

FUNDAÇÃO CENTRO DE ANÁLISE PESQUISA E INOVAÇÃO TECNOLÓGICA
INSTITUTO DE ENSINO SUPERIOR FUCAPI
CENTRO DE PÓS-GRADUAÇÃO E EXTENSÃO (CPGE)

DIEGO LINS DE FREITAS

**ANÁLISE DO PADRÃO DE PROJETO MODEL VIEW PRESENTER NA PLATAFORMA
ANDROID USANDO MÉTRICAS DE QUALIDADE OO**

MANAUS

2014

DIEGO LINS DE FREITAS

**ANÁLISE DO PADRÃO DE PROJETO MODEL VIEW PRESENTER NA PLATAFORMA
ANDROID USANDO MÉTRICAS DE QUALIDADE OO**

Monografia apresentada ao curso de especialização em Engenharia de Software do Centro de Pós-graduação e Extensão - CPGE - FUCAPI, como requisito parcial para obtenção do título de Especialista em Engenharia de Software.

Orientador: Danny Lopes M.Sc.

MANAUS

2014

Resumo

A plataforma android foi adotada por vários fornecedores de smartphones e tablets promovendo uma abrangente disseminação do sistema operacional. Formou-se um grande mercado a ser explorado com fornecimento de aplicativos para as mais variadas finalidades. Para atender essa demanda é necessário desenvolver aplicativos de qualidade e com altos índices de produtividade. Porém, por se tratar de aplicativos destinados a atender necessidades específicas e com um ciclo de desenvolvimento curto, há pouca preocupação com a arquitetura da solução resultando em produtos com baixa qualidade. Dentro da engenharia de software existem os padrões de projetos que ajudam no desenvolvimento do sistema. Este trabalho tem como objetivo apresentar a aplicação do padrão de projeto Model View Presenter no desenvolvimento de aplicativos android para aumentar a qualidade do aplicativo. Os efeitos que esse padrão tem sobre o objeto de estudo são avaliados sobre uma perspectiva orientada a objetos. O método experimental é aplicado de forma iterativa usando o processo de refatoração incremental e coletando métricas de qualidade de código para fazer uma análise quantitativa. Os resultados mostram que a aplicação do padrão aumentou a coesão do código, porém, com um pequeno aumento da complexidade devido a inclusão de novos componentes usados para dividir as responsabilidades.

Palavras-chaves: Android. Model View Presenter. Padrões de projeto.

Abstract

The android platform has been adopted by several vendors of smartphones and tablets promoting a comprehensive dissemination of the operating system. Formed a large market to be exploited for application delivery the most varied purposes. To meet this demand is necessary develop quality and high productivity applications. However, for dealing with applications designed to meet specific needs and with a short, the development cycle there is little concern with the architecture of solution resulting in products with poor quality. Within the engineering there are software design patterns that help in the development of sistema. This work aims to present the application of the design pattern Model View Presenter in android application development to increase application quality. The effects that this pattern has on the object of study are assessed on an object-oriented perspective. The experimental method is iteratively applied using the process and incremental refactoring collecting quality metrics of code to do a quantitative analysis. The results show that the application of standard increased the cohesion of the code, however, with a small increase in complexity due to the inclusion of new components used to divide responsibilities.

keywords: Android. MVP. MVC. Design Patterns.

Lista de figuras

Figura 1 – Processo de Experimentação	15
Figura 2 – Exemplo de avaliação da métrica RFC	19
Figura 3 – Exemplo de avaliação da métrica CBO	20
Figura 4 – Exemplo de avaliação da métrica LCOM	21
Figura 5 – Exemplo de avaliação da métrica LCOM	22
Figura 6 – Exemplo de avaliação da métrica DIT	23
Figura 7 – Exemplo de avaliação da métrica NOC	24
Figura 8 – Exemplo de avaliação da métrica WMC	25
Figura 9 – Representação dos componentes do MVC	27
Figura 10 –MVP - Passive View	30
Figura 11 –MVP - Supervising Controller	31
Figura 12 –Android Stack	32
Figura 13 –Exemplo de execução do CKJM	33
Figura 14 –Exemplo de resultado da análise do CKJM	34
Figura 15 –Diagrama de pacotes do aplicativo de contatos	35
Figura 16 –Tela de lista de grupos	36
Figura 17 –Tela de visualização de um grupo	36
Figura 18 –Tela de edição e criação de grupos	36
Figura 19 –Diagrama de pacotes de grupos	37
Figura 20 –Gráfico da métrica WMC	40
Figura 21 –Gráfico da métrica DIT	42
Figura 22 –Gráfico da métrica CBO	43
Figura 23 –Gráfico da métrica RFC	44
Figura 24 –Gráfico da métrica LCOM	45
Figura 25 –Pacote após refatoração	46
Figura 26 –Variação das métricas ao longo das iterações	48

Lista de tabelas

Tabela 1 – Métricas CK da versão de referência	37
Tabela 2 – Dados métrica WMC	40
Tabela 3 – Dados métrica NOC	41
Tabela 4 – Dados métrica DIT	42
Tabela 5 – Dados métrica CBO	43
Tabela 6 – Dados métrica RFC	44
Tabela 7 – Dados métrica LCOM	45

Lista de siglas

MVP	Model View Presenter
MVC	Model View Controller
API	Application Programming Interface
CKJM	Chidamber and Kemerer Java Metrics
WMC	Weighted methods per class
DIT	Depth of Inheritance Tree
NOC	Number of Children
CBO	Coupling between object classes
RFC	Response for a Class
LCOM	Lack of cohesion in methods
NPM	Number of Public Methods for a class
Ce	Afferent coupling
SDK	Software Development Kit
MVVM	Model View ViewModel
MVP-VM	Model View Presenter ViewModel
MVPC	Model View Presenter Controller

Sumário

1	Introdução	9
1.1	Motivação	10
1.2	Problematização	10
1.3	Hipótese	10
1.4	Objetivos	11
1.5	Trabalhos Relacionados	11
1.6	Contribuições	13
1.7	Metodologia	13
1.7.1	Objeto de Estudo	14
1.7.2	Processo de Experimentos	14
1.8	Organização do Trabalho	16
2	Referencial Teórico	17
2.1	Princípios e Padrões de Projetos	17
2.2	Métricas de qualidade OO	18
2.3	Model View Controller	24
2.4	Model View Presenter	28
2.5	Framework Android	30
3	Execução da Pesquisa	33
3.1	Ferramentas Aplicadas	33
3.2	Análise do objeto de estudo	34
3.3	Arquitetura Proposta	38
3.4	Resultados dos Experimentos	38
3.4.1	WMC	39
3.4.2	NOC	41
3.4.3	DIT	41

3.4.4	CBO	42
3.4.5	RFC	43
3.4.6	LCOM	45
3.5	Discussão dos Resultados	46
4	Conclusão	49
4.1	Trabalhos Futuros	50
	Referências	51

1 Introdução

Segundo (IDC, 2014) o mercado de dispositivos móveis ultrapassou a marca de mais de um bilhão de aparelhos vendidos em 2013. A plataforma Android é a líder de mercado dominando 78% desse universo com mais de 226.1 milhões de Smartphones produzidos e embarcados com android no último quadrimestre de 2013. Baseado nesses dados é possível concluir que existe uma demanda latente no desenvolvimento de novos aplicativos que agreguem valor à esta plataforma.

Para produzir aplicativos de qualidade é necessário aplicar boas práticas de desenvolvimento de software. Levando-se em consideração que a linguagem de programação usada para o desenvolvimento na plataforma android é Java, é natural que se aplique as práticas definidas pelos princípios de projeto orientado a objetos. Tais princípios guiam o desenvolvedor na definição da estrutura interna do software, afetando diretamente características de qualidade como manutenibilidade, performance e outras (YANG; TEMPERO; MELTON, 2008).

Os padrões de projeto são aplicados na engenharia de software como forma de reproduzir soluções para problemas recorrentes melhorando a manutenibilidade e o reuso de componentes de software (GAMMA et al., 1995). O uso de padrões de projetos é um meio de aplicar esses princípios.

Boas práticas de engenharia de software promovem o desenvolvimento dos artefatos que constituem o sistema de forma coesa. Os conjuntos desses artefatos podem ser classificados de acordo com suas responsabilidades, surgindo uma divisão clara em camadas. Um sistema pode ter diversas camadas, responsáveis por acesso a um banco de dados, comunicação com serviços externos, encapsular regras de negócios. Uma dessas camadas que constitui um aplicativo android é a camada de apresentação ou interface com o usuário.

Segundo Pressman (2006), “A interface com o usuário pode ser considerada o

elemento mais importante de um sistema ou produto baseado em computador“. Dada a importância da camada apresentação em um sistema, será tratado nesse trabalho o uso de padrões de projeto para desenvolvimento da camada de apresentação de aplicativos android.

1.1 Motivação

O autor deste trabalho atua em projetos de desenvolvimento de aplicativos android em uma instituição de pesquisa e desenvolvimento localizada em Manaus. O comprometimento com a qualidade promoveu o uso de ferramentas de código aberto para monitoração da qualidade dos projetos, além do processo de testes adotado pela empresa. Com uma equipe composta por mais de 30 desenvolvedores produzindo aplicativos com diversas finalidades, houve a necessidade de padronização da arquitetura.

1.2 Problematização

Dado que existem uma vasta diversidade de padrões de projeto a serem aplicados no desenvolvimento de software, qual padrão de projeto é aplicável para o desenvolvimento da camada de apresentação de aplicativos android para melhorar a qualidade do produto final?

1.3 Hipótese

O padrão de projeto Model-View-Presenter é aplicável no desenvolvimento da camada de apresentação de aplicativos android, gerando um aumento da qualidade do produto final.

1.4 Objetivos

Este trabalho fará um estudo sobre a arquitetura de aplicativos android com o propósito de melhorar a qualidade desses softwares, causando uma redução nos riscos relacionados às mudanças que acontecem durante o ciclo de desenvolvimento e promovendo a reutilização de componentes, além de permitir o incremento de funcionalidades com baixo impacto. Este estudo fornecerá insumos para que os desenvolvedores possam ter uma visão geral da aplicabilidade dos padrões de projetos e analisar quais técnicas contribuem para seus projetos. Os seguintes objetivos específicos serão alcançados:

- Estudar os padrões de projeto que podem ser usados para a implementação da camada de apresentação de um aplicativo android;
- Avaliar os componentes do framework android, identificando seus papéis e responsabilidades de acordo com os padrões estudados, levando em consideração características que podem dificultar a aplicação dos padrões de projeto;
- Identificar os impactos nas características de qualidade do aplicativo de acordo com o paradigma da Orientação a Objetos, por meio de análise de métricas de qualidade de código;
- Propor um modelo para implementação da camada de apresentação de um aplicativo android utilizando padrões de projetos.

1.5 Trabalhos Relacionados

O uso de um dispositivo móvel permite inovar ao usufruí-lo para atender necessidades tanto para computação pessoal e corporativa. Conforme a variedade de aplicativos surgem, o interesse em projetá-los com mais qualidade também aumenta. Existem estudos sobre arquitetura de aplicativos Android como demonstrado por [La e Kim \(2010\)](#), onde é elaborado uma arquitetura baseada no padrão de projeto MVC que

é estendida para as camadas do sistema que estão presentes em um serviço remoto para aproveitar as vantagens que a conectividade desses aparelhos ofereçam. Outro trabalho relacionado é de [Fiawoo e Sowah \(2012\)](#) que desenvolvem um aplicativo dentro de um contexto corporativo que é usado para consultar dados a partir de um banco de dados central.

Não foi encontrado na literatura disponível, referências sobre a aplicação do MVP para desenvolvimento de aplicativos android. Outros trabalhos mostram a aplicação do padrão em outros contextos e plataformas ([ALLES et al., 2006](#)), ([ZHANG; LUO, 2010](#)). Esses trabalhos enfatizam que o MVP melhora a testabilidade, que é uma característica de qualidade de software.

A principal referência presente neste trabalho é [Chidamber e Kemerer \(1994\)](#) que elaboram um conjunto de métricas para descrever de forma quantitativa a qualidade de software orientado a objetos. [Chidamber e Kemerer \(1994\)](#) aplicam um conjunto de seis métricas em dois projetos distintos desenvolvidos em Smalltalk e C++ analisando a relação dos resultados obtidos e os fatores que influenciaram no desenvolvimento dos sistemas como por exemplo decisões arquiteturais e a qualidade dos sistemas. Essas métricas serão usadas para avaliar a qualidade do código resultante do processo de refatoração.

As métricas [Chidamber e Kemerer \(1994\)](#) tem se demonstradas como uma forma de avaliar a qualidade de um sistema orientado a objetos. [Dubey e Rana \(2011\)](#) resumem uma série de estudos que mostram o aumento da manutenibilidade do código ao manter os valores das métricas baixos. Outras características de qualidade de software são verificadas por meio de métricas de código como demonstrado por [Khalid, Zehra e Arif \(2010\)](#) que usa essas métricas para avaliar a complexidade e a testabilidade de sistemas orientados a objetos. Segundo [Briand et al. \(2000\)](#), características como acoplamento, coesão e herança estão relacionados com uma probabilidade maior de ocorrer falhas, e avalia essa relação usando métricas para analisar a estrutura de projetos de software. [Chidamber, Darcy e Kemerer \(1998\)](#) relacionam os resultados dessas métricas com a produtividade, esforço de retrabalho e projeto.

Na dissertação de [Türk \(2009\)](#) é feita uma análise dos impactos na qualidade do código ao aplicar cinco padrões de projetos em um software de comunicação TCP/IP. Os resultados do trabalho citado mostram que a qualidade do software aumenta ao aplicar padrões de projetos, sendo que o principal atributo de qualidade aferido é a manutenibilidade. O método utilizado para a execução desta pesquisa se baseia no trabalho citado.

1.6 Contribuições

Tendo em vista que padrões de projetos descrevem soluções abstratas, este trabalho tem como principal contribuição uma interpretação do padrão de projeto Model-View-Presenter, proporcionando uma referência prática aplicada ao framework android. A diversidade de projetos de software torna difícil inferir se as métricas podem ser consideradas um parâmetro para determinar a qualidade. O presente trabalho se propõe a avaliar a aderência do padrão MVP a projetos android por meio de métricas de qualidade de código.

1.7 Metodologia

Esta pesquisa se caracteriza como experimental pois é selecionado um objeto de estudo que será analisado do ponto de vista de métricas bem definidas. Essas métricas serão influenciadas pelos experimentos de refatoração embasados na revisão da literatura existente sobre orientação a objetos e seus princípios.

Para validar a hipótese apresentada neste trabalho a pesquisa terá uma abordagem quantitativa por meio da análise de dados estatísticos de métricas que expressem atributos de qualidade em software orientado a objetos. O conjunto de métricas a serem usadas nessa validação será o elaborado por [Chidamber e Kemerer \(1994\)](#).

Essas medidas serão coletadas por meio de um procedimento experimental em laboratório utilizando um processo iterativo-incremental para executar refatorações

no código do objeto de estudo afim de introduzir o padrão de projeto Model-View-Presenter e a cada iteração será feita a coleta das métricas. Para executar esse processo será identificado um conjunto de funcionalidades que o aplicativo atende, relacionada com a camada de apresentação com a qual o usuário interage.

1.7.1 Objeto de Estudo

O projeto a ser refatorado será o aplicativo de Contatos do android¹, que é um dos aplicativos básicos pré-instalados com o sistema operacional. Este projeto é opensource mantido pelo The Android Open Source Project suportado pela Google com contribuições de desenvolvedores do mundo todo.

Os critérios para a escolha do objeto de estudo são:

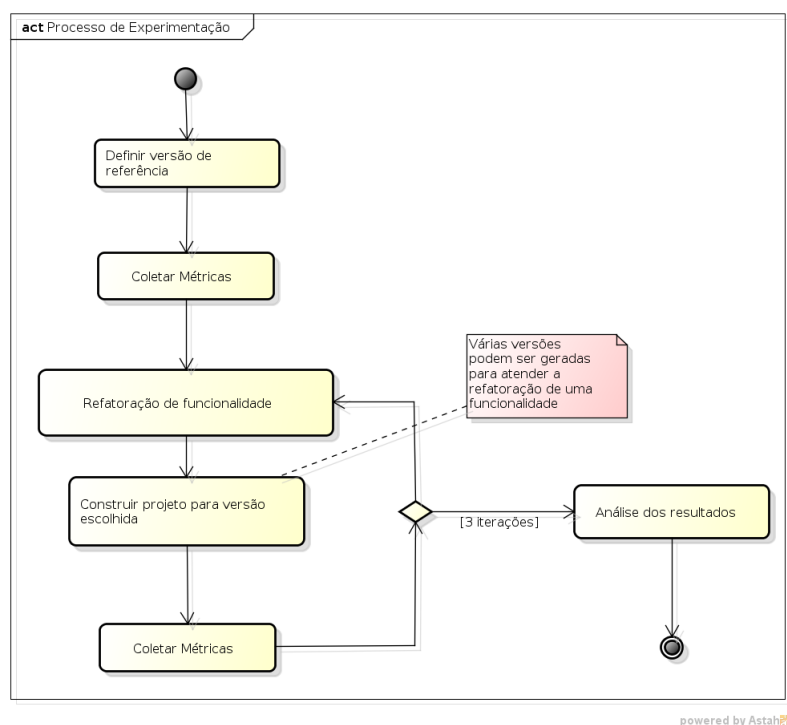
1. Tamanho/Complexidade - Um projeto muito complexo iria inviabilizar a pesquisa devido ao esforço para fazer a refatoração. Métricas coletadas a partir de projetos simples e triviais não forneceria dados suficientes para uma análise satisfatória.
2. Código Aberto - Além de permitir o acesso ao código fonte sem limitações para a pesquisa, tem a contribuição de vários desenvolvedores com experiência e formação em programação diversificada que se refletem no código fonte.
3. Origem do projeto - A escolha de um aplicativo mantido sobre o mesmo gerenciamento que o sistema operacional android foi feita com o intuito de fazer a pesquisa em um código fonte que expressasse as técnicas e práticas de programação difundidas nesse ecossistema.

1.7.2 Processo de Experimentos

A Figura 1 demonstra as atividades do processo de experimentação adotado:

¹ <https://android.googlesource.com/platform/packages/apps/Contacts>

Figura 1 – Processo de Experimentação



Fonte: Próprio Autor

Definir versão de referência - Delimitar um marco do estado do código no repositório.

Refatoração de funcionalidade Esta atividade tem como objetivo aplicar os padrões propostos em uma funcionalidade do aplicativo.

Construir projeto Executar o processo de construção do projeto que inclui a compilação das classes para fazer a coleta das métricas.

Coleta de Métricas Este passo tem como objetivo fazer a coleta das métricas do código que se encontra no repositório a partir de uma revisão para fazer a avaliação dos efeitos da refatoração na qualidade do código.

Análise dos resultados Discussão dos impactos das alterações executadas nas métricas.

1.8 Organização do Trabalho

O presente trabalho está estruturado em 4 capítulos dos quais este é o Capítulo 1 que apresenta a proposta do trabalho, contexto do problema, a motivação para este estudo, os objetivos a serem alcançados e a estrutura do trabalho. Também é abordada a metodologia aplicada neste trabalho mostrando o objeto de estudo a ser analisado e os passos a serem executados, enfatizando o método experimental. O Capítulo 2 discorre sobre a fundamentação teórica sobre padrões de projetos, métricas de qualidade orientada a objetos e as tecnologias com as quais o objeto de estudo foi desenvolvido. O Capítulo 3 contém uma análise do objeto de estudo e define como serão feitas as implementações e mostra os resultados da execução da pesquisa. O Capítulo 4 apresenta a análise final, conclusões do trabalho e sugestões para trabalhos futuros.

2 Referencial Teórico

2.1 Princípios e Padrões de Projetos

Desenvolver software orientado a objetos é um desafio. Criar uma representação computacional de uma faceta da realidade em que seus constituintes trabalhem de forma harmoniosa para atingir as necessidades que o software se propõe a atender requer experiência, conhecimento do domínio do problema e um processo de análise e projeto. Apesar de existirem várias abordagens para se conceber um sistema orientado a objetos([EVANS, 2004](#)),([GOMAA, 2011](#)), um sistema bem construído apresenta características fundamentais como alta coesão e baixo acoplamento.

A Coesão é uma característica de um componente de software que se refere ao grau de relacionamento entres os membros desse componente. No contexto de uma classe, é levado em consideração as relações entre os métodos e atributos. Classes com coesão baixa demonstram grande complexidade pois os atributos e métodos que não se relacionam indicam que a classe tem muitas responsabilidades.

O acoplamento descreve as dependências entre componentes. Quanto maior a quantidade dessas dependências entre classes, mais complexo ela se torna, pois dificulta alterações na classe. Além disso, o acoplamento aumenta o risco de uma classe ser afetada devido à alterações em suas dependências.

Um padrão, dentro do contexto de estudo deste trabalho pode ser definido como uma técnica efetiva cuja a sua aplicabilidade é aceita e difundida dentro de uma área de conhecimento com a intenção de atingir um objetivo([METSKE; WAKE, 2006](#)). Ao seguir um padrão evita-se o retrabalho de resolver um problema cuja a solução já existe.

Um padrão de projeto descreve uma solução para algum problema recorrente em um sistema orientado a objeto ([GAMMA et al., 1995](#)). Essa descrição envolve as

motivações para a utilização daquele padrão, a estrutura de classes e objetos participantes do padrão e contextualização de sua aplicabilidade. Em desenvolvimento de software, o catálogo mais difundido de padrões de projetos orientado a objetos é elaborado por [Gamma et al. \(1995\)](#), contendo um total de 23 padrões formalmente documentados que acumulam experiências bem sucedidas em diversos sistemas. Esses padrões têm a seguinte classificação:

Criacionais Padrões que definem como criar novas instâncias de classes.

Estruturais Foca na estruturação das classes e objetos.

Comportamentais Definem como as classes e objetos interagem entre si e suas responsabilidades.

Analisando a forma como um padrão de projeto é concebido, com uma definição dos papéis de cada elemento participante e como eles interagem entre si, pode-se concluir que o uso dos padrões de projeto promove maior coesão, melhor separação de interesses e baixo acoplamento no sistema. Todas essas características contribuem para um software de melhor qualidade.

2.2 Métricas de qualidade OO

Com o advento de novas técnicas de desenvolvimento de software é necessário obter informações do impacto dessas inovações nos resultados de um projeto. Com esse objetivo [Chidamber e Kemerer \(1994\)](#) criaram um conjunto de métricas para medir a qualidade de sistemas desenvolvidos usando o paradigma orientado a objetos que não se limitasse a uma linguagem de programação, fácil de coletar e com forte embasamento teórico na ontologia de Bunge que é um modelo usado pra analisar algumas propriedades estáticas e dinâmicas de um sistema de informação([WAND; WEBER, 1990](#)). As métricas são:

Response sets for Class (RFC) Quantidade de métodos que são executados quando um objeto recebe uma mensagem, incluindo os métodos de outras classes. É levado em consideração somente os métodos que são chamados diretamente.

Figura 2 – Exemplo de avaliação da métrica RFC



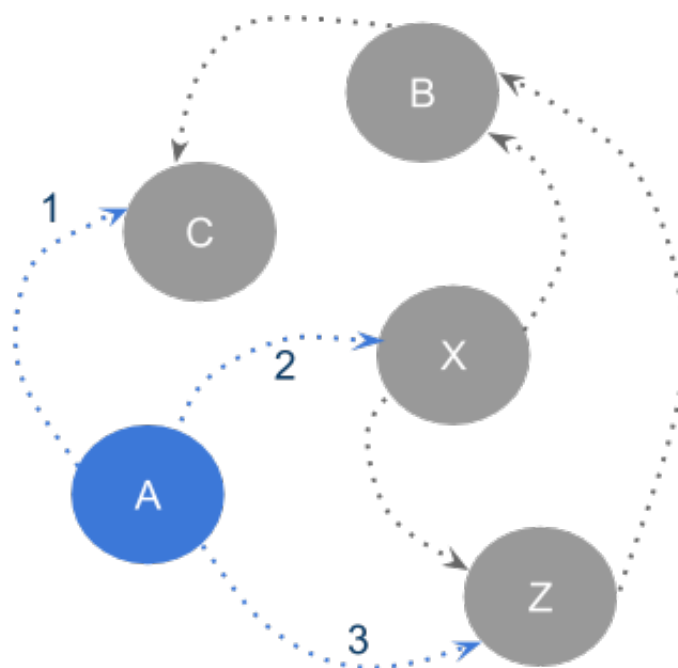
Fonte: Próprio Autor

Na figura 2, a classe de exemplo tem um total de três métodos definido nela. A implementação do método "MetódoA" chama o método "MétodoB", o método "MétodoW" que é de outra classe referenciada pelo atributo "Atributo2" e o método "MétodoC". Assumindo que os métodos "MétodoB" e "MétodoC" não contém chamadas a nenhum outro método, o valor da métrica RFC para a classe ilustrada é de seis métodos.

Coupling Bettwen Objects (CBO) Em um sistema orientado a objetos, diversas classes são implementadas para exercer uma respoonsabilidade, as instâncias dessa classes interagem entre si para atingir o objetivo do sistema. Uma classe está acoplada a outra quando o método de um classe invoca o método de uma variável de instância de outra classe que gera uma dependência entre essas classes. Quanto mais dependências existirem, mais difícil é reutilizar esses componentes em outras partes do sistema, além do aumento do risco de efeitos

colaterais que ocorrerem ao modificar uma classe altamente acoplada. O valor da métrica CBO para uma classe é o número de outras classes que a classe analisada depende.

Figura 3 – Exemplo de avaliação da métrica CBO



Fonte: Próprio Autor

A figura 3 ilustra um exemplo de acoplamento entre objetos. Um classe chamada “A” depende das classes “C”, “X” e “Z”. O valor de CBO para a classe “A” é a soma de todas as suas dependências, neste caso totalizando três dependências.

Lack of Cohesion in Methods (LCOM) Usado para avaliar a coesão de uma classe por meio dos relacionamentos entre seus atributos e métodos. O valor dessa métrica é fornecido por meio da quantidade de métodos que não tem nenhum atributo em comum, menos a quantidade de métodos que tem atributos em comum.

No exemplo da figura 4 a classe ilustrada tem três métodos implementados e cada método tem um conjunto de atributos que cada um utiliza. O método “MétodoA” utiliza o atributo “Atributo1” formando o conjunto $Ma = \{\text{Atributo1}\}$,

Figura 4 – Exemplo de avaliação da métrica LCOM



Fonte: Próprio Autor

o método “MétodoB” não utiliza nenhum atributo formando um conjunto vazio $M_b = \{\}$, e o método “MetodoC” referencia o atributo “Atributo2” formando o conjunto $M_c = \{\text{Atributo2}\}$. É necessário descobrir a interseção de cada um desses grupos para saber quantos grupos tem atributos em comum. A seguir é mostrado o resultado dessa avaliação:

$$M_a \cap M_b = \{\}, M_a \cap M_c = \{\}, M_c \cap M_b = \{\}$$

Com esses resultados em mãos pode-se calcular o valor de lcom para a classe exemplos. O total de conjuntos resultates vazios é três e não foi identificado nenhum conjunto com algum atributo que seja compartilhado entre os métodos. Portanto o LCOM para para a classe “ClasseA” é, três conjuntos vazios, menos zero conjuntos não vazios, que é igual a, LCOM três.

Outro exemplo é mostrado na figura 5 . Aplicando a mesma nomenclatura do exemplo anterior da figura 4 teremos a seguinte análise para calcular o valor de LCOM para a classe “ClasseB”: Dado, $M_a = \{\text{Atributo1}\}$; $M_b = \{\}$; $M_c = \{\text{Atributo1}, \text{Atributo2}\}$.

$$M_a \cap M_b = \{\}, M_a \cap M_c = \{\text{Atributo1}\}, M_c \cap M_b = \{\}$$

Figura 5 – Exemplo de avaliação da métrica LCOM



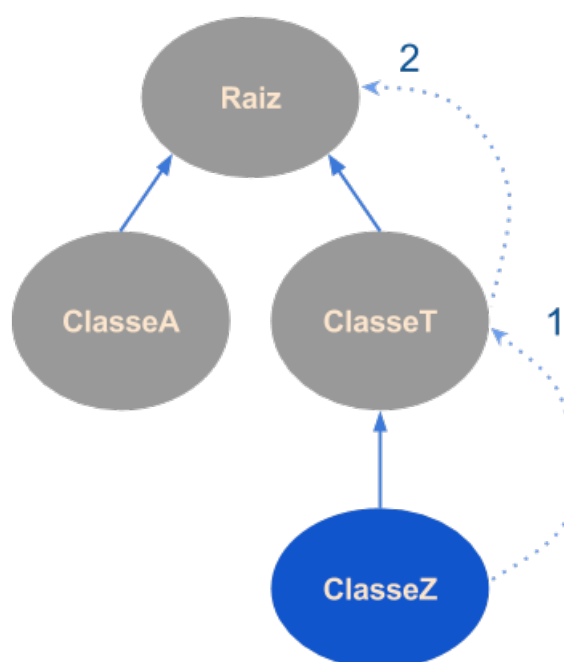
Fonte: Próprio Autor

Totalizando dois conjuntos vazios e um conjunto não vazio. Portanto o valor da métrica LCOM para a classe “ClasseB” é de um conjunto. Podemos concluir que a classe “ClasseB” é mais coesa que a classe “ClasseA”.

Depth Inheritance Tree (DIT) Nível de uma classe na hierarquia de herança. Reflete o número máximo de nós dentro da árvore de classes até a raiz, o que aumenta a complexidade conforme a quantidade de elementos envolvidos se eleva, diminuindo a previsibilidade do comportamento da classe com vários métodos e atributos sendo herdados, principalmente com o uso de sobrecarga de métodos. No exemplo da figura 6 a classe “ClasseZ” tem um relacionamento de herança com a classe “ClasseT” que se relaciona também por meio de herança com a classe “Raiz”. A distância entre a classe “ClasseZ” até a classe “Raiz” na hierarquia é de dois nós. Portanto o valor da métrica DIT para a classe “ClasseZ” é dois.

Number of Children (NOC) Número de subclasses imediatas de uma classe analisada. Essa medida é um indicativo de mau uso de herança conforme seu valor aumenta e mostra o impacto que uma classe pode ter no sistema requerendo maior atenção e testes.

Figura 6 – Exemplo de avaliação da métrica DIT



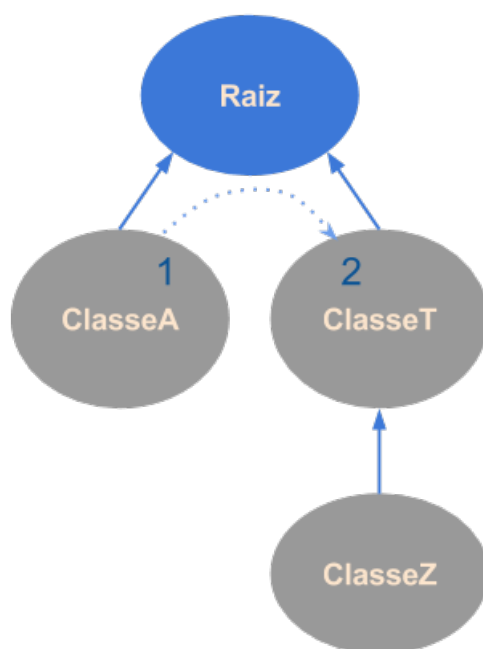
Fonte: Próprio Autor

Na figura 7 a classe “ClasseA” e a classe “ClasseT” são as únicas classes que tem uma relação de herança direta com a classe “Raiz”, portanto o valor da métrica NOC para a classe “Raiz” é dois.

Weighted Methods per Classe (WMC) Serve para expressar o nível de complexidade de uma classe com base na soma da complexidade dos métodos que ela possui. Isso afeta o esforço de manutenção da classe, além de impactar nas classes filhas que herdarão esses métodos. Também é um indicativo de que a classe tem métodos específicos dificultando o seu reuso. A unidade de complexidade definida por Chidamber e Kemerer (1994) é o próprio método e não considera outros fatores como número de linhas, número de parâmetros ou quantidade de estruturas de decisão. A figura 8 ilustra como a métrica WMC é avaliada em uma classe.

No exemplo da figura 8, a classe representada tem um total de cinco métodos implementados. Seguindo estritamente o que Chidamber e Kemerer (1994) definem para a métrica, o valor da métrica WMC para a classe é cinco métodos.

Figura 7 – Exemplo de avaliação da métrica NOC



Fonte: Próprio Autor

Todas essas métricas tem uma relação inerente com a coesão e acoplamento dos objetos, sendo uma forma confiável para a análise da qualidade em sistemas orientados a objetos. Os aplicativos desenvolvidos para a plataforma android são escritos usando a linguagem de programação Java que emprega esse paradigma de desenvolvimento, o que justifica o uso das métricas de [Chidamber e Kemerer \(1994\)](#) para validação dos projetos orientados a objetos.

2.3 Model View Controller

O padrão *Model View Controller* surgiu como uma solução genérica para que usuários de um sistema de planejamento manipulem dados complexos [Reenskaug \(1979\)](#). Posteriormente, [Krasner e Pope \(1988\)](#) implementam um framework MVC para o ambiente gráfico da linguagem de programação Smalltalk-80 como uma forma de promover a reusabilidade e plugabilidade dos componentes de um sistema.

Segundo [Reenskaug \(1979\)](#) o principal objetivo do MVC "...é representar o

Figura 8 – Exemplo de avaliação da métrica WMC



Fonte: Próprio Autor

modelo mental do usuário de um espaço de informações relevantes e permitir que o usuário inspecione e altere estas informações.”(tradução livre). Esse modelo mental é como o usuário percebe o domínio do problema que está inserido no qual executará suas atividades sobre dados de seu interesse. Para que o usuário de um sistema de informação possa interagir com a representação computacional de seu modelo mental três componentes são definidos:

Model - É o componente constituído de uma composição de classes que implementam as regras de negócio referentes as funcionalidades que o programa provê, representa o conhecimento que o usuário tem e como manipulá-lo. Atende mensagens da view requisitando seu estado e mensagens do controller para mudar seu estado,

View - Representação específica de um model na interface com o usuário, é responsável por toda a manipulação visual, recuperando um estado do model e exibindo os dados, podendo ser composta por sub-views e ser parte de views mais complexas.

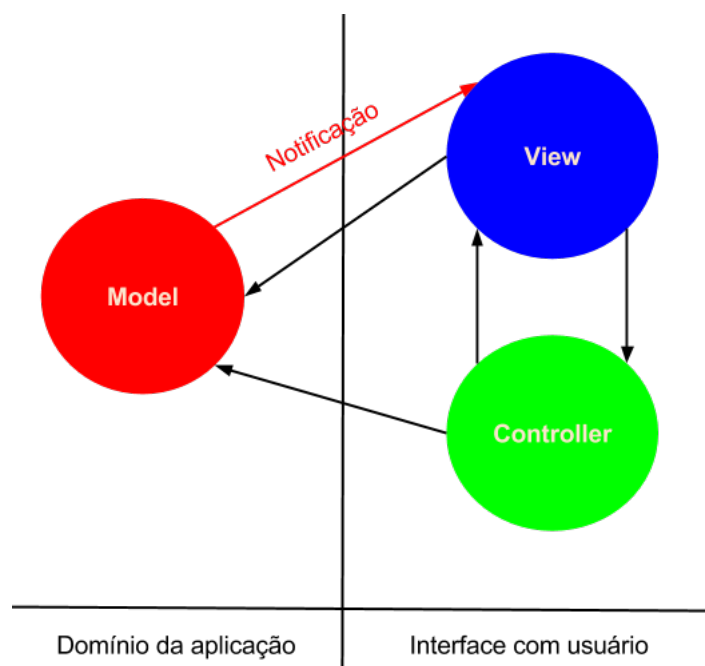
Controller - Interpreta as ações do usuário provenientes de um dispositivo de entrada(Teclado, Mouse) alterando estado da view ou do model.

Krasner e Pope (1988) descrevem a estrutura do MVC onde a view tem seu controller exclusivo mantendo uma dependência cíclica entre ambos. Tanto a View quanto o Controller tem referências diretas para o model por meio de atributos de classe, porém, o model não deve conhecer seus respectivos pares de View-Controller para promover maior reuso de código e encapsulamento do model. As alterações do estado do model são feitas na maioria das vezes pelo controller, e o model é responsável por notificar todas as views que o representa para que se atualizem refletindo o novo estado. No caso de um model ser usado por vários pares de View-Controller as mensagens de notificação de um novo estado do model podem ser parametrizadas assim cada view pode verificar se a alteração é de seu interesse.

Segundo Fowler (2002) "...esta separação da apresentação e modelo é uma das mais fundamentais heurísticas de bom projeto de software"(tradução livre). O controller poderia ser o responsável por publicar as alterações no estado do model devido sua relação direta com o mesmo, mas em casos onde o model é alterado por outro componente que não é um dos controladores que os utilizam, é necessário que o model conheça as views que devem ser notificados do novo estado. Para que essas alterações de estado sejam propagadas a view e o controller são registrados como dependentes de seu model. O padrão é descrito levando em consideração as aplicações desenvolvidas em Smalltalk usando características específicas dessa linguagem de programação que dá suporte à implementação dos três componentes como por exemplo o gerenciamento dos objetos que são dependentes do model definido na classe Object que o model deve estender. A Figura 9 esclarece a interação entre os componentes.

Gamma et al. (1995) cita Krasner e Pope (1988) fazendo uma análise dos objetos que compõem o MVC relacionando-os com outros padrões de projeto descritos em seu catálogo. O desacoplamento entre a View e o Model, somado à propagação das mudanças de estado no model para a View e o Controller podem ser descritos como uma implementação do padrão Observer. Segundo (GAMMA et al., 1995) o padrão Observer "...Define uma dependência de 1-N entre objetos para quando um

Figura 9 – Representação dos componentes do MVC



Fonte: Próprio Autor

objeto mudar de estado, todos os seus dependentes são notificados e atualizados automaticamente”(tradução livre). [Krasner e Pope \(1988\)](#) descrevem essa estrutura com o conceito de Dependents onde a View e o Controller são registrados como dependentes do model. O padrão Observer é composto por uma estrutura em que um objeto precisa publicar mudanças em seu estado, detém a referência para os objetos a serem notificados. A organização das classes de acordo com o que o padrão descreve permite que várias Views possam se atualizar dada uma modificação no Model.

A View é composta por vários objetos com a responsabilidade em comum de representar parte ou todo o estado do Model. Este é um exemplo do padrão de projeto Composite. Segundo [Gamma et al. \(1995\)](#) o padrão Composite “... Integra objetos em uma estrutura de árvore para representar uma hierarquia parte-todo, deixando clientes tratar objetos individuais e composições de objetos uniformemente”(tradução livre). Uma view pode ser constituída por sub-views para compor Views complexas. No Composite um conjunto de componentes visuais podem ser tratadas de forma encapsulada onde cada um implementa a mesma abstração.

Segundo [Gamma et al. \(1995\)](#) o padrão de projeto Strategy "...define uma família de algoritmos, encapsula cada um, e torna-os substituíveis entre si permitindo que variem independentemente dos clientes que o usam"(tradução livre). O padrão Strategy define uma abstração cuja as implementações podem ser trocadas de acordo com algum critério. Esse conceito pode ser aplicado ao Controller que encapsula o algoritmo que vai alterar a View e o Model, permitindo sua substituição, por exemplo, por uma outra implementação que deixa de responder às interações com o usuário.

Segundo [Krasner e Pope \(1988\)](#) o Model "...pode ser simples como um valor numérico inteiro (como o modelo de um contador) ou um valor literal (como o modelo de um editor de texto), ou pode ser um objeto complexo"(tradução livre). O Model pode ser implementado usando o padrão Facade para simplificar as interações entre o Controller e o Model. O padrão Facade provê uma abstração mais simples e unificada de várias interfaces de um subsistema([GAMMA et al., 1995](#)). Esta forma de interpretar o Model pode ser aplicada dependendo da complexidade do domínio que ele representa.

2.4 Model View Presenter

O MVP é um modelo de programação para implementação de interfaces com o usuário desenvolvido como um framework para C++ e Java, criado por uma subsidiária da IBM chamada Taligent, Inc. Este padrão é baseado no MVC e descreve vários componentes que tem as responsabilidades de como gerenciar os dados da aplicação e como o usuário interage com esses dados, tendo como objetivo promover o encapsulamento do Model, reuso de lógica de negócio e o polimorfismo da View. São Eles:

Model Tem as mesmas responsabilidades que o Model definido pelo MVC.

Selections - Abstração para selecionar um subconjunto dos dados existentes no model.

Commands Representa as operações a serem executadas sobre uma Selection do Model.

View Responsável por exibir o model assim como no MVC.

Interactor Mapeia as interações do usuário na view como eventos do mouse.

Presenter O papel do presenter é interpretar os eventos iniciados pelo usuário executando a lógica de negócio correspondente implementada em um command para manipular o model (POTEL, 1996).

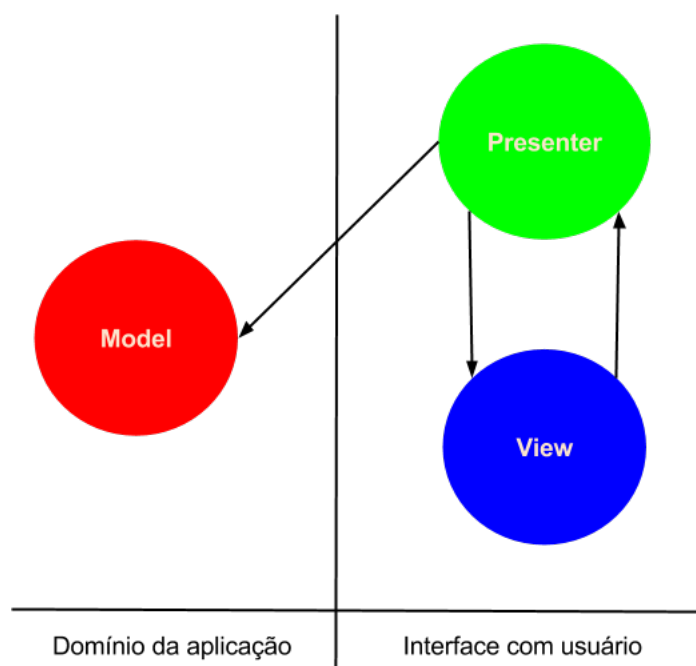
Os conceitos do MVP são descritos em Potel (1996) de forma genérica permitindo interpretações para uma implementação efetiva. Bower e McGlashan (2000) descrevem a implementação de um framework para Dolphin Smalltalk¹ adotando os conceitos do MVP onde salienta que a maioria dos sistemas operacionais com ambiente gráfico fornece um conjunto de componentes (Widgets) nos quais estão contidas as responsabilidades do Controller, de acordo com a descrição do padrão MVC. A maior parte do comportamento com o usuário é implementada no Presenter que está diretamente associado à View.

Ainda acerca das responsabilidades do Presenter, Fowler (2006a) descreve o que é chamado de Passive View, onde toda a lógica do comportamento da view é implementado no presenter deixando a view enxuta com o intuito de isolar ao máximo a API gráfica do resto da aplicação. Dessa forma o model não se comunica com a view por meio do padrão Observer, sendo que a view será atualizada pelo presenter como pode ser observado na Figura 10.

Outra variação do MVP descrita por Fowler (2006b) é o Supervising Controller. Nesta abordagem o Model utiliza algum mecanismo de notificação, por exemplo o padrão Observer, para atualizar a View. Este comportamento é parecido com o do padrão MVC como pode ser observado na figura 11.

¹ Implementação da Linguagem de programação Smalltalk - <http://www.object-arts.com>

Figura 10 – MVP - Passive View



Fonte: Próprio Autor

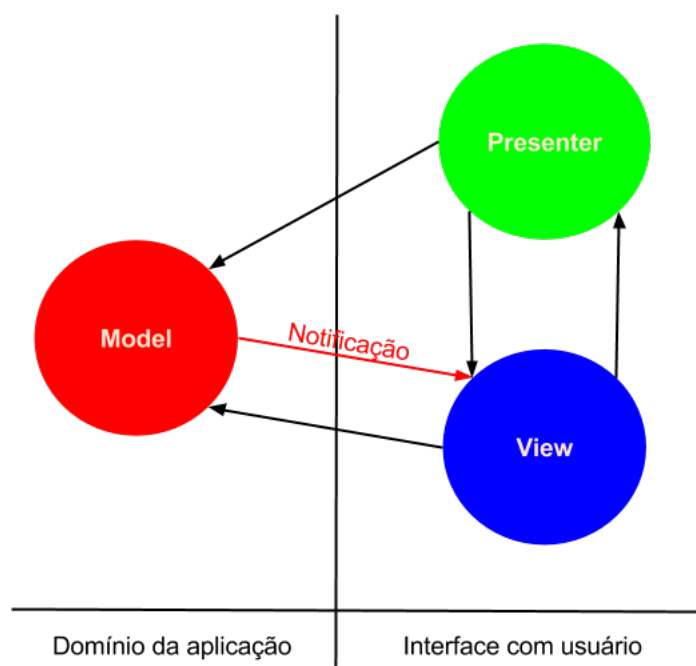
MVP se adequa melhor as API's gráficas existentes e define de forma mais clara os componentes necessários para desenvolver uma aplicação, sendo o ponto de maior discussão reside em quais os limites das responsabilidades no que tange a mediação do Model e a View por parte do Presenter.

2.5 Framework Android

O android é uma plataforma baseado no linux mantido pela Google para ser embarcado em dispositivos podendo ser aplicado em carros, televisão, placas controladoras mas seu destaque é a utilização em smartphones e tablets, que é o foco deste trabalho. A plataforma é constituída por API's e frameworks tendo em sua base o sistema operacional e seus drivers seguido da máquina virtual que executa os aplicativos android e bibliotecas auxiliares e aplicativos básicos como é demonstrado na figura 12.

Para desenvolvimento é usada a API disponível no SDK que define os blocos

Figura 11 – MVP - Supervising Controller



Fonte: Próprio Autor

de construção de um aplicativo, a saber:

Activity Representa uma atividade que o usuário executa no aplicativo em um determinado momento. É um agregador de componentes visuais e responde à interações do usuário.

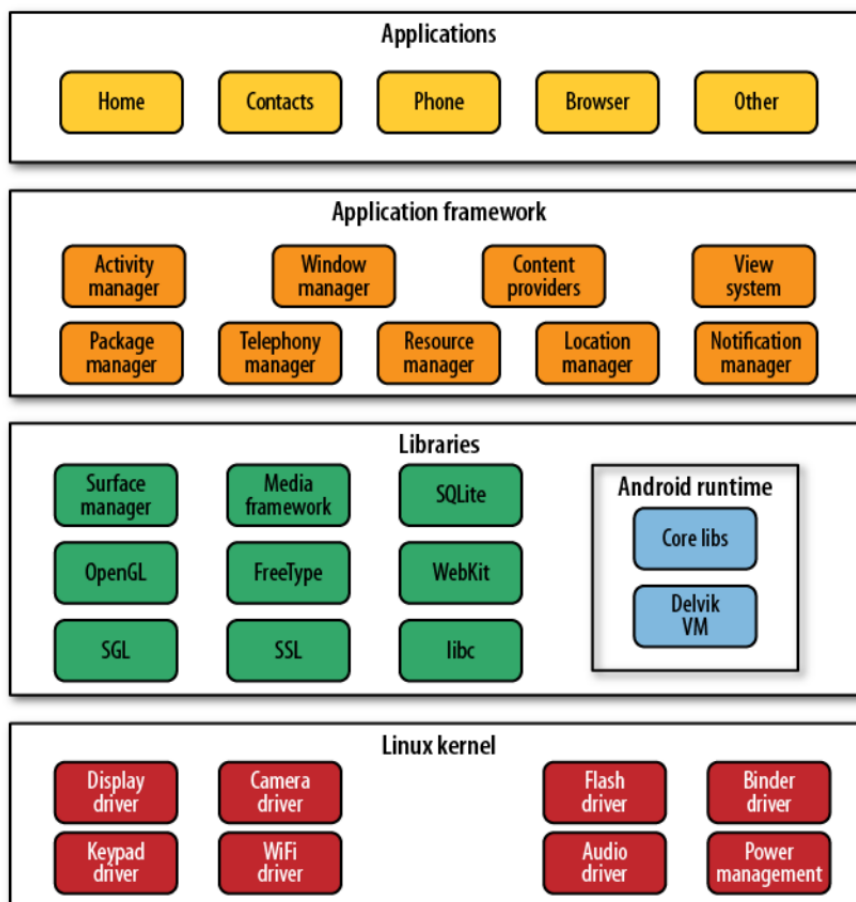
Fragment Representa uma parte de interface com o usuário em uma Activity.

Service Responsável por executar uma operação sem interface gráfica indicado para processamentos longos como por exemplo a execução de uma música ou download de arquivos.

Broadcast Receiver Implementação do padrão publish/subscribe

Content Provider Usado para expor dados de uma aplicativo para outros aplicativos. Os dados podem ser provenientes de qualquer forma de armazenamento como um arquivo ou banco de dados.

Figura 12 – Android Stack



Fonte: [Gargenta \(2011\)](#)

ApplicationContext Representa a aplicação em execução provendo acesso a recursos.

AsyncTask Usado para facilitar o uso de Threads evitando o uso da linha de execução principal do aplicativo que é responsável por tratar as interações com o usuário.

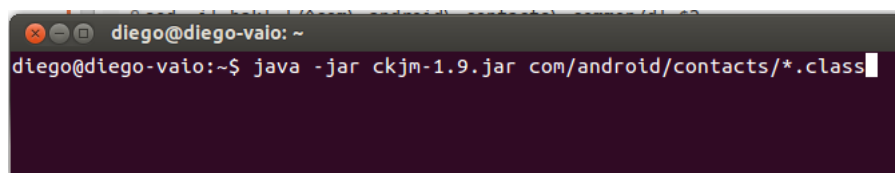
Com base nos componentes de framework e literatura revisada é possível fazer uma análise dos mesmos e projetar uma camada de apresentação utilizando o padrão MVP para ser usada como referência de implementação a ser aplicada.

3 Execução da Pesquisa

3.1 Ferramentas Aplicadas

O código será versionado no Github¹ onde será feito o gerenciamento das versões de cada iteração. As ferramentas utilizadas para a refatoração serão a IDE Eclipse(Juno) com plugin ADT v21 para facilitar a edição do código e ferramentas de construção do projeto existentes no próprio repositório do android, tendo em vista que todo o processo de compilação e empacotamento não visa ser usado em uma IDE. Para realizar a coleta das métricas é necessário que a ferramenta analise código java e contemple todas as métricas descritas na seção 2.2. O programa Chidamber and Kemerer Java Metrics(CKJM)² atende esses critérios, além de ser um projeto de código aberto. O CKJM é uma aplicação java sem interface gráfica, executado por linha de comando. Ele analisa o código java compilado, conhecido como byte codes contido em arquivos com extensão .class. Um exemplo de uso do CKJM é mostrado a seguir:

Figura 13 – Exemplo de execução do CKJM

A terminal window with a dark background. The prompt is 'diego@diego-vaio: ~'. The command entered is 'java -jar ckjm-1.9.jar com/android/contacts/*.class'. The command is highlighted with a white cursor at the end.

```
diego@diego-vaio: ~  
diego@diego-vaio:~$ java -jar ckjm-1.9.jar com/android/contacts/*.class
```

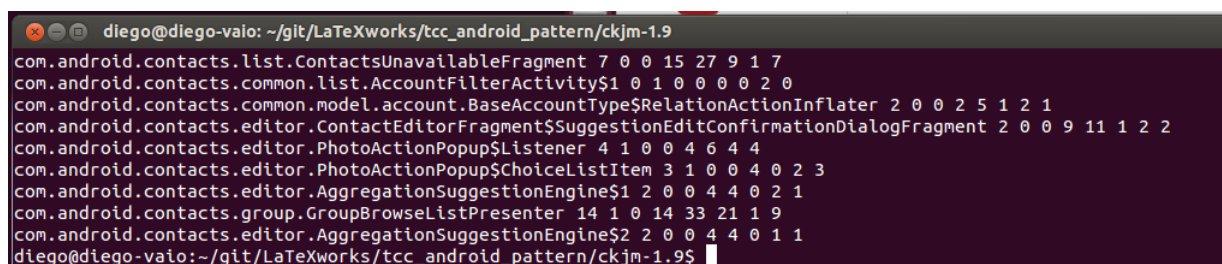
Fonte: Próprio Autor

Os resultados são escritos na saída padrão do sistema, neste caso, no terminal de execução, mostrando uma lista com os nomes completos das classes analisadas, seguidas dos respectivos valores das métricas na seguinte ordem: WMC, DIT, NOC, CBO, RFC, LCOM, Ce, and NPM sendo que as métricas Ce e NPM são desconsideradas para esta pesquisa. a figura 14 ilustra o resultado de uma coleta.

¹ https://github.com/diegofreitas/platform_packages_apps_contacts

² <https://github.com/dspinellis/ckjm>

Figura 14 – Exemplo de resultado da análise do CKJM



```
diego@diego-vaio: ~/git/LaTeXworks/tcc_android_pattern/ckjm-1.9
com.android.contacts.list.ContactsUnavailableFragment 7 0 0 15 27 9 1 7
com.android.contacts.common.list.AccountFilterActivity$1 0 1 0 0 0 0 2 0
com.android.contacts.common.model.account.BaseAccountType$RelationActionInflater 2 0 0 2 5 1 2 1
com.android.contacts.editor.ContactEditorFragment$SuggestionEditConfirmationDialogFragment 2 0 0 9 11 1 2 2
com.android.contacts.editor.PhotoActionPopup$Listener 4 1 0 0 4 6 4 4
com.android.contacts.editor.PhotoActionPopup$ChoiceListItem 3 1 0 0 4 0 2 3
com.android.contacts.editor.AggregationSuggestionEngine$1 2 0 0 4 4 0 2 1
com.android.contacts.group.GroupBrowseListPresenter 14 1 0 14 33 21 1 9
com.android.contacts.editor.AggregationSuggestionEngine$2 2 0 0 4 4 0 1 1
diego@diego-vaio:~/git/LaTeXworks/tcc_android_pattern/ckjm-1.9$
```

Fonte: Próprio Autor

3.2 Análise do objeto de estudo

O aplicativo a ser refatorado tem funcionalidades para gerenciamento de contatos e é composto 153 classes organizadas em 13 pacotes. O padrão de projeto MVP será aplicado em uma parte do aplicativo. Será refatorado o pacote referente ao gerenciamento de grupos de contatos presente no pacote **com.android.contacts.group**. A figura 15 mostra os pacotes que compõem o aplicativo. O pacote a ser refatorado está destacado em azul.

A cada iteração será selecionada uma tela do aplicativo para a aplicação do padrão MVP. Cada tela é implementada por uma classe e suas interfaces são ilustradas nas figuras 16, 17 e 18.

As classes a serem refatoradas são:

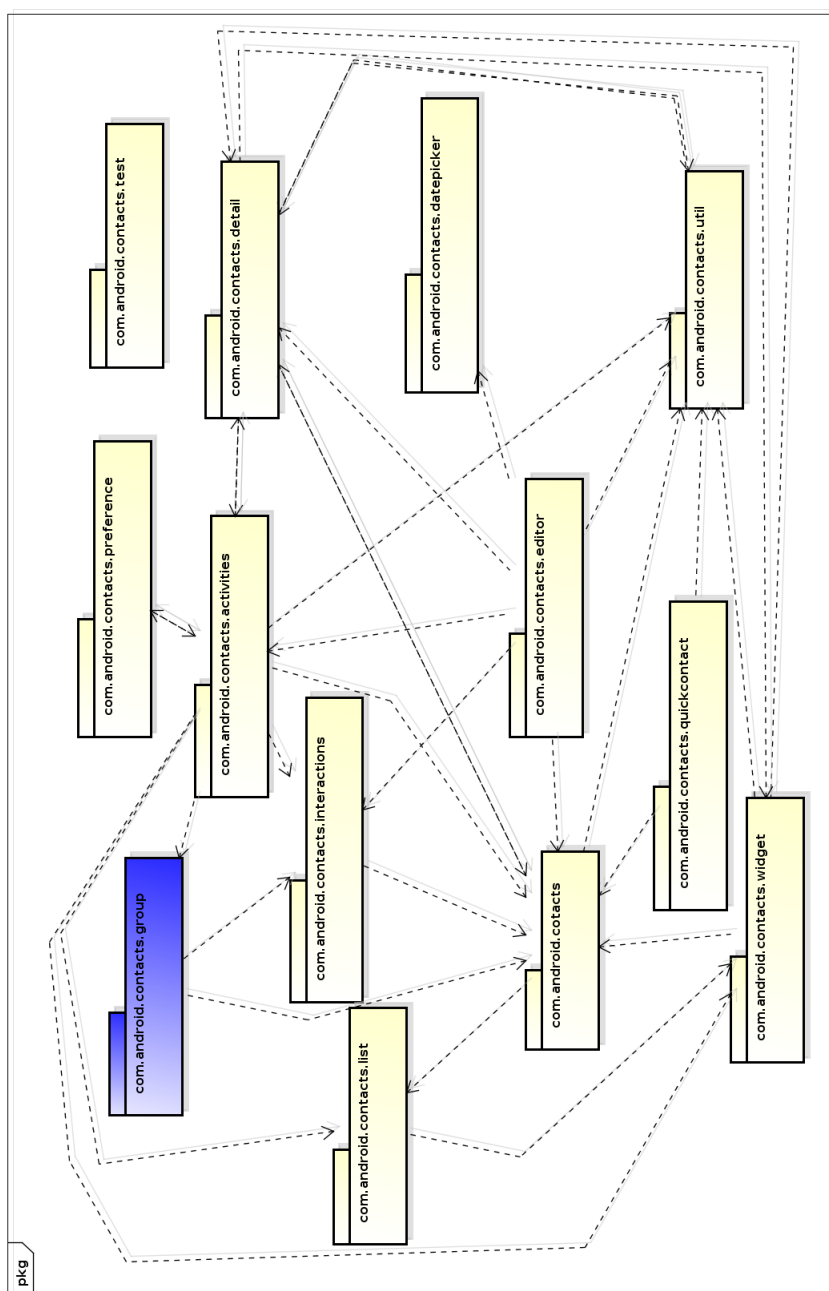
GroupDetailFragment.java Exibe os dados de um grupo de contatos.

GroupBrowseListFragment.java Fornece uma lista de grupos.

GroupEditorFragment.java Disponibiliza um formulário para edição dos dados de um grupo.

Essas classes estão presentes no pacote **com.android.contacts.group** como mostra a figura 19.

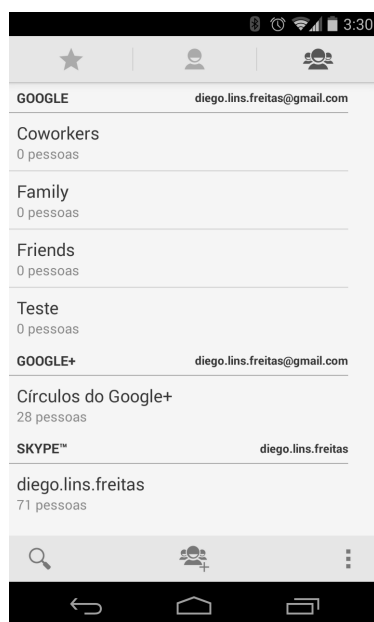
Figura 15 – Diagrama de pacotes do aplicativo de contatos



Fonte: Próprio Autor

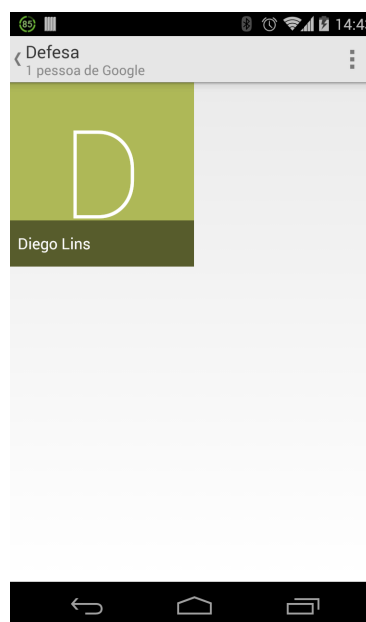
Estas interfaces com o usuário são usadas dentro de Activities que controlam uma parte do fluxo de interação e se comportam de forma diferente conforme o tipo de dispositivo móvel utilizado (tablet ou smartphone). Devido a essa complexidade, não será feita nenhuma alteração na interface pública dos componentes refactorados, evitando efeitos colaterais em outras partes do aplicativo. Os componentes

Figura 16 – Tela de lista de grupos



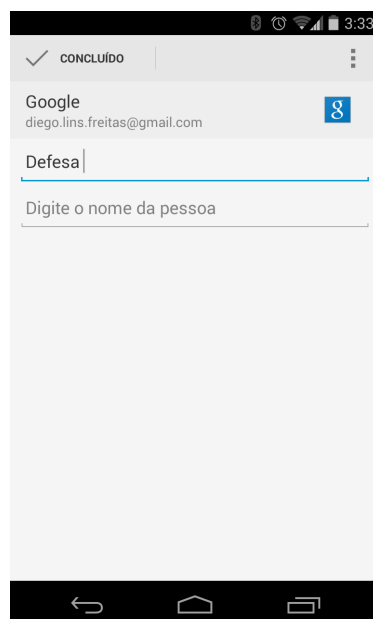
Fonte: Próprio Autor

Figura 17 – Tela de visualização de um grupo



Fonte: Próprio Autor

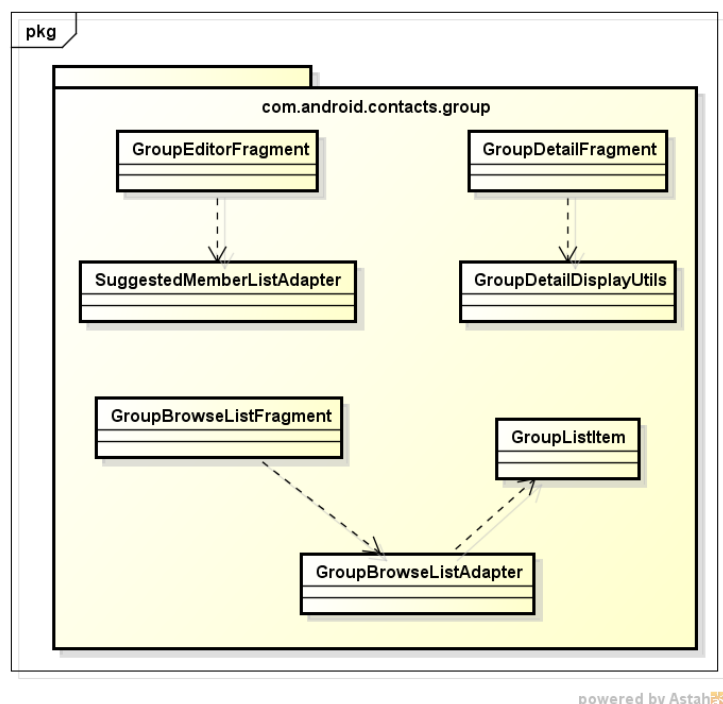
Figura 18 – Tela de edição e criação de grupos



Fonte: Próprio Autor

elencados contêm código não somente relacionado com a lógica de apresentação como também interagem diretamente com classes destinadas ao acesso de dados e serviços existentes nas dependências do projeto, por exemplo, gerenciamento de contas do usuário. Cada iteração consistirá na refatoração de cada um dos componentes

Figura 19 – Diagrama de pacotes de grupos



Fonte: Próprio Autor

descritos. O marco de referência de dados das métricas presentes na tabela 1 será feita a partir da versão 4.4.2_r1 do aplicativo.

Tabela 1 – Métricas CK da versão de referência

Métrica	Média
WMC	8.5161290323
DIT	0.7741935484
NOC	0
CBO	10.1612903226
RFC	23.7419354839
LCOM	57.4838709677

Fonte: Próprio Autor

Os dados de referência apresentados na tabela 1 foram coletados usando a ferramenta CKJM. Os dados são referentes ao pacote de grupos. Foram coletados os valores de cada métricas para cada uma das classes presentes no pacote de grupos e para cada métrica foi calculado a média. A cada refatoração será executado o mesmo procedimento de coleta descrito para fazer uma análise na variação das médias de

cada métrica.

3.3 Arquitetura Proposta

Será aplicado nos experimentos a variação do padrão MVP chamada *Passive View*, pois dessa forma, o Model não precisa publicar alterações de seu estado para a view, Logo, evita-se alterações no código referente às classes que fazem parte da camada de Model do aplicativo de contatos. A organização do código fonte no repositório dificulta a implementação, isto porque esses componentes estão localizados fora do projeto afetado e são compartilhados.

As classes que estendem *Fragment* terão a responsabilidade da View, pois é neste componente que a interface com o usuário é construída. A classe *Activity* fornece vários métodos para recuperação de recursos de imagens, textos, inicialização de serviços, entre outros. Isso ocorre porque a classe *Activity* é uma subclasse de *Context*, herdando diversos métodos não relacionados ao gerenciamento da interface.

Segundo [Reenskaug \(1979\)](#) "... Os papéis da View e do Controller podem ser exercidos pelo mesmo objeto quando eles estão muito acoplados. Exemplo: Um Menu."(tradução livre). Porém, isso requer uma boa análise do problema em questão para decidir o nível de granularidade que esses componentes podem ter. Portanto, é recomendável manter sempre essa separação para manter uma boa coesão nas classes. O *Presenter* será uma classe auxiliar à View e pode ser implementada como uma classe java simples.

3.4 Resultados dos Experimentos

Esta seção tem como objetivo mostrar os resultados obtidos com o processo de refatoração aplicando o padrão MVP. Cada métrica é apresentada com seus dados para cada iteração mostrando os efeitos desses valores na qualidade.

3.4.1 WMC

A métrica WMC é usada para medir o tempo e esforço necessário para desenvolver e manter uma classe. Levando em consideração o método como uma unidade de complexidade, quanto menos métodos um classe tiver menos complexa ela será, portanto, é recomendado que esta métrica tenha valores baixos. Entretanto, uma classe terá a quantidade de métodos necessária para exercer seu papel no sistema. É inviável desenvolver um sistema cujas todas as classes tenham um único método com a implementação de todas as funções de uma classe. Portanto, não existe um valor ideal para a métrica WMC. Essa métrica deve ser analisada levando em consideração o contexto da classe de interesse além da complexidade expressa pela quantidade de métodos.

A possibilidade de reuso de uma classe reduz, pois a grande quantidade de métodos indica que ela tem funções muito específicas(CHIDAMBER; KEMERER, 1994). Neste caso o WMC é um indicador de que é necessário fazer um refatoração para extrair funções comuns a outras partes do aplicativo, além do pacote de Groups, como por exemplo funções para exibição de dados em uma lista, validação de dados em componentes de texto, entre outros. O escopo de atuação do padrão MVP é mais amplo, abrangendo o caso de uso realizado pela interface refatorada. Logo, o MVP não tem impacto na reusabilidade.

Os efeitos colaterais em uma classe filha é maior quando é feito alguma alteração na classe pai que tenha o número de métodos muito alto(CHIDAMBER; KEMERER, 1994). Isso dificulta a manutenabilidade e o esforço de testes. Porém, As classes refatoradas tem uma função muito específica dentro da aplicação e não são extendidas por outras classes. O MVP está relacionado com a separação de responsabilidades e sua aplicação não interferiu na hierarquia de classes que foram refatoradas. A tabela 2 e a figura 20 mostram os valores dessa métrica no projeto.

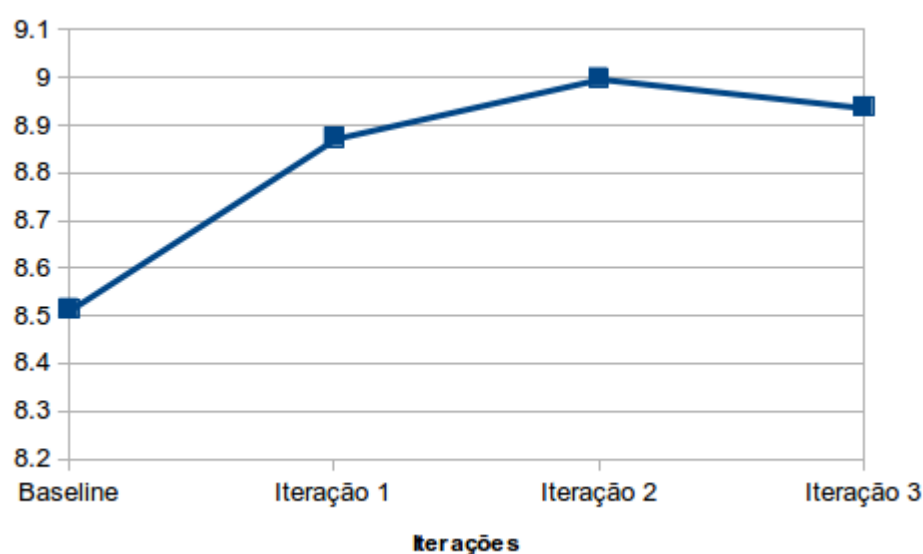
De acordo com o que foi exposto, o MVP não deveria afetar a métrica WMC. Porém, houve um aumento nos valores dessa métrica . Isso é consequência da divisão

Tabela 2 – Dados métrica WMC

Iteração	Média
Baseline	8.5161290323
Iteração 1	8.875
Iteração 2	9
Iteração 3	8.9393939394

Fonte: Próprio Autor

Figura 20 – Gráfico da métrica WMC



Fonte: Próprio Autor

de responsabilidades entre a View e o Presenter. Antes da refatoração, os métodos das classes de View implementavam o acesso a dados, controle dos componentes visuais e regras específicas das operações executadas na tela, como por exemplo, a atualização de um componente visual quando nenhum dado está disponível para exibição. Os métodos da classe de View, afetados pela refatoração, foram divididos em no mínimo dois métodos. Sendo que, o método que permanece na View manipula os componentes visuais, enquanto o novo método criado no Presenter faz o acesso à camada de Model e implementa as regras de negócio da operação em resposta à interação do usuário na View. Como consequência a quantidade de métodos aumenta, porém, os métodos tornam-se mais simples pois implementam funções específicas.

Levando isso em consideração, apesar dos valores de WMC terem aumentado, a complexidade diminuiu, pois os métodos estão mais concisos. No contexto dessa refatoração, a métrica WMC não pode ser considerada um indicador determinante para a avaliação da qualidade do objeto de estudo.

3.4.2 NOC

Altos valores para a métrica NOC são indicativos de que existe maior reuso de código. Assim como DIT, a métrica NOC é um indicador da influência da classe no comportamento das classes filhas o que aumenta o esforço de testes. Quando os valores desta métrica estão aberrantes em relação às outras classes, há grande chance de que a abstração está sendo usada de forma incorreta. Dado esses cenários, está é um métrica cujos seus valores devem ser analisados caso a caso.

Durante o processo de refatoração não foi aplicada herança em nenhuma das classes afetadas, portanto, essa métrica permaneceu intacta durante as iterações como pode ser observado na tabela 3.

Tabela 3 – Dados métrica NOC

Iteração	Média
Baseline	0
Iteração 1	0
Iteração 2	0
Iteração 3	0

Fonte: Próprio Autor

3.4.3 DIT

A métrica DIT está relacionada complexidade e reusabilidade. Quanto mais abaixo na hierarquia de herança a classe estiver, menos previsível será seu comportamento devido a quantidade de métodos que serão herdados das classes acima nessa hierarquia. Entretanto, a métrica também indica um potencial reuso de código por meio da herança de métodos, isso torna os parâmetros de avaliação da métrica dependentes do contexto a ser analisado. O aumento demonstrado na DIT se deve ao

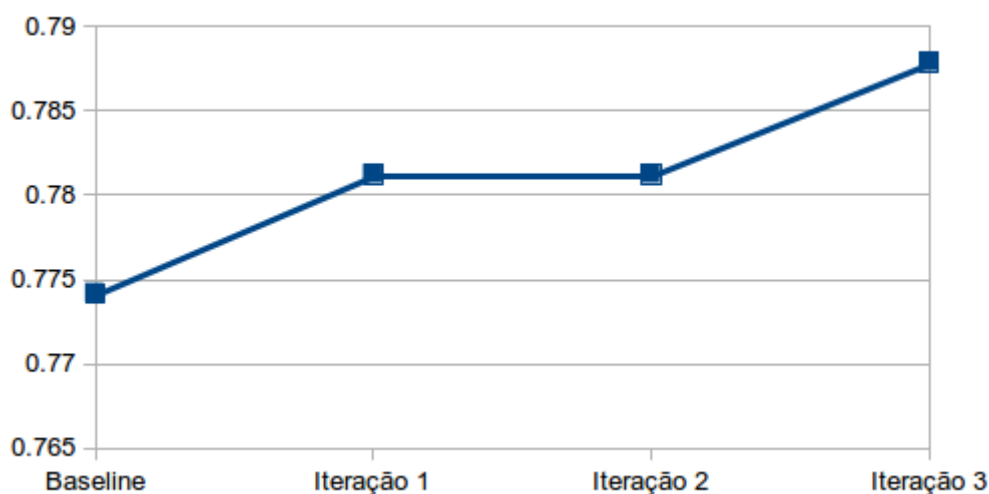
fato que o CKJM considera a implementação de interface como herança. A interface define somente o contrato que a classe deve implementar. Sendo uma interface, não há nenhuma implementação que possa interferir no comportamento da classe, portanto essa alteração na métrica DIT não deve ser considerada. O aumento no valor da métrica é mostrado na tabela 4 e figura 21

Tabela 4 – Dados métrica DIT

Iteração	Média
Baseline	0.7741935484
Iteração 1	0.78125
Iteração 2	0.78125
Iteração 3	0.7878787879

Fonte: Próprio Autor

Figura 21 – Gráfico da métrica DIT



Fonte: Próprio Autor

3.4.4 CBO

Valores baixos de CBO são indicativos de boa modularidade e encapsulamento que se reflete na independência da classe, o que a torna mais fácil de reutilizar, manter e testar. Na primeira iteração foi aplicado o padrão em um componente mais simples e foi possível remover qualquer dependência que não fosse relacionada a interface,

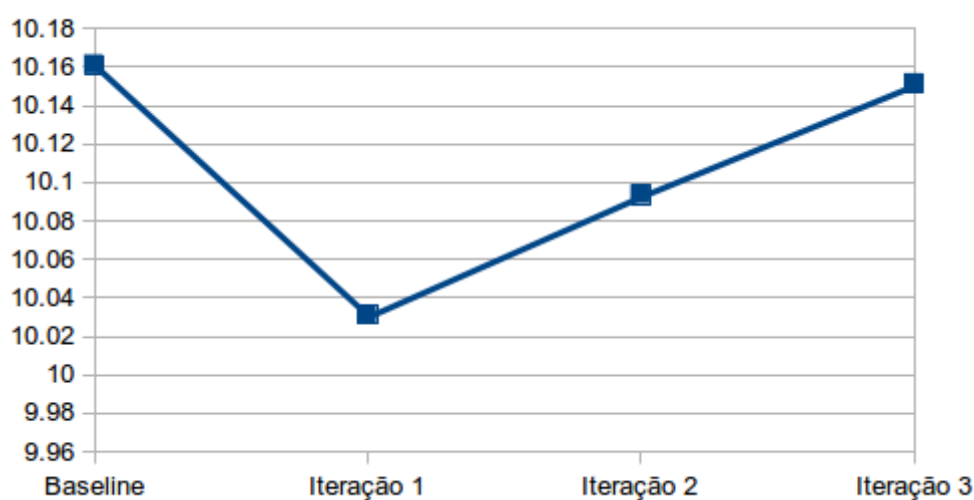
diminuído significativamente o valor da métrica. Nas iterações seguintes foram re-fatoradas interfaces mais complexas onde é mais difícil desacoplar as dependências. Algumas dependências não relacionadas a camada de apresentação permaneceram na View, dessa forma, tanto a View como o Presenter tem referências para essas dependências aumentando o valor da métrica. Além disso, existe o acréscimo de duas dependências entre a View e o Presenter. A variação da métrica é exposta na tabela 5 e na figura 22

Tabela 5 – Dados métrica CBO

Iteração	Média
Baseline	10.1612903226
Iteração 1	10.03125
Iteração 2	10.09375
Iteração 3	10.1515151515

Fonte: Próprio Autor

Figura 22 – Gráfico da métrica CBO



Fonte: Próprio Autor

3.4.5 RFC

Altos valores para a métrica RFC indica que uma quantidade grande de métodos são chamados a partir de uma classe tornando-a mais complexa de testar e fazer

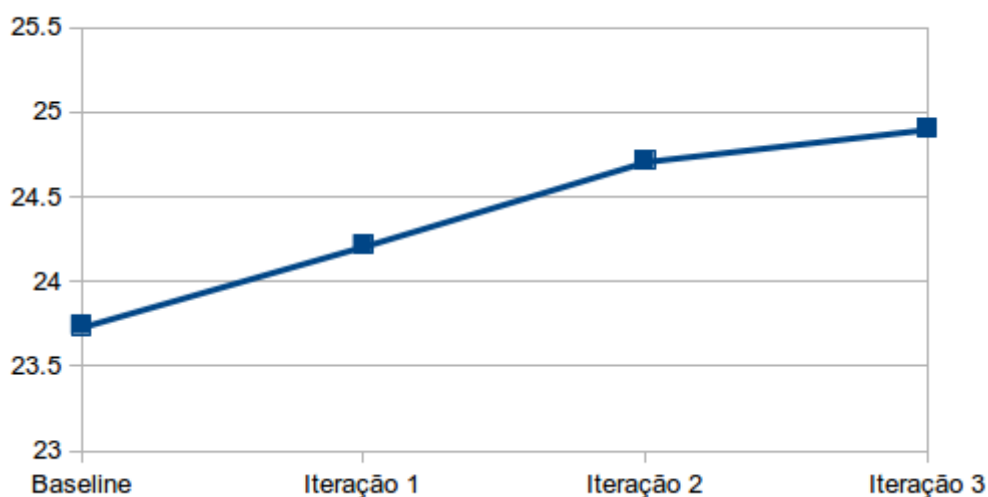
manutenção. Logo, esta métrica deve diminuir para expressar maior qualidade no código. A métrica RFC tende a aumentar a cada iteração, o que sugere um aumento da complexidade do código conforme é apresentado na tabela 6 e Figura 23.

Tabela 6 – Dados métrica RFC

Iteração	Média
Baseline	23.7419354839
Iteração 1	24.21875
Iteração 2	24.71875
Iteração 3	24.9090909091

Fonte: Próprio Autor

Figura 23 – Gráfico da métrica RFC



Fonte: Próprio Autor

A justificativa para o aumento da métrica WMC também se aplica neste caso. Após cada iteração de refatoração, as Views passaram a delegar responsabilidades para o Presenter por meio de chamada de métodos, além disso, o Presenter interage com a View da mesma forma para atualizá-la. Portanto, a quantidade de chamada de métodos aumentaram e isso se refletiu na métrica RFC.

3.4.6 LCOM

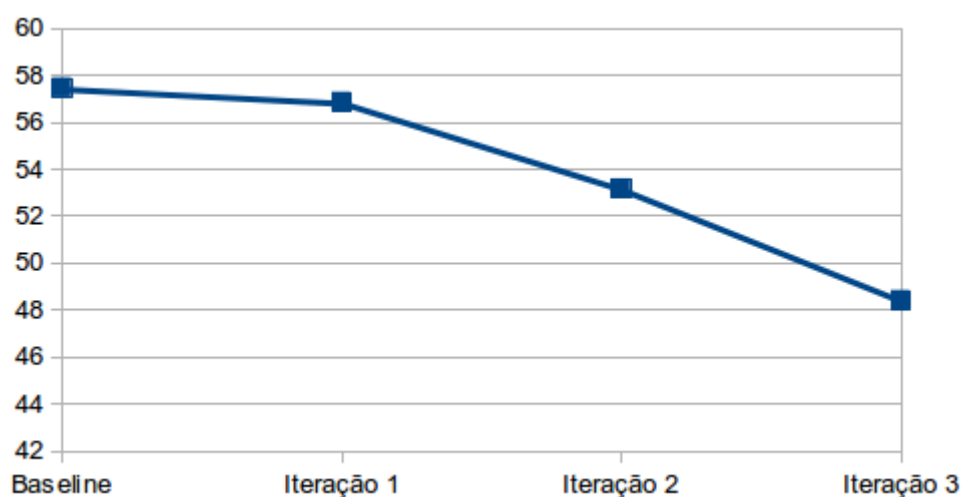
Segundo [Chidamber e Kemerer \(1994\)](#) “Um valor alto de LCOM indica uma disparidade na funcionalidade provida pela classe.”. Analisando a relação entre os métodos da classe e seus atributos é possível dizer se a classe tem muitas responsabilidades e é necessário dividi-la em duas ou mais classes. Esta métrica ajuda a identificar má qualidade na estrutura do código quando os valores são altos, apontando aumento da complexidade e pouco encapsulamento. A [tabela 7](#) e a [figura 24](#) mostram uma queda significativa na métrica LCOM. Isto indica que a coesão do código melhorou após cada iteração.

Tabela 7 – Dados métrica LCOM

Iteração	Média
Baseline	57.4838709677
Iteração 1	56.875
Iteração 2	53.1875
Iteração 3	48.4242424242

Fonte: Próprio Autor

Figura 24 – Gráfico da métrica LCOM

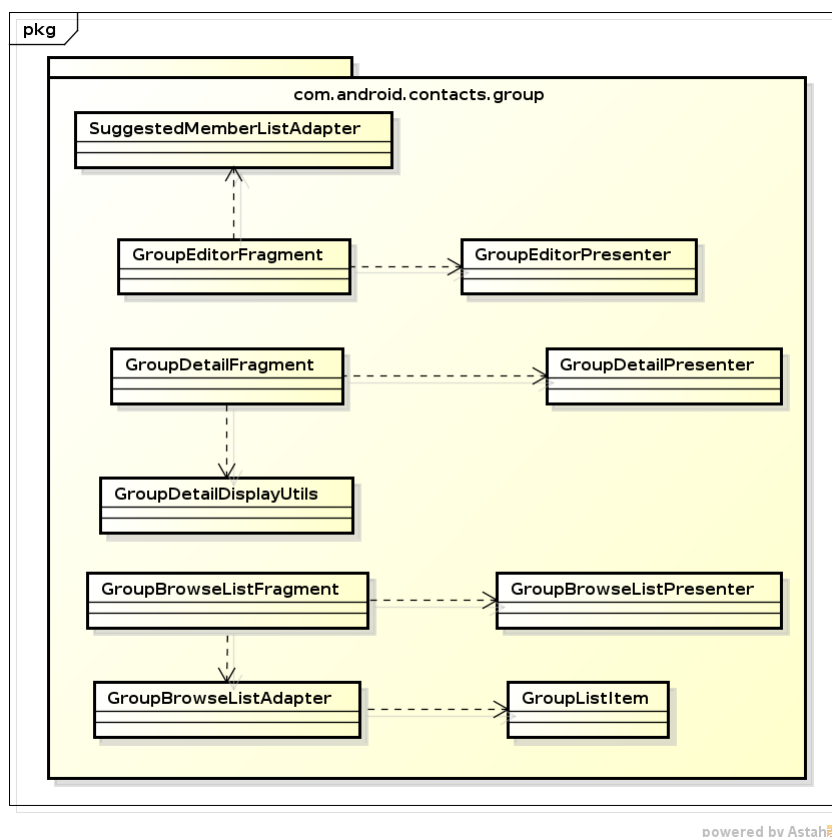


Fonte: Próprio Autor

3.5 Discussão dos Resultados

A figura 25 mostra a disposição das classes após a refatoração.

Figura 25 – Pacote após refatoração



Fonte: Próprio Autor

As classes destinadas à implementação da interface no framework android fornecem acesso a recursos que servem para implementação de responsabilidades não relacionadas com a interface. Essa característica do framework android leva à implementação da camada de View com diversas responsabilidades que não são inerentes à interação com o usuário. Isso dificultou a refatoração, pois as classes que exercem o papel de Presenter necessitam interagir com as classes de View para acessar esses recursos, além de atualizar o estado da View. O uso do padrão de injeção de dependência³ pode ser aplicado para acessar esses recursos e serviços sem a

³ Padrão de projeto onde um objeto recebe as referências para as suas dependências sem conhecer os processos de construção das mesmas. http://en.wikipedia.org/wiki/Dependency_injection

necessidade de interação com a classe de View.

Houve diminuição na métrica CBO nas classes alteradas pois diversas responsabilidades que utilizam essas dependências foram movidas para a classe de Presenter. Analisando de forma geral, essas dependências permanecem no pacote além de ser criado mais um acoplamento entre a View e a nova classe Presenter.

Ao usar o padrão MVP, a quantidade de linhas de métodos diminui pois cada um dos componentes implementaram uma parte do caso de uso, aumentando o número de métodos que se reflete na métrica WMC. Na implementação original onde um método que era implementado na classe View, tinha a responsabilidade de tratar os eventos do usuário, acessar os componentes visuais e recuperar os dados, validar esses dados, acessar o Model para persistir o dados e atualizar a tela como o novo estado. Tudo isso resulta em métodos com muitas linhas de código com várias estruturas de controle e iteração, isso aumenta a complexidade da classe.

Ao dividir as responsabilidades entre os componentes definidos no padrão, um método complexo que era implementado na View quebrado em pelo menos três métodos menores, para que a View possa receber as interações com o usuário e dados de entrada para então delegar o processamento ao Presenter que vai interagir com as classes que exercem o papel de Model, e para finalizar, o Presenter chama algum método da View que irá atualizar os componentes visuais com o novo estado dos dados.

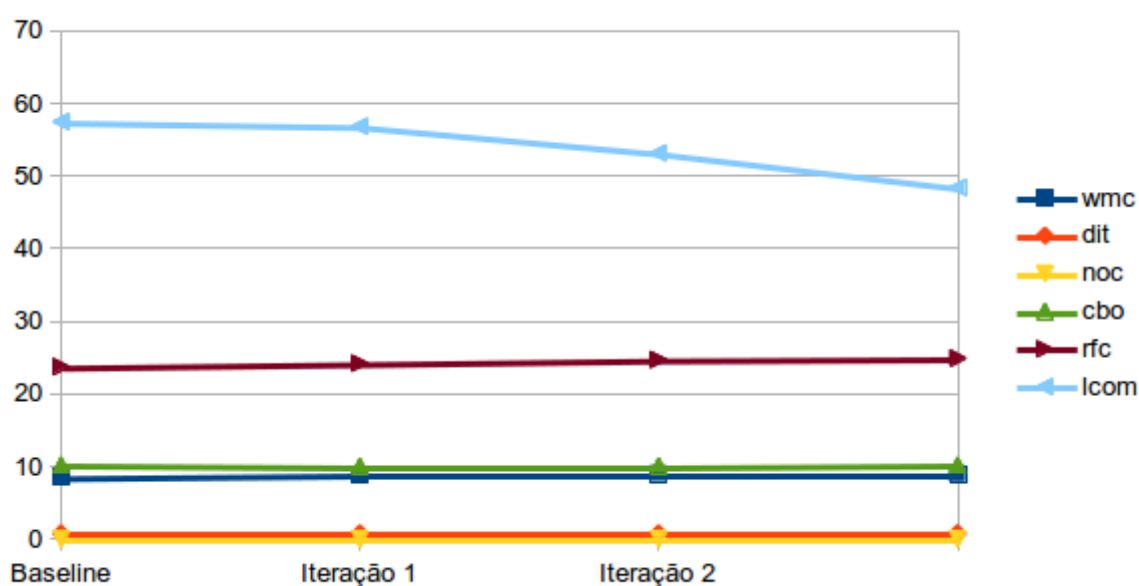
Tendo em vista que o método foi usado como unidade para calcular o WMC, essa métrica aumentou ao aplicar o padrão MVP, pois mais métodos foram criados. A métrica WMC mostra a complexidade de uma classe, mas apesar do aumento nos valores, a complexidade diminuiu, ao ser usado a implementação com métodos mais simples. A divisão de responsabilidades também afetou negativamente a métrica RFC, pois com o aumento da quantidade de métodos, a troca de mensagens entre componentes e com a própria classe aumentaram.

A métrica WMC está relacionada com as métricas DIT e NOC. Como houve

pouca variação no DIT e nenhuma variação no NOC, o aumento da WMC não tem impacto relevante.

Os experimentos demonstraram que a aplicação do padrão MVP promoveu de forma significativa maior coesão no aplicativo. Isso é demonstrado nos resultados da métrica LCOM que foi a mais afetada pelo uso do padrão MVP no projeto como pode ser observado na figura 26.

Figura 26 – Variação das métricas ao longo das iterações



Fonte: Próprio Autor

É possível observar nos resultados a relação entre LCOM e as métricas WMC, RFC que aumentaram conforme o LCOM diminuía. O aumento da complexidade indicado pelas métricas WMC e RFC é pequeno em comparação ao aumento de coesão indicada pela métrica LCOM. Pode-se chegar a essa conclusão, não somente analisando os resultados, como também ao analisar o código em que as classes estão menores, mais coesas, com responsabilidades bem definidas. Portanto, a arquitetura proposta melhorou a qualidade do objeto de estudo porque promoveu maior coesão e diminuiu a complexidade.

4 Conclusão

O presente trabalho avaliou os padrões de projetos Model View Controller e o Model View Presenter mostrando em que contextos cada um se aplica para o desenvolvimento da camada de apresentação de um sistema. O padrão MVP define melhor as responsabilidades dos componentes para implementação de interfaces levando em consideração o modelo de programação usado no framework android onde as responsabilidades do Controller(definidos no padrão MVC) estão implementadas nos próprios componentes visuais.

O padrão MVP foi implementado usando a variação chamada Passive View onde o Model fica totalmente isolado da View. Esta variante foi usada pois evitou-se fazer alterações em classes correspondentes a camada de Model, devido ao alto acoplamento e grande complexidade do aplicativo, dessa forma foi possível evitar que a refatoração gerasse algum erro no aplicativo, mantendo-o funcional. A maioria das alterações estão presentes nas classes criadas para exercerem o papel de Presenter.

As classes de View selecionadas para a refatoração estendem a classe Fragment. A classe Fragment é usada para criar agrupamentos de componentes visuais que possam ser reutilizados em outras partes do aplicativo. O uso da classe Fragment foi mantido e parte das implementações presentes nessas classes foram delegadas para as classes de Presenter correspondentes.

As métricas descritas por [Chidamber e Kemerer \(1994\)](#) foram usadas para avaliar os impactos na qualidade do aplicativo após a refatoração usando o padrão MVP. Foi usado o projeto CKJM para coletar os dados para fazer a análise.

Conforme um software é desenvolvido, novas classes e troca de mensagens são implementadas afetando negativamente as métricas WMC, RFC e CBO. Não é possível relacionar diretamente a métrica CBO com base na variação dos resultados para esta métrica, apesar de que, os valores se mantiveram abaixo dos encontrados da

versão de referência, isso requer mais estudos abordando outros padrões de projeto.

A maioria das métricas tendem a aumentar durante o processo de refatoração. É válido resaltar que as três classes refatoradas são as mais complexas do pacote de grupos com valores anômalos para as métricas. Portanto, somente após uma refatoração em uma amostragem maior de classes que se é possível determinar um valor de referência para as métricas, inclusive para um projeto construído do zero. Com os dados coletados durante o desenvolvimento é possível definir esses valores limites para identificar anomalias nas classes e tratar cada caso isolado.

4.1 Trabalhos Futuros

Existem outros padrões de projetos para o desenvolvimento da camada de apresentação de um software que não foram analisados nesse trabalho, a saber: MVVM, MVP-VM, MVPC. Essas variações no padrão MVC surgiram em contextos diversificados e podem agregar algum benefício à qualidade do aplicativo. Este trabalho não aborda o impacto do padrão MVP em outras métricas de qualidade de código derivadas do CK. Não foi feita uma avaliação dos impactos na performance do aplicativo devido ao uso do padrão MVP. A inclusão de mais objetos interagindo trocando mensagens pode depreciar a performance, levando-se em consideração sua execução em ambientes mais restritos, como no caso de um aparelho móvel.

Referências

- ALLES, M. et al. Presenter first: organizing complex gui applications for test-driven development. In: *Agile Conference, 2006*. [S.l.: s.n.], 2006. p. 10 pp.–288. Citado na página 12.
- BOWER, A.; MCGLASHAN, B. Twisting the triad: The evolution of the dolphin smalltalk mvp application framework. In: *ESUG2000 Conference*. Southampton, UK: [s.n.], 2000. Disponível em: <<http://www.object-arts.com/downloads/papers-/TwistingTheTriad.PDF>>. Citado na página 29.
- BRIAND, L. C. et al. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, Elsevier Science Inc., New York, NY, USA, v. 51, n. 3, p. 245–273, maio 2000. ISSN 0164-1212. Disponível em: <[http://dx.doi.org/10.1016/S0164-1212\(99\)00102-8](http://dx.doi.org/10.1016/S0164-1212(99)00102-8)>. Citado na página 12.
- CHIDAMBER, S.; DARCY, D.; KEMERER, C. Managerial use of metrics for object-oriented software: an exploratory analysis. *Software Engineering, IEEE Transactions on*, v. 24, n. 8, p. 629–639, Aug 1998. ISSN 0098-5589. Citado na página 12.
- CHIDAMBER, S.; KEMERER, C. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476 – 493, 6 1994. Citado 8 vezes nas páginas 12, 13, 18, 23, 24, 39, 45 e 49.
- DUBEY, S. K.; RANA, A. Assessment of maintainability metrics for object-oriented software system. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 36, n. 5, p. 1–7, set. 2011. ISSN 0163-5948. Disponível em: <<http://doi.acm.org/10.1145/2020976.2020983>>. Citado na página 12.
- EVANS, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. [S.l.]: Addison-Wesley, 2004. Citado na página 17.
- FIAWOO, S.; SOWAH, R. Design and development of an android application to process and display summarised corporate data. In: *Adaptive Science Technology (ICAST), 2012 IEEE 4th International Conference on*. [S.l.: s.n.], 2012. p. 86–91. Citado na página 12.
- FOWLER, M. *Patterns of Enterprise Application Architecture*. Boston, MA, USA: Addison-Wesley, 2002. ISBN 0321127420. Citado na página 26.
- FOWLER, M. *GUI Architectures*. 2006. GUI Architectures. Disponível em: <<http://www.martinfowler.com/eaDev/uiArchs.html>>. Acesso em: 20.8.2013. Citado na página 29.
- FOWLER, M. *Supervising Controller*. 2006. Supervising Controller. Disponível em: <<http://martinfowler.com/eaDev/SupervisingPresenter.html>>. Acesso em: 20.8.2013. Citado na página 29.

- GAMMA, E. et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison Wesley, Boston, 1995. (Professional Computing Series). Citado 6 vezes nas páginas 9, 17, 18, 26, 27 e 28.
- GARGENTA, M. *Learning Android*. 1. ed. Upper Saddle River, NJ: O'Reilly Media, Inc., 2011. ISBN 978-1-449-39050-1. Citado na página 32.
- GOMAA, H. *Software modeling and design : UML, use cases, patterns, and software architectures*. [S.l.]: Addison-Wesley, 2011. Citado na página 17.
- IDC. *Android and iOS Continue to Dominate the Worldwide Smartphone Market with Android Shipments Just Shy of 800 Million in 2013, According to IDC*. 2014. IDC - Worldwide Quarterly Mobile Phone Forecast. Disponível em: <<http://www.idc.com/getdoc.jsp?containerId=prUS24676414>>. Acesso em: 3.5.2014. Citado na página 9.
- KHALID, S.; ZEHRA, S.; ARIF, F. Analysis of object oriented complexity and testability using object oriented design metrics. In: *Proceedings of the 2010 National Software Engineering Conference*. New York, NY, USA: ACM, 2010. (NSEC '10), p. 4:1–4:8. ISBN 978-1-4503-0026-1. Disponível em: <<http://doi.acm.org/10.1145/1890810-1890814>>. Citado na página 12.
- KRASNER, G.; POPE, S. A description of the Model-View-Controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, n. 3, p. 26–49, 1988. Citado 4 vezes nas páginas 24, 26, 27 e 28.
- LA, H. J.; KIM, S. D. Balanced mvc architecture for developing service-based mobile applications. In: *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*. [S.l.: s.n.], 2010. p. 292–299. Citado na página 11.
- METSKER, S. J.; WAKE, W. C. *Design Patterns in Java*. 2. ed. Upper Saddle River, NJ: Addison-Wesley, 2006. ISBN 978-0-321-33302-5. Citado na página 17.
- POTEL, M. *MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java*, Taligent Inc. [S.l.], 1996. Citado na página 29.
- PRESSMAN, R. S. *Engenharia de Software*. 6. ed. [S.l.]: McGraw-hill Interamericana, 2006. Citado na página 9.
- REENSKAUG, T. M. H. *Thing-Model-View-Editor – an Example from a planning system*. [S.l.], 1979. Acesso em: 1.8.2013. Citado 2 vezes nas páginas 24 e 38.
- TÜRK, T. *THE EFFECT OF SOFTWARE DESIGN PATTERNS ON OBJECT-ORIENTED SOFTWARE QUALITY AND MAINTAINABILITY*. Dissertação (Mestrado) — Middle East Technical University: METU, Çankaya Ankara/ Turquia, Setembro 2009. Citado na página 13.
- WAND, Y.; WEBER, R. An ontological model of an information system. *IEEE Trans. Softw. Eng.*, IEEE Press, Piscataway, NJ, USA, v. 16, n. 11, p. 1282–1292, nov. 1990. ISSN 0098-5589. Disponível em: <<http://dx.doi.org/10.1109/32.60316>>. Citado na página 18.

YANG, H. Y.; TEMPERO, E.; MELTON, H. An empirical study into use of dependency injection in java. In: *Software Engineering, 2008. ASWEC 2008. 19th Australian Conference on*. [S.l.: s.n.], 2008. p. 239–247. ISSN 1530-0803. Citado na página [9](#).

ZHANG, Y.; LUO, Y. An architecture and implement model for model-view-presenter pattern. In: *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*. [S.l.: s.n.], 2010. v. 8, p. 532–536. Citado na página [12](#).