Resumo do livro "Domain-Driven Design"

Leonardo Leite

8 de agosto de 2016

Atenção: este documento trata-se ainda de uma versão em desenvolvimento.

1 Introdução

Este documento é um resumo do livro "Domain-Driven Design, Atacando as Complexidades no Coração do Software", escrito por Eric Evans em 2003. Os objetivos deste resumo são:

- Em primeiro lugar, fazer um exercício de fixação e assimilação de minha leitura do livro.
- Servir de referência rápida para que eu tente aplicar os conceitos do DDD no desenvolvimento de software.
- Fornecer a outras pessoas uma ideia rápida sobre o que trata o livro Domain-Driven Design, assim como apresentar alguns de seus principais conceitos.

Uma consideração importante ao leitor: embora minha prática do desenvolvimento de software já tenha vários pontos de acordo com o DDD, este resumo foi escrito logo após minha leitura do livro. Ou seja, não se trata de um texto baseado em minha experiência com DDD, mas somente na leitura do livro.

Destaco também que o livro que eu li foi a primeira edição da tradução para o português (2009). Aliás, uma tradução não tão boa. O pior exemplo de tradução que achei foi a tradução de *graphs* (grafos) e *edges* (arestas) respectivamente por gráficos e bordas. Não sei dizer se a tradução em si foi melhorada na segunda edição da tradução (2011).

Minha impressão geral sobre o livro: gostei bastante da mentalidade do autor, bem condizente com princípios da Programação Extrema (XP) e ao mesmo tempo com uma boa dose de pragmatismo. No entanto o livro é um tanto quanto grande e verboso. Embora seja definitivamente um clássico, já é relativamente antigo (2003). Dessa forma, acho que talvez possam existir por aí livros mais resumidos que possam transmitir a essência do DDD sem tantas considerações e mais direto ao ponto considerando o atual estado-da-arte do desenvolvimento de software. Não que o livro tenha deixado de ser atual, mas algumas considerações menores poderiam hoje ser deixadas de lado. Ao mesmo tempo, já surgiram alguns paradigmas modernosos como *micro-serviços*, que são bem relacionados com o tema do livro. Portanto, sua decisão de ler ou não o livro vai depender bastante do quanto você deseja se aprofundar no assunto.

Outra consideração bem interessante é, pra mim, foi bem positivo ter demorado pra ler esse livro. Já tinha ouvido falar do livro há alguns anos, mas somente agora finalmente

o priorizei. E isso foi bom porque a experiência recente que eu tive desenvolvendo um sistema com um complicado modelo de domínio me ajudou bastante a ler o livro com melhor proveito. Outros sistemas em que trabalhara antes não tinham um modelo de domínio assim tão complexo. Ler o livro e comparar o que o autor descreve com suas próprias experiências é de grande valor para uma melhor leitura do livro. Dessa forma, dou a ousada sugestão de que se você ainda é um desenvolvedor iniciante leia apenas um livro básico (ou mesmo esse resumo) sobre o assunto e espere alguns anos até ler o livro Domain-Driven Design.

"A principal finalidade de um software é servir aos usuários. Mas, primeiro, esse mesmo software tem que servir aos desenvolvedores".

Antes de chegar ao que interessa, mais uma consideração. A abordagem de DDD, tal como apresentada no livro, é intimamente ligada à orientação a objetos (OO). Caso você ainda não domine a OO em profundamente, não se preocupe, estudar DDD é até uma boa maneira de reforçar o aprendizado em OO. Mas um alerta se faz necessário. Muitas pessoas ainda vendem a ideia de que o foco principal da OO consiste no mapeamento de entidades do mundo real para o código e a utilização de encapsulamento, herança e polimorfismo. Embora todas essas coisas estejam inclusas na OO, é mais útil considerar que o foco do OO é a decomposição do código de um sistema em unidades coesas e fracamente acopladas. Essas unidade são tanto as pequena unidades (classes), quanto as grandes unidades (pacotes). Uma consequência almejada dessa decomposição é que a manutenção do sistema fica facilitada, pois é possível encontrar rapidamente o ponto a ser alterado e a mesma alteração não precisa ser repetida em diversos pontos do código. Um ótimo livro que ajuda a passar essa ideia sobre a OO é o "Orientação a Objetos e SOLID para Ninjas" [Aniche, 2015].

Obs: *"trechos nesta formatação são citações diretas do livro".*

Bom, chega de enrolação. Vamos ao que interessa...

2 O modelo do domínio de um software

O *domínio* de um software é o assunto relacionado às atividades desempenhadas pelos usuários desse software. Exemplos de domínios são compra de passagens aéreas, gestão de pessoas, tradução de texto para outras línguas, o processo legislativo, segurança veicular, previsão do tempo etc.

Para desenvolvermos um software que auxilie as atividades humanas em um dado domínio, precisamos de um *modelo* desse domínio que seja útil à criação desse software. *"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 que se tem em mãos ignorando os detalhes estranhos".* Ao ignorar seletivamente esses detalhes, modelos nos ajudam a tratar a sobrecarga de volume e complexidade das informações relacionadas ao domínio.

Diagramas não são o modelo. O modelo é uma abstração que se reflete em diversos artefatos concretos, como requisitos, diagramas, código-fonte e estrutura do banco de dados. Todos esses artefatos devem estar alinhados com o modelo do domínio. Esse alinhamento diz respeito até mesmo às palavras utilizadas na comunicação entre os membros do time (incluindo desenvolvedores, analistas de negócio e cliente).

O modelo do domínio não pode ser facilmente definido antes da construção do software. O processo de desenvolvimento traz um grande aprendizado para a equipe sobre o domínio. Esse aprendizado deve se refletir em evoluções incrementais do modelo ao longo

do desenvolvimento. E enquanto o modelo evolui, os demais artefatos (diagramas, código etc.) também devem evoluir para acompanhar a evolução do modelo.

3 A linguagem onipresente

A situação típica é de que o time de desenvolvimento não esteja habituada ao domínio do software a ser construído. Especialistas do domínio possuem jargões que os desenvolvedores não conhecem. Assim surgem complicações na comunicação entre desenvolvedores e especialistas do domínio. Até mesmo entre os próprios desenvolvedores pode haver confusão, pois cada um pode criar um entendimento diferente sobre o domínio em sua cabeça.

Para fortalecer a comunicação dos envolvidos em um projeto de software e garantir a precisão do modelo, Evans defende extensivamente ao longo do livro a utilização de uma *linguagem onipresente*. Essa linguagem onipresente deve ser utilizada na comunicação oral e escrita do time, além de ser utilizada nos diagramas, código-fonte etc. Assim, a linguagem onipresente faz a ligação entre a comunicação do time e a implementação do software.

Lacunas na linguagem onipresente evidenciam a falta de um conceito no modelo. Palavras estranhas podem evidenciar imprecisões do modelo ou divergências entre a implementação e os objetivos do negócio. Mudanças na linguagem onipresente são mudanças no modelo. Assim ela deve ser mantida e zelada por todo o time. Especialistas do domínio devem impedir a existência de termos inapropriados para o domínio, enquanto que desenvolvedores devem impedir ambiguidades e inconsistências.

Na comunicação diária, tanto especialistas quando desenvolvedores ainda utilizarão termos fora da linguagem onipresente. Esses termos podem ser, por exemplo, termos técnicos ligados à implementação do software, ou conceitos mais avançados do negócio, não necessários à implementação do software. Tais termos devem ser complementares à linguagem onipresente, e não conflitantes com a linguagem onipresente.

Embora seja rápido e fácil falar sobre a linguagem onipresente, ela é uma ideia central ao longo do livro. Em praticamente todos os capítulos, o autor aproveita para reforçar o assunto, aplicando-o nos mais diferentes contextos, seja na produção de código, seja na conversa com os especialistas.

4 Camadas de um software

O primeiro passo para um design dirigido pelo domínio é o isolamento da lógica do negócio de outras características que o software deve implementar. Isso é feito com a separação do software em camadas. Assim temos uma camada do domínio que fica desacoplada do restante do sistema. Isso facilita a manutenção, pois podemos criar designs mais coesivos e identificar rapidamente os pontos de mudança no software. Além disso, o isolamento das regras de negócio potencializa o reuso de código, pois a mesma regra de negócio pode ser reaproveitada em um aplicativo móvel ou por um web service, por exemplo.

As camadas podem ser classificadas em interface, aplicativo, domínio e infra.

Interface: interage com o usuário, coletando entradas e exibindo saídas. Esse usuário pode ser tanto uma pessoa quanto outro sistema.

Aplicativo: implementa as funcionalidades disponíveis para o usuário. Essa camada tende a ser bem magra e apenas dirige a camada de domínio para que a funcionalidade seja entregue.

Domínio: é o coração do sistema, onde estão os conceitos, regras e estado do negócio. É a parte do sistema onde menos os *frameworks* e plataformas podem ajudar. É onde mais a criatividade e as capacidades de análise e design se fazem necessárias. E é nessa camada que o livro Domain-Driven Design é focado.

Infra: recursos técnicos como persistência, troca de mensagens etc.

"O princípio essencial é de que qualquer elemento de uma camada depende somente dos outros elementos da mesma camada ou dos elementos das camadas "abaixo" dela. A comunicação para cima deve passar por algum tipo de mecanismo indireto". Ou seja, o importante aqui é que a camada de domínio seja auto-contida e não dependa de nenhuma das outras camadas do software.

A aplicação do princípio da inversão de dependência para possibilitar que a camada de domínio não dependa de outras camadas é descrito nesses dois posts do Uncle Bob: "The Clean Architecture"[1] e "A Little Architecture"[2]. Citando um trechinho: *"A web é um detalhe. O banco é um detalhe. Nós deixamos essas coisas do lado de fora, onde elas não podem trazer muito dano."*

Muitas pessoas exaltam a importância da infra-estrutura na definição de arquitetura de um software. Alguns acham que a escolha do Sistem Gerenciador de Bancos de Dados é a mais importante. Algumas outras se degladiam pela escolha do *framework* de desenvolvimento. Mas o post "A Little Architecture" esclarece como essas coisas na realidade importam muto pouco. As regras de negócio tendem a ser mais duradouras que as tecnologias empregadas. E queremos ser capazes de facilmente substitui tecnologias obsoletas se preciso. Entendemos que as regras de negócio são a parte mais valiosa do sistema. Por isso a importância do estudo do DDD, que consiste no aprimoramento de nossa capacidade em escrever a camada do domínio.

5 Elementos do domínio

Os elementos do domínio classificados por Evans são: entidades, objetos de valor e serviços.

Entidades representam algo com identidade e com uma continuidade rastreável por diferentes estados e até diferentes implementações. A definição de um objeto como esse não está em seus valores, pois objetos diferentes com valores iguais ainda devem ser distinguíveis. A definição da identidade é a principal característica de uma entidade. Um exemplo de entidade é um veículo que deve ser identificável em diferentes partes do sistema como correspondente a um determinado veículo único existente no mundo real. Dois veículos Fox, da Volkswagen, ambos de cor preta e demais características de fábrica idênticas ainda são veículos distintos.

Objetos de valor descrevem o estado de alguma coisa. Por exemplo, um veículo pode ter uma cor, que é descrita por uma combinação numérica RGB. Dois veículos podem

[1] https://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html
[2] http://blog.cleancoder.com/uncle-bob/2016/01/04/ALittleArchitecture.html

ter cores idênticas, mas não precisamos nos preocupar com que os dois objetos veículos apontem exatamente para o mesmo objeto cor. Ou seja, nesse caso a cor não possui uma identidade. Ela é meramente um valor. Para reforçar: enquanto uma entidade define *quem* ela é, um objeto de valor define *o que* ele é. Objetos de valor devem ser imutáveis. Assim, um veículo pode mudar de cor, mas não é a cor do veículo que se altera. Ou seja, o objeto veículo passa a apontar para um novo objeto cor e o antigo objeto cor é esquecido. A imutabilidade traz várias vantagens ao design de software [Bloch, 2008], e podemos aproveitar essas vantagens nos objetos de valor.

Serviços são operações oferecidas como interfaces, isoladas no modelo e que não possuem estado. Acomodam operações que possuem significado no domínio, mas que não se encaixam em entidades ou objetos de valor. Serviços devem ser criados com parcimônia, pois a utilização de serviços deixa o software menos orientado a objeto e mais procedural. Mas às vezes são convenientes para que regras complicadas de orquestração dos elementos do domínio não vazem para a camada de aplicação. Um exemplo seria a transferência de dinheiro entre contas bancárias, caso a operação "transferência" não se encaixe adequadamente no objeto "conta". Serviços existem em todas as camadas do software. Um exemplo de serviço de infra é o envio de e-mail. Mas é preciso ter os serviços de cada camada bem caracterizados e separados.

Módulos, chamados de pacotes na linguagem Java, são agrupamentos de classes. Devem possuir significado no domínio. A hierarquia de módulos deve refletir uma hierarquia do domínio. O nome de um módulo deve ser expressivo e pertencer à linguagem onipresente. A coesão e acoplamento dos módulos também deve ser controlada. A separação dos módulos deve isolar conceitos para facilitar o entendimento dos mesmos. A refatoração da estrutura de módulos é mais complicada do que a refatoração de classes, mas não menos importante.

6 Entidades e suas regras de negócio

Uma prática adotada por muitos desenvolvedores é a separação das definições de dados e comportamentos de uma entidade em objetos separados. Em algumas nomenclaturas seriam as *entities* e os *BCs* (*business classes*). A página 107 do livro inicia uma discussão importante ao criticar essa separação. Quando possível, deixe o comportamento de uma entidade no mesmo objeto que define seus dados. Assim, Evans sugere que as regras de negócio que ficam isoladas nos BCs passem a integrar as entidades. Além disso, regras do BC que sejam na verdade regras que manipulem o ciclo de vida da entidade (criação, alteração e deleção) podem se acomodar melhor em fábricas ou repositórios (ver próxima seção).

Essa separação de entidade com dados de um lado e BCs com regras de negócio do outro gera o que é chamado por aí de modelo anêmico. Alguns textos de rápida leitura que corroboram com o desencorajamento do modelo anêmico: "O que é Modelo Anêmico? E por que fugir dele?"[3] e "AnemicDomainModel"[4].

Mas a adoção dessa prática deve possuir alguns cuidados, como não criar acoplamentos indesejados. Exemplo: *empresa.possuiFuncionarioProcessado()* parece uma interes-

[3] `http://blog.caelum.com.br/o-que-e-modelo-anemico-e-por-que-fugir-dele/`

[4] `http://www.martinfowler.com/bliki/AnemicDomainModel.html`

sante adesão a um modelo não anêmico. Mas se a princípio o modelo não exige que o objeto "empresa" tenha conhecimento de objetos de "processos", esse método pode não ser uma boa ideia. Nesse caso, melhor talvez seria algo como *processoRepositorio.possuiFuncionarioProcessado(empresa)*.

Contudo, ainda há controvérsias... No livro Clean Code, Uncle Bob faz uma distinção entre objetos e estruturas de dados [Martin, 2008]. Ele considera que no padrão *active record*[5], os modelos (classes que fazem a ponte com o banco de dados para a persistência das entidades) devem ser simples estruturas de dados. Ele critica a colocação de regras de negócio nessas estruturas de dados e termina concluindo que *"a solução é, claro, tratar o Active Record como uma estrutura de dados e criar objetos separados que contenham as regras de negócio e que escondam seus dados (que são provavelmente apenas instâncias do Active Record)"*. Alguns trechos a mais desse capítulo do Clean Code podem ser encontrados no post "Clean Code: objetos não são estruturas de dados!"[6]

Essa solução no fundo vai de encontro à recomendação geral de não expor publicamente todos os dados de uma entidade. Mas forçar essa separação da entidade em classes diferentes parece um tanto burocrático e cair no caso criticado por Evans, no qual há uma separação da entidade conceitual em diferentes objetos, sendo um para manter a estrutura de dados e o outro as regras de negócio da entidade.

Para concluir a controvérsia, minha posição neste momento é de que a postura mais equilibrada parece ser implementar suas entidades como classes contendo os dados da entidade nos atributos e as regras de negócio em métodos públicos, mas evitando a publicização desnecessária dos dados da entidade. Em Java isso seria não criar *getters* e *setters* desnecessariamente. Uma posição similar pode ser depreendida do post "Como não aprender Java e Orientação a Objetos: getters e setters"[7]. Mas essa é uma discussão pra lá de complicada!

7 Padrões do ciclo de vida de um objeto do domínio

agregados: possui uma entidade e vários objetos d valor. Membros externos ao agregado acessam a entidade e não podem modificar diretamente os objetos d valor. Entidade garante as invariantes do grupo. Ex: carro vs penu. Ex: laudo vs proprietario, veiculo vistoriado....

fábricas: A função da fábrica é instanciar um objeto potencialmente complexo a partir dos dados. Fábrica não cuida de persistência. A fábrica pode instanciar um novo objeto ou reconstituir um objeto já existente a partir de seu formato serializado.

repositórios: faz o CRUD do objeto. Interface do repositório é ligado ao modelo, e não à infra. O repositório usa um DAO para ter acesso a infra. Ou seja, no repositório ficam as regras de negócio de CRUD da entidade, não os mecanismos tecnológicos de persistência. Esses últimos ficam no DAO, que é da camada de infra.

Então um cenário típico de uso, para a criaçaõ de um novo objeto, fica assim:

1. Cliente pede pra fábrica criar objeto.

[5] `https://pt.wikipedia.org/wiki/Active_record`
[6] `http://polignu.org/artigo/clean-code-objetos-n%C3%A3o-s%C3%A3o-estruturas-de-dados`
[7] `http://blog.caelum.com.br/nao-aprender-oo-getters-e-setters/`

2. Cliente pede pra inserir objeto no repositório.

3. Repositório pede pro DAO persitir o objeto.

Já para a reconstituição de um objeto, podemos ter:

1. Cliente pede objeto para repositório.

2. Repositório acessa o banco via DAO para obter dados persistidos do objeto.

3. Repositório repassa dados persistidos do objeto a fábrica.

4. Fábrica reconsrói objeto, que é devolvido ao repositório, que é devolvido ao cliente para a edição.

A complexidade desses passos decorre do caso mais complicado no qual a reconstituição do objeto não se resume simplesmente a restaurar os dados persistidos no banco. O que é o banco guarda, pode ser apenas parte do que é objeto é em tempo de execução. Esse é um dos motivos pelo qual é muito arriscado permitir que o sistema de uma aplicação tenha acesso direto ao banco de dados de outra aplicação. Por mais que impor algumas restrições no modelo de dados do banco possa ser uma ideia em caráter de garantia extra, essas regras podem não dar conta de manter complexas restrições semânticas decorrentes do domínio.

Detalhe importante: mesmo a interface que o DAO fornece não deve conter detalhes da tecnologia. Assim, tudo o que o domínio sabe é que a entidade está sendo persistida em algum lugar, mas sem nenhuma dependência com alguma tecnologia específica. Alguns programadores leval isso mais a sério ao criar interfaces para os DAOs, fazerem o domínio depender da interface, e aí criar implementações acopladas ao banco, mas que são desconehcidas do domínio. Essa complicação faz sentido, mas eu acredito que esse trabalho possa ser evitado enquanto se mantenha primeira ideia do parágrafo: a interface do DAO (i.e. nome de classe e assinaturas dos métodos) não pode conter detalhes específicos de tecnologia.

8 Evolução do modelo do domínio

Refatoração constante.

Evoluções passo a passo são as vezes interrompidas por uma oportunidade de avanço onde as coisas mudam bastante de uma só vez. São as oportunidades de avanço:

* Busca por um modelo mais profundo.

* Evitar regras de casos particulares.

* As vezes há uma oportunidade de avanço para um modelo mais profundo que requer uma grande refatoração, daquelas que é difícil concluir sem deixar o código quebrado por um tempo. Mas vale a pena.

* Refatoração técnica (melhora o design, mas sem alterar o modelo do domínio) vs refatoração para refinamento do modelo. A primeira é algo mecânico, a segunda é mais subjetiva.

O autor enfatiza o como sessões em que desenvolvedores e analistas de negócios se juntam para rabiscar diagramas pode ser proveitosa (para evoluir o modelo). Nessa toada, uma coisa que o autor valoriza são os diagramas informais. Se prender às normas restritas da UML pode não ser o ideal para passar uma certa mensagem. Afinal, um bom diagrama

é sobre comunicar sucintamente algo à equipe. Do meu ponto de vista, acho aborrecedor com as diversas ferramentas UML por aí que inserem restrições diversas (ex: homens palitos só podem ser inseridos em diagramas de caso de uso).

9 Outras dicas de design

* Associações bidirecionais (e referências circulares) são problemáticas para o design (mas as vezes necessárias).

* A direção da travessia muitas vezes capta uma visão aprofundada do domínio, aprofundando o próprio modelo.

* Explicite os conceitos implícitos.

* É preciso muita iteração e conversa com os especialistas do domínio. Rabiscar diagramas no quadro branco ajuda bastante.

* Acertar o design de primeira não existe.

* Ler livros sobre o domínio pode ajudar também.

* Outros conceitos além de substantivos/verbos para fazerem parte do modelo: restrições, processos de negócios e especificações.

Uso de especificações...

* Interface reveladora de intenções. Diminuir esforço cognitivo do desenvolvedor. Se a mente do desenvolvedor está transbordando de detalhes internos sobre o objeto utilizado, sua mente não está limpa para resolver o problema. Se é preciso considerar a implementação de um componente para utilizá-lo, o valor do encapsulamento é perdido. Utilizar bons nomes é um caminho para resolver esses problemas. TDD (testes de unidade antes da implementação) também ajuda.

* Funções (devolvem valores) vs comandos (têm efeitos colaterais). Um método deve ser ou uma função ou um comando, não misture os dois. Prefira funções. Comandos devem ser bem simples. Cálculos complexos ficam melhor em objetos de valor (não em entidades).

* Se esforce para reduzir as dependência entre classes e módulos (diminuir o acoplamento). Se possível, crie classes autônomas, que podem ser estudadas por si só. Isso diminui o esforço cognitivo em compreender/usar/testar os elementos do domínio. Obs: dependências visíveis na interface são piores que dependências internas; dependências com elementos fora do módulo são piores que dependência com elementos dentro do módulo.

* Se possível, crie operação *fechadas* sobre elas mesmas. Ou seja, a operação retorna um objeto do mesmo tipo que o objeto que contém a operação (x.inverse()). Também ajuda: retorno do mesmo tipo que o argumento.

* O autor vê com certa reserva abordagens como design declarativos, sistemas baseados em regras e linguagens declarativas. São interessantes, mas há limitações. O autor encoraja a utilização dos padrões apresentados ao longo do capítulo para que o código de objetos tenha um estilo mais declarativo. Essa discussão é bem parecido com a questão se fazer BDD com ou sem a linguagem de alto nível, mais próxima da linguagem natural.

* Como estilo declarativo o autor dá alguns exemplos de combinações de especificações usando operadores lógicos. Assim chegamos a códigos como "Spec both = ventilated.and(armored)", onde ventilated e armored são Spec. No geral, creio que padrões de "linguagens fluentes"ajudem nesse propósito. Em geral esses padrões são construídos que retornam o próprio objeto invocado (ex: design pattern Builder). Pode-se até chegar ao ponto de verificar se uma spec contém outra spec: manSpec.inclui(mortalSpec).

Depois de uma discussão sobre alterar o modelo, pode ser legal esperar alguns dias pra discutir novamente, com a ideia mais madura, antes de de fato por a mão na massa e alterar o design ("dormir com o problema"ajuda).

Refatore quando: * O design não expressa o entendimento atual da equipe sobre o domínio. * Conceitos importantes estão implícitos no design. * Você vê uma oportunidade de tornar mais flexível alguma parte importante do design.

Design e emergência... citar caindo na real.

10 Arquitetura de larga escala

Os capítulos finais do livro (cap 14 em diante) apresentam algumas ideias interessantes sobre como lidar com grandes sistemas de software, nos quais fica difícil manter um único modelo de domínio para todas as funcionalidades. Um caso particular bastante abordado pelo o autor é a integração com sistemas legados.

Embora tais técnicas sejam interessantes, não são tão essenciais quanto o início do livro, pois se trata de técnicas que não se aplicam a todos os projetos. Por isso, dependendo da sua situação (experiência e tipos de projeto com o qual trabalha), pode ser uma opção interessante em um primeiro momento ler o livro apenas até o capítulo 13. Mas vou dar aqui uma breve explanada sobre essas ideias finais do livro.

Referências

[Aniche, 2015] Aniche, M. (2015). *Orientação a Objetos e SOLID para Ninjas*. Casa do Código.

[Bloch, 2008] Bloch, J. (2008). Item 15: Minimize mutability. In *Effective Java*, chapter 4 Classes and Interfaces, pages 73–80. Addison-Wesley, 2nd edition.

[Martin, 2008] Martin, R. C. (2008). Chapter 6. Objects and Data Structures. In *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall.