Pengaruh Design Pattern Terhadap Maintainability Aplikasi Mobile

I Gusti Bagus Vayupranaditya Putraadinatha[1], Dawam Dwi Jatmiko Suwawi[2], Shinta Yulis Puspitasari[3]

[1,2,3] Universitas Telkom, Bandung
[1]vayupranaditya@student.telkomuniversity.ac.id, [2]dawamdjs@telkomuniversity.ac.id,
[3]shintayulia@telkomuniversity.ac.id

Abstrak

Dalam sebuah *software development lifecycle* pada pengembangan aplikasi *mobile*, tahap *maintenance* merupakan tahap yang membutuhkan biaya dan waktu yang sangat besar. Hal ini ditunjukkan dengan banyaknya aplikasi yang harus dihapus oleh Google pada toko aplikasi mereka karena tidak di-*maintain* dengan baik. Hal ini menunjukkan bahwa masih banyak aplikasi yang tidak dipersiapkan dengan memikirkan aspek *maintainability*. Penelitian sebelumnya menunjukkan *maintainability* pada sistem berorientasi objek seperti aplikasi *mobile* dapat ditingkatkan dengan penggunaan *design pattern* yang sesuai. Sayangnya penelitian tersebut tidak memaparkan secara detail analisis terhadap pemilihan masing-masing *design pattern* yang digunakan serta pengaruh pemilihan tersebut terhadap aspek-aspek *maintainability*. Penelitian ini berfokus pada pemilihan *design pattern* yang dilakukan pada sebuah aplikasi *mobile* dan melihat pengaruhnya terhadap *maintainability* seperti *analyzability, changeability, stability*, serta *testability*. *Source code* aplikasi dianalisis untuk mengetahui *design problem* yang ada serta pengaruhnya terhadap *OO metric*, lalu mencari *design pattern* yang sesuai untuk melihat pengaruh dari penggunaan beberapa *design pattern* secara terpisah, serta kombinasi dari *design pattern* tersebut terhadap aspek *maintainability*. Dari analisis tersebut ditemukan 2 hal. Pertama pemilihan *design pattern* pada aplikasi *mobile*, khususnya Android dapat dilakukan dengan mencari *design problem* dengan penyesuaian seperti merubah *constructor* menjadi method-method penunjang *activity lifecycle*, serta tidak semua *design problem* dapat diselesaikan seperti jumlah *parent class* dari Activity yang pada dasarnya melebihi batas yang diterima. Kedua implementasi *design pattern* dapat memperbaiki nilai *maintainability* aplikasi *mobile* yang dipilih antara 11.35% sampai 17.83% karena dapat menerapkan *separation of concern*. Walau begitu, aspek *stability* dapat memburuk karena perlu ada penambahan *class* yang dilakukan sehingga meningkatkan *dependency*.

Kata kunci : design pattern, pengembangan aplikasi mobile, maintainability, non-functional requirement, rekayasa perangkat lunak

Abstract

In a software development lifecycle in mobile application development, the maintenance stage is a stage that requires very large costs and time. This is indicated by the number of applications that must be removed by Google in their application store because they are not properly maintained. This shows that there are still many applications that are not prepared by considering the maintainability aspect. Previous research has shown that maintainability in object-oriented systems such as mobile applications can be improved by using appropriate design patterns. Unfortunately, this research does not describe in detail the analysis of the selection of each design pattern used and the effect of the selection on maintainability aspects. This study focuses on selecting a design pattern for a mobile application and looking at its effect on maintainability such as analyzability, changeability, stability, and testability. The application source code is analyzed to determine the existing design problems and their effect on OO metrics, then look for the appropriate design pattern to see the effect of using several design patterns separately, as well as the combination of these design patterns on maintainability aspects. From the analysis found 2 things. First, the selection of design patterns in mobile applications, especially Android, can be done by looking for design problems with adjustments such as changing the constructor into methods that support the activity lifecycle, and not all design problems can be solved, such as the number of parent classes from the Activity which basically exceeds the accepted limit. The two implementations of the design pattern can improve the maintainability value of the selected mobile application between 11.35% to 17.83% because it can apply separation of concern. However, the stability aspect can deteriorate because it is necessary to add classes to increase dependencies.

Keywords: design pattern, mobile application development, maintainability, non-functional requirement, software engineering

1. Pendahuluan

Latar Belakang

PengembanganPengembangan aplikasi*mobile* merupakan sebuah investasi yang membutuhkan waktu dan biaya yang besar. Dalam pengembangannya, waktu dan biaya terbesar berada pada tahap*maintenance* [7][5]. Banyak aplikasi yang berhasil dikembangkan tetapi ternyata tidak di-*maintain* dengan baik. Bahkan Google harus menghapus banyak aplikasi di toko aplikasi mereka yang tidak di-*maintain* dalam kurun waktu tertentu. [5][16]

Dalam tahap*maintenance*, pengembang aplikasi perlu menekan jumlah galat yang ada. Selain itu pengembang juga perlu memastikan fungsionalitas aplikasi untuk selalu berkembang mengikuti kebutuhan bisnis[6][9]. Di sisi lain, pengembangan fungsionalitas dapat menimbulkan dampak yang tidak diinginkan pada fungsionalitas-fungsionalitas yang sudah ada[9]. Oleh karena itu pengembang perlu memperhatikan aspek*maintainability*.

Panca, dkk.[12] dalam penelitiannya memaparkan bahwa*maintainability* pada software berbasis objek seperti Android dapat ditingkatkan dengan pemilihan*design pattern* yang sesuai. Sayangnya Panca tidak menjelaskan lebih lanjut alasan pemilihan*design pattern* khususnya efeknya terhadap aspek*maintainability*. Untuk itu, diperlukan penelitian lebih lanjut terhadap pemilihan*design pattern* pada aplikasi*mobile* serta efeknya terhadap aspek*maintainability*.

Topik dan Batasannya

Dalam penelitian ini, dilakukan analisis terhadap pemilihan *design pattern* yang sesuai terhadap keperluan aplikasi mobile, serta efeknya terhadap aspek *maintainability*. Analisis dilakukan pada penerapan satu *design pattern* dalam satu aplikasi, serta pada beberapa *design pattern* dalam satu aplikasi.

Penelitian dilakukan terhadap sebuah aplikasi *open source* Android pada toko aplikasi *open source* Fossdroid yang masih mengalami perubahan serta penambahan fitur untuk memastikan bahwa aplikasi tersebut memang digunakan dan perlu dilakukan *maintenance*.

Tujuan

Penelitian ini dilakukan untuk mendalami cara pemilihan *mobile* pada aplikasi *mobile* khususnya Android serta pengaruhnya pada aspek *maintainability* baik saat implementasi satu *mobile* dalam satu aplikasi maupun implementasi beberapa *mobile* dalam satu aplikasi.

Organisasi Tulisan

Untuk memudahkan pembaca, tulisan ini dibagi menjadi beberapa bagian: Pertama, pemaparan penelitian terdahulu mengenai aspek *maintainability* pada aplikasi *mobile* serta penggunaan *mobile* untuk memperbaikinya; Kedua, metodologi dan eksperimen pada aplikasi *mobile* yang dipilih; Ketiga, analisis hasil eksperimen; Keempat, kesimpulan dari penelitian ini.

2. Studi Terkait

2.1 *Maintainability* Pada Aplikasi *Mobile*

Panca, B. dkk. [12] menjelaskan *maintainability* pada aplikasi *mobile* dapat ditingkatkan dengan menerapkan *design pattern*. Panca membandingkan *maintainability* pada aplikasi *mobile* yang menggunakan *anti-pattern* dengan aplikasi *mobile* yang menggunakan kombinasi beberapa *design pattern*, yaitu Singleton, Memento, State, Iterator, Factory, Builder, dan Flyweight. Panca menemukan bahwa penerapan *design pattern* tersebut secara konsisten dapat meningkatkan *maintainability* pada aplikasi *mobile* karena penerapan *design pattern* tersebut membagi kompleksitas pada masing-masing *class*, kecuali pada Flyweight. Flyweight tidak secara konsisten meningkatkan *maintainability* karena tidak menambahkan *class* baru seperti *design pattern* lain.

Di sisi lain, Khomh, F. dan Gueheneuc, Y. [10] melakukan penelitian secara kualitatif mengenai pendapat mereka terhadap penggunaan *design pattern*. Mereka menemukan bahwa penerapan *design pattern* dapat menyelesaikan *design problem*, *design pattern* juga dapat berdampak negatif terhadap *quality attribute* sebuah *software*, di antaranya *simplicity, learnability* dan *reusability*. Oleh karena itu mereka menyimpulkan bahwa penggunaan *design pattern* harus dilakukan secara berhati-hati.

2.2 Perhitungan Nilai *Maintainability*

Aplikasi *mobile* dikembangkan dengan paradigma pemrograman berbasis objek. Oleh karena itu banyak peneliti menganggap penggunaan *OO metric* merupakan cara yang sesuai untuk menghitung *maintainability* pada aplikasi *mobile* [3][2][15]. Albeladi, A., dkk. dalam Saifan, A. dan Rabadi, A. [13] memaparkan 4 aspek dalam *maintainability* yaitu *analyzability, changeability, stability*, dan *testability*. Masing-masing aspek tersebut memiliki beberapa *submetric* yang dihitung dengan rumus berikut.

$$Maintainability = Analyzability + Changeability + Stability + Testability \quad (1)$$

$$Analyzability = CL_WMC + CL_CMOF + IN_BASES + CU_CDUSED \quad (2)$$

$$Changeability = CL_STAT + CL_FUN + CL_DATA \quad (3)$$

$$Stability = CL_DATA_PUBL + CU_CDUSERS + IN_NOC + CL_FUN_PUBL \quad (4)$$

$$Testability = CL_WMC + CL_FUN + CU_CDUSED \quad (5)$$

Tabel 1. *Submetric* yang digunakan untuk menghitung *OO metric*

Submetric	Deskripsi	Batasan
CL_WMC	Total kompleksitas masing-masing *method* pada *class*	0-11
CL_CMOF	Rasio *lines of comment* dengan *total lines of code*	0-100
IN_BASES	Banyaknya *parent class* dalam suatu *class*	0-4
CU_CDUSED	Banyaknya *class* lain yang digunakan dalam *class* ini	0-6
CL_FUN	Banyaknya *method* dalam *class*	0-9
CL_FUN_PUBL	Banyaknya *public method* dalam *class*	0-7
CL_DATA_PUBL	Banyaknya *public attribute* dalam *class*	0-7
CL_DATA	Banyaknya *attribute* dalam *class*	0-25
IN_NOC	Banyaknya *child class*	0-5
CU_CDUSERS	Banyaknya *class* yang menggunakan *class* ini	0-3
CL_STAT	Banyaknya *statement* yang dapat dijalankan	0-7

Dalam tabel 1 di atas, dapat diketahui bahwa masing-masing *submetric* memiliki batas bawah 0 serta tidak terdapat nilai negatif. Oleh karena itu dapat dikatakan bahwa semakin kecil nilainya, *submetric* tersebut semakin baik. Nguyen[4] menjelaskan banyak *statement* yang dapat dijalankan dapat diartikan sebagai bagian dari *code* yang melaksanakan suatu aksi saat *runtime* ataupun *compile time*. Nilai tersebut dapat dihitung dengan pendekatan *logical source lines of code* (*logical SLOC*) [13][4][1]. Adithyan, T., dkk.[1] menjelaskan bahwa penghitungan *logical SLOC* dilakukan dengan menghitung banyaknya *logic statement* dan *statement-statement* yang berkaitan.

Berdasarkan perhitungan pada rumus (1, 2, 3, 4, 5) serta batasan pada tabel 1, dapat diketahui batas atas yang diterima pada *analyzability, changeability, stability, testability*, serta *maintainability* masing-masing adalah 121, 41, 22, 26, serta 210.

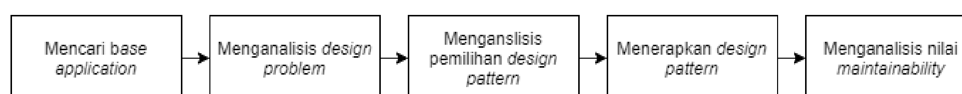
2.3 Static Code Analyzer

Marcilio, D.[11] menjelaskan bahwa *static code analyzer* dapat digunakan untuk menganalisis beberapa aspek dalam source *code* tanpa harus menjalankan program yang dianalisis. *Static code analyzer* merupakan *software* yang dapat menganalisis *software quality attribute* suatu aplikasi tanpa harus menjalankan aplikasi yang dianalisis.

2.4 Design Pattern

Gamma, E., dkk. [8] menjelaskan bahwa *design pattern* merupakan solusi yang dapat diterapkan secara berulang pada permasalahan yang berulang kali terjadi. Terdapat 22 *design pattern* yang dibagi dalam 3 kelompok besar berdasarkan sifatnya, yaitu Creational yang memberi solusi dengan pendekatan pada mekanisme pembuatan *object*; Structural yang memberi solusi dengan pendekatan pada penyusunan *class-class* untuk membentuk suatu struktur yang lebih kompleks; serta Behavioral yang memberi solusi dengan berfokus pada algoritma serta pada *separation of concern*. Setiap *design pattern* terdiri atas nama, pemaparan permasalahan yang sesuai, pemaparan solusi yang ditawarkan, serta konsekuensi penggunaannya.

3. Metodologi Penelitian



Gambar 1. Tahapan penelitian

Penelitian ini dilakukan dengan tahapan yang ditunjukkan dalam gambar 1. Penelitian dimulai dengan pemilihan *base application* yang sesuai tujuan dan batasan masalah. *Base application* yang ditemukan nantinya akan menjadi acuan dalam penelitian ini. Kedua dilakukan analisis *design problem* yang ada pada *base application*. Hasil analisis tersebut menjadi dasar pemilihan *design pattern* yang juga merupakan tahap ketiga dari penelitian ini. Keempat dilakukan implementasi *design pattern* yang sudah dipilih pada tahap ketiga. Hasil implementasi ini yang akan dibandingkan dengan *base application* dan dianalisis pada tahap kelima.

3.1 Pemilihan Base Application

Dalam pemilihan *base application*, dilakukan analisis terhadap beberapa aplikasi yang sesuai dengan batasan masalah. *Base application* dicari pada toko aplikasi *open source* Fossdroid yang terdapat pada bagian Most Popular, sudah dikembangkan selama lebih dari satu tahun dan masih dikembangkan dalam waktu satu bulan terakhir. Didapatkan sebuah aplikasi bernama Currency yang dikembangkan bersama oleh 9 orang pengembang dan telah mengalami penambahan dan perubahan fungsionalitas serta algoritma yang dapat diketahui melalui *release note*. Aplikasi ini merupakan aplikasi Android yang berguna untuk melakukan konversi nilai mata uang. Terdapat beberapa fungsionalitas, di antaranya dapat memperbaharui nilai mata uang secara daring; membandingkan beberapa nilai mata uang secara bersamaan; serta menampilkan grafik nilai mata uang dalam suatu rentang waktu.

3.2 Menganalisis Design Problem Pada Base Application

Source code aplikasi dianalisis menggunakan beberapa *static code analyzer* kemudian dibandingkan dengan batasan yang diterima pada tabel 1 di atas. Hasil penghitungan yang dituliskan dengan warna merah menunjukkan nilai di luar batas yang diterima. Perhitungan dilakukan menggunakan MetricsReloaded versi 1.11.2 untuk mengukur CL_WMC dan CL_CMOF, ProjectCodeMeter versi 2.4.1 untuk menghitung CL_STAT, dan SonarQube Community versi 8.7.1 untuk menghitung *submetric* lain.

class	cl_wmc	cl_cmof	in_bases	cu_cdused	cl_fun	cl_fun_publ	cl_data_publ	cl_data	in_noc	cu_cdusers	cl_stat
AboutPreference	7	4.636	3	10	2	1	0	0	0	0	23
ChartActivity	85	4.376	5	31	23	6	12	28	0	1	410
ChartParser	4	5.800	1	8	5	4	0	3	0	1	30
ChoiceAdapter	10	7.556	2	11	5	5	0	7	0	1	35
ChoiceDialog	31	6.429	5	16	6	6	0	5	0	2	91
CurrencyAdapte	12	8.889	2	11	5	5	0	8	0	1	44
Data	8	5.750	1	8	7	5	0	4	0	1	24
HelpActivity	9	7.625	5	13	3	3	0	0	0	1	32
Main	126	5.750	5	49	25	12	25	55	0	5	612
Parser	4	6.333	1	8	4	4	0	2	0	2	30
SettingsActivity	6	5.625	5	5	2	1	0	0	0	1	19
SettingsFragmer	9	6.250	3	10	5	5	0	0	0	1	30
Singleton	9	5.889	1	8	8	6	0	5	0	1	28

Gambar 2. Perhitungan *submetric* pada aplikasi Currency

Terdapat permasalahan yang berasal dari Android API sehingga tidak dianggap sebagai sebuah design problem sehingga dan diabaikan. Permasalahan tersebut adalah nilai IN_BASES dari *class* Activity yang berada pada batas yang diterima. Permasalahan seperti ini menyebabkan semua *class* yang diturunkan dari Activity pasti memiliki nilai IN_BASES di luar batas yang diterima. Pada gambar 2 dapat dilihat bahwa batasan CU_CDUSED dilewati pada hampir seluruh *class* yang ada, sedangkan batasan CL_STAT dilewati pada seluruh *class*. Karena keterbatasan waktu, penelitian ini tidak dapat menganalisis semua *class* yang melewati batasan pada CU_CDUSED dan CL_STAT.

Pada akhirnya dianalisis empat *class* yang memiliki *submetric* di luar batasan yang diterima. Dari keempat *class* tersebut, ditemukan *design problem* sebagai berikut.

Tabel 2. *Design problem* yang terdapat pada aplikasi Currency

***Class* yang Terkait**	***Submetric* yang terkait**	***Design Problem***
CurrencyAdapter, Main, ChartAdapter	CL_WMC	Kompleksitas setter untuk menunjang lifecycle
Main, ChoiceDialog, ChartAdapter	CL_WMC, CU CDUSED, CL_FUN, CL_STAT	Terdapat banyak event handler
Main, ChoiceDialog	CL_WMC, CU CDUSED, CL_FUN, C_FUN_PUBL, CL_STAT	Terdapat beberapa fungsionalitas berbeda tergantung mode yang sedang aktif

Design problem di atas digunakan sebagai acuan untuk memilih *design pattern* yang dijelaskan pada bagian 3.3.

3.3 Menganalisis Pemilihan Design Pattern

Buku *Design Patterns: Elements of Reusable Object-Oriented*[8] memaparkan enam pendekatan memilih *design pattern* yang sesuai. Berdasarkan pendekatan tersebut, *design pattern* dianalisis sesuai *intent*, permasalahan, pemecahan masalah, dan keterhubungan dengan *design pattern* lain sesuai permasalahan yang ada pada *source code* aplikasi Currency berdasarkan penelitian-penelitian sebelumnya [8][14]. Pada studi kasus aplikasi Currency, terdapat beberapa kesamaan yang ada pada beberapa *design problem*. Analisis tersebut dapat dilihat pada tabel 3 pada lampiran. Untuk mempermudah pembaca, *design problem* yang memiliki karakteristik yang sama disatukan dalam satu baris. Dari tabel 3 dapat diketahui bahwa tidak semua *design problem* dapat diselesaikan dengan suatu *design pattern*. Berdasarkan analisis tersebut dapat diketahui terdapat 2 *design pattern* yang dapat diterapkan yaitu Command dan Strategy.

Eksperimen dilakukan dengan menerapkan *design pattern* secara terpisah untuk melihat efeknya secara *atomic* serta dengan menggabungkan beberapa *design pattern* sesuai dengan *class* dan block of code yang sesuai untuk melihat efeknya saat digabungkan. Terdapat 3 eksperimen yang dilakukan.

- Eksperimen 1: Menerapkan Command Pattern
- Eksperimen 2: Menerapkan State Pattern
- Eksperimen 3: Menerapkan Command Pattern dan State Pattern

3.4 Menerapkan Design Pattern

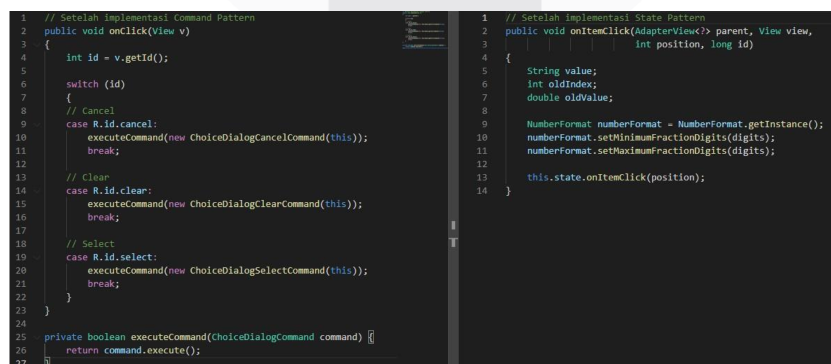
Masing-masing eksperimen yang ditentukan pada 3.3 diimplementasikan pada *base application*. *Class diagram* implementasi *design pattern* dapat dilihat pada gambar 7, 8, dan 9 pada bagian lampiran. Modifikasi pada *base application* dapat berupa pengurangan, penambahan, maupun modifikasi *method* serta *attribute class*. *Method* maupun *attribute* berlatar belakang hijau menunjukkan penambahan, warna merah menunjukkan pengurangan, serta tanpa latar belakang menunjukkan modifikasi. Untuk menyederhanakan tampilan, *method* serta *attribute* lain tidak ditampilkan.

Pada eksperimen 1, terdapat banyak *separation of concern* dalam hal fungsionalitas. *Separation of concern* dapat berupa memindahkan sebuah *method* menjadi *class* tersendiri, maupun memindahkan bagian dari sebuah *method* menjadi *class* tersendiri.

Pada eksperimen 2, tidak terdapat sebuah *method* yang dipindahkan menjadi *class* tersendiri karena pada suatu *method*, terdapat algoritma untuk masing-masing *state*.

Pada eksperimen 3, tidak terdapat bagian dari Command Pattern yang juga merupakan bagian dari State Pattern. Implementasi keduanya dapat dilakukan secara terpisah.

Contoh implementasi Command Pattern serta State Pattern dapat dilihat pada gambar 3. Implementasi tersebut merupakan hasil *refactor* dari *base application* yang dapat dilihat pada gambar 10 pada bagian lampiran. Dapat dilihat pada gambar 3 bahwa terdapat bagian dari *method* pada gambar 10 yang hilang. Potongan *method* tersebut dipindahkan ke dalam *class* tersendiri sesuai *design pattern*-nya.



Gambar 3. Implementasi Command Pattern dan State Pattern

3.5 Menganalisis Nilai Maintainability

Masing-masing *submetric* dihitung kembali menggunakan *static code analyzer* kemudian dianalisis penyebabnya berdasarkan sifat dan penerapan *design pattern* pada masing-masing *source code*.

4. Evaluasi

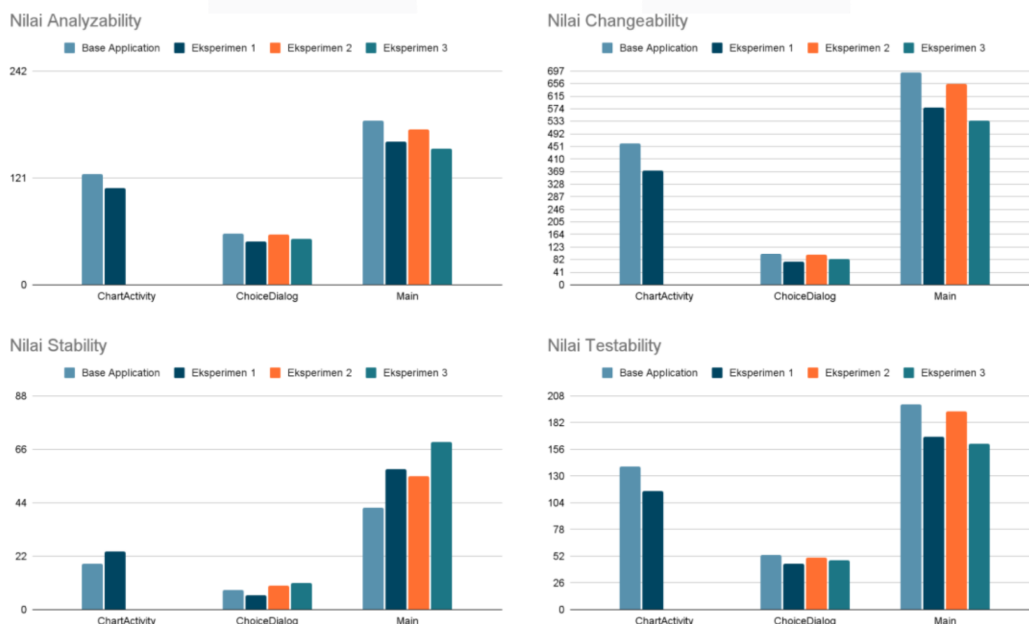
4.1 Hasil Pengujian

Perubahan nilai masing-masing *submetric* dapat dilihat pada gambar 4 di bawah. Berdasarkan nilai-nilai tersebut, didapatkan nilai pada masing-masing aspek yang ditunjukkan pada gambar 5. Nilai berwarna hijau menunjukkan hasil yang semakin baik dibandingkan *base application*, sedangkan nilai berwarna merah menunjukkan hasil yang semakin buruk dibandingkan *base application*.

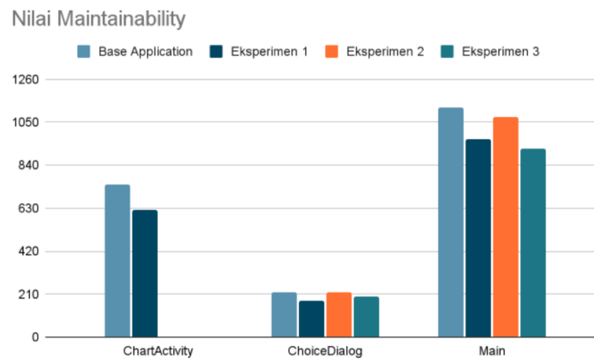
	cl_wmc	cl_cmof	in_bases	cu_cdused	cl_fun	cl_fun_publ	cl_data_publ	cl_data	in_noc	cu_cdusers	cl_stat
Base Application											
ChartActivity	85	4.38	5	31	23	6	12	28	0	1	410
ChoiceDialog	31	6.43	5	16	6	6	0	5	0	2	91
Main	126	5.75	5	49	25	12	25	55	0	5	612
Eksperimen 1											
ChartActivity	68	4.34	5	32	15	6	16	28	0	2	329
ChoiceDialog	22	5.83	5	16	7	3	0	5	0	3	63
Main	104	5.94	5	47	17	12	35	55	0	11	506
Eksperimen 2											
ChoiceDialog	27	7.56	5	17	7	7	0	6	0	3	86
Main	117	3.61	5	50	26	13	34	56	0	8	572
Eksperimen 3											
ChoiceDialog	24	6.30	5	17	7	7	0	6	0	4	72
Main	95	6.11	5	48	18	14	42	56	0	13	462

Gambar 4. Nilai *submetric* setelah implementasi *design pattern*

Gambar 5 menunjukkan klasifikasi aspek-aspek *maintainability* berdasarkan batasan yang diterima sesuai studi terdahulu. Semakin kecil nilai maka semakin baik. Batas yang diterima ditunjukkan dengan nilai pada sumbu Y pertama. Dengan kata lain jika suatu nilai di bawah batas pertama pada sumbu Y, nilai tersebut dianggap cukup baik. Nilai *maintainability* dapat dilihat pada gambar 6. Berdasarkan grafik-grafik tersebut dapat diketahui bahwa semua aspek kecuali *stability* dapat diperbaiki pada semua eksperimen. Walaupun terdapat aspek yang memburuk, nilai *maintainability* tetap membaik pada semua eksperimen.



Gambar 5. Nilai pada aspek-aspek *maintainability*. Semakin kecil semakin baik. Aspek dianggap cukup baik jika nilainya di bawah batas sumbu Y pertama.



Gambar 6. Nilai *maintainability*. Semakin kecil semakin baik. *Maintainability* dianggap cukup baik jika nilainya di bawah batas sumbu Y pertama.

4.2 Analisis Hasil Pengujian

Berdasarkan hasil pengujian pada 4.1, ditemukan bahwa implementasi *design pattern* dapat memperbaiki semua aspek kecuali *stability*. Walaupun begitu, nilainya bergantung pada *design problem* yang ada pada aplikasi. Berdasarkan perubahan tersebut, dapat dilihat bahwa implementasi *design pattern* dapat memperbaiki *analyzability* sehingga hasilnya dapat dikategorikan sebagai mudah dianalisis. Hal ini dimungkinkan karena terdapat *separation of concern* yang menyederhanakan tugas dari masing-masing *class*.

Di sisi lain, implementasi *design pattern* secara umum memperburuk nilai *stability* karena *design pattern* perlu menambah beberapa *class* sehingga perlu menambah *dependency* pada *client class* serta mengubah beberapa *method* dan *attribute* menjadi *public*.

Nilai *submetric* pada gambar 4 menunjukkan bahwa Command Pattern berhasil menerapkan *separation of concern* pada masing-masing fungsionalitas sehingga terdapat penurunan kompleksitas *class*. Oleh karena terdapat *separation of concern*, terdapat beberapa *class* baru yang dependant terhadap client *class* sehingga meningkatkan *dependency*. Pada *submetric* lain, nilainya mengikuti banyaknya *block of code* yang dipisah dibanding *block of code* baru yang diperlukan untuk menerapkan Command Pattern.

Dalam hal ini State Pattern juga berhasil melakukan *separation of concern*. Tidak seperti Command Pattern, State Pattern perlu menambah sebuah *attribute* yang menyimpan *current state* sehingga terdapat peningkatan pada jumlah *attribute*.

Eksperimen 3 menunjukkan bahwa Command Pattern dapat diimplementasikan dengan State Pattern secara terpisah. Masing-masing *design pattern* memiliki efek sesuai eksperimen 1 dan 2. Dapat dilihat pada gambar 7, 8, dan 9 pada bagian lampiran bahwa Command Pattern mengurangi 9 *method* serta menambah 1 *method*, sedangkan State Pattern menambah 1 *method*. Pada kombinasi keduanya, terdapat total pengurangan 9 *method* serta penambahan 1 *method*.

5. Kesimpulan

Berdasarkan penelitian yang telah dilakukan, dapat diketahui 2 hal. Pertama pemilihan *design pattern* pada aplikasi *mobile*, khususnya Android dapat dilakukan dengan mencari *design problem* dengan penyesuaian seperti merubah *constructor* menjadi *method-method* penunjang *activity lifecycle*, serta tidak semua *design problem* dapat diselesaikan seperti jumlah *parent class* dari Activity yang pada dasarnya melebihi batas yang diterima. Kedua implementasi Command Pattern pada aplikasi Android yang dipilih dapat memperbaiki aspek *analyzability* antara 12.79% sampai 16.43%, *changeability* antara 16.47% sampai 26.47%, *testability* antara 15.09% sampai 17.27%, dan dapat memperbaiki *stability* sebesar 25% dan juga memperburuk *stability* sampai 38.1%. Secara umum implementasi Command Pattern dapat memperbaiki nilai *maintainability* antara 13.74% sampai 20.59%. Implementasi State Pattern dapat memperbaiki aspek *analyzability* antara 3.21% sampai 5.46%, *changeability* antara 2.94% sampai 5.49%, *testability* antara 3.5% sampai 3.77%, dan memperburuk *stability* antara 25% sampai 30.95%. Secara umum implementasi State Pattern dapat memperbaiki nilai *maintainability* antara 2.2% sampai 3.85%. Implementasi Command dan State Pattern secara bersamaan dapat memperbaiki aspek *analyzability* antara 10.48% sampai 17.03%, *changeability* antara 16.67% sampai 22.54%, *testability* antara 9.43% sampai 19.5%, dan memperburuk *stability* antara 37.5% sampai 64.29%. Secara umum implementasi Command dan State Pattern secara bersamaan dapat memperbaiki nilai *maintainability* antara 11.35% sampai 17.83%. Design pattern dapat memperbaiki nilai *maintainability* aplikasi *mobile* yang dipilih karena dapat menerapkan *separation of concern*. Walau begitu,

nilai *stability* dapat memburuk karena perlu ada penambahan *class* yang dilakukan sehingga meningkatkan *dependency*. Penelitian lebih lanjut dapat dilakukan terhadap aplikasi iOS agar dapat mengetahui *design problem* serta penyesuaian implementasi *design pattern* yang diperlukan pada lingkungan pengembangan aplikasi iOS.

Referensi

[1] T. A. Adithyan et al. Nature inspired algorithm. *national Conference on Trends in Electronics and Informatics*, 2017.

[2] A. Agrawal and R. K. Singh. Empirical validation of oo metrics and machine learning algorithms for software change proneness prediction. *Towards Extensible and Adaptable Methods in Computing*, 2018.

[3] A. Bakar et al. Review on 'maintainability' metrics in open source software. *International Review on Computers and Software*, 2012.

[4] D. N. Balaji, N. Shivakumar, and V. V. Ananth. Software cost estimation using function point with non algorithmic approach. *Global Journal of Computer Science and Technology*, 2013.

[5] S. Cherednichenko. What's the cost to maintain and support an app in 2020. https://www.mobindustry.net/blog/whats-the-cost-to-maintain-and-support-an-app-in-2020/, 2020. Online; Acces- sed 26 February 2021.

[6] L.-V. Cobaleda et al. Reference software architecture for improving modifiability of personalised web applications - a controlled experiment. *International Journal of Web Engineering and Technology*, 2016.

[7] Galorath. Accurately estimate your software maintenance cost. https://galorath.com/ software-maintenance-costs/. Online; Accessed 24 August 2021.

[8] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented*. Addison-Wesley, 1994.

[9] A. Gupta and S. Sharma. Software maintenance: Challenges and issues. *International Journal of Computer Science Engineering*, 2015.

[10] F. Khomh and Y.-G. Gueheneuc. Do design patterns impact software quality positively? *European Conference on Software Maintenance and Reengineering*, 2008.

[11] D. Marcilio et al. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019.

[12] B. S. Panca, S. Mardiyanto, and B. Hendradjaya. Evaluation of software design pattern on mobile application based service development related to the value of maintainability and modularity. *2016 International Conference on Data and Software Engineering (ICoDSE)*, 2016.

[13] A. A. Saifan and A. Al-Rabadi. Evaluating maintainability of android applications. *2017 8th International Conference on Information Technology (ICIT)*, 2015.

[14] A. Shvets. *Dive Into Design Patterns*. 2019.

[15] Y. Singh, A. Kaur, and R. Malhotra. Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal*, 2010.

[16] Stastista. Number of available applications in the google play store from december 2009 to september 2020. https://www.statista.com/statistics/266210/ number-of-available-applications-in-the-google-play-store/, 2020. Online; Accessed 24 August 2021.