Java Building

Princípios

- Herança
- Polimorfismo

Nomenclatura

Nomenclatura

A escrita de código passa inevitávelmente por criar nomes. Nomes de classes, de métodos e de variáveis. Práticamente todo o codigo que escrevemos que não é uma palavra reservada da linguagem é um nome que tivemos que criar em algum momento, ou o nome que alguem já criou. Porque tanto do código é um nome é útil e conveniente ter uma politica de nomenclatura.

Nomes cuidadosamente escolhidos tornam o código mais legivel não apenas do ponto de vista da lingua, mas também do propósito. Ao ser mais claro o propósito apenas pela leitura do código diminui a necessidade de comentários que expliquem o que o codigo faz. Pela diminuição de comentários, e clareza do codigo ele é a sua própria melhor documentação.

O objetivo de uma politica de momenclatura é escolher certas regras que se seguidas, não importa por quem, produzem um nome legivel, com significado e com a mesma estrutura. Para escolher entre as regras temos que saber quais principios elas devem obdecer, pois se criarmos uma regra de nomenclatura contrária a estes principios, estaremos complicando o processo de criar nomes e não simplificando-o.

Principios de Nomenclatura

Expresse intenções

A primeira coisa que um nome deve fazer é nomear. Ou seja, identificar verbalmente algo. Esta identificação deve trazer à mente de quem lê uma imagem e partiular, no código, o nome deve revelar a intenção por detrás daquele artefato (classe, método, variável, etc...) [1] [2].

O nome deve ser claro, simples e único. Claro significa que não ha ambiguidade na identificação do nome com um, e apenas um, conceito. Simples significa que o nome deve ser composto do minimo de palavras possivel sem que obscureça o significado. Idealmente deve ser composto de apenas uma palavra. Único significa que o nome deve ser usado apenas para referir um único conceito no sistema.

Criar um nome único depende da lingua natural em que os nomes são escolhidos.

Facilite a comunicação

Se nomeiamos uma coisa é porque gostariamos de nos referir a ela no futuro apenas usando o nome. Gostariamos também que as outras pessoas entendessem ao que estamos nos referindo, apenas dizendo o nome. Para isto o nome tem que ser pronunciável. Nomes que são conjuntos de letras e nomes como a1 e bc3to onde somos obrigados a pronunciar as letras e os numeros um por um, são exemplo de maus nomes.

Outro fator que dificulta a pronunica é a utilização de prefixos sem significado como começar todas as variáveis internas com *underline* ou o uso da Notação Hungara, muito famosa nos anos 70 em linguagens com modelos de tipos pouco definidos ou voláteis demais.

Outro fator que pode atrapalhar a comunicação é o uso de nomes com apenas uma letra como *a* ou *b*. Esta regra pode ser quebrada se estamos usando as letras i,j ou k como contado, pois pela tradição todos acabam entendendo que se trata de uma variável burucrática de iteração. Outro ponto onde usar uma só letra é quando estamos resolvendo um problema matemático onde o uso de x, y e z é facilmente entendivel.

Para facilitar a comunicação os nomes não devem ter segundos significados, ou significados engraçados para a equipa e não devem ser usados sinónimos espertinhos em vez da palavra que seria esperada. Por exemplo um método que faz a aplicação terminar deve se chamar "terminar" e não "daOFora".

Em caso de duvida, ambiquidade ou dificuldade em encontrar um nome, nomes que estão relacionados aos conceitos tecnicos usados no design devem ser preferidos. Em outras palavras nomes do dominio da solução devem ser preferidos a nomes do dominio do problema [1]. A razão para isto é que quem irá ler o código é um outro programador. Como programador ele terá que entender o propósito da classe apenas conhecendo o seu nome. Se usamos nomes do dominio do problema, o programador pode não estar familizado com o problema e portanto não entender o que está sendo feito pelo codigo.

Esta regra não é aplicável, obviamente, a classes que apenas existem como representações do dominio do problema; especialmente entidades.

Esta regra é util para poder reaproveitar nomes fácilmente usando nomes de padrões de projeto. Por exemplo um `ServiceRegestry` é uma classe que segue o padrão Registry e é usada para guardar e recuperar serviços.

Estilos de escrita

Todos os nomes devem seguir o padrão CamelCase em que as palavras são aglutinadas sem espaços ou sinais não alfabéticos. As palavras são diferenciadas colocando a primeira letra em caixa alta.

Nomes de tipos (interface, interfaces, enumerações e anotações) devem seguir o estilo CamelCase com a primeira letra da palavra grifada maisucla.

Nomes de métodos ou variáveis devem seguir o estilo CamelCase em que a primeira letra é grifada em minuscula.

Nomes de pacotes são escritos sempre em minuscula, com as várias partes que compoem o nome separadas por ponto final (.). Cada parte do nome é formada apenas por uma única palavra.

Abreviações não devem ser utilizadas na formação de nenhum nome. Existem algunas exceções como a abreviação Id que usadas tradicionalmente, mas como regra abreviações não devem ser usadas. Nomes devem ser completos e fácilmente legiveis, abreviações não cooperam com estes principios [5].

Siglas como HTTP, URL e AJAX (entre outro) devem seguir o estilo de caixa normal como se fossem palavras, por exemplo um método acessor que recupere um URL seria `getUrl()` [5] e não `getURL`. A API Java padrão não adota uma politica especifica para estes casos. Por um lado termos a classe HttpServlet e por outro as classes URL e URLConnection demonstrando uma falta ou falha nas regras de nomenclatura. Com certeza um exemplo a não seguir.

Nomeando Pacotes

A plataforma de desenvolvimento não força regras para a nomenclatura de pacotes, contudo é recomendado que os nomes sejam escolhidos tal que o pacote seja universalmente único [4]. Para alcançar este objetivo é normalmente usado um caminho de dominio invertido como por exemplo `com.javabuilding` como pacote principal. O pacote principal é seguido do nome do projeto/aplicação. Dentro de pacote da aplicação são criados os pacotes para as partes da aplicação conforme o design utilizado.

Nomes dos pacotes internos da aplicação devem expressar o propósito das classes lá dentro da forma mais simples possivel. Por exemplo, um pacote chamado `org.enterprise.app.validation` deverá conter classes que auxiliam a validação. Nomes de pacotes não devem ser usados para separar entres camadas ou andares da sua aplicação. Podem ser usados em casos especiais para separar entre nodos como `org.enterprise.app.server` e `org.enterprise.app.client`.

Nomeando Classes

Os nomes das classes devem ser substantivos que designam o conceito ou a responsabilidade que a classe ocupa no sistema. Os nomes devem ser especificos o suficiente para estarem relacionados ao problema ou à solução e nomes genéricos como coleção, dados, informações, controlador ou processador, por exemplo.

Nomes de classes podem ter prefixo ou sufixos que tentam orientar o programador sobre o nivel que a classe ocupa em uma hierarquia ou no design do sistema. Contudo prefixo ou sufixo que não adicionem informação devem ser evitados. Por exemplo, chamar um classe de NotafiscalPOJO ou NotaFiscalBean não faz sentido pois já é sabido que essas classes serão beans e POJOs. Especialmente classes que representam entidades do dominio como Cliente, Produto, Uusuario, não precisam de qualquer afixo.

Outro prefixo que é inutil é um que identifique a aplicação [1]. Isso deve ser feito pelo nome de pacote da classe e não pelo nome da classe em si.

Nomeando Interfaces

As regras de nomenclatura de interfaces devem seguir os mesmos principios das regras para classes. A única diferença é que podemos escolher não apenas substantivos, mas também adjetivos e advérbios.

Muitos gostam de prefixar suas interfaces com um I maiusculo. Esta prática não é recomendada [1] por violar explicitamente o encapsulamento fornecendo informação demais. Além disso, se nos arrependemos depois e quizermos mudar para uma classe essa mudança irá provocar uma refactoramento desnecessário do codigo inteiro e até de código cliente já escrito por outras pessoas.

Nomeando Classes Abstractas e Implementações

A nomenclatura para classes abstratas é a mesma que para classes normais e interfaces normalmente adicionada do prefixo *Abstract*. Este recurso é especialmente importante no desenvolvimento de API de uso geral onde os elementos principais são definidos como interfaces e classes abstratas são implementadas para fornecer implementações padrão para a maior parte dos métodos da interface. O prefixo *Abstract* não é usado para significar que a classe precisa ser implementada, mas sim para posicionar a classe num nivel superior da

hierarquia de classes (logo abaixo de interfaces, e acima de classes comuns). O prefixo *Abstract* no nome não força que a classe seja abstracta de fato (marcada com `abstract`), contudo, o fato da classe ser marcada com `abstract` força que o nome seja prefixado com *Abstract*.

A nomenclatura para implementações de classes abstratas ou interfaces segue a regra de usar o mesmo nome da classe/interface pai com um prefixo ou sufixo que a destingua como sendo a implementação. É comum utilizar o sufixo "impl" para designar a implementação. Contudo esta não é uma boa prática, pois no momento que queremos uma implementação diferente não mais podemos usar esse prefixo e muito menos podemos usar prefixos como "impl2". Além disso o uso do prefixo "impl" viola o principio da Facilidade de Comunicação ao forçar a pronunica de um afixo abreviado e sem dar informação sobre no que aquela implementação é diferente das outras.

Prefixos e sufixos de implementações devem explicitar a forma em que aquela implementação é especifica e diferente das outras. Normalmente este afixo está relacionado ao algoritmo ou à tecnologia usada internamente naquela implementação. Exemplos classicos são `ArrayList` que é a implementação de `List` usando um array, ou `TreeSet` que é a implementação de `Set` usando um algorimo de arvore binária.

Quando estamos implementando uma interface ou classe abstrata mas não conseguimos ver à partida várias opções de implementação afixos como "Simples" e "Default" são comuns; como em `SimpleDateFormat` e `DefaultTableModel`).

Nomeando métodos e construtores

Nomes de métodos devem ser verbos (enviar, pagar, ...) com a possibilidade de serem compostos com adverbios ou substantivos relativos ao papel dos argumentos (enviarProposta, pagarImpostos) Métodos que representem acessores ou modificadores devem seguir a nomenclatura JavaBeans [3] começando com o prefixo `get` (acessor) ou com `set` (modificador). Nomes de métodos devem expresar a ação que será realizada ou a mudança de estado que se espera que aconteça.

Construtores são obrigados a ter o mesmo nome da classe (pelo menos em Java e C#) o que significa que se o construtor tiver parametros não temos como oferecer um auxiliar semantico para o significado de cada um. Neste caso é conveniente declarar métodos estáticos corretamente nomeados quen informem o propósito dos argumentos e deixar os construtores com parametros privados seguindo o padrão Static Method Factory.

Nomeando Exceções

Os nomes devem revelar a causa da exceção e devem terminar com o sufixo que representa a sua hirarquia de exceção: `Exception` ou `Error`. Por exemplo, em `ArithemeticException` sabemos que se trata de uma classe derivada de `Exception`, e não de `Error`, que foi causada por algum problema em uma operação aritmética.

Modelos de Momenclatura

Algumas classes com utilidade especifica têm formas especificas de serem nomeadas. A seguir são listados alguns modelos de nomenclatura para estes conjuntos de classes, baseados nas regras anteriores.

Entidades

O nome de uma entidade é uma exceção ao principio de Preferência de Dominio Tecnico. Nomes de entidades não devem conter sufixo ou prefixos de nenhum tipo, nem mesmo o popular *Entity*. Nomes de entidades, são exactamente isso, nomes. Substantivos que são reconhidos do dominio do problema e que normalmente guiam o objetivo ou a operação do software.

Repositório

A nomenclatura do padrão Repository seguue o modelo: `[Entidade]Repository`.

Esta nomenclatura segue o principio de Preferência de Dominio Tecnico deixando o nome do padrão no fim do

nome. O nome é precedido pelo nome da entidade a que este repositorio se refere.

Servico

O padrão Service tem duas peças: a interface do serviço e a implementação do serviço. A nomenclatura para a interface do padrão Service segue o modelo: `[Proposito]Service`. Esta nomenclatura segue o principio de Preferência de Dominio Tecnico deixando o nome do padrão no fim do nome. O nome é precedido pela descrição do proposito em forma resumida, por exemplo: `EmailSendingService`.

O nome da implementação do serviço segue o modelo: `[Tecnologia][Proposito]Service`, em que ao nome da interface do serviço é adicionado um prefixo que denota o tipo de tecnologia ou algoritmo que o serviço utiliza para satisfazer a interface, por exemplo: `JavaMailEmailSendingService`

DAO

O padrão DAO, semelhante ao padrão Service tem duas peças: a interface do serviço/estratégia e a implementação. Para classes de DAO associados a entidades, o nome da entidade é adicinado como prefixo para destinguir entre os contratos de cada entidade. O nome da interface segue o modelo `[Entidade]DAO` e o da implementação o modelo: `[Tecnologia][Entidade]DAO`. Esta nomenclatura segue o principio de Preferência de Dominio Tecnico deixando o nome do padrão no fim do nome.

O nome da implementação começa com o nome da tecnologia de persistência que a implementação do DAO usa, seguindo pelo nome da interface que está sendo implementada, por exemplo `JdbcCustomerDAO`.

Inglês vs Português

Muito do conceito de linguagem de programação é baseado na ideia de escrever o mais próximo da linguagem natural possivel. Acontece que esta lingua é normalmente o inglês. Por isso que as palavras reservadas são em inglês. A ideia é que se misturem bem com o resto dos nomes para prover uma leitura o mais fluida possivel. É esta leitura fluida que promove a melhor documentação e o código mais limpo [2].

O problema acontece quando a lingua natural do programador não é o inglês. Um programador cuja lingua natural não é o inglês terá o entendimento do código dificuldado à partida pois a leitura não mais é fluida. Mesmos assim, é possivel adaptar regras de nomenclatura especificas à lingua natural e ter os mesmos beneficios que antes. Todos excepto um: quem não souber ler aquele lingua não entenderá o código. O problema não é apenas para quem tem o Português como lingua natural, mas para todos aqueles que têm uma lingua natural diferente do inglês. Por exemplo, o japonês. Embora java aceite codigo fonte em UTF-16 (permitindo assim a escrita de caracteres janoneses) apenas programadores que saibam ler e dominem o idioma podem entender o codigo.

Existem ocasiões em que obrigatoriamente teremos que usar o inglês. Afinal, o inglês é a lingua franca *de fato* em programação. Isto acontece sempre que precisamos partilhar o codigo com outras pessoas em outros paises ou culturas. Projetos Open Source são um exemplo. Projetos em Outsource são outro. Na realidade existem poucos exemplos de quando o seu código não seria partilhado com pessoas de outros paises ou culturas.

Deve estar pensando que a sua aplicação é muito simples, que você vai vendê-la para um só cliente e nunca sairá das fronteiras do seu pais ou do seu controle. Isso pode ser verdade durante a vida inicial do software, mas se ele durar, se houver necessidade de dar manutenção, se você quiser ganhar mais dinheiro comercializando o fonte ou se simplesmente quiser fazer *outsource*, ou open source, do desenvolvimento porque os custos estão muito caros... você não poderá. Escrever código em uma lingua natural que não o inglês limita automaticamente a evolução, expansão e vida util do seu software.

Um detalhe que leva muita gente a não escrever em inglês é achar que certos conceitos são muito particulares da cultura e não podem ser expressos em inglês (como por exemplo o CPF ou o CNPJ). Então, mesmo quando escrevendo o resto do aplicativo em inglês é comum ver classes do dominio com nomes na lingua natural do programador. Isto é um problema porque esses nomes não traduzem o conceito, violando diretamente o principio de Expressar Intensão. Ele é o nome dado no dominio do problema a um certo conceito. Porque o programador, não sabe ou não consegue abstrair esse conceito ele não consegue atribuir outro nome além

daquele que conhece na sua lingua natural. Ao abstrair o conceito, é mais fácil encontrar um equivalente em inglês. Por exemplo, o CPF e o CNPF representam apenas um conceito: o numero de indentificação tributário. Que podemos facilmente nomear em inglês como TributaryIdentificationNumber ou TaxIdentificationNumber (numero de identificação para pagamento de impostos).

Não é necessário que o nome em inglês seja a tradução direta porque isso pode causar outros problemas relativamente ao dominio. Por exemplo, nos EUA, os impostos são identificados usando o Social Security Number, que é o mesmo numero que a pessoa usa para se identificar em outras agencias do governo, não apenas a que recolhe os impostos e faz, para essa entidade o mesmo papel que o RG. Portanto, tentar a tradução direta é muitas vezes um erro em si mesmo porque não ha uma contraparte na cultura inglesa. Por isso temos que nos concentrar em criar um nome, e não em traduzir. Por exemplo, boletos bancários são uma prática que não existe fora do Brasil assim como não existem no Brasil a prática de emitir Cashier's Checks. Boleto bancário é, assim, intradutivel diretamente. Contudo, atendendo ao conceito facilmente podemos criar o nome de *PrintableBankCreditTitle* (Titulo impresso de crédito bancário).

Analizar o nome do conceito na sua lingua natural e tentar passá-lo para o inglês o ajudará a criar maior poder de abstração e um maior poder de abstração o poderá ajudar a modelar orientado a objetos de uma forma mais simples e natural. Por outro lado, ao abstrair o conceito você pode ver novas hierarquias de objetos e classes que ajudarão a simplificar o seu modelo e o seu software.

Ninguém o vai impedir de criar nomes em linguas naturais diferentes do inglês, mas tenha em mente que ao fazer essa escolha estará inveitávelmente restringindo a evolução do software que está escrevendo.

Referências

[1] **Clean Code: A Handbook of Agile Software Craftsmanship**

Livro:Clean Code: A Handbook of Agile Software Craftsmanship

[2] **Code Complete**

Livro:Code Complete

[3] **JavaBeans Tecnology**

URL: http://java.sun.com/products/javabeans/docs/spec.html

[4] **Code Conventions for the JavaTM Programming Language: Naming Conventions**

URL: http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html

[5] **Effetive Java**

Livro:Effetive Java