Análise Técnica de Arquiteturas de Software no desenvolvimento de aplicativos para Smartphones

João Pedro Pereira Lemos[1], **Igor Rafael Silva Valente**[1]

[1]Instituto Federal de Educação, Ciência e Tecnologia do Estado do Ceará (IFCE)

joao.pedro.pereira62@aluno.ifce.edu.br, igor@ifce.edu.br

Abstract. *When creating applications for smartphones, developers make important decisions that directly impact the long-term growth of a project. The software architecture is a determining factor for the success of an application. It is of paramount importance that this is well defined and has characteristics that allow its evolution and maintenance in a sustainable way. This work compares two widely used development approaches, analyzing their characteristics. The MVC pattern is compared with the Clean Architecture, using the scenario-based analysis of software architecture. The main contribution of this paper is to provide an analysis of the chosen architectures in an application for smartphones.*

Resumo. *Ao criar aplicações para smartphones, os desenvolvedores tomam decisões importantes que impactam diretamente no crescimento de um projeto no longo prazo. A arquitetura de software é um fator determinante para o êxito de um aplicativo. É de suma importância que esta seja bem definida e possua características que permitam sua evolução e manutenção de maneira sustentável. Este trabalho compara duas abordagens de desenvolvimento amplamente empregadas, analisando suas características. O padrão MVC é comparado com a Arquitetura Limpa, utilizando o método de análise de arquiteturas de software baseado em cenários. A principal contribuição deste artigo é fornecer uma análise das arquiteturas escolhidas em uma aplicação para smartphones.*

1. Introdução

Os smartphones fazem parte do cotidiano da sociedade, de modo que muitas tarefas podem ser realizadas por intermédio deles. Comunicação, entretenimento, notícias, educação, saúde, serviços, entre outros. As operações por meio de smartphones já representam mais de 51% das transações bancárias no Brasil. Empresas do setor financeiro estão investindo muito neste segmento. Estima-se que só em 2020 foram aplicados mais de R$ 25,7 bilhões [FEBRABAN and Deloitte 2021]. Segundo uma pesquisa realizada em 2022, existem 447 milhões de dispositivos digitais em uso no Brasil, sendo 242 milhões destes, smartphones. Há mais de 1 smartphone por habitante em território nacional [de Souza Meirelles 2022]. O Banco Central do Brasil estabeleceu em 2020 um novo meio de pagamento — o Pix. Esse meio de pagamento nasceu com a ideia de utilizar o smartphone para enviar e receber dinheiro, dada a sua massiva utilização. Apenas no mês março de 2022, mais de 784 milhões de transações foram realizadas por meio do Pix [BANCO CENTRAL DO BRASIL 2022].

Dada a importância que os smartphones têm na sociedade, é importante garantir que as aplicações desenvolvidas tenham um padrão de qualidade. Empresas de tecnologia estão preocupadas com o *time to market*, termo que designa o tempo decorrido desde

a concepção da ideia até a publicação de um produto. Assim, diversas abordagens são utilizadas para entregar o produto no menor tempo possível, em muitos casos sacrificando a qualidade, mas que geram débitos técnicos dentro do projeto. Em geral, nota-se que mudanças na regra de negócio e eventuais adições de funcionalidades podem ter uma complexidade maior quando a arquitetura do projeto é muito permissiva, ou não é bem definida. A garantia de qualidade muitas vezes está concentrada apenas no processo de QA, e o código-fonte produzido não é levado em consideração. Nem tudo pode ser prevenido, e *bugs* geralmente são um reflexo disso [Grønli and Ghinea 2016]. O presente artigo compara duas abordagens diferentes para se desenvolver um aplicativo para smartphones, analisando como se comportam em face à evolução do software. Inicialmente, são apresentados trabalhos de outros autores que se relacionam com o tema escolhido. Depois, os conceitos de Engenharia de Software que permearão este trabalho são discorridos brevemente. Na sequência a metodologia adotada é apresentada. Em seguida, o padrão MVC é comparado à Arquitetura Limpa, utilizando método de análise de arquiteturas de software baseado em cenários. Ao final, são apresentadas as conclusões da análise realizada.

2. Trabalhos relacionados

Em [Humeniuk 2019] a arquitetura VIPER é comparada com o MVP na plataforma Android, com o objetivo de verificar a viabilidade de cada abordagem. Uma aplicação de notas é desenvolvida em ambas as arquiteturas. Modificabilidade, testabilidade, manutenibilidade e performance são usadas como métricas para a análise. O trabalho conclui que ambas as abordagens são viáveis, e que o MVP é mais adequado para projetos pequenos, ao passo que o VIPER é adequado para projetos maiores. A comparação apresenta métricas específicas para a plataforma Android, que podem não se aplicar em outras plataformas, como o iOS e frameworks híbridos. Em [Ghafoor 2021], é apresentada uma análise de diversos padrões arquiteturais para o desenvolvimento na plataforma Android. Abordagens como o MVC, MVP, MVVM, VIPER e a Arquitetura Limpa são comparadas subjetivamente em relação ao acoplamento, coesão, testabilidade e manutenibilidade, a fim de investigar qual padrão arquitetural é o mais adequado para a plataforma escolhida. De acordo com o estudo, uma combinação de MVVM com Arquitetura Limpa atingem os atributos desejados. A comparação não conta com um exemplo real de código para cada abordagem. O presente artigo justifica-se por trazer uma análise baseada em cenários, a partir de um projeto de exemplo, em cada uma das abordagens selecionadas. São considerados aspectos importantes, como a facilidade de implementar novos recursos, o impacto de modificações futuras, possibilidade de modularizar a aplicação, curva de aprendizado de cada abordagem e a facilidade de escrever testes unitários. Também, a análise realizada é válida tanto no desenvolvimento em plataformas específicas, como o Android e iOS, quanto em frameworks híbridos, como Flutter ou React Native.

3. Fundamentação teórica

Padrões arquiteturais são descrições de alto nível dos componentes de um software e seus relacionamentos. Em [Reenskaug 1979], é apresentado o padrão MVC (*Model-View-Controller*), que visava padronizar a forma que os sistemas com interface gráficas eram construídos, dividindo as responsabilidades da aplicação em alguns componentes.

Em [Martin 2017] apresentado a *Clean Architecture* (Arquitetura Limpa, em português), uma abordagem que unifica as ideias presentes no *Domain-Driven Design*

[Evans 2003] e na *Hexagonal Architecture* [Cockburn 2005] com os princípios SOLID. A arquitetura visa permitir que o software desenvolvido seja plenamente testável, por desacoplar dependências externas. Além disso, a arquitetura almeja possibilitar que o cliente da aplicação seja independente da lógica de negócio.

SOLID é um acrônimo em inglês para 5 princípios de design de software, sendo estes o Princípio da Responsabilidade Única, Princípio do Aberto-Fechado, Princípio da Substituição de Liskov, Princípio da Segregação de Interfaces e Princípio de Inversão de Dependências. Seu objetivo é guiar a criação de componentes que tolerem mudanças, sejam de fácil entendimento e sirvam de base para a criação outros projetos de software [Martin 2017].

Coesão diz respeito ao relacionamento que os membros de um determinado componente possuem entre si. Componentes coesos possuem uma relação forte, onde os membros estão intimamente ligados em prol de um objetivo comum. Acoplamento se refere a quanto um componente é dependente de outro para funcionar corretamente. Componentes desacoplados tornam a aplicação mais flexível a mudanças, além de torná-los reutilizáveis. Em geral, componentes com um baixo acoplamento possuem alta coesão, e vice-versa [Stevens et al. 1974].

4. Metodologia

Para atingir o objetivo deste artigo, foram desenvolvidas duas versões de um mesmo aplicativo de notas, uma utilizando o padrão MVC, e outra utilizando a Arquitetura Limpa. A aplicação permite o cadastro e autenticação por meio de credenciais ou de uma conta Google. Após autenticar-se, o usuário é capaz de criar, visualizar, editar e apagar notas que existam em sua conta. Os dados são armazenados no serviço remoto Firebase [Google 2011], além de serem persistidos localmente no dispositivo. O aplicativo foi construído utilizando a linguagem Dart, fazendo uso do framework Flutter [Google 2017]. Na Figura 1 é possível visualizar cinco capturas de tela do aplicativo.

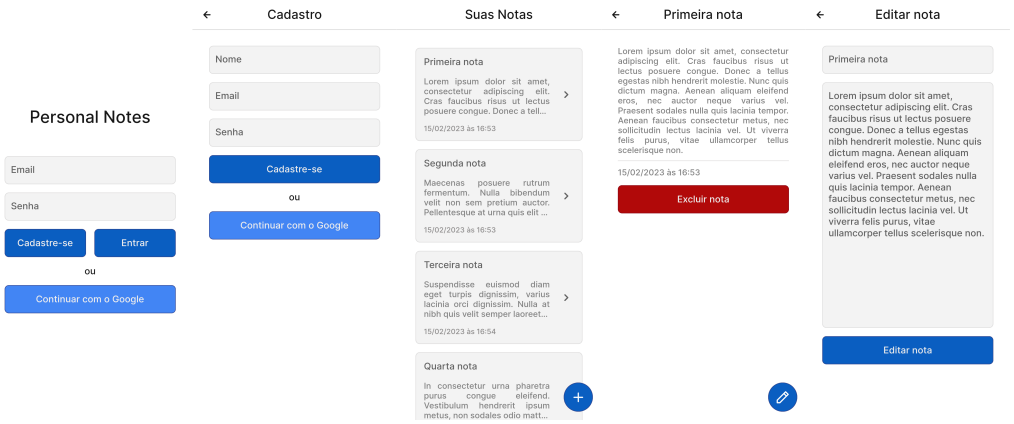

Figura 1. Capturas de tela do aplicativo

4.1. Método de análise de Arquiteturas de Software baseado em cenários

A análise se dará por meio do método baseado em cenários, proposto em [Kazman et al. 1996]. Um cenário é uma breve narrativa de usos esperados ou antecipa-

dos do sistema, tanto do ponto de vista do usuário quanto do desenvolvedor. Ele fornece uma visão de como o sistema satisfaz atributos de qualidade em vários contextos de uso.

Inicialmente, é realizada uma descrição da arquitetura candidata. Após esta etapa, os cenários são desenvolvidos, de modo a ilustrar os tipos de atividades que o sistema deve suportar e as mudanças que são esperadas ao longo da vida de um software. Então, para cada cenário, é avaliado se a arquitetura necessita de alterações para executar o cenário. Em caso afirmativo, o cenário é classificado como direto. Caso contrário, é classificado como indireto.

Quando dois ou mais cenários indiretos necessitam de mudanças em algum componente, diz-se que eles interagem. As interações mostram quais módulos estão envolvidos em quais tarefas. Um alto nível de interação pode significar que um componente possui um alto acoplamento.

4.1.1. Cenários propostos

Serão analisados os seguintes cenários:

1. Mudança no banco de dados remoto
   É desejado que o banco de dados remoto seja substituído, de modo a utilizar a infraestrutura de outro serviço externo.
2. Implementação da pesquisa de notas por título e conteúdo
   É desejado que seja possível realizar a pesquisa por notas, tanto no banco de dados remoto quanto no local. Os resultados devem ser apresentados em ordem cronológica.
3. Implementação de um novo método de autenticação
   É desejado que seja possível autenticar-se utilizando uma conta do GitHub, além dos métodos já existentes.

4.2. O padrão MVC (*Model-View-Controller*)

[Reenskaug 1979] apresenta uma proposta para lidar com conjuntos de dados grandes e complexos dentro de uma aplicação, enquanto permite que a interface de usuário seja atualizada conforme as alterações ocorrem. São apresentados três componentes principais: *Model* (modelo), *View* (visualizador) e *Controller* (controlador).

A *Model* é representada como uma estrutura de dados unificada com os métodos necessários para manipulá-los. Serviços externos, como APIs e bancos de dados podem ser consumidos por ela. Dada uma *Model* qualquer, há uma ou mais *Views* associadas a ela, capazes de exibir alguma representação gráfica ou textual da *Model* em questão. A *View* pode consultar a *Model* para decidir sobre a exibição dos dados. O *Controller* é a ponte entre o usuário e uma ou mais *Views*. Ele é responsável por validar e adequar a entrada do usuário à *Model*, e repassar o resultado do processamento para as *Views* correspondentes. A Figura 2 apresenta um diagrama que ilustra o funcionamento do padrão MVC.

4.3. A Arquitetura Limpa

[Martin 2017] mostra a Arquitetura Limpa, uma abordagem que tem por objetivo delimitar responsabilidades para cada um dos componentes em um software. Esta abordagem

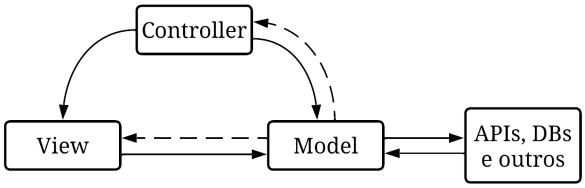

Figura 2. Diagrama do padrão MVC

produz softwares independentes de *frameworks*, interfaces de usuário, bancos de dados e de qualquer agente externo, de modo que as regras de negócio sejam plenamente testáveis.

A Arquitetura Limpa divide as responsabilidades em quatro níveis: entidades, casos de uso, adaptadores de interfaces e agentes externos. As entidades encapsulam regras de negócio inerentes ao domínio da aplicação. Uma entidade pode ser um objeto com métodos, ou um conjunto de estruturas de dados e funções.

Os casos de uso encapsulam as regras específicas para o funcionamento da aplicação. Eles fazem uso das regras de negócio presentes nas entidades para atingir os seus objetivos. Não é esperado que mudanças em agentes externos impactem nos casos de uso.

Os adaptadores de interface são componentes responsáveis por converter dados de um formato para outro. Eles são utilizados, por exemplo, para converter os dados de entidades para um banco de dados ou interface de usuário, e vice-versa.

A camada de agentes externos é composta por *frameworks*, bibliotecas de terceiros, bancos de dados e qualquer outro componente externo que auxilie na execução das regras de negócio.

A Figura 3 ilustra a Arquitetura Limpa como círculos concêntricos, onde cada círculo representa uma camada. As camadas mais internas não interagem diretamente com as camadas mais externas. São utilizadas abstrações para permitir que as camadas se comuniquem, respeitando assim o Princípio de Inversão de Dependências.

5. Resultados obtidos

A seguir são apresentados os resultados obtidos com a aplicação dos cenários propostos em cada uma das arquiteturas analisadas.

5.1. Cenário 1: mudança no banco de dados remoto

No MVC, as operações relacionadas ao banco de dados remoto se encontram nos *Models*, os quais são *NoteModel* e *UserModel*. Toda a lógica para obter e adicionar dados está concentrada nestes dois componentes. Eles fazem uso direto das bibliotecas externas que são necessárias para acessar o Firebase. Com isso, faz-se necessário alterar esses dois componentes quase que em sua totalidade para se adequarem ao cenário proposto. Com a alteração destes dois componentes, também seria necessário efetuar modificações nos *Controllers* da aplicação que interagem diretamente com os *Models*.

Na arquitetura limpa, as operações relacionadas ao banco de dados remoto se encontram na camada de agentes externos. Existem duas classes abstratas que definem

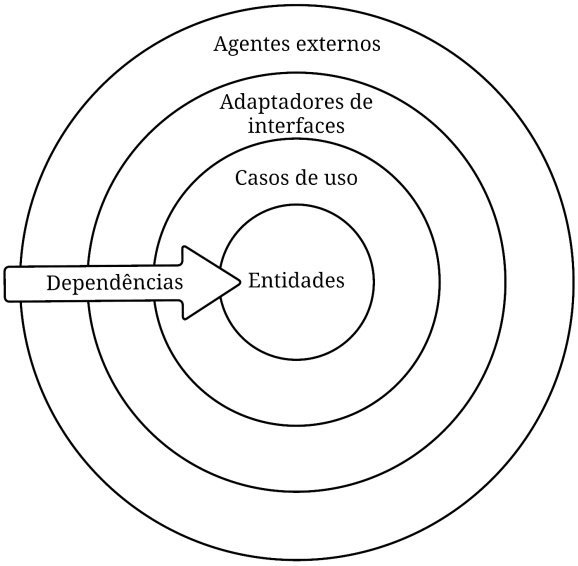

Figura 3. Camadas da Arquitetura Limpa e suas dependências

operações no banco de dados remotos: *INotesRemoteDataBase* e *IUsersRemoteDataBase*. Essas duas interfaces possuem suas respectivas implementações utilizando o Firebase. Os repositórios que fazem uso do banco de dados remoto recebem uma instância das implementações por meio de injeção de dependências, pois estão dependendo de uma abstração, e não de uma implementação concreta. Para se adequar ao cenário proposto, seria necessário criar implementações de *INotesRemoteDataBase* e *IUsersRemoteDataBase* no novo banco de dados remoto, e realizar a injeção deles na aplicação. Não é necessário alterar nenhum componente.

5.2. Cenário 2: implementação da pesquisa de notas por título e conteúdo

No MVC, para adequar-se ao cenário proposto, é necessário criar dois métodos em *NoteModel* para realizar a busca, tanto no banco de dados local como no remoto. Os dois métodos farão uso direto das bibliotecas externas que são necessárias. Um terceiro método para ordenar os resultados de forma cronológica também poderá ser criado, e utilizado em ambos os métodos de busca. Além disso, é necessário criar novos métodos no *Controller* responsável pela listagem de notas, de modo a executar os métodos de busca presentes em *NoteModel*. A *View* que realiza a listagem de notas também sofrerá alterações.

Já na arquitetura limpa, é necessário criar um novo caso de uso que realize a validação dos dados de entrada do usuário, solicite dados ao repositório das notas e que ao recebê-los, realize a sua ordenação de forma cronológica. A abstração e implementação do repositório de notas deverão ser modificadas, adicionando-se um método para realização da busca. O mesmo deve acontecer com as abstrações e implementações dos bancos de dados, que por sua vez, farão uso direto das bibliotecas externas que são necessárias. Assim como no MVC, é necessário criar novos métodos no *Controller* responsável pela listagem de notas, de modo a consumir o recém criado caso de uso, bem como realizar alterações na *View* correspondente.

5.3. Cenário 3: implementação de um novo método de autenticação

No MVC, as operações relacionadas ao registo e autenticação dos usuários se encontram no *UserModel*. Ele faz uso das bibliotecas externas que são necessárias para comunicar-se com os serviços de autenticação e o Firebase. Para se adequar ao cenário proposto, é necessário criar um método que realize a autenticação com o *GitHub*. Também é necessário adicionar novos métodos no *Controller* que lida com a autenticação para executar o novo método criado. A *View* que realiza a autenticação também sofrerá modificações.

Na arquitetura limpa, é necessário criar um novo caso de uso que realize a validação dos dados do usuário e encaminhe-os por meio do repositório de autenticação. Uma abstração e implementação de um serviço que realize a autenticação com o *GitHub* deverão ser criados. Por sua vez, é necessário modificar a abstração e implementação do repositório de autenticação, incluindo a dependência da abstração do novo serviço criado, bem como um novo método que irá consumir o serviço. Assim como no MVC, novos métodos serão criados no *Controller* que lida com a autenticação, que passará a consumir o novo caso de uso, além de modificações na *View* correspondente.

6. Conclusões e trabalhos futuros

No presente artigo, o padrão MVC e a arquitetura limpa são comparadas por meio do método de análise de arquiteturas de software baseado em cenários. São utilizados três cenários: mudanças no banco de dados remoto, implementação da pesquisa de notas por título e conteúdo, e por fim, a implementação de um novo método de autenticação. Em cada cenário, são observados os componentes que precisam ser criados ou alterados.

A partir da análise realizada, é possível observar que cada abordagem possui seus pontos positivos e negativos. O MVC permite que as aplicações sejam desenvolvidas com mais facilidade e rapidez, pois não existem muitos limites arquiteturais a serem respeitados. A lógica da aplicação se concentra nas *Models* e, em alguns casos, também nos *Controllers*. Entretanto, em um cenário em que é preciso realizar modificações profundas, como em uma mudança de banco de dados remoto, as *Models* são reescritas quase que em sua totalidade, e os *Controllers* precisam adaptar-se às mudanças que acontecem. Isso ocorre pois os *Models* fazem uso direto de agentes externos, como bibliotecas de terceiros, sem qualquer camada de abstração, o que os deixam mais suscetíveis a falhas inesperadas. Pelo mesmo motivo, escrever testes unitários pode ser um desafio. Caso a dependência externa não possua uma forma de simular o seu comportamento por meio de uma classe de *Mock*, nem sempre será possível realizar um teste unitário. Neste cenário, o teste de integração se torna a melhor opção para garantir a corretude do código, mas com a desvantagem de ser mais custoso. Além disso, é possível observar que as *Models* tendem a ficar cada vez maiores com a adição de novas funcionalidades, o que pode dificultar a legibilidade e manutenção do código, além de atribuir muitas responsabilidades à um único componente.

Por sua vez, a arquitetura limpa torna o processo de desenvolvimento mais demorado, já que abstrações são utilizadas em várias camadas, e há a necessidade de isolar serviços externos do domínio da aplicação. Seguir por essa abordagem facilita a realização de mudanças mais complexas, conforme observado no cenário da mudança do banco de dados. A lógica da aplicação se concentra nas entidades e nos casos de uso, que não são afetados quando alguma dependência externa sofre alterações. Assim, é possível

realizar testes unitários com facilidade, pois os componentes que lidam com as regras de negócio estão isolados, e não dependem de nada externo para funcionarem corretamente. Entretanto, novas funcionalidades na aplicação exigem a criação de muitos componentes, bem como as interfaces que realizarão a ponte entre as camadas. Pode ser um desafio para desenvolvedores que não estão familiarizados entender todos os conceitos por trás desta abordagem, já que existem muitos componentes com responsabilidades distintas.

De modo geral, o desenvolvimento utilizando a arquitetura limpa é mais demorado em relação ao MVC, além de necessitar que as regras de negócio sejam definidas antes do desenvolvimento ser iniciado. Porém, uma vez desenvolvida, é mais simples realizar a manutenção e os testes unitários da aplicação, já que os componentes estão desacoplados entre si. Já no MVC, o inverso ocorre: é mais rápido desenvolver, mas a manutenção e testes unitários são mais custosos.

Do ponto de vista da experiência de desenvolvimento, o código produzido utilizando o MVC tende a ser extenso, com classes que possuem muitos atributos e métodos. No caso da arquitetura limpa, há a necessidade de se criar muitas classes concretas e abstratas e, por esse motivo, muitas pastas e arquivos diferentes são criados. Essa característica permite que um time trabalhe em partes diferentes da mesma aplicação sem que hajam tantos conflitos, pois arquivos diferentes são utilizados. Ainda, há a possibilidade de se implementar *Micro Frontends* com mais facilidade, isto é, a modularização da aplicação em pequenos pacotes, um para cada funcionalidade, com um único módulo compartilhado entre todos os pacotes.

Em projetos de pequeno porte, provas de conceito e MVPs (*Minimum Viable Product*, ou produto mínimo viável, em inglês), onde se deseja um rápido desenvolvimento, não se planeja realizar grandes mudanças no futuro, e a regra de negócio não é muito complexa, o MVC se torna uma ótima opção. Em grandes projetos, onde a aplicação necessita de estabilidade, possui regras de negócio complexas, esteiras de testes unitários, ou até mesmo uma modularização em pequenos pacotes, a arquitetura limpa mostrou-se mais adequada.

Referências

BANCO CENTRAL DO BRASIL (2022). Estatísticas do pix. `https://www.bcb.gov.br/estabilidadefinanceira/estatisticaspix`.

Cockburn, A. (2005). Hexagonal architecture. `https://alistair.cockburn.us/hexagonal-architecture`.

de Souza Meirelles, F. (2022). Pesquisa anual do uso de ti. `https://eaesp.fgv.br/producao-intelectual/pesquisa-anual-uso-ti`.

Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison Wesley, Boston, MA, 1 edition.

FEBRABAN and Deloitte (2021). Pesquisa febraban de tecnologia bancária. `https://cmsarquivos.febraban.org.br/Arquivos/documentos/PDF/pesquisa-febraban-relatorio.pdf`.

Ghafoor, N. A. . S. (2021). Analysis of architectural patterns for android development.

Google (2011). Firebase. `https://firebase.google.com/`.

Google (2017). Flutter. `https://flutter.dev/`.

Grønli, T.-M. and Ghinea, G. (2016). Meeting quality standards for mobile application development in businesses: A framework for cross-platform testing. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5711–5720, Koloa, HI, USA. IEEE.

Humeniuk, V. (2019). Android architecture comparison: Mvp vs. viper.

Kazman, R., Abowd, G., Bass, L., and Clements, P. (1996). Scenario-based analysis of software architecture. *IEEE Software*, 13(6):47–55.

Martin, R. C. (2017). *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, Boston, MA, 1 edition.

Reenskaug, T. (1979). The original mvc reports. Blindern, Oslo. Department of Informatics, University of Oslo.

Stevens, W. P., Myers, G. J., and Constantine, L. L. (1974). Structured design. *IBM Systems Journal*, 13(2):115–139.