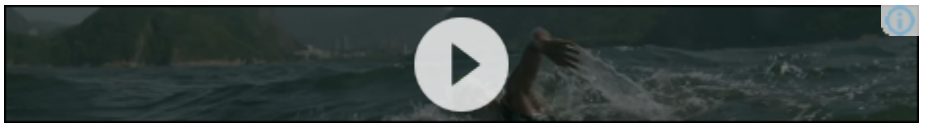
Java Building



Nomenclatura

Aug/11
20
1 Comentário | Arquivado em : Boas Práticas, Desenvolvimento

Pode não parecer, mas a nomenclatura ajuda bastante a manter um código limpo, coeso , coerente e de fácil entendimento. Nos tempos em que se fala muito de DDD (Domain Driven Development) muitos se esquecem que técnicas como o glossário de projeto e o uso dos nomes do domínio nas entidades sempre foram boas práticas. Estas práticas forma pedidas no tempo por várias razões mas principalmente pela deficiência das linguagens de programação em libertar o programador e deixá-lo usar os nomes que quisesse. Técnicas como a notação húngara muito famosa nos tempos áureos de linguagens como VB (pré .NET) e Delphi e que é usada até hoje na programação e nomenclatura do Windows, por exemplo, ajudaram a empobrecer e apodrecer essas boas práticas relacionadas a dar o nome certo à coisa certa.

O Java, e mais propriamente a Sun no seu compêndio de boas práticas, deixaram claro que a nomenclatura é vital para o sucesso de um API. A nomenclatura tem várias facetas, e todas elas devem ser consideradas.

Tipografia

A tipografia dos nomes é importante. Em java foi criado o padrão de usar nomes em Camel Case. Você deve conhecer o Upper Case (Caixa Alta) que significa que todas as letras da palavra são maiusculas – por exemplo : SERVICODECOBRANCA -, o Lower Case (Caixa Baixa) em que todas as letras das palavras são minusculas – por exemplo : servicodecobranca. O Camel Case (Caixa Camelo) é quando todas as letras inicias das palavras são maiusculas e o resto minusculas – por exemplo: ServicoDeCobranca. Note como os seus olhos entendem melhor o camel case do que qualquer outro case já que as maiusculas atuam como separadores naturais.

Outras linguagens adotam padrões diferentes como o uso de *underline* (por exemplo: servico_de_cobranca) que o java utiliza também em certas circunstancias. A tipografia de uma constante é uma mistura do Upper Case com o uso de underline (SERVICO_DE_COBRANCA).

É importante que todo o seu codigo seja escrito com a mesma tipografia. Muitas pessoas se habituaram a usar o eclipse e as suas milhentas formas de pintar texto para separar as coisas, mas usando uma tipografia padronizada (a que a Sun criou e recomendou por anos) você não precisa de cores e estilos.

Justaposição

A justaposição é um dos mecanismos que existe nas línguas para criar novas palavras. A justaposição se caracteriza por simplesmente justapor (colocar as palavras juntas) sem nenhum tipo de modificação das palavras. Por exemplo, guarda-chuva e passatempo. Note que as palavras não foram modificadas. Enquanto que outras formas de criação de palavras como a aglutinação levam à modificação das palavras originais. Por exemplo , planalto (= plano + alto) em que o ultimo 'o' de planalto desaparece.

A justaposição é a forma mais simples de criar nomes para classes, métodos e variáveis já que não é necessário modificar a palavra original, e, no caso, não nos precisamos preocupar com sinais como o hífen já que o Camel Case separa as coisas para nós.

Língua

É importante escolher a língua do seu código. Isto é mais complexo do que parece. A língua inglesa é mais simples de usar já que em inglês a justaposição é muito natural e soa bem. Em português nem sempre soa bem juntar várias palavras juntas. Além disso nomes de padrões de projeto são em inglês , e como veremos adiante, é comum usar esses nomes ao compor nossas nomenclaturas. DesktopSingleton (Desktop + Single + ion) soa bem melhor que TopoDeMesaSolteirao (Top da Mesa + Solteiro + ião). Poderiamos pensar num DesktopSolteiao ou TopoDeMesaSingleton, mas ai é misturar o pior de cada parte. Usar inglês também tem a vantagem de não usar acentos e outros dilacrílicos (é por isso que se escreve facade e se lê façade), que embora a língua inglesa os aceite (Façade é uma palavra inglesa escrita com ç porque vêm do francês) não é comum vermos usar (porque os teclados as pessoas que falam inglês nativamente, não têm o caractere ç).

A resistência de muitos a usar o inglês advém de dois problemas : 1) falta de vocabulário e 2) pode violar a regra de que se deve usar um glossário de projeto. O primeiro motivo é fútil e qualquer dicionario pode resolver isso. O segundo motivo é mais sério. Se o cliente fala português nativamente e define seus conceitos de negocio em português porque então traduzir isso para inglês? Algumas coisas até seriam triviais : produto -> product , cliente -> costumer , mas fatalmente cairíamos no Nota fiscal -> ? , Pedido -> ? ou CPF -> ? . Assim muitos preferem usar uma mistura de inglês com português, usando o inglês para código de infra e padrões e o português para objetos de negocio.

O meu argumento é que se você realmente quiser você consegue usar apenas nomes em inglês. O lance é utilizar o domínio em inglês também. Por exemplo, a nota fiscal é um documento que prova que o objeto é seu, que você o comprou. Desse ponto de vista ele é uma "nota de venda" e a tradução seria "bill of safe"(literalmente nota (bill) de (of) venda (sale)). Pedido seria Invoice, embora invoice possa ter significados mais refinados ligeiramente diferentes de pedido, mas contém os mesmos itens básicos: quais os itens, quantos, de quem, para quem. Já o CPF é algo próprio do pais e nem sequer do domínio do negocio de compra e venda. Está mais relacionado a impostos. Contudo poderíamos criar um nome que traduzisse o conceito em vez de traduzir o nome, por exemplo Individual Tax Registry Code (ITRC). Rebuscado ? Depende. Para um sistema feito em pais de língua oficial portuguesa para pessoas que falam português nativamente, definitivamente. Mas para software que será usado no estrangeiro ou que pretende se adequar a vários mercados, ou que se pretende que seja open source, talvez não seja tão absurdo.

A moral aqui é que deve ser considerado o objetivo e o publico alvo do software e do código de forma que para os usuários do software o código possa fazer sentido. Isto em DDD se chamaria de usar a linguagem ubiqua, mas na realidade estamos adequando o máximo possível o código ao glossário. Por outro lado, se você consegue fazer código de infra em inglês por que não código de negocio ? A resposta é que você conhece o domínio "infra" muito melhor, e é natural para si – programador – usar inglês. Apenas isso.

Depois que decidir que língua usar , ou como misturar duas línguas, mantenha-se fiel à escolha.

Substantivos

Vimos várias coisas que ajudam na nomenclatura, mas existem regras para criar os nomes eles mesmos ? Existem.

Classes e Interfaces têm nomes simples que refletem diretamente os conceitos do modelo de negocio. Sem

prefixos ou sufixos. Se quer modelar um cliente use “Cliente” como nome da classe. Se for uma interface não use “ICliente” por exemplo. A razão disto é simples : polimorfismo. Se eu digo “Veiculo” você não sabe se é uma interface ou uma classe. Ótimo. É assim mesmo que tem que ser. Se eu usar “IVeiculo” automaticamente estou dizendo que é uma interface, o que viola o proposito do polimorfismo.Além de que aquilo que hoje definiu como interface, amanhã pode virar um classe. O modelo evolui. O uso do I é uma remanescencia da notação hungara e é usado na corrente .NET já que essa corrente herdou várias coisas dos antepassados VB, Delphi e do codigo do windows, como comentei antes. É lícito usar o I em .NET porque essa é a convenção nesse ambiente, mas em pura orientação a objetos , e no java, não.

Se eu precisar criar um objeto utilizando o padrão Builder então justaponho os nomes como ClienteBuilder ou VeiculoBuilder. Normalmente os builders são mais usados com interfaces, mas isso não obrigatorio. Aqui a lógica é utilizar o nome do padrão como sufixo e aproveitar o nome do conceito. Usando este padrão de nomenclatura teriamos por exemplo ClienteProxy, ClienteRepository, ClienteFactory, etc… Contudo não se utiliza este padrão com singleton nem para as implementações do padrão Service. Não se nomeia ClienteSingleton. Isto pela mesma razão do uso do I em interfaces : quebra o polimorfismo e o que hoje é singleton, amanhã poderá deixar de ser.

Na questão do padrão service, definimos uma interface como o contrato do serviço e mais do que uma implementação. Cada implementação é especializada por alguma razão e isso distingue uma implementação do serviço das outras.Por exemplo , para o serviço PrintService poderiamos ter um PDFLocalFilePrintService , um SystemPrinterPrintService e um WebRemotePrintService (como exercício, experimente colocar estes nomes em português). Todas as implementações terminam com o nome do serviço mas descrevem o objetivo da implementação. É claro que a primeira implementação irá criar um PDF em disco local, a segunda usará uma impressora de fato e a terceira algum serviço de impressão via web (sem dizer se será em uma impressora ou arquivo). Isto se aplica também ao padrão DAO (que é uma especialização do padrão de serviço) onde teremos coisas como JDBCClienteDAO ou BigTableClienteDAO ou LDAPClienteDAO. O uso de Impl com o sufixo (ClienteDAO e ClienteDAOImpl, por exemplo) além de ambíguo e desinformador (como é a implementação ?) é pura falta de imaginação.

A mesma regra de nomenclatura pode ser seguido em geral com interfaces onde antes do nome da interface é dito algo sobre como aquela implementação é diferente das outras. Exemplos classicos são ArrayList , LinkedList e HashSet e LinkedHashSet (este é duplamente qualificado)

Este regra de momenclatura também funciona bem quando você quer juntar conceitos que não são necessáriamente padrões de projeto, mas que você definiu um conjunto de classes com um proposito semlhante, por exemplo ClienteManager se vc criou o conceito de Manager ou XMLTextTransformer se você criou o conceito de TextTransformer. Não necessáriamente estas classes herdam de Manager ou TextTransformer, pode ser uma classificação puramente conceptual. (Se for puramente conceptual recomenda-se que crie uma interface marcadora e faça todos os objetos que partilham o mesmo conceito implementá-la. Leia Herança e Interfaces para mais detalhes deste assunto)

Uma variante do padrão de sufixo indicando padrões de projeto, embora não utilize o nome de um padrão é o sufixo “Utils”. MathUtils ou DateUtils ou ClienteUtils. Este tipo de classes contém apenas métodos estáticos é final e sem construtor publico. Em certos casos é possível utilizar a variante no plural do nome como Clientes ou, retirando exemplos do próprio java : Collections e Arrays. Contudo é mais raro que esta forma de nomenclatura soe bem ou encaixe em qualquer situação.

Para classes abstratas é útil utiliza Abstract como prefixo AbstractCliente ou AbstractPrintService. Este padrão pode ser usado mesmo quando a classe não é abstrata no sentido técnico (não é definida como “abstrat class”) mas é abstrata no sentido do modelo ou do negocio. Ou seja, olhando para o nome sabemos que não é suposto instanciarmos esta classe e sabemos também que devem haver classes herdando dela. Uma classe com Abstract no nome pode herdar de outra com Abstract no nome, contudo sabemos que no fim da linha de herança existirá pelo menos uma classe não abstrata que poderemos utilizar. Este padrão é especialmente util quando o nome a seguir a abstract se refere a uma interface, por exemplo, AbstractList do java.

Especialmente para interfaces muitas vezes elas caracterizam não entidades (substantivos) mas ações que podem ser feitas (advérbios). Em inglês é muito fácil transformar qualquer palavra em adverbio e aqui é um dos exemplo onde é vantagem utilizar o inglês. Para um ação como Print, teriamos Printable (que pode ser impresso) ou mais complexo como Mergeable (que pode ser jeito merge). Em português também funciona se usar o prefixo “-vel” ou suas variações como “Juntável” , “Imprimível” , “Fundível”, mas é aqui que começa a

ficar esquisito. Na API do java temos Serializable e Clonable como exemplos desta forma de nomenclatura.

Um outro padrão de projeto que tem uma nomenclatura diferenciada é o Adapter. Aqui estamos tentando usar uma classe com a "cara" de outra. Para o adapter é bom usa a regra [InterfacePublica][ObjectoPrivado]Adapter. Por exemplo, InjectorSpringContextAdapter. Daqui podemos inferir que se trata de um objeto no padrão adapter, que implementa uma interface chamada Injector e usa um SpringContext por baixo dos panos. Ou seja, o contexto do spring está sendo adaptado ao contrato de um Injector.

Resumo

Nomenclatura coerente das suas classes e interfaces é uma boa forma de documentação do código, ajuda a manter os conceitos de negocio e os conceitos técnicos frescos, evita ambiguidade e, se bem usada, ajuda a identificar a camada a que pertence a classe. São regras simples que todos podemos seguir, e com isso ajudar que todos entendamos o código uns dos outros.

Mais informação

Informação adicional pode ser encontrada em Princípios : Nomenclatura.

« Se7e Pecados
Modelando do Zero »

Um comentário para "Nomenclatura"

1. **David**, em December 16th, 2011 às 16:00 disse:

   Muito bom o artigo! Porém, uma ressalva: apesar de singleton também significar "solteiro", acredito que no contexto de design patterns signifique "um conjunto com apenas um elemento", que é uma das traduções do verbete (o nosso "Conjunto Unitário).

Comente

O seu nome - obrigatório

O seu email (não será publicado) - obrigatório

O seu site

Enviar

Enquete

Sobre quais assuntos gosta de ler neste blog ?

- Arquitetura e Design de Software em geral
- Boas práticas em Java
- Design Patterns em Java

- [ ] Scrum
- [ ] Carreira de Desenvolvedor

Vote

View Results

JavaBuilding

- Academia
- Arquitetura
- Design Patterns
- Livros
- Oficina
- Principios

Tags

agil arquitetura boas práticas Camadas carreira conceitos contrato decorator design design pattern Design Patterns diretivas escolhas gerencia ideia java liderança linguagens mercado mitos monad MVC más práticas opinião pacotes plano plataforma portabilidade processo produto programação qualidade risco scrum tecnologia tendência valores

Artigos

Artigos Select Month

1