Estudo sobre Clean Architecture
Projeto Disciplina Engenharia de Software

Augusto Calado Bueno

São Paulo
2021

Visão Geral

O design e implementação de sistemas de software é uma atividade extremamente complexa e exigente. Projetar um sistema de tal forma que seja possível garantir sua manutenibilidade e extensibilidade para novas features e componentes é uma tarefa desafiadora.

Conforme o software avança em seu ciclo de vida, esses aspectos tornam-se cada vez mais caros e difíceis de se atingir e manter, pois linearmente com o crescimento do código de um sistema de software, sua complexidade e custo de mudança também aumentam [1].

A arquitetura na qual o desenvolvimento do software é baseado têm um papel fundamental para mitigar a complexidade de inclusão de novas features, dificuldade de manutenção e custo, além de prover a estrutura necessária para atingir os comportamentos esperados do sistema. Ela tem como principal objetivo oferecer suporte ao desenvolvimento e manutenção durante todo o ciclo de vida do projeto [1].

Arquitetura

Arquitetura de um sistema de software nada mais é do que a forma dada ao sistema por aqueles que o implementam [1]. A forma pode ser percebida na maneira como ocorreu a divisão do sistema em componentes, na estruturação desses componentes, e na maneira como a comunicação entre eles ocorre [1].

Dependendo de como a arquitetura é abordada dentro de um projeto de software, seja com mais ou menos ênfase, o impacto em futuros trabalhos é imenso, podendo, ou contribuir para a facilidade de inclusão de novas features e comportamentos, ou tornar cada vez mais custoso a manutenção e, eventualmente, mudanças acabam praticamente impossíveis de se realizar para parte ou para todo o sistema [1].

Portanto, uma arquitetura minuciosamente estudada reduz consideravelmente o custo e os entraves de implementação e, também, comunica as necessidades operacionais.

Clean Architecture

Os sistemas de software podem ser decompostos em dois elementos principais: políticas e detalhes [1]. Os elementos relacionados à política incorporam todas as regras de negócio, são neles onde o verdadeiro valor do sistema reside.

Os detalhes são os facilitadores e habilitadores necessários para que o sistema consiga atingir seu objetivo, mas estes não impactam ou possuem efeito sobre elementos da regra de negócios.

O propósito da arquitetura é criar e estabelecer as formas do sistema, de tal modo que as regras de negócio sejam os elementos centrais, ao mesmo tempo em que torna os detalhes “irrelevantes” perante essas regras [1].

Um dos maiores benefícios que se pode alcançar ao seguir essa abordagem arquitetural é o desenvolvimento das políticas de alto nível sem se comprometer com os detalhes que as cercam. Fazendo isso, é possível adiar decisões sobre os detalhes por um período maior de tempo e, quanto mais no futuro essas decisões forem tomadas, mais informações existirão para ajudar no processo de tomada de decisão [1].

Robert C. Martin, reuniu uma série de técnicas, princípios e conceitos e cunhou o termo *Clean Architecture* para definir um padrão de arquitetura para estruturar e organizar sistemas de software a fim de satisfazer os aspectos que arquiteturas de sistemas necessitam cumprir.

Clean Architecture utiliza uma estrutura em camadas, similar a estrutura definida por Jeffrey Palermo em *Onion Architecture* [2]. Os componentes definidos nas camadas externas dependem dos componentes definidos nas camadas internas, mas não o contrário [5].

A base para o modelo de arquitetura proposto por Robert C. Martin são algumas das regras e princípios listados abaixo:

- Regra da Dependência *(Dependency Rule*), que descreve a ideia de que as relação de dependência entre os elementos do sistema só podem apontar para o centro da arquitetura, local onde residem os as regras de neǵocio;
- Princípio de Inversão de Dependência (*Dependency Inversion Principle*), que afirma que módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações;
- Princípio de abstrações estáveis (*Stable Abstractions Principle*); os componentes estáveis devem ser abstratos. Um possível exemplo de um componente estável e abstrato é uma política de alto nível que é alterada por extensão seguindo o *open-closed principle* [1].
- Princípio da Responsabilidade Única (*Single Responsibility Principle*), que descreve a ideia de que uma classe deve ter um, e apenas um, motivo para mudar;
- Princípio Comum de Fechamento (*Common Closure Principle*), que afirma que classes que mudam juntas devem ser agrupadas;
- Arquitetura Ports and Adapters de Alistair Cockburn, para estabelecer comunicação entre as camadas [6]. As portas representam os limites entre as

camadas. Os adaptadores são, essencialmente, a implementação das portas. Essa arquitetura visa mitigar o acoplamento entre as camadas [6].

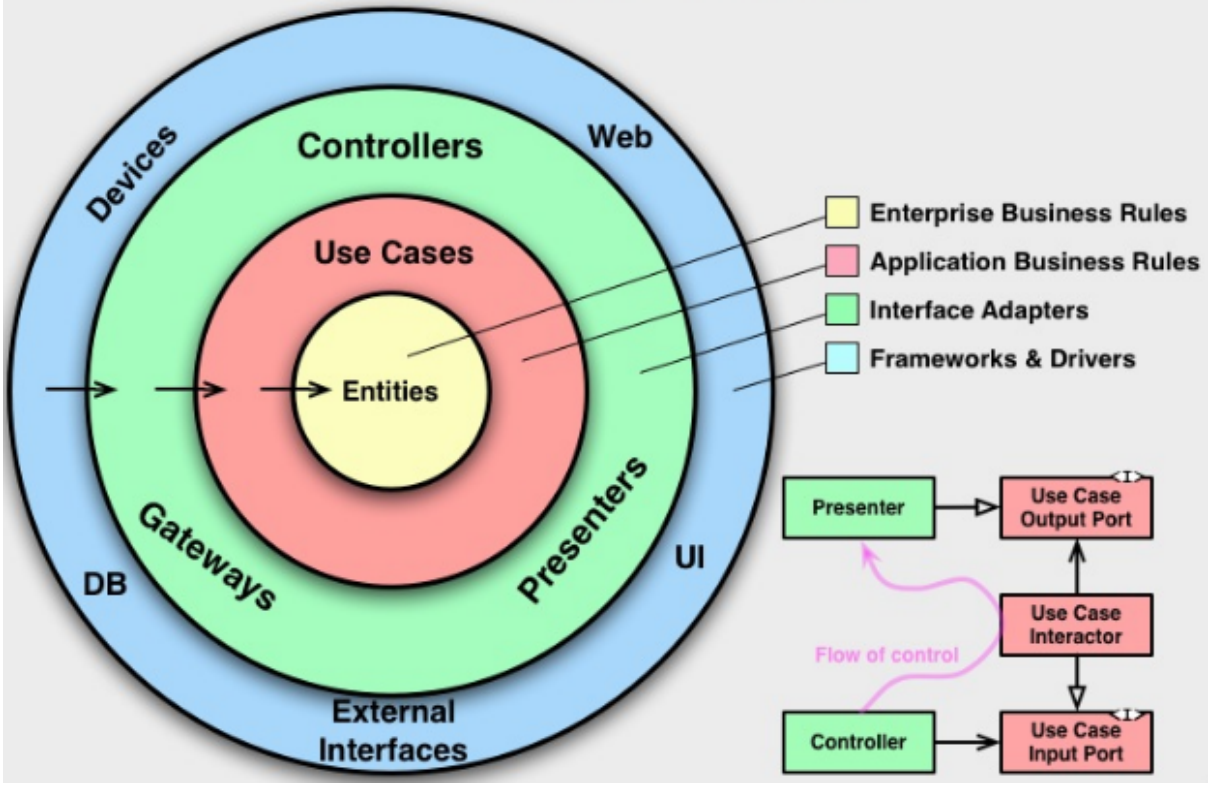

Figura. 1. Camadas estabelecidas em *Clean Architecture* e sua orientação para dentro dependências [2].

As camadas principais do *Clean Architecture* são:

- **Entidades (*Entities*)**: Camada responsável por conter e encapsular as regras críticas de negócio de toda a organização.
- **Casos de Uso (*Use Cases*)**: Camada responsável por conter regras de negócio específicas da aplicação. Essa camada encapsula e implementa os casos de uso do sistema [2], e realiza a manipulação e controle do fluxo de dados que entra e saí da camada de entidade [1].
- **Interfaces e Adaptadores (*Interface e Adaptors*)**: Camada intermediária que estabelece os limites entre *Use Cases* e camadas externas, além de servir de conversor dos dados que circulam entre as camadas.
- **Frameworks e Drivers:** Camada onde os “detalhes” habilitadores do funcionamento do sistema são inseridos. Exemplo: Banco de dados, Frameworks Web e UI.

Exemplo Clean Architecture

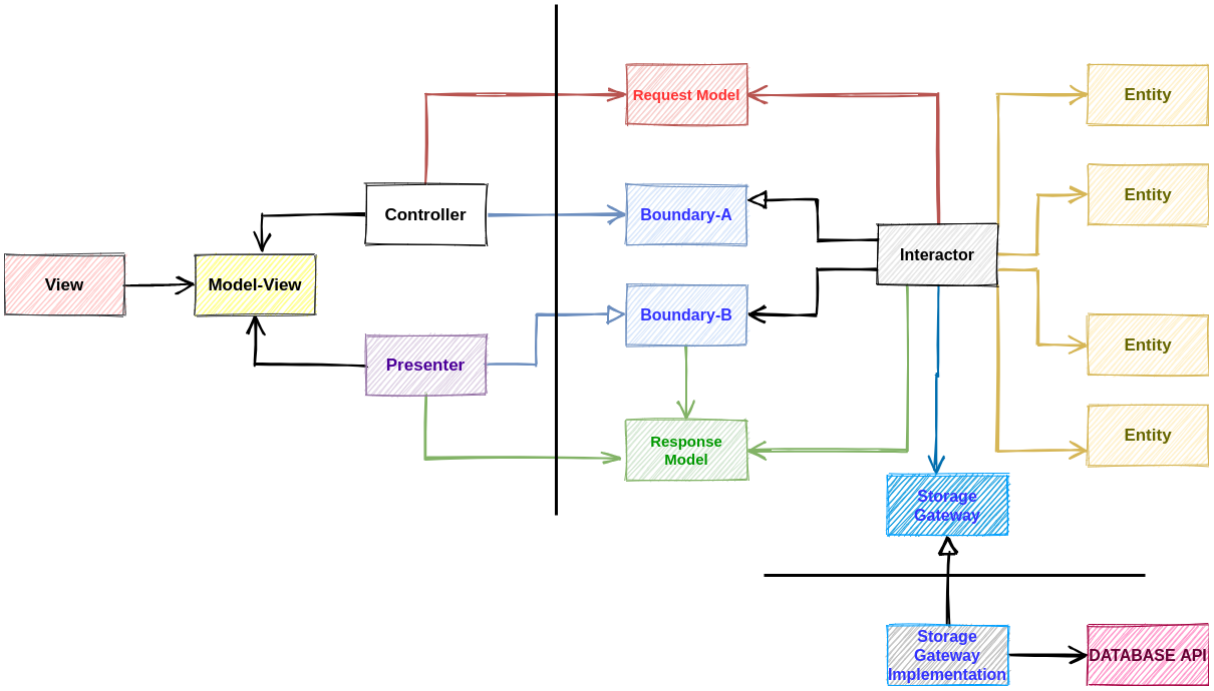

Figura. 2. Estrutura de classes e interfaces Clean Architecture. (fonte: Banco de dados do autor)

A Figura 2 representa um modelo de um sistema genérico aplicando os princípios definidos no *Clean Architecture*. As flechas abertas representam relações de uso, as flechas fechadas representam relação de herança.

O *interactor* representa um *use case* qualquer, ele é responsável por coordenar e acionar as regras definidas nos objetos *entity*. O *interactor* faz uso da interface *boundary-a* para conseguir enviar dados, e implementa a interface *boundary-b* para receber dados.

Os objetos *response* e *request model* são estruturas de dados sem nenhuma lógica embutida Eles são utilizados para transportar dados de uma camada para outra.

O objeto *presenter* que implementa a interface *boundary-a* possui a função de coletar os dados contidos no *response model* gerado pelo *interactor* e reformatá-los para que possam ser exibidos na *view.* Um exemplo de formatação realizado pelo *presenter* pode ser a conversão de um objeto, presente no *response model*, que armazena a data e hora dentro para uma string e enviá-lo para *view* através do objeto *view-model*.

O objeto *controller* converte os dados recebidos da *view* por meio do objeto *model-view* para o objeto *request-model*. Uma vez feita a conversão, o objeto

request-model é enviado para o *interactor* através da utilização da interface *boundary-b*.

Caso haja a necessidade de substituir o esquema de armazenamento ou a forma de input para o *interactor,* pelo fato dos limites estarem bem definidos por meio das interfaces *boundary-a, boundary-b* e *storageGateway*, o impacto no *interactor e entities* é mínimo. Dessa forma as regras de negócio e fluxos do sistema se mantêm estáveis e inalteradas quando alguma alteração é exigida nas camadas externas.

Vantagens do Modelo Arquitetural *Clean Architecture*

A forma como *Clean Architecture* propõe a organização dos elementos de um sistema coloca os casos de uso (*use case*) em evidência, deixando claro qual é a intenção dos componentes. Tal formatação pode ser aplicada em diversos níveis da hierarquia da arquitetura, até mesmo em níveis mais baixos, como classes de um programa.

O comportamento estando visível, contribui para a manutenibilidade, tanto para a diminuição do seu custo, quanto na sua eficácia, pois o entendimento sobre o software é feito de forma mais rápida e acurada, e determinar o melhor local para incluir ou alterar funcionalidades se torna mais fácil e preciso.

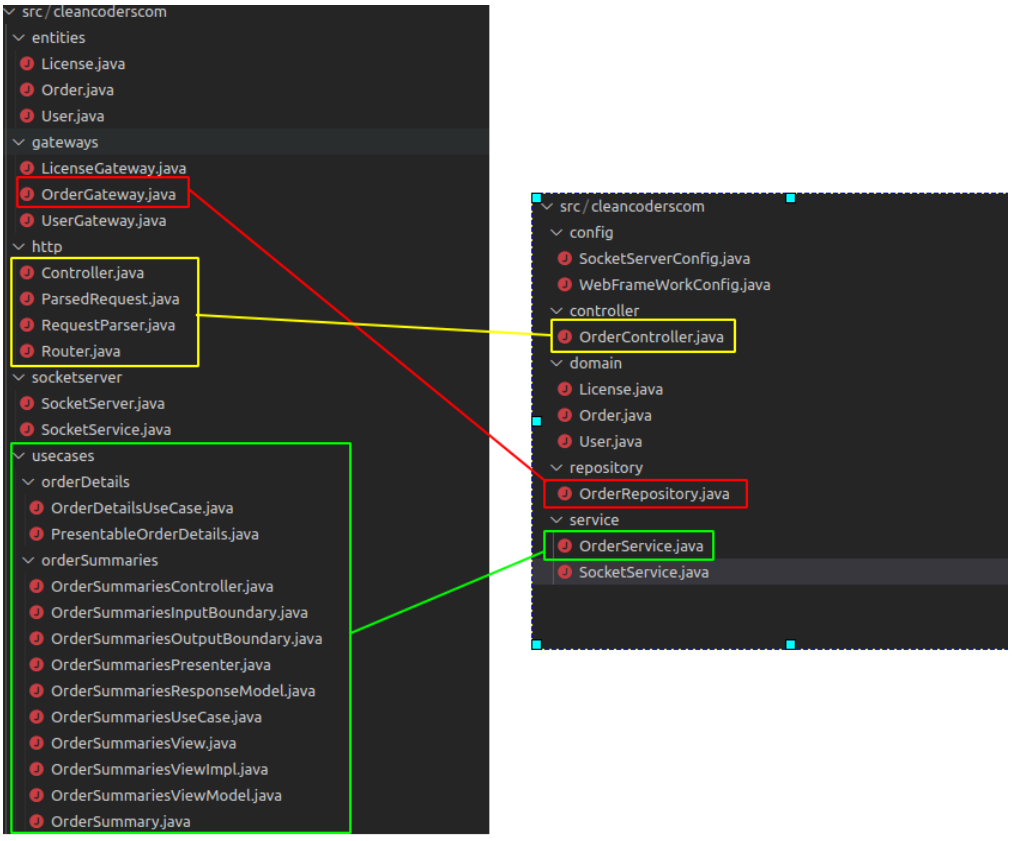

Figura. 3. Estrutura de pastas e arquivos Clean Architecture (esquerda) vs MVC (direita). (fonte: Banco de dados do autor)

A estrutura de pastas e classes do lado esquerdo da figura três é uma possível aplicação do *Clean Architecture*. Notar que há um aumento considerável de classes quando comparado com o modelo MVC, porém, em contrapartida, a intenção do sistema está muito mais nítida.

Além da nitidez da intenção, outras vantagens são obtidas, dentre elas a separação de responsabilidades. No modelo MVC *OrderService concentra* todas as funcionalidades de *OrderSummaries e OrderDetails*. Essa abordagem, apesar de simples, quando alguma alteração é feita no serviço *OrderService* é muito mais suscetível à geração de efeitos colaterais nas funcionalidades que compartilham o mesmo escopo, mas que não seriam o alvo final da mudança.

Clean architecture propõem desenvolver as regras de negócio centrais primeiro, independente de quaisquer detalhes das camadas externas [1]. Para se atingir isso é necessário que haja o desacoplamento entre esses dois tipos de elementos. A utilização de abstrações seguindo diversos princípios como *Dependency Inversion Principle*, *Common Closure Principle e, também a* arquitetura *Ports and Adapters* para estabelecer relações e comunicação entre as camadas diminui o acoplamento.

Ao organizar o sistema de software dessa maneira, uma série de benefícios são obtidos, dentre eles, a capacidade de teste das camadas centrais (use case e entidades) e experimentação dos detalhes (banco de dados, frameworks, UI, etc).

Como a relação dos componentes de diferentes camadas se dá por meio de abstrações e interfaces, é possível configurar, através de mocks, o comportamento das camadas externas para testar os componentes das camadas internas.

A possibilidade de se realizar experimentações dos detalhes, é muito importante para a tentativa de se atingir requisitos operacionais. Imagine uma aplicação que possui uma demanda por escrita em banco muito elevada, dado a estrutura do *Clean Architecture,* que se vale do desacoplamento das camadas centrais em relação às camadas externas, é fácil de se experimentar qual banco de dados melhor atende os requisitos operacionais que o sistema demanda sem haver impactos nas regras e políticas centrais.

Desvantagens do Modelo Arquitetural *Clean Architecture*

A redundância gerada pelo *Clean Architecture* pode ser considerada como uma desvantagem, pois ao colocar em prática as ideias de desacoplamento e independência entre os componentes, é possível que ocorra redundância de código e lógica. Um exemplo que podemos citar é o caso das estruturas de dados que trafegam entre as camadas externas, como a camada de banco de dados; para as

camadas internas. Esses objetos podem ser exatamente iguais, mas para desacoplar as camadas internas do banco de dados, replica-se a estrutura do objeto armazenado em banco em um objeto DTO (*Data transfer Object*), para impedir que as *Entities* ou *Use Cases* conheçam esse objeto estritamente vinculado a camada de implementação da persistência de dados.

Para se atingir o desacoplamento das camadas centrais da arquitetura das demais camadas, para cada integração ou relacionamento que eventualmente venha a ocorrer, interfaces terão de ser criadas. Dependendo do número de comunicações e relações que se estabelecem entre os níveis, muitas interfaces e abstrações podem existir.

Em projetos pequenos a utilização do *Clean Architecture* talvez não seja a melhor abordagem, pois a complexidade de implantação é muito elevada para os ganhos que esse sistema de menor porte consiga obter. Isso faz com que haja limitações de contexto onde esse tipo de organização arquitetural, ao ser aplicada, realmente faça a diferença.

Processos Relacionados a ISO 12207

ISO 12207 Systems and software engineering — Software life cycle processes, é um padrão internacional e seu principal objetivo é definir e estipular padrões dos processos necessários para desenvolver e manter sistemas de software. Dentre as diversas atividades propostas pela ISO 12207/2017, algumas podem ser diretamente endereçadas pelo padrão de arquitetura proposto por Robert C. Martin, *Clean Architecture*.

Tarefas ISO 12207 Relacionadas com *Clear Architecture*

> - *Architecture Definition process*, **atividade:** A - *Prepare for architecture definition*, **tarefa:** *Identity stakeholders concerns*. (ISO 12207, 2017)

O escopo dessa atividade trata-se da identificação de algumas restrições que o sistema precisa atender, dentre elas operacionais como disponibilidade e usabilidade. Devido a possibilidade de experimentação dos detalhes (são as camadas mais externas na arquitetura, não vinculadas as regras de negócio e políticas de alto nível) proporcionado pelo *Clean Architecture*, essa tarefa pode ser mais facilmente executada à medida que testes de diferentes tecnologias podem ser aplicados para se atingir alguns dos requisitos operacionais do *stakeholder* sem exigir mudanças nas camadas internas da arquitetura.

- *System/Software Requirements Definition Process*, **atividade:** A - *Prepare for System/Software Requirements Definition*, **tarefa:** *Define the functional boundary of the software system or element in terms of the behavior and properties provided.* (ISO 12207, 2017)

- *Architecture Definition process*, **atividade:** C - *Develop models and views of candidate architecture*, **tarefa:** *Define the software system context and boundaries in terms of interfaces and interactions with external entities.* (ISO 12207, 2017)

- *Architecture Definition process,* **atividade:** D - *Relate the architecture to design,* **tarefa:** *Define the interfaces and interactions among the software system elements and external entities.* (ISO 12207, 2017)

Nestas atividades da ISO 12207, discute-se a etapa de identificação e definição dos limites de interação do sistema que está sendo desenvolvido com elementos externos, como sistemas previamente existentes. *Clean Architecture* estimula o desenvolvimento das principais regras de negócio agnósticas as demais camadas [1]. Portanto, por padrão, a orientação da ISO 12207 de definição dos limites do sistema são fortemente seguidos e discutidos nos primeiros momentos da projeção da arquitetura clean (aquela que segue os princípios do *Clean Architecture*), e elas normalmente são implementadas por meio da criação de gateways/ports (interfaces) para a estabelecimento dos relacionamentos e comunicação.

- *Architecture Definition process*, **atividade:** C - *Develop models and views of candidate architecture*, **tarefa:** *Identify architectural entities and relationships between entities that address key stakeholder concerns and critical software system requirements.* (ISO 12207, 2017)

Nesta tarefa há a discussão de que a arquitetura não necessariamente preocupa-se com todos os requisitos, mas devota-se, principalmente, aos requisitos que impactam diretamente a arquitetura. A abordagem que o *Clean Architecture* segue para desenvolver primeiro as políticas centrais em detrimento dos detalhes [1] suporta plenamente a execução dessa tarefa. A ideia é deixar em aberto certos tipos de decisões que estão relacionadas ao sistema, mas não possuem impacto direto no core do negócio. Um exemplo que podemos comentar é qual banco de dados o sistema adotará. Isso é uma preocupação, mas não uma preocupação de nível arquitetural.

- *Architecture Definition process*, **atividade:** C - *Develop models and views of candidate architecture*, **tarefa:** *Allocate concepts, properties, characteristics,*

behaviors, functions, or constraints that are significant to architecture decisions of the software system to architectural entities. (ISO 12207, 2017)

Como o título da tarefa três sugere, nesta etapa é onde se determina alguns conceitos e comportamentos esperados da execução do sistema. A definição e priorização das regras de negócio centrais prescrito em *Clean Architecture* estão profundamente relacionadas a essa atividade, pois o que guia a definição da arquitetura são as *Entities* e *Use case*, e são nessas camadas que comportamento e fluxos estão definidos, e são a partir dessas camadas que todo o projeto de software será orientado.

- *Architecture Definition process*, **atividade:** D - *Relate the architecture to design,* **tarefa:** *Define principles for the software system design and evolution.* (ISO 12207, 2017)

- *Design Definition process*, **atividade:** A - *Prepare for software system design definition,* **tarefa:** *Select and prioritize design principles and design characteristics.* (ISO 12207, 2017)

Nesta tarefa discute-se a definição dos conceitos e técnicas que um sistema ao ser implementado deve seguir. *Clean architecture* define uma série de princípios, como SOLID e outros para nortearem o desenvolvimento e evolução do projeto de software.

Relacionamentos Indiretos ISO 12207 e *Clean Architecture*

A adoção do *Clean Architecture* como arquitetura do sistema, pode contribuir positivamente para algumas atividades estipuladas na ISO 12207 de forma indireta. Como por exemplo a tarefa *Define each function that the software system or element is required to perform*, da atividade B - *Define system/software requirements*, do processo de *Definição de Requisitos de Sistemas/Software*, e a tarefa *Identify required states or modes of operation of the software system*, da atividade B - *Define system/software requirements* do processo de Definição de Requisitos de Sistemas/Software. A primeira tarefa comenta sobre a identificação e definição de cada função que um sistema de software deve fazer (casos de uso); já a segunda descreve a determinação dos requisitos operacionais em que o sistema deve executar.

No início do desenvolvimento de um projeto de software nem todos os casos de uso ou requisitos operacionais são conhecidos de antemão, a formatação que o *Clean Architecture* gera no sistema de software que o aplica, permite que o impacto pelo não conhecimento de alguns casos de uso e restrições operacionais sejam baixos [1]. A inclusão de novos casos de uso tendem a gerar menos impactos nos

casos de uso já existente do sistema, pois o sistema foi organizado de tal forma que cada caso de uso é logicamente separado uns dos outros.

Com relação aos requisitos operacionais, devido a divisão do sistema, fica mais fácil aplicar ações isoladas para se atingir e respeitar as restrições operacionais fazendo-se uso da experimentações dos detalhes, como discutido anteriormente na seção “Vantagens do Modelo Arquitetural Clean Architecture”.

O processo de *Knowledge Management* definido na ISO-12207 também é positivamente impactado. Devido a forma como o sistema é arranjado dentro do *Clean Architecture* onde as camadas *Entity* e *Use Case* armazenam as políticas e comportamentos de alto nível que o sistema deve executar, e também pela forma como os componentes de mais baixo nível são organizados (por meio da utilização de princípios como Common Closure Principle) , a intenção do sistema fica explícita, a arquitetura diz o que o sistema faz [1], auxiliando na facilitação do conhecimento contido no próprio sistema.

Referências

[1] MARTIN, Robert C. The Clean Architecture: A Craftsman‘s Guide to Software Structure and Design. Pearson Education, 2017.

[2] MARTIN, Robert C. The Clean Architecture. 13 ago. 2012. Disponível em: https://8thlight.com/blog/uncle-bob/2012/08/13/the-clean-architecture.html. Acesso em: 4 maio 2021.

[3] MARTIN, Robert C. A Handbook of Agile Software Craftsmanship. Pearson Education, 2008.

[4] IVANICS, Péter. An Introduction to Clean Software Architecture. 2017. Disponível em: https://pivanics.users.cs.helsinki.fi/portfolio/docs/publications/Peter_Ivanics-Clean_Software_Architecture.pdf. Acesso em: 4 maio 2021.

[5] PALERMO, Jeffrey. The Onion Architecture. 2008. Disponível em: https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/. Acesso em: 4 maio 2021.

[6] COCKBURN, Alistair. Hexagonal architecture. 2005. Disponível em: https://alistair.cockburn.us/hexagonal-architecture. Acesso em: 4 maio 2021.

[7] KNILL, Matthew. Refactoring with Clean Architecture, 2019. Disponível em: http://courses.cecs.anu.edu.au/courses/CSPROJECTS/19S2/reports/u6052043_report.pdf. Acesso em: 4 maio 2021.

[8] INTERNATIONAL STANDARD. ISO 12207: Systems and software engineering — Software life cycle processes. 2017.