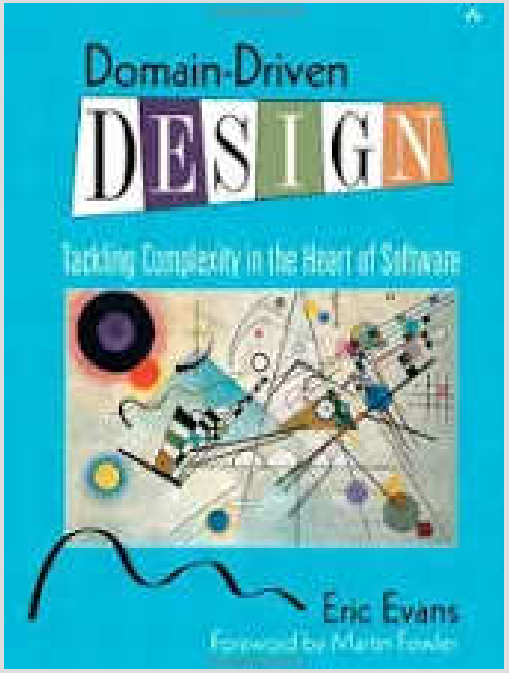
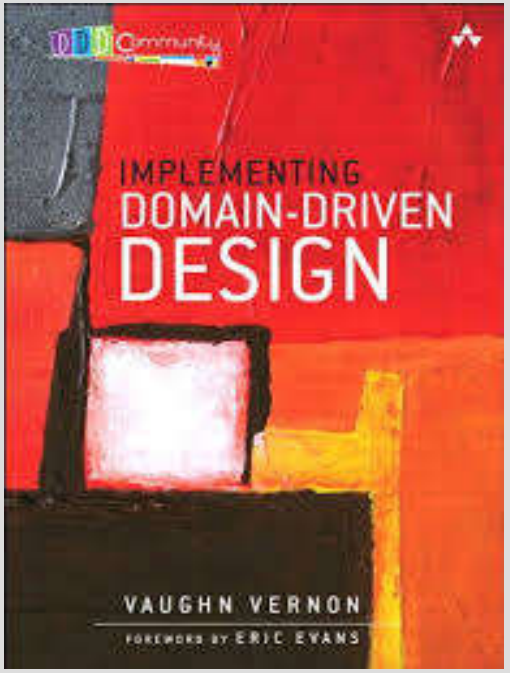
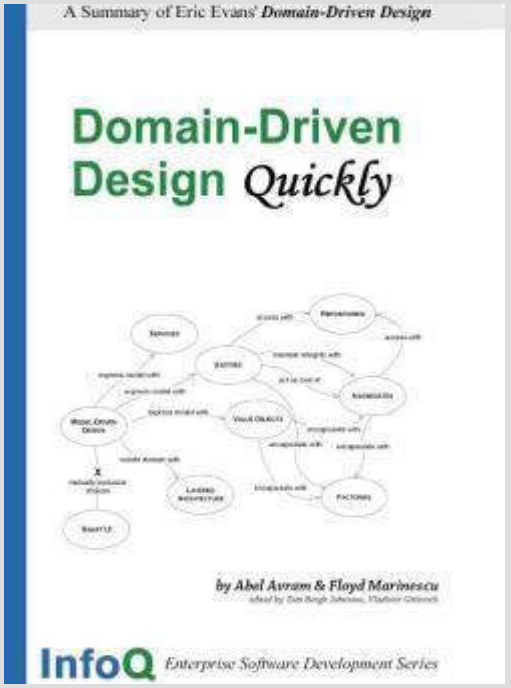
Domain Driven Design - DDD







O que é DDD ?

O Domain-Driven Design é a reunião de um conjunto de princípios e práticas que surgiu com o intuito de atenuar problemas que atormentam desenvolvedores há algum tempo.

Não se trata de nenhum novo conceito, mas sim uma espécie de “guia” para a aplicação de práticas já existentes de forma a melhorar o processo de desenvolvimento.

Utilidade

O Domain-Driven Design apresenta um conjunto de valores e práticas que têm como objetivo auxiliar desenvolvedores a despender mais tempo no que realmente importa, que é o domínio de uma aplicação.

Quanto mais complexo o domínio, mais retorno a aplicação do DDD trará.

Especialista de Domínio

O time para criação de um software deve conter de um lado especialistas de negócio e de outro os desenvolvedores e arquitetos.
No processo ágil defendido pelo MDD a criação do modelo abstrato deve ser feita em grupo, com todas as pessoas juntas.
Se arquitetos e analistas de negócio criarem o modelo sem a participação dos programadores, corre-se o risco de criar um modelo que não é implementável ou que usará uma tecnologia inadequada.

Especialista de Domínio

Da mesma forma, se os programadores codificarem sem se basear num modelo consistente, provavelmente desenvolverão um software que simplesmente não serve para o domínio. Em DDD, parte das pessoas que modelam o domínio são necessariamente pessoas que colocam a mão em código. Se os programadores não se sentirem responsáveis pelo modelo ou não entenderem como o modelo funciona, então o modelo não terá relação alguma com o software produzido por essas pessoas.

Linguagem Ubíqua

Para ter um software que atenda perfeitamente a um determinado domínio, é necessário que se estabeleça, em primeiro lugar, uma Linguagem Ubíqua. Nessa linguagem estão termos que fazem parte das conversas entre especialistas de domínio e times de desenvolvimento. Todos devem usar os mesmos termos tanto na linguagem falada quanto no código.

Linguagem Ubíqua

"Temos que emitir a fatura para o cliente antes da data limite"

- Uma classe para a entidade Cliente;
- Uma classe para a entidade Fatura;
- Algum serviço que tenha um método emitir;
- Algum atributo com o nome de data limite;

Linguagem Ubíqua

Essa linguagem ubíqua deve ser compreendida por todos e não pode haver ambiguidades. Toda vez que alguém perceber que um determinado conceito do domínio possui várias palavras que o represente, essa pessoa deve tentar re-adequar tanto a linguagem falada e escrita, quanto o código.

Domain Model

Domain = o problema

Model = a solução

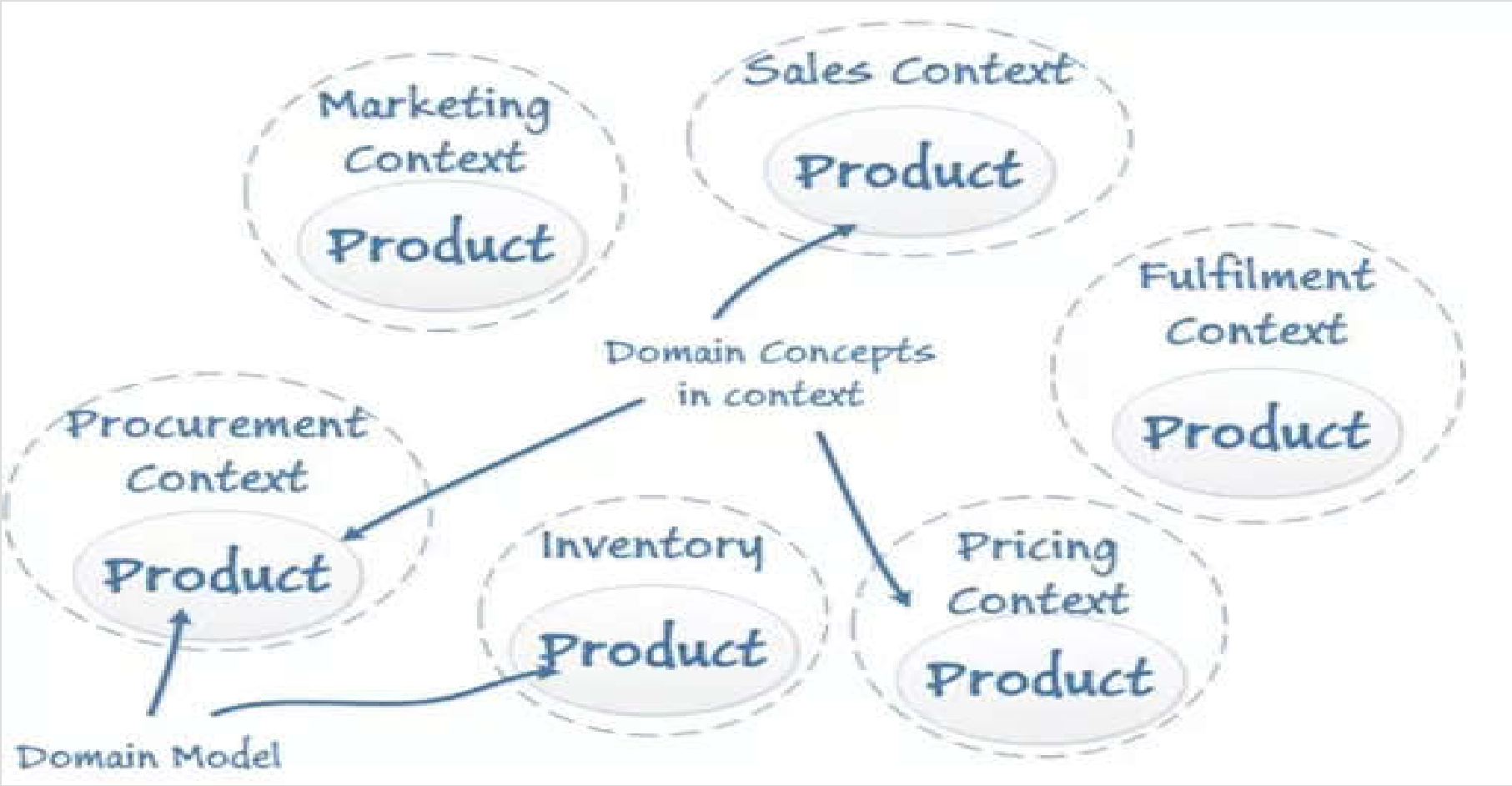
Domain Model ?

Domain Model

- organização e estruturação do conhecimento do problema
- pode ser um diagrama , exemplos de código ou até a documentação do problema.

O importante é que ele deve ser acessivel e entendido por todos os envolvidos com o projeto.

Context Map


Marketing Context
Product
Sales Context
Product
Fulfilment Context
Product
Domain Concepts in context
Procurement Context
Product
Inventory
Product
Pricing Context
Product
Domain Model

Context Map

Pode ser representado através de imagens e uma simples documentação do tipo de relacionamento entre os contextos.

Modelagem estratégica além de delimitar os contextos engloba outros conceitos como Sub-Domain, Shared Kernel, Customer/Supplier, Conformist, Anti-Corruption Layer, Separate Ways, Open Host Service e Published Language.

Arquitetura

Cada contexto pode possuir uma arquitetura independente dos demais.

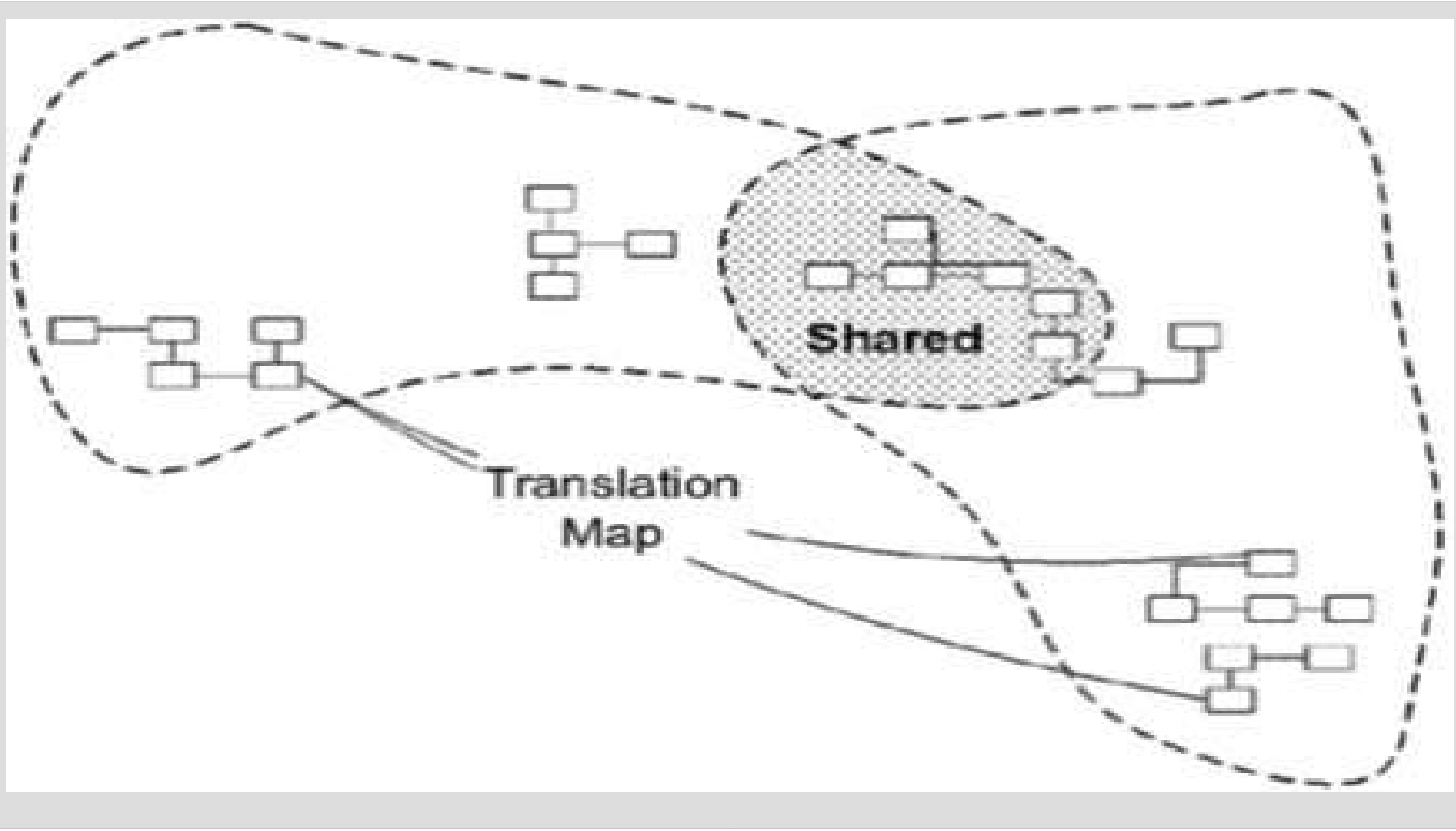
O DDD não prega a necessidade de uma arquitetura de 4 camadas (Presentation, Application, Domain e Infra). Pelo contrário, o arquiteto tem a liberdade de definir o melhor estilo arquitetural para atender a necessidade da aplicação.

Arquitetura

Existem diversos estilos arquiteturais como a clássica Arquitetura Cebola, Arquitetura Hexagonal proposta pelo Vernon em seu livro Implementing Domain Driven Design ou até mesmo os populares Microservices.

Alguns destes estilos inclusive podem fazer uso de patterns como CQRS, Event Sourcing, etc.

Shared Kernel



Shared Kernel

Dois contextos compartilham um subconjunto do modelo.

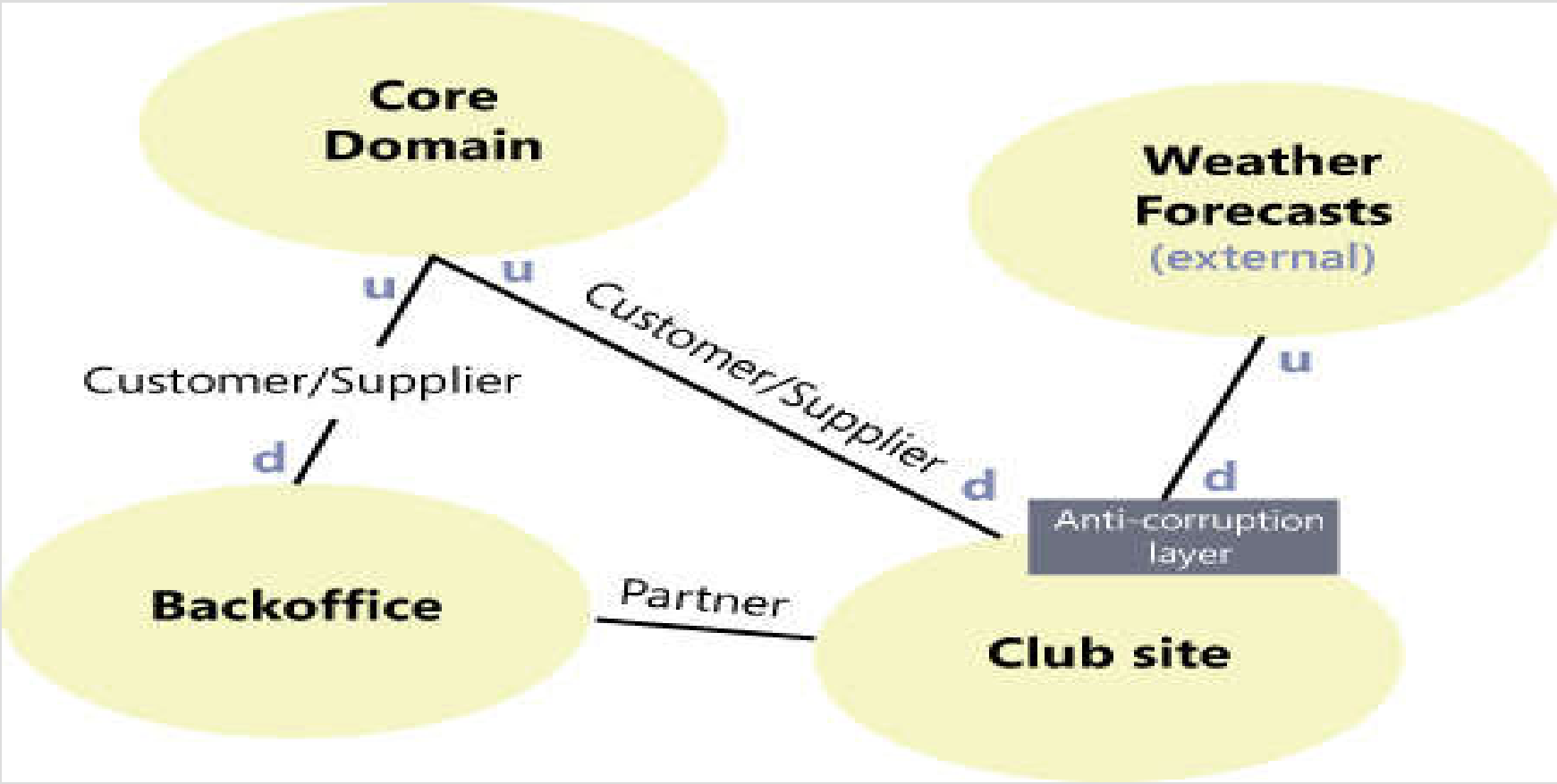
Portanto, os contextos são fortemente acoplados e nenhuma equipe pode alterar o kernel compartilhado sem sincronizar com a outra equipe.

Customer/Supplier

Dois contextos estão em um relacionamento clássico **u**pstream / **d**ownstream, em que o fornecedor é o upstream.

As equipes, no entanto, trabalham juntas para garantir que nenhum dado desnecessário seja enviado.

Customer/Supplier

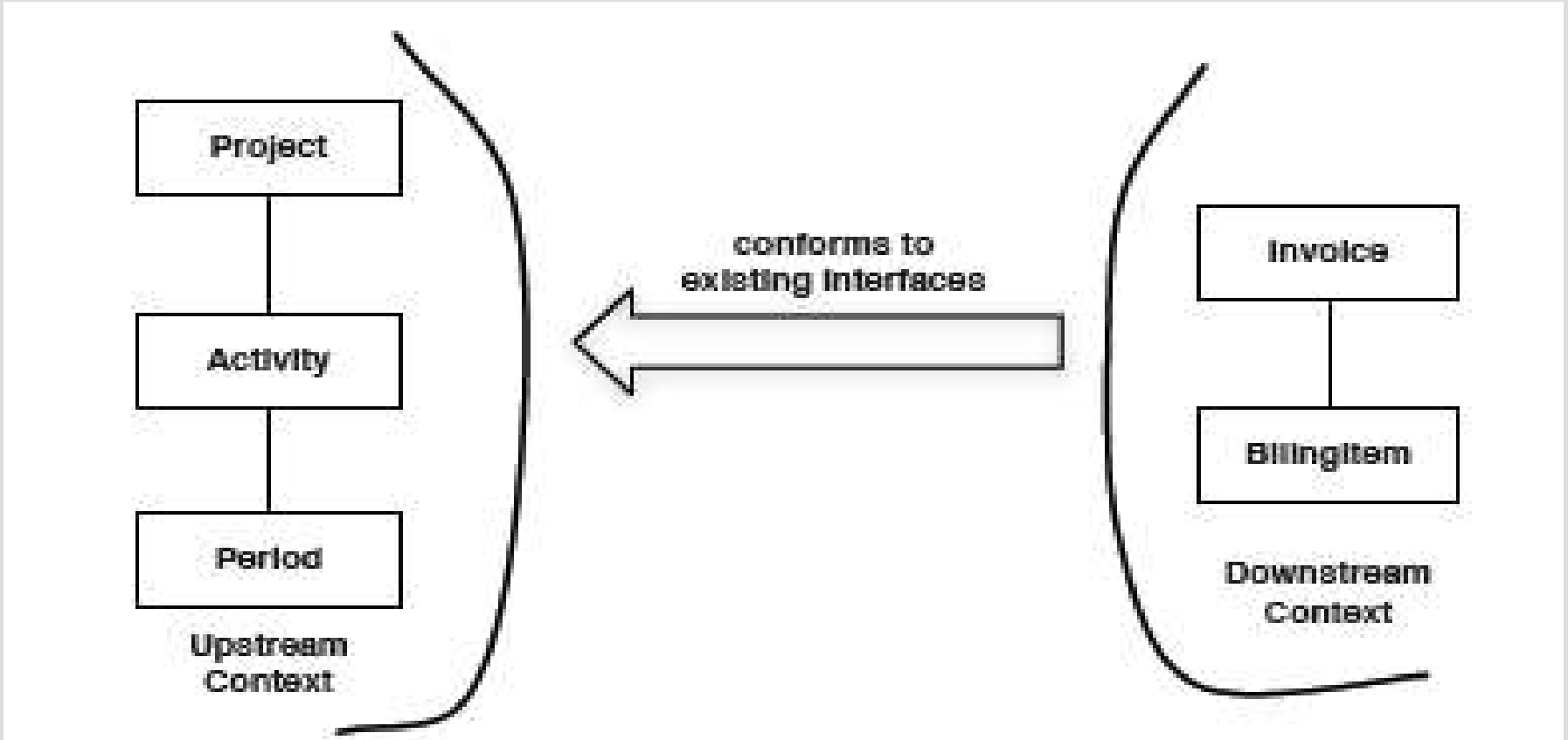


Conformist

O contexto de downstream se adapta passivamente a qualquer modelo que o contexto de upstream criar.

Normalmente, o padrão conformista é uma abordagem mais leve que a ACL e o contexto downstream também recebe dados de que talvez não seja necessário.

Conformist

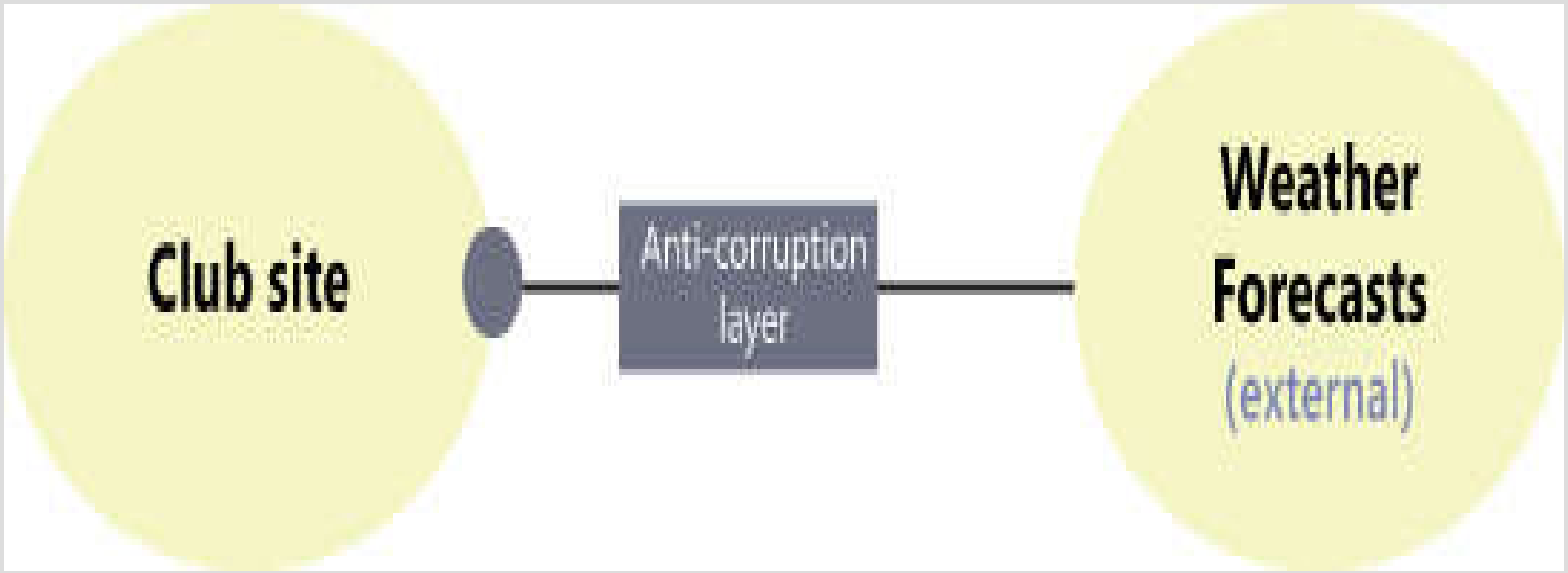

Project
Activity
Period
Upstream Context
conforms to existing interfaces
Invoice
BillingItem
Downstream Context

Anti-Corruption Layer

Indica uma camada extra de código que oculta no contexto de downstream quaisquer alterações implementadas em algum momento no contexto de upstream.

É a maneira mais segura, porque todas as alterações necessárias para manter os contextos sincronizados quando sofrem alterações são isoladas nesta camada.

Anti-Corruption Layer



Separate Ways

Se dois conjuntos de funcionalidades não tiverem um relacionamento significativo, eles poderão ser completamente separados um do outro.

Referências

http://ddd.fed.wiki.org

https://www.microsoftpressstore.com/articles/article.aspx?p=2248811&seqNum=3

https://www.jp-institute-of-software.com

https://www.infoq.com/br/articles/ddd-10-anos/

https://www.eduardopires.net.br/2016/08/ddd-nao-e-arquitetura-em-camadas/

https://www.infoq.com/minibooks/domain-driven-design-quickly/

Eric Evans

Conselhos para pessoas que estão tentando aprender DDD ?

- Modeladores precisam codificar (hands on)
- Foco em cenários concretos
- Não tente aplicar DDD em tudo
- Experimente muito e espere cometer muitos erros.

Video Adicional

Canal dotNet no youtube de 23/09/2019

Arquitetura de Software: Fatos, Mitos, Verdades