sesudah penerapan *Dependency Injection* secara otomatisasi menggunakan *Hilt* pada *Model-View-Presenter*. Uji-t berpasangan digunakan untuk membandingkan dua buah data metrik. Menurut Yusuf Prabowo [15], Uji-t berpasangan digunakan ketika kedua populasi bersifat tidak bebas atau dalam makna lain bahwa keduanya berkaitan erat satu sama lain.

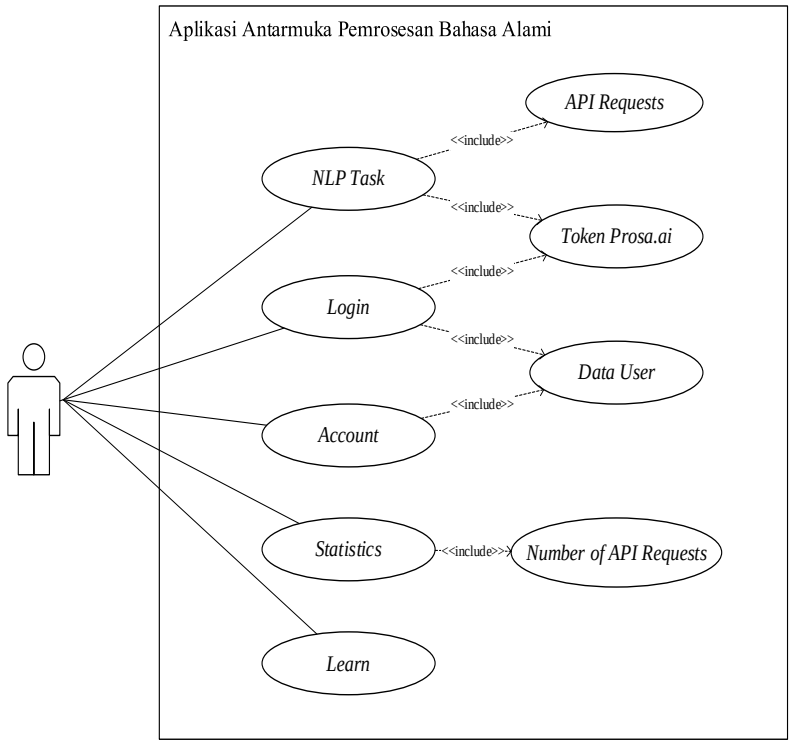
Teknik pengumpulan data yang digunakan adalah observasi dalam proses pengumpulan data untuk menentukan studi kasus masalah. Teknik analisis dan perancangan fungsional menggunakan metode perancangan berorientasi objek, dalam hal ini menggunakan bahasa pemodelan *Unified Modeling Language* (UML) sebagai standar dalam proses perancangan berorientasi objek. Alat yang digunakan untuk implementasi perancangan fungsional menggunakan *Android Studio* (versi *Arctic Fox* 2020.3.1), dan bahasa pemrograman *Java* (versi 1.8). Dalam pengujian aplikasi fokus pada aspek *Usability.*

Teknik observasi digunakan dalam proses pengumpulan data untuk menentukan studi kasus masalah. Hasil observasi dari sebuah layanan *Software as Service* yaitu Prosa.ai yang berfokus pada pemrosesan bahasa alami, bahwa Prosa.ai dalam menyediakan layanan *Webservice* API (*Application Programming Interface*) belum menyediakan *Standard Development Kit* (SDK) yang memudahkan penggunaan dan integrasi API pada suatu sistem. Menyediakan SDK untuk pengguna adalah hal umum dalam perusahaan *Software as Service*. Dengan latar belakang penelitian tentang analisis objek kopling pada aplikasi *Android*, maka penelitian ini menggunakan hasil rancang bangun aplikasi antarmuka untuk pemrosesan bahasa alami yang terintegrasi dengan Prosa.ai API sebagai objek penelitian, seperti terlihat pada Table 2 tentang layanan *webservice* API sebagai objek penelitian.

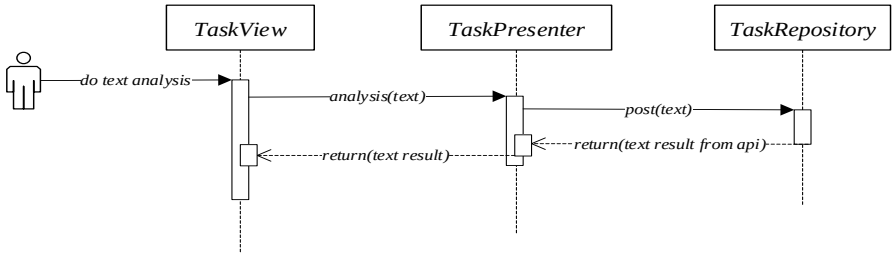
Tabel 2. Prosa.ai API yang digunakan dalam pemrosesan bahasa alami

No	URL	Maksud
1	https://api.prosa.ai/v1/entities	*Named Entity Recognizer* API digunakan untuk menemukan bagian dalam dokumen teks seperti nama orang, lokasi, produk, acara, organisasi, nomor telepon, dan sebagainya.
2	https://api.prosa.ai/v1/hates	*Hate Speech Detector* API digunakan untuk identifikasi ujaran kebencian pada suatu dokumen teks berupa kalimat atau paragraf.
3	https://api.prosa.ai/v1/normals	*Word Normalizer* API digunakan untuk mengoreksi kata-kata dalam kalimat yang diperlukan untuk menghasilkan kalimat yang baik dan benar.
4	https://api.prosa.ai/v1/quotes	*Quotation Extractor* API digunakan untuk mengambil ungkapan atau pendapat satu pihak atau lebih terhadap suatu kasus atau objek dalam sebuah artikel berupa kalimat atau paragraf.
5	https://api.prosa.ai/v1/syntax	*Syntactic Analyzer* API digunakan dalam menganalisis sebuah teks untuk memahami maknanya.
6	https://api.prosa.ai/v1/topics	*News Topic Classifier* API digunakan untuk klasifikasi topik berita berdasarkan 14 topik yang sesuai dengan topik yang telah ditentukan sebelumnya.

Analisis bertujuan untuk menentukan fungsional aplikasi yang dapat dilakukan oleh pengguna pada sistem. Hasil analisis digambarkan menggunakan *Use Case Diagram*. Dari hasil analisis yang ditunjukan pada Gambar 1, ditentukan 5 *Use Case* yang dapat dilakukan pengguna pada aplikasi, yaitu: *NLP Task*, *Learn*, *Statistics*, *Login,* dan *Account*. Selain *Use Case* yang dapat dilakukan pengguna pada aplikasi terdapat juga 4 *Use Case* yang berhubungan dengan internal sistem, yaitu: API (*Application Programming Interface*) *Request, Number of API Request, Token Prosa.ai,* dan *Data User*.



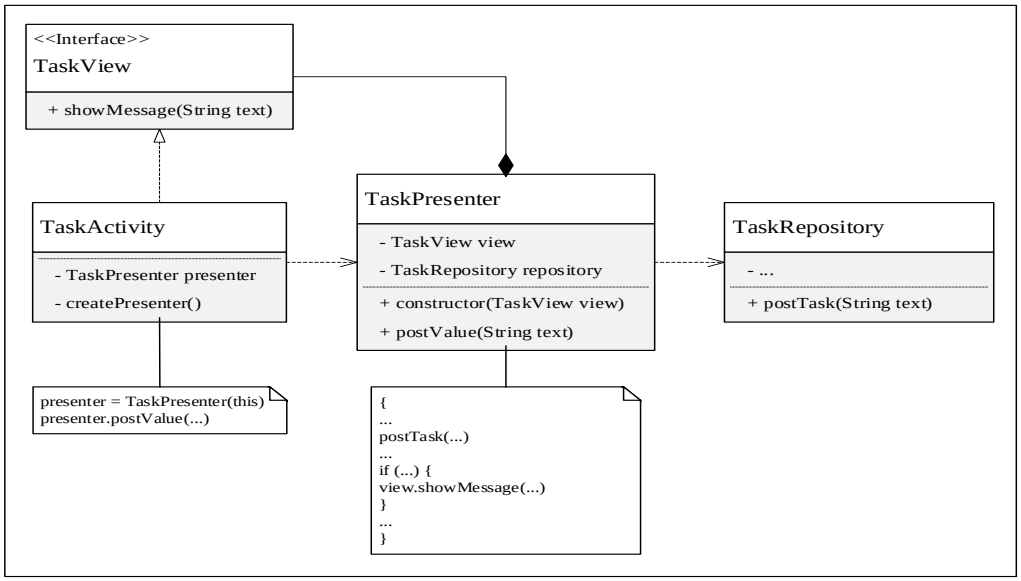
Gambar 1 *Use Case Diagram* analisis fungsional



Gambar 2 *Sequence Diagram* analisis fungsional

Proses bisnis dari setiap *Use Case* yang berkaitan dengan interaksi pengguna dengan sistem dapat digambarkan menggunakan *Sequence Diagram.* Objek (*TaskView, TaskPresenter, TaskRepository*) pada Gambar 2 *Lifetime* objek menggunakan pola yang sama seperti pada *Activity Diagram* yaitu *Model-View-Presenter*. *TaskView* mengirim pesan ke *TaskPresenter* dan kemudian *TaskPresenter* mengirim pesan ke *TaskRepository*, hasil proses dari *TaskRepository* akan dikirim balik ke *TaskPresenter* dan kemudian dari *TaskPresenter* mengirim balik ke *TaskView.*

Pewarisan dan komposisi memainkan peran utama dalam perancangan berorientasi objek. Pada tahapan ini, *Class Diagram* digunakan untuk memodelkan domain objek dan menentukan struktur data dari suatu sistem yang mendekati representasi asli pada tahap implementasi. Gambar 3 menunjukkan pola *Model-View-Presenter* pada *Class Diagram*. Kelas *TaskPresenter* menyimpan referensi objek *TaskView*. Untuk mendapatkan referensi objek *TaskView*, hal ini mengharuskan *TaskActivity* mengimplementasikan *Interface TaskView*, kemudian mengirim referensi objeknya (*Dependency Injection*) lewat konstruktor kelas *TaskPresenter* yaitu *TaskPresenter(this)*.



Gambar 3 Pola *Model-View-Presenter* pada *Class Diagram*

Perancangan antarmuka sistem bertujuan untuk menentukan batasan-batasan perancangan yang akan berinteraksi dengan pengguna. Perancangan *Wireframe* digunakan dalam perancangan antarmuka sistem yang kemudian diimplementasikan pada tahapan implementasi sistem.

Teknik implementasi sistem dilakukan setelah tahapan perancangan fungsional. dalam hal ini implementasi dilakukan menggunakan alat *Android Studio* dan Pengujian sistem lebih mengarah pada aspek *Usability*, yaitu dengan cara aplikasi dicoba langsung oleh pengguna, ini sekaligus mewakili uji fungsionalitas pada aplikasi. Pengujian *Usability* dilakukan berdasarkan instrumen kuesioner *Computer System Usability Questionnaire*

3. Hasil Dan Pembahasan

Jumlah populasi sebanyak 197 kelas dan 84 paket. Ini diambil dari total kelas dan paket hasil rancang bangun aplikasi yang menerapkan *Dependency Injection* secara manual dan otomatisasi menggunakan *Hilt* pada *Model-View-Presenter*. Sampel yang digunakan dalam penelitian ini sebanyak 28 kelas dan 14 paket, diambil dari domain *NLP Task* (*View dan Presenter*) dan *Repository* (*Model*) yang terdiri dari 14 kelas dan 7 paket sebelum dan sesudah penerapan *Hilt*.

Domain *NLP Task* digunakan sebagai domain sampel karena merupakan fungsionalitas utama yang ada pada aplikasi. Walau pun domain *Login*, *Account*, *Statistics*, dan *Learn* menggunakan pola *Model-View-Presenter* yang sama seperti domain *NLP Task*, tidak terjadi bias nilai sebelum dan sesudah menerapkan *Dependency Injection* secara otomatisasi menggunakan *Hilt*. *Repository* masuk dalam sampel dikarenakan sebagai sumber data yang berasosiasi dengan *NLP Task*. Hal ini, untuk memenuhi syarat dari pola *Model-View-Presenter* yang terdiri dari *View*, *Presenter*, dan *Model*.

Domain *NLP Task* terdiri dari 6 sub domain, yaitu: *Name Entity Extractor, Hates Speech Detector, Word Normalizer, Quotation Extractor, Syntax Analyzer,* dan *News Topics Classifier*. Tabel 3 dan Tabel 4 di atas menunjukkan nama kelas dan nama paket yang digunakan sebagai sampel dalam tahapan analisis data.

Tabel 3 Sampel kelas dari domain *NLP Task* dan *Repository*

No	Nama Kelas	Domain
1	*com.bahasa.model.local.repository .AccountRepository*	*Repository*
2	*com.bahasa.model.local.repository .RequestRepository*	
3	*com.bahasa.presentation.task.entity .NameEntityExtractorActivity*	*Name Entity Extractor*
4	*com.bahasa.presentation.task.entity .NameEntityExtractorPresenter*	
5	*com.bahasa.presentation.task.hates .HateSpeechDetectorActivity*	*Hates Speech Detector*
6	*com.bahasa.presentation.task.hates .HateSpeechDetectorPresenter*	
7	*com.bahasa.presentation.task.normals .WordNormalizerActivity*	*Word Normalizer*
8	*com.bahasa.presentation.task.normals .WordNormalizerPresenter*	
9	*com.bahasa.presentation.task.quote .QuotationExtractorActivity*	*Quotation Extractor*
10	*com.bahasa.presentation.task.quote .QuotationExtractorPresenter*	
11	*com.bahasa.presentation.task.syntax .SyntacticAnalyzerActivity*	*Syntax Analyzer*
12	*com.bahasa.presentation.task.syntax .SyntacticAnalyzerPresenter*	
13	*com.bahasa.presentation.task.topics .NewsTopicClassifierActivity*	*News Topics Classifier*
14	*com.bahasa.presentation.task.topics .NewsTopicClassifierPresenter*	

Tabel 4 Sampel paket dari domain *NLP Task* dan *Repository*

No	Nama Paket	Domain
1	*com.bahasa.model.local.repository*	*Repository*
2	*com.bahasa.presentation.task.entity*	*Name Entity Extractor*
3	*com.bahasa.presentation.task.hates*	*Hates Speech Detector*
4	*com.bahasa.presentation.task.normals*	*Word Normalizer*
5	*com.bahasa.presentation.task.quote*	*Quotation Extractor*
6	*com.bahasa.presentation.task.syntax*	*Syntax Analyzer*
7	*com.bahasa.presentation.task.topics*	*News Topics Classifier*

Pengambilan Metrik dari setiap sample pada domain *NLP Task* dan *Repository*. Metrik diambil menggunakan alat *MetricsReloaded* dan *MetricsTree* pada *Android Studio*. Pada tingkatan kelas, metrik yang digunakan adalah *Coupling between Object Classes* (CBO) dan *Response for a Class* (RFC), dan metrik pada tingkatan paket menggunakan metrik *Afferent Coupling* (AC), *Efferent Coupling* (EC), dan *Instability* (I).

Metrik tingkat kelas (CBO dan RFC) dan tingkat paket (AC, EC, dan I) dilakukan Uji-t berpasangan untuk mengetahui nilai rata-rata dari kelompok sampel yang sama apakah memiliki perbedaan yang atau tidak. tahapan ini membandingkan rata-rata dua variabel dari sampel domain *NLP Task* dan *Repository* sebelum dan sesudah menerapkan *Dependency Injection* secara otomatisasi menggunakan *Hilt.*

Tabel 5 Hasil metrik CBO

	Sebelum	Sesudah
Mean	10,428	10,642
Variance	0,725	0,708
Observations	14	14
Pearson Correlation	0,122	
Hypothesized Mean Difference	0	
Df	13	
t Stat	-0,714	
P(T<=t) one-tail	0,243	
t Critical one-tail	1,7709	
P(T<=t) two-tail	0,487	
t Critical two-tail	2,1603	

Pada Tabel 5 rata-rata nilai metrik CBO sesudah penerapan *Hilt* lebih tinggi daripada nilai sebelum penerapan *Hilt*. Hal ini terlihat dari nilai rata-rata sebelum penerapan yaitu 10,428 sedangkan nilai rata-rata sesudah penerapan *Hilt* adalah 10,642. Selisih 0.214 antara keduanya. Dapat disimpulkan telah terjadi penambahan jumlah *Coupling* kelas sesudah penerapan *Hilt*. CBO adalah ukuran jumlah banyaknya kelas yang berasosiasi dengan kelas lain. Semakin berkurang nilai CBO maka semakin berkurang usaha dalam memahami suatu kelas dan pengaruh kelas independen terhadap kelas dependen.

Tabel 6 Hasil metrik RFC

	Sebelum	Sesudah
Mean	30,642	28,5
Variance	111,324	133,346
Observations	14	14
Pearson Correlation	0,999	
Hypothesized Mean Difference	0	
Df	13	
t Stat	7,806	
P(T<=t) one-tail	1,4608	
t Critical one-tail	1,7709	
P(T<=t) two-tail	2,921	
t Critical two-tail	2,1603	

Pada Tabel 6 rata-rata nilai metrik RFC sebelum penerapan *Hilt* lebih tinggi daripada nilai sesudah penerapan *Hilt*. Hal ini terlihat dari nilai rata-rata sebelum penerapan yaitu 30,642 sedangkan nilai rata-rata sesudah penerapan *Hilt* adalah 28,5. Selisih 2,142 antara keduanya.

Dapat disimpulkan bahwa penerapan *Dependency Injection* secara otomatisasi menggunakan *Hilt* dapat mengurangi tanggung jawab kelas terhadap *Coupling* kelas. Semakin kecil nilai RFC maka semakin kecil pula dampak dari kemungkinan perubahan yang terjadi pada kelas dan mengurangi kompleksitas dalam jumlah pengujian unit. RFC adalah jumlah metode yang digunakan dalam merespon pesan pada sebuah kelas.

Tabel 7 Hasil metrik I

	Sebelum	Sesudah
Mean	0,922	0,748
Variance	0,0000424	0,000526
Observations	6	6
Pearson Correlation	1	
Hypothesized Mean Difference	0	
Df	5	
t Stat	25,998	
P(T<=t) one-tail	7,865	
t Critical one-tail	2,015	
P(T<=t) two-tail	1,573	
t Critical two-tail	2,57	

Hasil Uji-t berpasangan dari metrik I terhadap paket domain *NLP Task* pada Tabel 7 bahwa rata-rata nilai metrik I sebelum penerapan *Hilt* lebih tinggi daripada nilai sesudah penerapan *Hilt*. Hal ini terlihat dari nilai rata-rata sebelum penerapan yaitu 0,922 sedangkan nilai rata-rata sesudah penerapan adalah 0,748. Paket yang stabil (metrik I mendekati 0), yang berarti paket *NLP Task* bergantung pada tingkatan yang rendah pada paket lain. Dapat disimpulkan bahwa penerapan *Dependency Injection* secara otomatisasi menggunakan *Hilt* memberikan dampak yang baik karena mengurangi pengaruh paket lain yang berasosiasi dengan paket *NLP Task.*

Table 8 Hasil metrik AC

Nama Paket	Sebelum	Sesudah
.local.repository	14	14

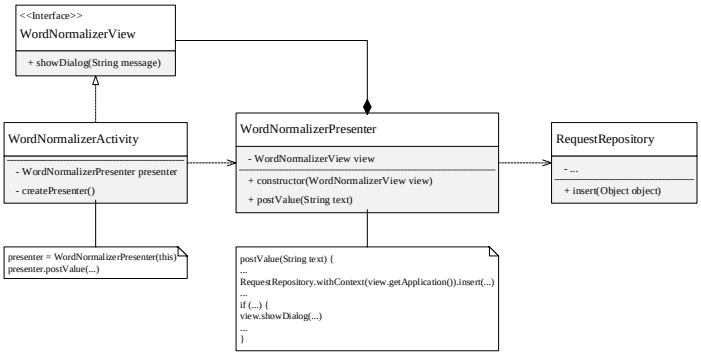
Table 9 Hasil metrik EC

Nama Paket	Sebelum	Sesudah
.local.repository	6	4

Table 10 Hasil metrik I

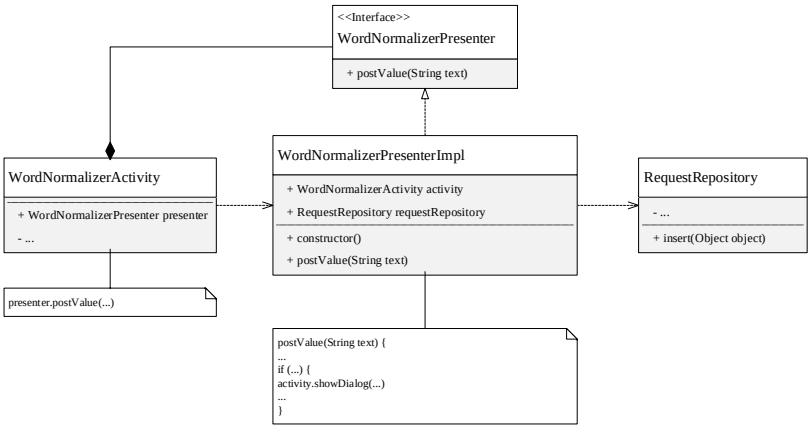
Nama Paket	Sebelum	Sesudah
.local.repository	0,3	0,2222

Hasil metrik I pada Tabel 8 yang merupakan rasio dari AC dan EC pada paket *Repository* terjadi penurunan sesudah menerapkan *Hilt*. Makna lainya adalah bahwa tingkat pengaruh *Repository* terhadap paket dependen menurun. Hal ini dapat diketahui dari nilai metrik I sebelum menerapkan *Hilt* yaitu 0,3 dan sesudah menerapkan *Hilt* yaitu 0,2222. Paket yang stabil (metrik I mendekati 0), yang berarti mereka bergantung pada tingkat yang sangat rendah pada paket lain. Nilai EC pada Tabel 10 sebelum menerapkan *Hilt* lebih tinggi dari pada sesudah menerapkan *Hilt*, ini disebabkan *Repository* membutuhkan objek *Android Context* yang di kirim dari *View* ke *Model*. Sesudah menerapkan *Hilt*, nilainya menurun karena *Repository* tidak lagi membutuhkan objek *Android Context* yang dikirim dari *View*, tetapi *Model* dapat menggunakan objek *Android Context* yang disediakan oleh *Hilt*. Dengan menggunakan Hilt, tingkat pengaruh paket *View* dan *Presenter* pada paket *Model* menurun. Hal ini dapat diketahui dari nilai metrik EC pada Tabel 9 sebelum menerapkan *Hilt* yaitu 6 dan sesudah menerapkan *Hilt* yaitu 4.



Gambar 4 Pola awal sebelum menerapkan *Hilt*

Hasil pengamatan kode pada domain *NLP Task* terjadi perubahan pola ketika menerapkan *Dependency Injection* secara otomatisasi menggunakan *Hilt*. Gambar 4 menunjukkan pola awal dari *Model-View-Presenter* sebelum menerapkan *Hilt*. Dan Gambar 5 menunjukkan pola setelah menerapkan *Dependency Injection* secara otomatisasi.



Gambar 5 Pola *Model-View-Presenter* Setelah Penerapan *Hilt*

Gambar 5 menunjukan bahwa penerapan *Dependency Injection* secara otomatisasi mengharuskan komunikasi lewat *Interface* yang dilakukan dari *View* ke *Presenter*, dan menghilangkan komunikasi lewat *Interface* dari *Presenter* ke *View* seperti pada pola awal yang ditunjukkan pada Gambar 4.

Dampak dari penerapan *Dependency Injection* secara otomatisasi memungkinkan *Presenter* dapat langsung memanggil semua metode turunan *Android Activity* tanpa melalui realisasi *Interface*. Teknik Ini menghilangkan perancangan kode yang berlebihan ketika menerapkan pola *Model-View-Presenter* yang menggunakan *Dependency Injection* secara manual. Walaupun *Hilt* dapat menghilangkan *Design Smell* pada saat tahapan implementasi, ukuran proyek bertambah karena *Hilt* melakukan generator kode untuk *Dependency Container* saat waktu kompilasi. *Hilt* mengharuskan sebagian atribut pada kelas yang dibuat harus bersifat publik, ini bertujuan untuk mendapatkan referensi objek dari *Hilt*.

Dapat dilihat pada Gambar 5 bahwa metode *"postValue (…)"* tidak mengirim "*view.getApplication()*" ke kelas *RequestRepository* seperti pada Gambar 4. Selain itu, dapat dilihat juga dari hasil analisis metrik EC dan I pada Tabel 3.7 dan 3.8 memperkuat kesimpulan bahwa *Model* yang membutuhkan objek *Android Context* tidak bergantung lagi dengan *View* yang menyediakan objek *Android Context*, dan tidak bergantung lagi dengan *Presenter* sebagai jembatan media pengiriman objek *Android Context* dari *View* ke *Model*.

Kompleksitas proyek bertujuan untuk memberi wawasan terhadap faktor-faktor yang mempengaruhi dalam menerapkan *Dependency Injection* secara otomatisasi. Faktor-faktor yang digunakan sebagai acuan kompleksitas proyek aplikasi adalah jumlah baris kode, kelas *Static*, kelas *Instance*, kelas *Interface*, dan kelas *Abstract* pada proyek aplikasi.

Tabel 11 Kompleksitas proyek aplikasi

No	Faktor Kompleksitas	Sebelum	Sesudah
1	Jumlah Baris Kode	1912	1939
2	Jumlah Kelas *Static*	6	6
3	Jumlah Kelas *Instance*	88	109
4	Jumlah Kelas *Interface*	27	27
5	Jumlah Kelas *Abstract*	1	17

Kompleksitas proyek aplikasi yang ditunjukkan pada Tabel 11 terjadi peningkatan dalam jumlah baris kode, kelas *Instance*, dan kelas *Abstract*. Penerapan *Dependency Injection* secara otomatisasi menggunakan *Hilt* tidak terjadi peningkatan yang cukup besar pada jumlah baris kode. terjadi peningkatan secara signifikan pada kelas *Abstract* dan kelas Instance dikarenakan *Hilt* membutuhkan deklarasi *Setter Injection* khusus pada kelas Abstract dan kelas Instance untuk memberitau *Hilt* kelas mana saja yang akan diberikan atau dikirim referensi objek pada deklarasi yang telah dibuat, hal tersebut berlaku ketika ingin mengirim referensi objek pada komponen pihak ketiga atau kelas *Interface*

4. Kesimpulan

Hasil analisis yang diperoleh dari Penerapan *Dependency Injection* secara otomatisasi menggunakan *Hilt* mempengaruhi pola *Model-View-Presenter* yang menggunakan *Dependency Injection* secara manual. Dan penerapan *Dependency Injection* secara otomatisasi menambah jumlah *coupling* kelas pada *View* dan mengurangi *coupling* kelas dan tanggung jawab kelas pada *Presenter* dalam konteks menggunakan bahasa pemrograman *Java*. Dengan menerapkan *Dependency Injection* secara otomatisasi menghilangkan *Design Smell* pada *Model* yang membutuhkan *Dependency* objek *Android Context*. Dari hasil uji coba d*esign Smell* menyebabkan kekakuan, kerapuhan.

Referensi

[1] M. Fowler. "Reducing coupling". IEEE Softw. vol. 18, no.4, pp.102–4, 2001
[2] H. Melton, E. Tempero." An empirical study of cycles among classes in Java". Empir Softw Eng. vol.12, no.4, pp.389–415,2007
[3] T.D. Oyetoyan, D.S.Cruzes, R. Conradi. "A study of cyclic dependencies on defect profile of software components". J Syst Softw. vol.86, no.12, pp.3162–82, 2013
[4] B. Walter, T. Alkhaeir. "The relationship between design patterns and code smells: An exploratory study". Inf Softw Technol. vol. 74, pp.127–42, 2016
[5] M. Aniche, G. Bavota, C. Treude, M.A. Gerosa, A. van Deursen. "Code smells for Model-View-Controller architectures". Empir Softw Eng. vol. 23, no.4, pp.2121–57, 2018
[6] A.D. Dennis, R.M. Roth, B.H. Wixom. "Systems Analysis and Design: An Object-Oriented Approach with UML". 2015.
[7] R.C. Martin. "Agile Software Development: Principles, Patterns, and Practices". 2014.
[8] Smartphone Market Share.
[9] Google. Android @ Google I/O: 3 things to know in Modern Android Development.
[10] Dependency Injection.
[11] A.C. Baramuli, A. Nugroho, A.A. Setiyanti. "Implementasi Model View Presenter dan Object Relational Mapping NHibernate pada Aplikasi eStop Card berbasis Web (Studi Kasus: PT. XYZ Jakarta)". J Inform. vol. 8, no. 2, pp. 189-208, 2013
[12] G. Prabowo, H. Suryotrisongko, A. Tjahyanto. "A Tale of Two Development Approach: Empirical Study on The Maintainability and Modularity of Android Mobile Application with Anti-Pattern and Model-View-Presenter Design Pattern". In: Proceedings - 2nd 2018 International Conference on Electrical Engineering and Informatics, ICELTICs 2018. IEEE; pp. 149–54, 2018.
[13] B. Rizki, P. Surya, A.P. Kharisma, N. Yudistira. "Perbandingan Kinerja Pola Perancangan MVC , MVP , dan MVVM Pada Aplikasi Berbasis Android (Studi kasus : Aplikasi Laporan Hasil Belajar Siswa SMA BSS)". JPTIIK (Jurnal Pengemb Teknol Inf dan Ilmu Komputer). vol. 4, no. 11, pp. 4089–95, 2020;
[14] S.R. Chidamber, C.F. Kemerer. "A Metrics Suite for Object Oriented Design". IEEE Trans Softw Eng. vol. 20, no. 6, pp.476–93, 1994
[15] W. Yusuf. "Metode Statistik". Yogyakarta: Gadjah Mada University Press, 2015