INTRODUÇÃO

A expressão "Documentação ágil" nesse trabalho, faz referência a atividade de documentação realizada em projetos de *software* que seguem metodologias ágeis ou estão alinhados com os preceitos do Manifesto Ágil.

Segundo Briand (2003), o problema da documentação é um problema ainda em aberto, dado que, se por um lado, há consenso tanto sobre os custos quanto sobre sua necessidade, por outro lado, não há consenso sobre suas soluções:

> Trabalhos recentes sugerem que a documentação é muitas vezes percebida como muito custosa, difícil de manter frente às pressões típicas que perpassam toda a indústria de *software*. O interessante é que a maioria das pessoas pensam que alguma forma de documentação é necessária, mas há pouca concordância sobre o que se necessita. E ainda mais surpreendente é que pessoas cotidianamente usam documentações incompletas e obsoletas e as consideram úteis (BRIAND, 2003, tradução).

O Manifesto Ágil é um ponto de inflexão sobre a documentação, contestando a abordagem orientada a documentos do processo tradicional. Contudo, o problema da documentação ágil não é claro no manifesto, dependendo de como cada metodologia ágil oferece uma solução.

No desenvolvimento ágil, o problema é mais grave, dado que "em projetos de desenvolvimento de *software* ágeis, os engenheiros de *software* priorizam a implementação sobre a documentação" (SAITO et al., 2017, p. 194, tradução).

A abordagem DDD (Domain-Driven Design) é um *framework* de decisões de projetos orientado ao domínio que oferece um modelo de documentação ágil com características próprias. Comparativamente, a abordagem DDD fornece uma documentação relativamente mais abrangente. Nesse contexto surge a questão se essa maior abrangência documental do DDD solucionaria os problemas de documentação em projetos ágeis.

O problema tratado nesta pesquisa é a questão: A abordagem *Domain-Driven Design* seria uma solução documental adequada para as deficiências de documentação em projetos ágeis identificadas em pesquisas como a de Saito et al. (2017), Stettina et al. (2012) e Voigt et al. (2016)?

O objetivo principal desta pesquisa foi apresentar a aderência do modelo de documentação da abordagem *Domain-Driven Design* (EVANS, 2016) ao processo ágil e como este modelo soluciona os problemas de documentação de projeto através dos seguintes objetivos específicos:

a) Fundamentar a documentação, Manifesto Ágil e DDD;
b) Apresentar o modelo de domínio como documentação ágil DDD.
c) Evidenciar os problemas de documentação em projetos ágeis;
d) Demonstrar a solução DDD para os problemas encontrados.

REFERENCIAL TEÓRICO

Manifesto Ágil

Organizado por Robert Cecil Martin em fevereiro de 2001 na cidade Wasatch, no estado norte-americano de Utah, uma reunião sobre "métodos leves" reuniu 21 signatários do que depois ficou conhecido como Manifesto Ágil.

> Foram convocados representantes do *Extreme Programming*, *Scrum*, DSDM, *Adaptive Software Development*, *Crystal*, *Feature-Driven Development, Pragmatic Programming* entre outros simpatizantes da necessidade de uma alternativa para o processo de desenvolvimento de software pesado orientado a documentação (MANIFESTO, 2019, tradução).

O termo "método leve" foi substituído por "método ágil", com a reunião resultando em um manifesto baseado em quatros valores (MANIFESTO, 2019):

a) Indivíduos e interações mais que processos e ferramentas;
b) Software em funcionamento mais que documentação abrangente;
c) Colaboração com o cliente mais que negociação de contratos;
d) Responder a mudanças mais que seguir um plano.

Além desses quatro valores, o Manifesto Ágil também prescreve os seguintes doze princípios que são regidos pelos quatro valores já citados:

a) Nossa maior prioridade é satisfazer o cliente, através da entrega adiantada e contínua de *software* de valor;
b) Aceitar mudanças de requisitos, mesmo no fim do desenvolvimento. Processos ágeis se adequam a mudanças, para que o cliente possa tirar vantagens competitivas;
c) Entregar *software* funcionando com frequência, na escala de semanas até meses, com preferência aos períodos mais curtos;
d) Pessoas relacionadas à negócios e desenvolvedores devem trabalhar em conjunto e diariamente, durante todo o curso do projeto;
e) Construir projetos ao redor de indivíduos motivados. Dando a eles o ambiente e suporte necessário, e confiar que farão seu trabalho;
f) O método mais eficiente e eficaz de transmitir informações para, e por dentro de um time de desenvolvimento, é através de uma conversa cara a cara;
g) *Software* funcional é a medida primária de progresso;
h) Processos ágeis promovem um ambiente sustentável. Os patrocinadores, desenvolvedores e usuários, devem ser capazes de manter indefinidamente, passos constantes;
i) Contínua atenção à excelência técnica e bom *design*, aumenta a agilidade;
j) Simplicidade: a arte de maximizar a quantidade de trabalho que não precisou ser feito;
k) As melhores arquiteturas, requisitos e *designs* emergem de times auto-organizáveis;
l) Em intervalos regulares, o time reflete em como ficar mais efetivo, então, se ajustam e otimizam seu comportamento de acordo.

Segundo a 3ª versão do Guia para o Corpo de Conhecimento da Engenharia de *Software* (SWEBOK), as metodologias ágeis, enquanto Método de Engenharia de *Software*, se definem como:

> Métodos ágeis são caracterizados por ciclos de desenvolvimentos iterativos e curtos, times auto-organizados, projetos mais simples, refatoração de código, desenvolvimento orientado a teste, envolvimento frequente com o cliente e ênfase em criar um produto de trabalho demonstrável em cada ciclo de desenvolvimento (BOURQUE et al., p. 9-9, 2014, tradução).

A Engenharia de *Software* é definida como:

> Aplicação sistemática, disciplinada e quantificável de uma abordagem para o desenvolvimento, operação e manutenção do *software* (IEEE apud BOURQUE et al., 2014, p. xxxi, tradução).

Ciência da Computação

Newel define a Ciência da Computação como o “estudo dos computadores” (NEWEL et al. apud MCGUFFEE, 2000, p. 74, tradução). A Engenharia de *Software* se divide nas seguintes áreas de conhecimento, conforme apresenta a Tabela 1 abaixo:

Tabela 1 - Áreas de conhecimento.

Áreas	**Descrição**
Requisito	Se refere a “(...) restrições impostas a um produto de *software* que contribuem para a solução de algum problema (...)” (BOURQUE et al., p. 1-1, 2014, tradução).
Projeto	Especifica “(...) arquitetura, componentes, interfaces e outras características de um sistema ou componente” (BOURQUE et al., p. 2-1, 2014, tradução).
Construção	Se refere a “combinação de codificação, verificação, teste de unidade, teste de integração e depuração”, visando reuso, mudança, simplicidade, verificação e padronização (BOURQUE et al., p. 3-1, 2014, tradução).
Teste	Se refere a “verificação dinâmica de (...) comportamentos esperados em um conjunto finito de casos de teste, adequadamente selecionados no domínio de execução geralmente infinito” (BOURQUE et al., p. 4-1, 2014, tradução).
Manutenção	Prover “suporte econômico para o software (...) [modificando] software existente enquanto preserva sua integridade” (BOURQUE et al., p. 5-1, 2014, tradução).
Processo	Visa “medir e melhorar a qualidade dos produtos de *software* de maneira eficiente; apoiar a melhoria de processos e fornece uma base para o suporte automatizado da execução do processo” (BOURQUE et al., p. 8-1, 2014, tradução).
Configuração	Se refere a “(...) orientação e supervisão técnica e administrativa para identificar e documentar as características funcionais e físicas de um item de configuração, controlar alterações nessas características, registrar e relatar o status de processamento e implementação de alterações e verificar a conformidade com os requisitos especificados” (BOURQUE et al., p. 6-1, 2014, tradução).
Gerência	Se refere ao “planejamento, coordenação, medição, monitoramento, controle e geração de relatórios - para garantir que os produtos e serviços de engenharia de *software* sejam fornecidos com eficiência, eficácia e benefício das partes interessadas” (BOURQUE et al., p. 7-1, 2014, tradução).
Modelos e métodos	O modelo “(...) fornece uma abordagem para a solução de problemas, uma notação e procedimentos para construção e análise de modelos. Os métodos fornecem uma abordagem para a especificação sistemática, projeto, construção, teste e verificação do *software* (..) e (..) associados” (BOURQUE et al., p. 9-1, 2014, tradução).
Qualidade	Se refere aos “(...) métodos de medição e os critérios de aceitação para avaliar o grau em que o *software* e a documentação relacionada atingem os níveis de qualidade desejados” (BOURQUE et al., p. 10-1, 2014, tradução).

Prática profissional	Se refere ao "(...) conhecimento, habilidades e atitudes que os engenheiros de *software* devem possuir para praticar a engenharia de *software* de maneira profissional, responsável e ética". (BOURQUE et al., p. 11-1, 2014, tradução).
Economia	Conceitos e práticas comuns de economia de engenharia de *software* para tomada de decisão em uma perspectiva de negócio em um ciclo de vida que destaca os riscos e incertezas gerenciais (BOURQUE et al., p. 12-1, 2014, tradução).
Fundamentos	Os fundamentos científicos da Engenharia de *Software* são a computação, enquanto parte da ciência da computação; a matemática, enquanto cobertura das técnicas básicas para modelagem usando provas matemáticas; e a engenharia, no uso de técnicas de engenharia aplicadas no desenvolvimento de *software*.

Fonte: BOURQUE et al., p. xxxii, 2014, tradução.

Engenharia de Software

Segundo o SWEBOK, os "métodos de Engenharia de Software providenciam uma abordagem organizada e sistemática para o desenvolvimento de software para um computador alvo" (BOURQUE et al., 2014, p. 9-7, tradução), sendo classificados como métodos heurísticos, formais, prototípicos e ágeis, conforme apresenta a Tabela 2 abaixo.

Tabela 2 - Métodos de Engenharia de *Software*.

Padrões	Descrição
Métodos heurísticos	"Métodos heurísticos são aqueles métodos de engenharia de *software* que são baseados na experiência que tem sido amplamente praticado na indústria de *software*" (BOURQUE et al., 2014, p. 9-7, tradução).
Métodos formais	"Métodos formais são métodos de engenharia de *software* usados para especificar, desenvolver e verificar o *software* através da aplicação de uma rigorosa notação e linguagem matematicamente baseada" (BOURQUE et al., 2014, p. 9-7, tradução).
Métodos prototípicos	"O engenheiro de *software* seleciona um método prototípico para entender os últimos aspectos entendidos ou componentes do primeiro *software*" (BOURQUE et al., 2014, p. 9-8, tradução).
Métodos ágeis	"Métodos ágeis são considerados métodos leves em que são caracterizados por ciclos de desenvolvimento curtos e iterativos" (BOURQUE et al., 2014, p. 9-9, tradução).

Fonte: BOURQUE et al., 2014, p. 9-9, tradução.

Métodos Ágeis

Segundo o SWEBOK, "muitos métodos ágeis estão disponíveis na literatura" (BOURQUE et al., 2014, p. 9-9, tradução), contudo, compartilham aspectos comuns que os identificam como um método ágil (seção 2.1.2):

> Métodos ágeis são caracterizados por ciclos de desenvolvimentos iterativos e curtos, times auto-organizados, projetos mais simples, refatoração de código, desenvolvimento orientado a teste, envolvimento frequente com o cliente e ênfase em criar um produto de trabalho demonstrável em cada ciclo de desenvolvimento (BOURQUE et al., 2014,p. 9-9, tradução).

Segundo o SWEBOK, métodos leves e pesados atendem a necessidades diferentes, havendo lugar para ambos, citando também uma tendência a formação de novos métodos

mistos que combinam elementos de métodos leves e com elementos de métodos pesados, como balanceamento entre ambos:

> No que que sempre haverá lugar para métodos de engenharia de software pesados e baseados em planos, bem como lugares onde brilham os métodos ágeis. Há novos métodos surgindo da combinação de processo ágil e métodos baseados em planos onde os praticantes estão definindo novos métodos que balanceiam características necessárias entre método pesado e leve baseado, primariamente, na prevalência das necessidades organizacionais do negócio (BOURQUE et al., 2014, p. 9-8, tradução).

Na Tabela 3 abaixo, é apresentada uma breve explicação sobre o funcionamento de alguns métodos ágeis, conforme amostra de metodologias ágeis mais populares selecionadas pelo SWEBOK:

Tabela 3 - Metodologias Ágeis.

Padrões	**Descrição**
Scrum (termo do *Rugby* para simbolizar o trabalho colaborativo)	É uma abordagem gerencial com um gerente de processo denominado de *Scrum Master* mediando as cerimônias (*meetings*) e um *Product Owner* como um analista de negócio priorizando demandas (*backlog*). O ciclo (*sprint*) é composto de reuniões diárias (*daily*), reunião de entrega (*review*) e reunião de discussão visando a melhoria contínua (*retrospective*).
RAD (*Rapid Application Development*)	Utiliza ferramentas especializadas para banco de dados viabilizando engenharia rápida de desenvolvimento, teste e implantação de certos tipos de aplicações tanto para sua criação quanto para sua evolução.
XP (*Extreme Programming*)	Utiliza histórias de usuário (*user stories*), eventualmente com cenários, como documentação de requisitos, promovendo práticas como o TDD, teste de aceitação com aproximação com o cliente, programação em par, micro-refatorações, integrações contínuas, entre outras.
FDD (*Feature-Driven Developement*)	Utiliza ciclos de desenvolvimento iterativo, curto e dirigido por modelo (model-driven), se dividindo em cinco fases: delimitar domínio, listar funcionalidade, planificar funcionalidades, planejar iterações das funcionalidades e integração contínua das funcionalidades (codificar, testar e integrar funcionalidades).

Fonte: BOURQUE et al., 2014, p. 9-9, tradução.

Documentações

Documentação de *software*

Segundo Kipyegen et al. (2013), em uma pesquisa sobre documentação com desenvolvimento ágil, a documentação é uma atividade crítica, sendo relevante ao ambiente de desenvolvimento e à manutenção de sistemas:

> Documentação de *software* é uma atividade crítica na Engenharia de *Software*. A documentação melhora a qualidade do produto de *software* e joga um importante papel no ambiente de desenvolvimento de *software* e manutenção de sistema (KIPYEGEN et al., 2013, p. 223, tradução).

Enquanto resultado ou produto da atividade de documentação, o documento é definido como "um artefato cujo propósito é comunicar informação sobre o sistema de *software* da qual faz parte" (KIPYEGEN, 2013, p. 223, tradução).

Na metodologia tradicional RUP, a documentação de *Software* abrange diagramas, modelos, formatos, guias, manuais, protótipos, requisitos, contratos, entre outros

(PRIESTLEY, 2000, p. 234, tradução), mas, conforme delimitação (seção 1.1), a análise dos artefatos se restringiu à documentação de projeto de *software*.

Documentação de projeto

Segundo SWEBOK, a documentação de projeto de *software* é classificada em duas visões: estática e dinâmica (BOURQUE et al., 2014, p. 2-8, tradução).

Visão estática

A visão estática "descreve e representa aspectos estruturais do projeto de *software* (...) para descrever componentes maiores e como estão interconectados" (BOURQUE et al., 2014, p. 2-8, tradução), descrito na Tabela 4 abaixo:

Tabela 4 - Notação estática.

Notação	**Descrição**
Arquitetura (ADL)	Linguagens utilizadas para descrever a arquitetura do *software* em termos de componentes e conectores.
Diagrama de Classe ou Objeto	Utilizados para representar um conjunto de classes (e objetos) e seus relacionamentos entre si.
Diagrama de componente	Utilizado para representar um conjunto de componentes com seus relacionamentos entre si.
Cartões (CRC)	Utilizado para denotar nomes de componentes (classe), suas responsabilidades e seus colaboradores.
Diagrama de implantação	Utilizado para representar um conjunto de nós físicos e suas inter-relações com aspectos físicos do *software*.
Entidade-Relacionamento (ERD)	Utilizado para representar modelos conceituais de dados armazenados em repositórios de informação.
Interface (IDL)	Utilizada para definir interfaces com linguagens de programação dos componentes de *software*.
Gráficos estruturais	Utilizado para descrever a chamada estrutural dos programas (qual módulo chamar, por exemplo).

Fonte: BOURQUE et al., 2014, p. 2-9, tradução.

Visão dinâmica

Segundo o SWEBOK, notações de visão dinâmica são "usadas para descrever o comportamento dinâmico de sistemas de software e componentes. Muito dessas notações são úteis muitas vezes, mas não exclusivamente, durante o detalhamento do projeto" (BOURQUE et al., 2014, p. 2-9, tradução), conforme tabela 5 abaixo:

Tabela 5 - Notação dinâmica.

Notação	**Descrição**
Diagrama de atividade	Utilizado para mostrar o controle de fluxo de atividade em execução. Pode ser usado para representar atividades concorrentes.
Diagrama de comunicação	Utilizado para mostrar as interações que ocorrem entre um grupo de objetos, enfatizando nos objetos, suas ligações, mensagens e trocas.

Diagrama de fluxo (DFD)	Utilizado para mostrar o fluxo de dados entre elementos, podendo ser usado para análise de segurança.
Tabela ou diagrama de decisão	Utilizado para representar combinações complexas de condições e ações, podendo ser uma tabela ou um diagrama.
Fluxograma	Utilizado para representar o fluxo de controle com suas ações associadas para serem executadas em uma dada ordenação de passos.
Diagrama de sequência	Utilizado para mostrar as interações entre um grupo de objetos com ênfase na ordem de tempo das mensagens passadas pelo objeto.
Diagrama de estado	Utilizado para mostrar o fluxo de controle de estado em estado e como o comportamento do componente muda baseado no estado atual.
Especificação formal	São linguagens textuais que usam noções básicas de matemáticas para rigorosamente e abstratamente definir o *software* e seu comportamento.
Pseudo-código (PDL)	São linguagens de programação estruturadas para descrever, geralmente em um estágio mais detalhado, os aspectos comportamentais.

Fonte: BOURQUE et al., 2014, p. 2-9, tradução.

Documentação ágil

Segundo Folwer (2005), as metodologias ágeis surgiram em reação às metodologias tradicionais. O processo tradicional contém as seguintes características:

a) Modelo de processo em cascata;
b) Processo guiado pela documentação;
c) Imposição do processo sobre o desenvolvimento;
d) Arquitetura prescritiva com pouca liberdade de implementação.

Segundo Shafiq (2018), a documentação ágil se diferencia da tradicional por se concentrar no suporte ao desenvolvimento do software:

> Na metodologia ágil, a documentação (...) se concentra no desenvolvimento. (...) a metodologia ágil suporta a documentação que é importante para a implementação (SHAFIQ, 2018, tradução).

Em projetos com metodologias ágeis, a priorização da implementação sobre a documentação ajuda a eliminar documentação desnecessária:

> Em projetos de *software* ágeis, engenheiros de *software* priorizam implementação sobre documentação como forma de eliminar documentação desnecessária. (SAITO et al., 2017, p. 194, tradução).

A Tabela 6 a seguir, relata uma pesquisa comparativa com algumas características do modelo de documentação de algumas metodologias ágeis:

Tabela 6 - Documentação nas metodologias ágeis.

Método	Papel da Documentação	Retenção da Informação
XP	Moderadamente gerenciado, menos documentação	Poucos documentos, verbal
Lean	Menos documentação para feedback e teste, documentação sumarizada	Comunicação cara-a-cara
Crystal	Moderada	Comunicação concisa, densa e formal

FDD	Por-demanda, leve	Combinado com o time
DSDM	Enxuta e rápida	Prototipação adiantada
MSF	Pesada	Conforme usuário-alvo
AUP	Pesada	Não se aplica
ASD	Moderada	Não se aplica

Fonte: Shafiq, 2018, tradução.

No Manifesto Ágil, a documentação é citada no segundo valor ágil, ao valorizar "software funcionando, mais do que documentação abrangente". Também ocorre a menção ao citar contrato no terceiro valor e plano no quarto valor.

Enquanto meio de comunicação, a documentação é também preterida em favor da conversação, conforme o sexto princípio no Manifesto Ágil. Como resultado da análise do Manifesto Ágil e das pesquisas sobre desenvolvimento ágil, as indicações favorecem uma documentação com uma orientação geral mais contida, sumarizadas nos três critérios a seguir como evidências de uma documentação ágil: utilidade, leveza e minimalismo.

a) Útil: conteúdo relevante que auxilie ao desenvolvimento;
b) Leve: documentação com redação concisa, objetiva e minimalista;
c) Mínima: número reduzido de artefatos de documentação diferentes.

DDD - DOMAIN-DRIVEN DESIGN

A abordagem DDD tem como problema-alvo o controle da complexidade do software, assumindo que a causa principal da complexidade reside no domínio.

> São várias as coisas que tornaram complexo o desenvolvimento de *softwares*. Mas o coração dessa complexidade está na dificuldade essencial do próprio domínio-problema. (...) [a] chave para controlar complexidades é um bom modelo de domínio (...), mas não é algo fácil de fazer (...). Este é um livro (...) sobre a arte da modelagem de domínios. (EVANS, 2016, p. xvii)

Segundo Evans (2016), a abordagem DDD se define como:

> (...) um *framework* para (...) decisões de projeto e um vocabulário técnico para discutir projeto de domínios. Uma síntese das melhores práticas aceitas e minhas próprias opiniões e experiências (EVANS, 2016, p. xix)

A abordagem DDD prescreve duas diretrizes gerais como meio de controle da complexidade no desenvolvimento de *software* (EVANS, 2016, p. xxi):

a) Projetar *software* focando no domínio (DDD);
b) Projetar domínios através de modelos (MDD);

A abordagem DDD se associa ao processo ágil através de sua fundamentação nos dois pré-requisitos seguintes (EVANS, 2016, p. xxii):

a) Desenvolvimento iterativo como "pedra fundamental" ágil;
b) Proximidade entre especialista (cliente) e desenvolvedor;

O conceito de modelo de domínio se constitui de dois conceitos chaves: domínio e modelo. O conceito de domínio se define como:

> Cada programa de *software* está relacionado a alguma atividade ou interesse do seu usuário. Essa área de assunto em que o usuário aplica o programa é o domínio do *software* (EVANS, 2016, p. 2).

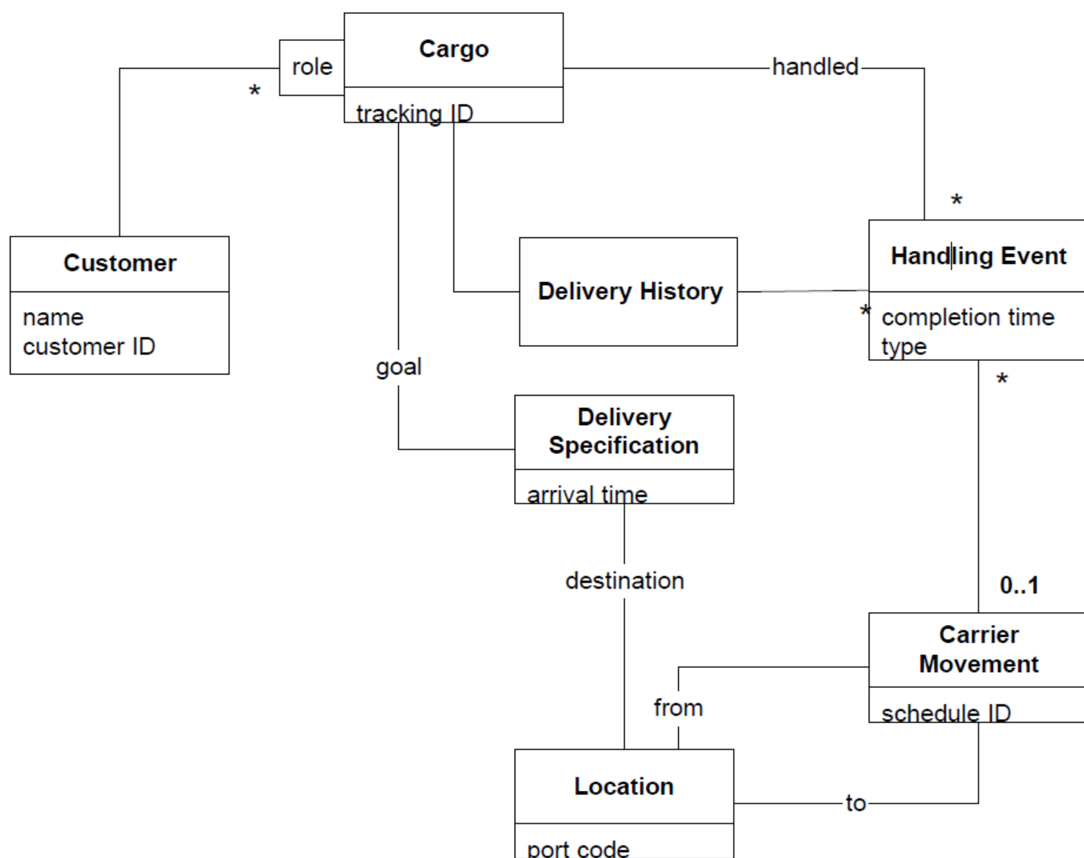
O conceito de modelo se define como:

> Mapas são modelos e cada modelo representa algum aspecto da realidade com uma ideia que seja de interesse. Um modelo é uma simplificação. Ele é uma interpretação da realidade que destaca os aspectos relevantes para resolver o problema (...), ignorando os detalhes (EVANS, 2016, p. 2).

Segundo Evans (2016), O modelo de domínio, apresentado na Figura 1, fornece as seguintes utilidades básicas em um projeto de *software* (EVANS, 2016, p.3):

a) Integração com o projeto;
b) Terminologia conceitual;
c) Conhecimento depurado.

Figura 1 - Modelo de Domínio.



Fonte: Evans, 2016.

Modelagem

As experiências a seguir são apresentadas como evidências de "exemplos vivos de como a prática de design de domínios pode afetar drasticamente os resultados de um desenvolvimento" (EVANS, 2016, p. xix).

> (...) [em um projeto que] faltava um modelo de domínio ou até uma linguagem comum no projeto (...). Um ano depois, a equipe se viu em

> apuros e impossibilitada de lançar uma segunda versão (EVANS, 2016, p. xx).

Em um segundo projeto é relatado a experiência exitosa:

> (...) em um domínio pelo menos tão complexo quanto o primeiro, (...) mas (...) seguido de sucessivas acelerações (...) [graças a] um modelo de domínio incisivo, repetidamente refinado e expresso em código (EVANS, 2016, p. xx).

A terceira experiência problematiza o alinhamento entre análise e projeto:

> (...) [mesmo] baseado em um modelo de domínio, (...) após anos de decepção, o projeto baixou suas expectativas (...). A equipe (...) deu especial atenção a modelagem. Mas (...) desassociou a modelagem da implementação (...) (EVANS, 2016, p. xx).

Foi evidenciado que o sucesso dependeu não só da modelagem do domínio, mas de seu alinhamento com o projeto (implementação). O conceito de Linguagem Ubíqua exerce esse papel através de uma terminologia rigorosa e presente na comunicação, documentação e codificação, iterativamente redigida entre desenvolvedores de software e especialistas do domínio durante a modelagem:

> Comprometidos em usar essa linguagem no contexto da implementação, os desenvolvedores poderão identificar imprecisões ou contradições (...), [enquanto especialistas podem] levantar questões (EVANS, 2016, p.24).

A Linguagem Ubíqua mitiga os custos de "tradução":

> O custo de toda a tradução, além do risco de entendimento errado, é simplesmente muito alto. Um projeto precisa de uma linguagem comum mais robusta que o mínimo denominador comum (EVANS, 2016, p.23).

O modelo de domínio sintetiza o modelo de análise e de projeto:

> O *Model-Driven Design* (projeto dirigido por modelo) descarta a dicotomia do modelo da análise e do projeto (design) e parte em busca de um modelo único que atende as duas finalidades (EVANS, 2016, p. 44).

A diagramação UML é problematizado nos modelos de domínio:

> Diagramas UML são muito bons em comunicar as relações entre objetos e são fiéis em mostrar as interações. Mas não transmitem as definições conceituais desses objetos. (...) Diagramas muito cheios de informação (..) falham ao comunicar ou explicar alguma coisa; eles sobrecarregam o leitor com detalhes e falta-lhes significado. (EVANS, 2016, p. 32).

A sobrecarga do detalhamento fica a cargo da documentação XP:

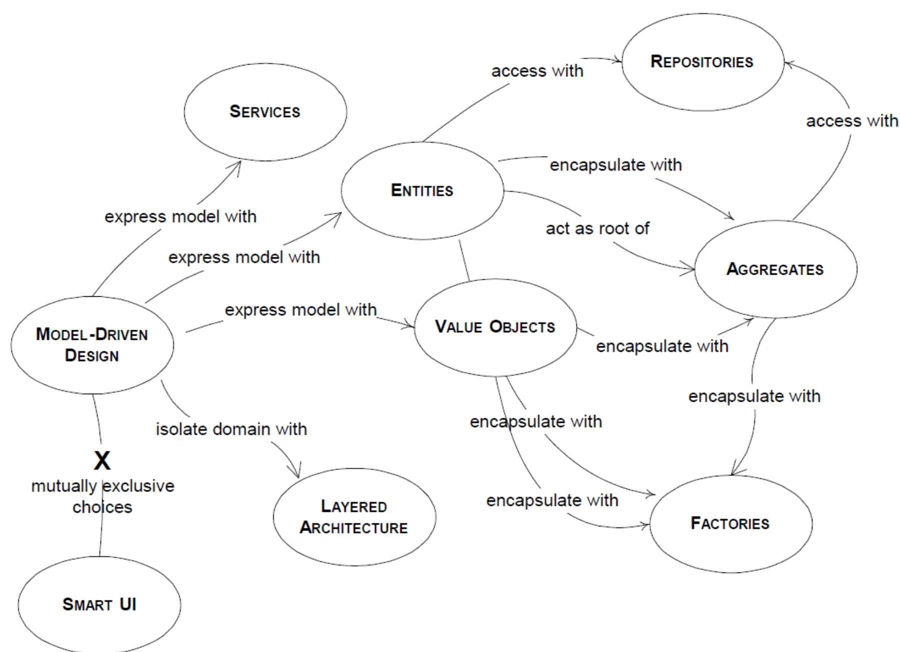
> *Extreme Programming* defende não usar (..) documento de projeto extra e deixar o código falar por si mesmo. (...) O código fornece os detalhes (...), outros documentos precisam esclarecer o significado (EVANS, 2016, p. 33).

Concluindo, um modelo de domínio é um artefato de documentação constituído de uma mescla do modelo de análise com o modelo de projeto em uma documentação de projeto abstrata que delega os detalhes ao código semântico (XP) e redigido através de Linguagem Ubíqua com moderado uso do UML.

Padronização

Os padrões de domínio, apresentado na Figura 2 abaixo, alinham o projeto de software com a abordagem DDD na forma de um projeto orientado ao modelo (Model-Driven Design).

Figura 2 - Padrões de Domínio.



Fonte: Evans, 2016.

O DDD segue o padrão arquitetural em camadas (BUSCHMANN, 2006. apud EVANS, 2016, p. 65) tendo como única restrição “a existência de uma camada específica (...), a camada de domínio”, dado que o “isolamento da implementação do domínio é um pré-requisito do Domain-Driven Design” (EVANS, 2016, p. 71).

O DDD sugere sete padrões de projeto classificados em dois grupos: padrões que expressam o modelo (entidade, objeto de valor, serviço e módulo) e padrões de ciclo de vida (agregado, fábrica e repositório), conforme Tabela 7 abaixo:

Tabela 7 - Padrões de Projeto de Domínio.

Padrões	Descrição
Entidade (*entity*)	Um objeto com atributo de identidade, cuja identidade atua como elemento de rastreio. Assim, entidades com os mesmos atributos podem ser diferentes.
Objeto de Valor (*value object*)	Um objeto imutável sem identidade, onde os objetos são identificados por seus atributos. Exemplo: cor, período, tempo, dinheiro, entre outros.
Serviço (*service*)	Um objeto apenas com comportamentos (métodos) que não encontra associação em nenhuma *entity* ou *value object* do modelo.
Módulo (*module*)	Agrupa conforme o domínio: assim, todas as classes relacionadas ao cliente ficariam no mesmo pacote (Java) ou *namespace* (C#).
Agregado (*aggregate*)	Uma relação de dois ou mais objetos que representam conceitualmente um único conceito no modelo, garantindo a integridade de suas “invariantes”.

Fábrica (*factory*)	Um objeto responsável por realizar montagens de objetos complexos, como *builders*, f*actory method*, *abstract factory*, entre outros.
Repositório (*repository*)	Encapsula a lógica de persistência, atuando no ciclo de vida com o papel de recuperação e evitando o acoplamento com a tecnologia de persistência usada.

Fontes: Evans, 2016, p. 77 / Evans, 2016, p. 117.

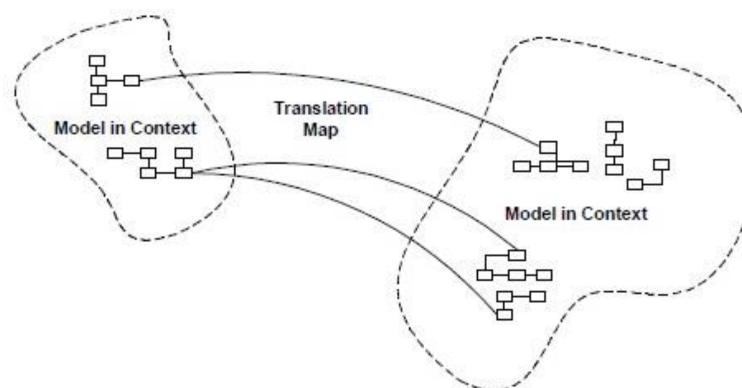
Estratégia

Como "a unificação total do modelo de domínio para um sistema muito grande não será factível ou econômica" (EVANS, 2016, p. 318), é sugerido um "projeto estratégico" com as seguintes três técnicas (EVANS, 2016, p. 314):

Contextualização

O modelo é dividido em Contextos Delimitados, conforme a Figura 3, se preservando com Integração Contínua e um Mapa de Contexto com seus relacionamentos, além de seguir padrões de cooperação, conforme a Tabela 8 abaixo, quando os contextos estiverem com times diferentes.

Figura 3 - Mapa de Contexto.



Fonte: Evans, 2016.

Tabela 8 - Padrões de Cooperação.

Padrões	Descrição
Núcleo compartilhado (*shared kernel*)	Uma biblioteca contendo as classes com o domínio principal sendo compartilhado entre os times.
Cliente-fornecedor (*customer-supplier*)	Um time assume o papel de fornecedor enquanto um time-cliente realiza demandas para evolução de uma biblioteca com as classes de domínio.
Conformista (*conformista*)	Quando um time-cliente, por alguma razão, não pode solicitar mudança ao time-fornecer, o time cliente deve se "conformar" com o que é fornecido.
Anti-Corrupção (*anti-corruption layer*)	É criada uma fachada que encapsula um código incompatível com o modelo, servindo como proteção para não corromper o modelo.
Caminhos separados (*separate ways*)	Quando o custo de qualquer grau de cooperação é muito alto, é sugerido que os times sigam cada um o seu próprio caminho.
Serviço de Host Aberto (*open-host service*)	É disponibilizado um serviço que passa a ser responsável por atuar como um tradutor geral, ao invés de delegar um tradutor para cada um.

Linguagem Publicada (*published language*)	Publicação online da documentação dos protocolos de consumo do Serviço de Host Aberto para quem for consumir o serviço.

Fonte: EVANS, 2016, p 338.

Destilação

A destilação no "modelo é uma destilação de conhecimento (...) [que extrai o] domínio principal" (EVANS, 2016, p. 379) com as seguintes estratégias, conforme Tabela 9 abaixo:

Tabela 9 - Estratégia de destilação.

Estratégias	Descrição
Domínio principal (*core domain*)	Contém os artefatos mais importantes do domínio, funcionando como um repositório de valor agregado ao projeto.
Subdomínios genéricos (*generic subdomains*)	Contém as partes genéricas do domínio que viabilizam aquisições prontas, como um sistema de pagamento ou geração de boleto.
Visão de domínio (*domain vision statement*)	Tal como uma Declaração de Visão gerencial explica o valor de um projeto, esse visa explicar o valor agregado pelo modelo de domínio.
Núcleo destacado (*highlighted core*)	Destaca (com marcações) os elementos essenciais contidos no documento de declaração de visão de domínio.
Mecanismos coesos (*cohesive mechanisms*)	São elementos funcionais coerentes entre si que formam uma estrutura a parte de caráter mais utilitário do que conceitual.
Núcleo segregado (*segregated core*)	Representa um resíduo temporário que mescla subdomínio genérico com domínio principal, subsistindo até sua separação definitiva.
Núcleo abstrato (*abstract core*)	Uma biblioteca desenvolvida quando se torna viável expressar o domínio através de interfaces polimórficas.

Fonte: EVANS, 2016, p 379.

Estruturação

Uma Estrutura de Larga-Escala é a opção para quando as técnicas de contextualização e destilação são insuficientes para o entendimento do modelo, sendo sugeridas as seguintes estratégias (EVANS, 2016, p. 417), apresentadas na Tabela 10:

Tabela 10 - Estratégias de estruturação.

Estratégias	Descrição
Ordem de evolução (*evolving order*)	Atrasa as restrições do projeto para definições posteriores, evolutivamente, e com isso não limitar as opções do projeto.
Metáfora de sistema (*system methaphor*)	Utiliza analogias úteis como conceitos transitórios no modelo para atuarem até surgir o conceito adequado que o substitua.
Camadas de responsabilidades (*responsability layers*)	Define linhas divisórias no modelo demarcando responsabilidades de modo a permitir um melhor entendimento do modelo.
Nível de conhecimento (*knowledge level*)	Uma notação como uma camada de abstração responsável por explicar elementos do modelo no modelo (meta-modelo).
Componentes plugáveis (*pluggable component*)	Utiliza um núcleo abstrato para viabilizar diversas implementações possíveis atuando como componentes plugáveis.

Fonte: EVANS, 2016, p 417.

METODOLOGIA

Nesse trabalho foi aplicada a metodologia de "Revisão de Literatura", através de pesquisa por meios de artigos científicos, livros-textos, conferências, entre outros. Segundo Rowley e Slack (2014), as revisões de literatura:

> (...) facilitam a obtenção de fontes de informação e contribuem para o entendimento de conceitos, análise e interpretação de resultados relacionados a um determinado assunto. (ROWLEY et al., 2004. apud JUNIOR, 2014, p. 3).

Foi utilizada como fonte principal, enquanto pesquisa e revisão da abordagem do Domain-Driven Design, a obra homônima de Evans (2016), em razão da mesma ser tratada como obra de referência sobre o assunto.

> Acredito e espero, que este seja um livro de grande influência. Um livro que acrescente estrutura e coesão a um campo bastante incerto ensinando ao mesmo tempo muitas pessoas a usarem uma valiosa ferramenta. Os modelos de domínio podem trazer grandes consequências para o controle do desenvolvimento de *software* (FOWLER apud EVANS, 2016, p. xviii).

As citações sobre o Extreme Programming (XP) também priorizaram Evans (2016) como fonte, com o propósito de se ater a forma como a abordagem DDD interpreta e incorpora o modelo de documentação do XP.

LEVANTAMENTO E ANÁLISE DOS DADOS

O termo "documentação-modelo" faz referência a documentação de modelo de domínio na abordagem Domain-Driven Design. O modelo de domínio, enquanto projeto dirigido por um modelo (MDD), abrange tanto modelo de análise (negócio) quanto modelo de projeto (técnico).

> O *Model-Driven Design* (projeto dirigido por modelo) descarta a dicotomia do modelo da análise e do projeto (design) e parte em busca de um modelo único que atende as duas finalidades (...). Isso requer que sejamos mais exigentes com relação ao modelo escolhido, pois ele deve realizar dois objetivos bastante diferentes (EVANS, 2016, p. 44).

Esse modelo análise-projeto é uma visão de negócio que é balanceada com uma visão técnica do projeto de software, através do processo iterativo entre especialistas de domínio e desenvolvedores de software:

> Sempre existem várias maneiras de abstrair um domínio e sempre existem várias "formas de projetar um software" que podem resolver um problema do aplicativo. Isso é o que torna prático ligar o modelo e o design. Esta ligação não deve acontecer à custa de uma análise enfraquecida, fatalmente comprometida por considerações técnicas. Nem podemos aceitar projetos malfeitos, refletindo ideias do domínio, mas fugindo aos princípios de projeto de um software. Essa abordagem exige um modelo que funcione bem tanto como análise quanto como projeto (EVANS, 2016, p. 45).

O modelo de domínio, não se limita ao UML, dado que "não é um diagrama específico; ele é a ideia que o diagrama pretende transmitir" (EVANS, 2016, p. 3):

> O problema aparece quando (...)[se tenta] transmitir o modelo ou o *design* inteiro através do UML (...). Com todo esse detalhamento, ninguém consegue ver o que realmente é importante (EVANS, 2016, p. 33).

O UML deve ser usado apenas na medida da necessidade do modelo:

> Diagramas UML são muito bons em comunicar as relações entre os objetos e são fiéis em mostrar as interações. Mas não transmitem as definições conceituais desses objetos (...). Em outras palavras, um diagrama UML não pode transmitir dois dos aspectos mais importantes de um modelo: o significado dos conceitos que ele representa e o que os objetos devem fazer. No entanto, não precisamos nos preocupar com isso, pois o uso cuidadoso da 'linguagem' pode cumprir esse papel". (EVANS, 2016, p. 32).

Enquanto o UML é responsável pelos relacionamentos, o "uso cuidadoso da linguagem" (Linguagem Ubíqua) fornece os conceitos (seção 2.3.1).

Em síntese, a documentação-modelo une os modelos de análise e projeto, utilizando um UML moderado e uma elaborada linguagem ubíqua.

Documentação-código

O termo "documentação-código" faz referência a forma como a abordagem DDD incorpora o modelo de documentação do XP (seção 2.3).

Segundo Shukla (2002), a documentação XP é o código-fonte:

> O XP enfatiza que os programadores não devem tentar prever necessidades futuras e projetar adequadamente. Outro aspecto do *Simple Design* é "O código-fonte é o projeto" (SHUKLA et al., 2002, tradução).

Segundo Evans (2016), o Extreme Programming (XP) se define:

> *Extreme Programming* defende não usar absolutamente nenhum documento (...) e deixar o código falar por si mesmo. Um código que funciona não mente, como poderia acontecer com qualquer outro documento. O comportamento de um código que funciona não gera dúvidas. (EVANS, 2016, p. 33)

Enquanto um código que se explica através de si mesmo, a documentação XP faz com que "essa dependência do código como meio de comunicação motive os desenvolvedores a manter o código limpo e transparente" (EVANS, 2016).

Para Evans (2003, p. 35), esse modelo de documentação XP encontra as seguintes limitações ao se deparar com domínios mais complexos:

a) Sobrecarrega a leitura com detalhes de implementação, caindo no mesmo erro das UML excessivamente detalhadas;
b) Apesar do comportamento do código não gerar dúvida nessa forma de documentação, o sentido conceitual não é transparente;
c) A documentação XP só é acessível aos desenvolvedores, mas não são apenas desenvolvedores que precisam acessar a documentação.

A documentação-modelo e a documentação-código se complementam no DDD em diferentes níveis de abstração, com uma documentação-modelo mais alto-nível (conceitual), acessível a especialistas e desenvolvedores; e uma documentação-código mais baixo-nível (comportamental), servindo aos desenvolvedores:

> Um documento não deve tentar fazer o que o código já faz bem. O código já fornece os detalhes. Ele é uma especificação exata do comportamento do programa. Outros documentos precisam esclarecer o significado, fazer entender as estruturas em larga escala e concentrar a atenção nos elementos principais (...). Documentos escritos devem complementar o código e a conversa (EVANS, 2016, p. 35).

Documentação-auxiliar

Na maioria dos casos, em modelos com dimensões moderadas, a documentação-modelo e a documentação-código são suficientes para atender as necessidades de documentação de projeto na abordagem DDD.

> À medida que os sistemas se tornam complexos demais para que possamos conhecê-los por completo em termos de cada objeto, precisamos de técnicas para manipular e compreender modelos maiores (EVANS, 2016, p. 314).

Nesse grau de complexidade, o DDD sugere um "projeto estratégico" com as técnicas de contextualização, destilação e estruturação (seção 2.3), podendo incorrer nesses casos, eventualmente, alguns artefatos extras de documentação, conforme Tabela 11:

Tabela 11 - Documentação auxiliar

Documento	Descrição
Mapa de contexto	Mapas de contexto podem ser uma notação estendida sobre o próprio modelo de domínio ou um documento a parte, servindo, como sugere o nome, como um mapeamento dos diferentes contextos através da técnica de contextualização.
Linguagem publicada	É um padrão de cooperação para a estratégia de contextualização, sendo uma documentação contendo o protocolo de consumo do serviço para padrão de cooperação denominado "Serviço de Hospedagem Aberta".
Declaração de visão de domínio	É uma estratégia de destilação que funciona como um documento equivalente a uma "declaração de visão gerencial", a visão de domínio sumariza e explica o valor agregado pela documentação de modelo de domínio.

Fonte: Elaborada pelo autor

O conceito de "documentação auxiliar" faz referência a documentação que complementa o modelo de domínio (secção 2.3.3), permitindo aprimorar semanticamente a compreensão sobre o modelo em um "projeto estratégico".

O Problema

Essa seção abordou pesquisas que trataram o problema da documentação em projetos que seguem metodologias ágeis, destacando a relação dos problemas identificados com os preceitos do Manifesto Ágil.

Documentação x implementação (valor)

Em uma pesquisa sobre a documentação no desenvolvimento ágil, alguns "agilistas" se mostram céticos a respeito da documentação:

> Alguns agilistas até veem a documentação como uma perda de tempo, uma vez que não contribui para o produto final (VOIGT et al., 2016, tradução)

Enquanto perda de tempo ou valor não agregado, esse ceticismo se refere ao primeiro princípio ágil que prioriza "(...) o cliente através da entrega contínua e adiantada de software com valor agregado" (MANIFESTO, 2019).

Nesse contexto e ao valorizar "software em funcionamento, mais que documentação abrangente" (MANIFESTO, 2019), as atividades de implementação e documentação entram em contradição no processo ágil:

> Em projetos de *software* ágeis, engenheiros de *software* priorizam implementação sobre documentação como forma de eliminar documentação desnecessária. (SAITO et al., 2017, p. 194, tradução)

Uma pesquisa relata negligência com a documentação:

> A documentação geralmente é negligenciada em projetos de *software* ágeis, mesmo que os desenvolvedores de *software* percebam a necessidade de uma boa documentação. (...) A identificação da falta de tempo é o argumento principal contra a documentação (VOIGT et al., 2016, tradução).

A contradição é associada a implicações do segundo valor ágil:

> Os métodos ágeis de desenvolvimento de *software* desempenham um papel importante na pesquisa e na prática. Os "tradicionalistas" e "agilistas" debatem a correta interpretação de valores e princípios no Manifesto Ágil. O segundo valor "*Software* funcionando acima de documentação abrangente" é frequentemente mal compreendido (VOIGT et al., 2016, tradução).

Documentação x conversação (iterativo)

A mesma tendência à documentação negligente foi evidenciada em outra pesquisa, mas identificando a sua causa no sexto princípio ágil:

> (...) escrever documentação era percebido como uma tarefa intrusiva, levando à especialização de tarefas e à alocação de documentação para membros menos qualificados da equipe. Consequentemente, isso dificultou a colaboração dentro da equipe (STETTINA et al., 2012, p. 31, tradução).

Segue a citação indireta ao sexto princípio ágil da pesquisa acima:

> Quando comparado ao desenvolvimento de *software* tradicional dirigido por documento, as práticas de desenvolvimento de *software* ágil advogam uma ênfase na comunicação direta. Este foco está embutido em práticas recorrentes como o desenvolvimento iterativo, envolvimento frequente com o cliente, reuniões diárias ou estimativas de esforço baseadas no time (STETTINA et al., 2012, p. 31, tradução).

A pesquisa citada ressaltou os riscos no projeto associados a essa baixa documentação em cenários de rotatividade do time de desenvolvedores:

> Enquanto que as práticas de compartilhamento do conhecimento baseadas em socialização dos membros do time habilitam agilidade e um entendimento comum em um nível de time, eles também podem resultar em problemas tais como brechas de conhecimento não documentado ou perda de integridade arquitetural entre projetos. Isto é particularmente problemático quando pessoas deixam o time ou a organização. Nesse sentido, o desenvolvimento ágil pode fazer a gerência de projeto e

> gerenciamento em nível de programa mais difíceis (STETTINA et al., 2012, p. 31, tradução).

Concluindo, o Manifesto Ágil problematiza a documentação de projeto em favor da conversação, dado sua comunicação indireta e processo em cascata.

Documentação x atualização (manutenção)

A eliminação do problema relacionado a manter uma documentação atualizada é uma das motivações principais do modelo de documentação de projeto na metodologia ágil XP (que é parte da abordagem documental do DDD):

> O *Extreme Programming* se concentra exclusivamente nos elementos ativos de um programa e nos testes executáveis. Até mesmo comentários adicionados ao código não afeta o comportamento do programa e, por isso, eles sempre saem de sincronia com o código ativo e o seu modelo condutor. Documentos externos e diagramas não afetam o comportamento do programa e, por isso, eles saem de sincronia (EVANS, 2016, p. 35).

Esse problema revela uma característica de qualquer documentação convencional, a necessidade de mantê-la atualizada e o risco de desatualização. Os custos para garantir a manutenção de todos os artefatos de documentação atualizados podem inviabilizar o projeto ou a própria documentação.

A documentação-código evita o problema da manutenção da documentação por não ser propriamente uma documentação no sentido convencional, dado que não produz artefatos de documentação, mas apenas define orientações visando uma codificação mais autoexplicativa, transparente e semântica.

Contudo, enquanto artefato de documentação convencional, a documentação-modelo e a documentação-auxiliar estão expostas ao problema da manutenção da documentação com seus custos e riscos envolvidos.

SOLUÇÕES PROPOSTAS

Conforme já adiantado (seção 2.3), a abordagem DDD segue o processo ágil na forma de dois pré-requisitos (EVANS, 2016, p. xxii):

> 1. O desenvolvimento é iterativo. O desenvolvimento iterativo tem sido defendido e praticado há décadas, e é a pedra fundamental dos métodos de desenvolvimento *Agile*. Existem várias boas discussões nos textos referente a desenvolvimento *Agile* e *Extreme Programming* (ou XP), entre elas, *Surviving Object-Oriented Projects* (Cockburn, 1998) e *Extreme Programming Explaned* (Beck, 1999).
>
> 2. Desenvolvedor e especialistas em domínio têm uma relação íntima. O DDD compacta uma grande quantidade de conhecimento em um modelo que reflete uma visão profunda do domínio e um enfoque nos conceitos principais. É uma colaboração entre quem conhece o domínio e quem sabe como construir softwares (EVANS, 2016).

Os dois pré-requisitos citados acima formalizam a fundamentação da abordagem DDD enquanto processo ágil. Contudo, o alinhamento aos preceitos ágeis vai além desses dois pré-requisitos, como evidenciado nas subseções a seguir.

Domínio como valor

Na fundamentação foram evidenciados três critérios para o conceito de documentação ágil: útil, leve e mínima (seção 2.2.3).

O valor agregado imediato de uma documentação é sua utilidade. Contudo, quando um modelo de documentação cede mais tempo à implementação, a documentação também agrega valor, indiretamente, liberando mais tempo para o processo de geração de valor através da implementação.

Os critérios ágeis de documentação leve e documentação mínima agregam esse valor indireto: com um número menor de artefatos para manter e tendo menos o que redigir neles, mais tempo é liberado para a implementação.

a) Utilidade

A documentação-modelo se qualifica como útil, enquanto “conhecimento depurado” (seção 2.3). Essa característica de um modelo de domínio fornece informações refinadas, amadurecidas e discutidas através de várias iterações entre os especialistas e os desenvolvedores do software, economizando retrabalho de análise e projeto desse conhecimento já absorvido pelo modelo.

b) Leveza e minimalismo

A documentação-domínio se qualifica como documentação leve, ao abstrair os detalhes e se focar no essencial, delegando o detalhamento à documentação-código e mantendo o uso moderado do UML.

A documentação-domínio se qualifica como documentação mínima, ao reduzir em um único artefato de documentação, não só os modelos de análise e projeto, assim como os outros modelos da documentação tradicional.

A documentação-código, como já adiantado (seção 3.2.3), não produz novos artefatos, dessa forma, não se aplica a essa categorização.

A documentação-auxiliar se qualifica como documentação mínima pela sua raridade de uso com poucos artefatos (seção 3.1.3). E também se qualifica como documentação leve, por se ater a documentar apenas a informações adicionais específicas que complementem a documentação-modelo.

Assim, o problema da contradição entre documentação e implementação (seção 3.2.1) se resolve em uma documentação mais abrangente, mas ainda ágil, dado sua utilidade, leveza e minimalismo. Com isso, permitindo custo baixo de manutenção e benefício considerável por sua utilidade no desenvolvimento.

Modelagem incremental

A modelagem é tradicionalmente associada a um processo em cascata, onde se cria um documento que em seguida será implementado daquela forma.

> (...) modelos de domínio não são modelados primeiros e, depois, implementados. (...) modelos de domínio verdadeiramente bons evoluem com o tempo (FOWLER apud EVANS, 2016, p. xviii).

O processo de documentação modelado, através de comunicação direta entre especialistas de domínio e os desenvolvedores de software, fornece uma documentação que elimina a contradição com o sexto princípio ágil (seção 3.2.3):

> O constante refinamento do modelo de domínio força os desenvolvedores a aprender os princípios importantes do negócio (...), em vez de gerar funções mecanicamente. Os especialistas do domínio geralmente refinam seu próprio entendimento (..) e passam a entender o rigor conceitual exigido pelos projetos de software (EVANS, 2016, p. 14).

Ao orientar o projeto ao modelo, mas ao mesmo tempo modelando essa documentação em um processo iterativo e com comunicação direta, a abordagem DDD converge documentação e conversação em um mesmo processo orgânico.

> À medida que o projeto avança, ele se torna uma ferramenta para organizar as informações que continuam a fluir pelo projeto. O modelo se concentra na análise dos requisitos, interagindo intimamente com a programação e o design (EVANS, 2016, p. 14).

O problema do processo documental em cascata é eliminado pelo processo iterativo da modelagem na abordagem DDD. Enquanto comunicação direta nesse processo, a contradição com o sexto princípio ágil é superada.

Documentação atualizada

A documentação-modelo, enquanto artefato de documentação, está exposta aos mesmos problemas de manutenção de uma documentação tradicional.

Contudo, a documentação-modelo consegue mitigar seus custos de manutenção através de sua alta abstração, que mitiga os impactos das alterações de detalhes concretos. Assim, com menor risco de mudança, menor o risco de desatualização, e, por conseguinte, menor o custo de manutenção.

Além da alta abstração que mitiga os impactos de mudanças, o processo de modelagem iterativa provoca o hábito não só de contínua atualização do modelo, mas de contato frequente com o modelo no processo de desenvolvimento.

Unindo menos propensão a mudanças com maior integração com o desenvolvimento, a documentação da abordagem DDD viabiliza um processo que permite manter a documentação atualizada com menor risco e baixo custo.

A Tabela 12 a seguir, resume os problemas relatados e as soluções analisadas identificadas durante a seção da discussão (seção 3.3).

Tabela 12 - Problemas x soluções.

Problema de documentação	Solução da documentação DDD
Não agrega valor ao projeto	Documento é útil por registrar o conhecimento depurado em diversas iterações entre desenvolvedores e especialistas.
Interrompe implementação	Ao mesclar diversos modelos em um e ao se concentrar no que é essencial, ele consome menos tempo da implementação.
Processo em cascata	A modelagem de domínio segue um modelo de processo iterativo-incremental entre especialistas e desenvolvedores.
É comunicação indireta	A modelagem iterativa, na abordagem DDD, ocorre com comunicação direta entre especialistas e desenvolvedores.
Não é ágil	Além de se fundamentar em dois pré-requisitos ágeis (seção 3.1.1), segue os critérios de uma documentação ágil (seção 2.2.3).

Fonte: Elaborada pelo autor.

CONCLUSÕES

A fundamentação teórica realizou uma revisão literária do conceito de documentação, metodologia ágil e modelo de domínio, como necessário embasamento teórico ao entendimento da documentação ágil de projeto software e a documentação de modelo de domínio na abordagem Domain-Driven Design (DDD).

O modelo de domínio foi apresentado como a documentação de referência na abordagem DDD, conceitualmente de alto nível (abstrata), mesclando modelo de análise e modelo de projeto e sendo identificada como “documentação-modelo”.

A abordagem DDD se refere também a uma documentação-código de baixo-nível (seguindo a documentação XP) e, eventualmente, uma documentação-auxiliar, ocorrendo apenas em situações bem específicas.

O problema-alvo da pesquisa foi apresentado através das pesquisas que abordaram a documentação em projetos ágeis, identificando falhas de documentação motivadas por um ou mais dos seguintes motivos:

a) Não agrega valor ao desenvolvimento;
b) É desperdício por tomar tempo da implementação;
c) Não é ágil por ser uma comunicação indireta;
d) Segue um modelo em cascata tradicional;
e) Alto custo e risco em se manter atualizado.

Na discussão foi tratada a solução documental da abordagem DDD, como contraposição aos problemas relatados, com os seguintes argumentos:

a) Agrega valor ao servir como repositório de valor;
b) Não se desperdiça em detalhes (abstrato);
c) É ágil por se basear em comunicação cara-a-cara;
d) Segue o modelo iterativo-incremental ágil;
e) Se mantém atualizado por se aproximar do desenvolvimento.

Foi observado que a documentação é um tema ainda problemático no desenvolvimento ágil, com muitas dúvidas e desafios, havendo muita discrepância entre as metodologias ágeis e em como tratar o problema da documentação.

Também foi observado como a abordagem DDD fornece uma prática de documentação alinhada com o Manifesto Ágil e especialmente relacionada com a metodologia ágil XP, enquanto um de seus três artefatos de documentação.

Um dos achados relevantes desta pesquisa foi relatado na solução, onde foi apresentado como a documentação da abordagem DDD forneceu soluções adequadas aos problemas citados sem se alienar de seus preceitos ágeis.

Assim, ficou evidenciado que a abordagem DDD, além de se adequar as práticas ágeis que hoje dominam a indústria de *software* e atender ao seu problema alvo: controlar a complexidade no *software*; também, colateralmente, possibilita resolver alguns problemas identificados na documentação em projetos ágeis.

Entretanto, a abordagem DDD, apesar de seguir um processo leve, seu completo domínio exige uma considerável curva de aprendizado, ainda que, como premissa, a abordagem DDD não exija muito:

> Nenhum projeto jamais empregará todas as técnicas discutidas neste livro. Mesmo assim, qualquer projeto comprometido com um *design* dirigido por domínio será reconhecível através de alguns aspectos. Sua característica definitiva é a prioridade pelo entendimento do domínio-alvo e a incorporação desse entendimento no *software*. Todo o resto parte dessa premissa (EVANS, 2016, p. 478).

Esse misto de soluções ágeis para vários problemas torna a abordagem *Domain-Driven Design* uma opção de bom custo-benefício para atender *softwares* que em seu ciclo de vida possam se enveredar por domínios de maior complexidade.

REFERÊNCIAS BIBLIOGRÁFICAS

BARKER, T. T. (1990). Software documentation: from instruction to integration. IEEE Transactions on Professional Communication, 33(4), 172–177. doi:10.1109/47.62811

BOURQUE, Pierre. Fairley, Richard. Guide to the Software Engineering Body of Knowledge, Version 3.0. 2014. IEEE Computer Society. AGILE. (2013). ITNOW, 55(2), 6–8. doi:10.1093/itnow/bwt002

BRIAND, L. C. (n.d.). Software documentation: how much is enough? Seventh European Conference onSoftware Maintenance and Reengineering, 2003. Proceedings. doi:10.1109/csmr.2003.1192406

EVANS, ERIC. Domain Driven Design: Atacando as Complexidades no Coração do Software. Traduzido por Tibério Júlio Couto Novais. 3 Edição revisada – Rio de Janeiro. Alta Books, 2016. 538 p.

FOWLER, Martin. The New Methodology. 2005. Disponível em: <http://www.martinfowler.com/articles/newMethodology.html>. Acessado em: 31 out. 2019

JUNIOR, Mauro Tomaz Silva. Queiroz, Fernanda Cristina Barbosa Pereira. Queiroz, Jamerson Viegas (2014). ISO 9001 – Uma Revisão Da Literatura Sobre Seus Benefícios, Motivações e Dificuldades. Congresso Nacional de Excelência em Gestão. ISSN 1984-9354

KIPYEGEN, Jemutai. KORIR, William P. K. Importance of Software Documentation. IJCSI International Journal of Computer Science Issues, Vol. 10, Issue 5, No 1, September 2013. ISSN (Print): 1694-0814 | ISSN (Online): 1694-0784

LE, D. M., Dang, D.-H., & Nguyen, V.-H. (2016). Domain-driven design patterns: A metadata-based approach. 2016 IEEE RIVF International Conference on Computing & Communication Technologies, Research, Innovation, and Vision for the Future (RIVF). doi:10.1109/rivf.2016.7800302

MANIFESTO, Ágil. Manifesto for Agile Software Development. Disponível em: < https://agilemanifesto.org>. Acessado em: 28/10/2019

MCGUFFEE, J. W. (2000). Defining computer science. ACM SIGCSE Bulletin, 32(2), 74–76. doi:10.1145/355354.355379

PRIESTLEY, M., & Utt, M. H. (n.d.). A unified process for software and documentation development. 18th Annual Conference on Computer Documentation. Ipcc Sigdoc 2000. Technology and Teamwork. Proceedings. IEEE Professional Communication Society International Professional Communication Conference and ACM Special Interest Group on Documentation Conference. doi:10.1109/ipcc.2000.887279

QURESHI, M. R. J. (2012). Agile software development methodology for medium and large projects. IET Software, 6(4), 358. doi:10.1049/iet-sen.2011.0110

SAITO, S., Iimura, Y., Massey, A. K., & Anton, A. I. (2017). How Much Undocumented Knowledge is there in Agile Software Development?: Case Study on Industrial Project Using Issue Tracking System and Version Control System. 2017 IEEE 25th International Requirements Engineering Conference (RE). doi:10.1109/re.2017.33

SHAFIQ, M., & Waheed, U. sman. (2018). Documentation in Agile Development. A Comparative Analysis. 2018 IEEE 21st International Multi-Topic Conference (INMIC). doi:10.1109/inmic.2018.8595625

SHUKLA, A., & Williams, L. (n.d.). Adapting extreme programming for a core software engineering course. Proceedings 15th Conference on Software Engineering Education and Training (CSEE&T 2002). doi:10.1109/csee.2002.995210

STETTINA, C. J., Heijstek, W., & Faegri, T. E. (2012). Documentation Work in Agile Teams: The Role of Documentation Formalism in Achieving a Sustainable Practice. 2012 Agile Conference. doi:10.1109/agile.2012.7