Java Building

A Herança e a Interface

Jun/11
27
16 Commentários | Arquivado em : Desenvolvimento

Uma das coisas que mais marcou a linguagem java foi ao mesmo tempo a sua semelhança com C++ e o seu departe do C++. Os criadores do java, ao mesmo tempo que aproveitaram a sintaxe do C++ deixaram de lado algumas das coisas mais recônditas dessa tecnologia. A principal ? Muitos diriam que o abandono do uso explicito de ponteiros – ou como dizem alguns da “aritmética de ponteiros” – mas eu acho que a principal diferença é a herança única.

O conceito de herança, era, segundo os criadores do Java abusado pelo uso de herança múltipla. O que em C++ era visto como reaproveitamento de código, em Java é visto como a falha em entender o que é herança para começo de conversa. A más linguas dirão que por causa da necessidade de usar herança múltipla o java inventou a Interface e portanto permitindo-a e proibindo-a simultaneamente.

Pois são coisas distintas e embora a sintaxe possa ser enganadora, o conceito não o é.E por isso mesmo, por ser verdadeiro ao conceito e deixando de lado o facilitismo programático o Java mostrou que é possível construir grandes coisas, complexas coisas, completas coisas sem herança múltipla.

A herança se baseia no conceito de classificação. Um objeto pode apenas pertencem a uma classe ou a uma linhagem (herança) de classes, mas nunca a duas classes ou a duas linhagens. É como um ser humano ter duas mães. Impossível. Impossível não pela natureza, mas pela própria definição de mãe. É um problema lógico, não um problema biológico.

O conceito de herança é baseado na identidade da natureza do objeto. Ou seja, não na identidade do objeto em si, mas na entidade do grupo (classe) a que pertence. Se um objeto é um Animal não pode ser uma Planta. Se é Alto não pode ser Baixo. Se é quente, não é frio, etc… Aquilo que o objeto é dentro da sua classe é imutável e corresponde a uma forma de o distinguir dos outros na multidão.Um objeto pode ser um Gato e ser um Animal, e um Cão pode ser um Animal, mas um Cão nunca será um Gato. Inerentemente estamos abstraindo o conceito de Cão e Gato, para encontrar algo que eles têm em comum e chamar a isso o conceito de “Animal” que se destingue do conceito de Planta, por exemplo. Mas “Animal” não é algo que exista no mundo real. Não é algo que se pode tocar, é apenas um conceito.

Objetos concretos, instanciáveis e que vivem na memória são concretos, mas os conceitos que os definem e os

distinguem são apenas fruto da nossa inteligencia, da nossa abstração e compreensão desses conceitos. Afinal, quem nunca pensou que pessoas física e pessoa jurídica são duas classes diferentes de "pessoa" ? (Podemos ver que não , necessariamente, são)

Aquilo que o objeto é só pode ser uma coisa. O Objeto não tem múltiplas identidades, nem é esquizofrênico. Aquilo que ele é , é completamente definido; mesmo que por um conceito abstrato. É por isso que o conceito tão poderoso, é também tão escasso. Uma vez que você escolher o que o objeto é, ele o será para sempre e mais do que isso, nunca será nenhuma outra coisa.

O conceito de Interface é muito diferente. Não importa o que é o objeto, mas como ele sabe se comportar ou do que ele é capaz. Em diferentes contextos, o mesmo objeto se comporta de formas diferentes e é capaz de coisas diferentes. Estas perspetivas diferentes do mesmo objeto são aquilo que define uma forma de outros objetos trabalharem com aquele objeto, ou melhor, outras classes de objetos trabalharem com aquele objeto , naquele contexto. O que o objeto é, realmente não interessa, desde que ele se comporte como esperado.

Em Java chamamos isto de interface. Em sociologia isto é chamado de "Contrato Social". Um acordo que ninguém fez, mas que todos cumprem. A Interface é um contrato com o resto da sociedade das classes de objetos. É um contrato, uma confirmação eterna de que aquelas capacidades serão possíveis. A Interface, ou "contrato" é usada para diferentes coisas conforme o contexto e a finalidade, já que ela não designa o que o objeto é, mas o que ele pode/sabe fazer. Em primeiro lugar temos a interfaces marcadoras. Interfaces como java.io.Serializable que designam que o Objeto pode ser usado em certo contexto, mas que ele não tem nenhuma ação particular (método). Ele simplesmente segue as regras esperadas por outros. Em seguida temos as interfaces funcionais. Ou seja, interfaces que na realidade definem assinaturas de funções através de métodos de um objeto. Por exemplo, Comparable. Depois temos os contratos de serviço; muito usados em WebServices, EJB, e outros mecanismos orientados a serviços. A interface define os métodos que podem servir um propósito sendo que o estado do objeto é irrelevante. Temos ainda as interfaces que definem métodos que auferem algumas responsabilidade explicita ao objeto, como Clonable, por exemplo. Por fim, temos as interfaces taxonômicas que criam classificações alternativas e paralelas às das classes como por exemplo, CharSequence e List, que ao mesmo tempo que se utilizam o conceito de contrato passam a ideia de "identidade".

Em inglês a nomenclatura de interfaces é sempre um advérbio de modo (as interfaces acabam em "able" que a forma mais simples de criar advérbios de modo). As interfaces taxonômicas que se utilizam de nomes abstratos. Em português é mais difícil criar estes nomes. Securable (that can be secured) é simples , embora, "Segurável" parece esquisito. Mas ambos significam "que pode ser/ter seguro". É por estas e por outras que prefiro modelar em inglês, até porque não escrevemos se-faça-enquanto e sim if-do-while.

A interface de contrato é extremamente útil porque permite que um mesmo objeto interaja com outros em diferentes contexto e ambientes. Aumenta o polimorfismo e isso é sempre bom ("Programe para interfaces"). Já as interfaces taxonômicas, igualmente uteis em muitos aspetos já permitem definir hierarquias não tão rígidas quanto uma classe enquanto mesmo assim não proibem de criar linhagens mais fechadas. O exemplo mais claro disto é a API de coleções que é toda baseada em interfaces, sendo as classes abstratas que as implementam meros utilitários programáticos que não incluem nenhum sentido de identidade da natureza da classe. Esta é, aliás, uma forma muito útil de definir suas hierarquias. Comece com um interface, e explore o conceito. Quando você descobrir que as implementações sempre repetem alguns pormenores é hora de criar uma classe abstrata no meio para ajudar na limpeza do código. Quando verificar que afinal *sempre* precisará desses pormenores então você encontrou a sua classe principal da linhagem.

Claro que ainda ha quem não esteja satisfeito com o problema de não poder definir métodos em interfaces. Afinal o *bytecode* não proíbe isso, apenas o compilador. Então, temos alternativas como os mixin do Scala e as definições default do Java 8(? ou será 7? já nem sei depois de tanto vai e volta da Oracle). O ponto é que a simplificação é muito poderosa e mesmo assim podemos fazer grandes coisas com ela. Mesmo sem os truques dos mixins, existem centenas de API java para fazer quase tudo o que imaginar. O que significa que com outras estrutruas talvez fosse mais fácil escrever, mas com certeza não seria tão fácil modelar.

O Java deu vários passos à frente depois de dar um passo atrás em relação ao C++. A famosa máxima "menos é mais" , se aplica aqui, com toda a sua classe.

« Lead and Leader
Especialistas, Generalistas e os Outros »

16 comentários para “A Herança e a Interface”

1. **Rafael**, em July 1st, 2011 às 17:58 disse:

   Sergio,

   Hoje em dia por incrível que pareça o uso de herança por parte da maioria dos desenvolvedores me deixa com vergonha alheia. Não sei o porque do uso vasto de herança, acho que por preguiça de escrever código e de estudar. Quem leu o livro Java Efetivo (obrigatório para qualquer desenvolvedor Java) entendeu através de diversas abordagens o porque “programar para interfaces”.

   Outro ponto interessante que não vi no seu texto é o fato de que usar interfaces facilita o desenvolvimento baseado em testes. Hoje em dia isso é uma realidade e qualquer desenvolvedor sério tira proveito disso.

   Enfim, acho horroroso o uso de herança em quase todos os casos em que me deparei com ela. Isso é tão comum, que quase todos os projetos que participei eu vi um “GenericDAO” da vida. Existe um exemplo de herança pior do que esse?!

2. **sergiotaborda**, em July 1st, 2011 às 21:50 disse:

   Sim, acho que não ha exemplo pior (se bem que pessoa-fisica/pessoa-juridica não fica muito atraz). 🙂
   Sim, usar interfaces facilita muito o teste, contudo nem sempre e possível (ai que entram as ferramentas e artifícios como bytecode re-write). O que sim é fato é que as interfaces permitem desacoplar muito melhor e isso permite facilitar os testes.

3. **Laudelino**, em July 25th, 2011 às 11:47 disse:

   Em primeiro lugar é o primeiro comentário que faço em seu blog. Apesar de acompanhar esse blog (e os outros de sua autoria) a mais de dois anos, me senti obrigado a lhe parabenizar pelo tempo, conhecimento e dedicação dedicados para levar conceitos, no mínimo de conhecimento obrigatório, a desenvolvedores de toda parte do globo.
   O seu texto me fez lembrar do tempo que eu fazia uso abusivo da herança para “reaproveitar” código, sendo que muitas vezes hierarquias complexas eram criadas sem nenhuma necessidade. Após começar a ler o livro da katy sierra (o da certificação mesmo) e do cay hosrtamnn (OO Design & Patterns), na parte onde o uso de interface foi abordado, é que a minha cabeça realmente se abriu e foi quando eu realmente comecei a criar códigos com algum grau de facilidade de manutenção e reaproveitamento. Não quero dar a entender que com herança não se consegue isso também, mas com interface é possível de forma mais flexível, podendo se focar no contexto no qual o objeto está inserido, para a partir desse ponto analisar qual deve ser o comportamento adotado por esse mesmo objeto.
   Em suma o teu texto foi brilhante em abordar o tema, mais uma vez esclarecendo e educando a todos que assim como eu buscam ser um desenvolvedor melhor a cada dia que passa.
   Muito obrigado!

4. **Nilson**, em July 26th, 2011 às 21:58 disse:

   Sérgio Taborda,
   Comecei a observar seus comentários somente agora e justo no momento que mais preciso de entendimentos(herança, composição, interface) pois iniciei meu TCC agora(8º per Sist. de Informação) e como você deve imaginar minha cabeça está a mil com tantas opiniões tanto a favor e contra de cada conceito. Em virtude de estar no começo de minha modelagem(DER) das tabelas do meu projeto, me surgiu a dúvida sobre como modelar PFisica, Pjurídica…e em outro post li sobre seu comentário a respeito de utilizar cliente como solução, mas ainda não consegui “visualizar” de forma correta o emprego, e se você puder explicar com detalhes sobre essa modelagem será de uma importância inestimável.
   []’s e parabéns pela iniciativa.

5. **sergiotaborda**, em July 27th, 2011 às 08:01 disse:

   Nilson,

A forma mais simples de modelar pessoa física e jurídica é não modelar. Ou seja, não criar nenhum tipo de herança ou mecanismo complicado. O mais simples é simplesmente criar um entidade pessoa, que tem um identificador fiscal e tem um tipo (juridica ou fisica) e um numero de indentificação fiscal (NIF). Aqui sim ha uma modelagem a fazer. O NIF é que realmente muda. CPF e CNPJ são dois NIF. Uma herança simples de um objeto de valor (Value Object) para este caso é suficiente. Vc pode usar o tipo de pessoa para saber qual classe instanciar. Lembre-se que nem todas as pessoas são fisicas ou juridicas , por exemplo, perfeituras não têm cnpj nem cpf, logo, o NIF pode ser nulo ou existir algum outro numero identificador que não seja relacionado ao fiscal.
Não existe uma forma universal de modelar, e tudo depende do seu problema e do seu contexto. Contudo, existe formas erradas e complexas de modelar que não lhe trazem vantagem. Criar herança neste caso é uma delas.
Se vc precisar ou quiser muito ter duas classes, use um interface para cada um, mas o truque é sempre deixar as interfaces iguais e modificar os objetos/valores das propriedades para sejam eles a ter a herança (que é muito mais simples, porque objetos de valor são auto-contidos)

6. **Nilson**, em July 27th, 2011 às 19:06 disse:

   Sérgio,

   Obrigado pela resposta, vou tentar entender e modelar partindo dessa idéia, vou ver se consigo desenhar e te enviar pra ver se estou no caminho certo.
   A propósito, você comentou…"por exemplo, perfeituras não têm cnpj nem cpf,"…uma correção: Prefeitura, Estado, União, todos tem cnpjs. Apesar desses órgãos terem sempre algum identificador a mais ex: banco tem cnpj e um nº de identificação (Bradesco=237, HSBC=399…) não dá pra ter uma classe universal contendo todos esses atributos de cada um né ? afetaria o desempenho do banco numa query para saber qtos tem no sistema q são PF, PJ, Órg. Público, banco correto? não sei se a linha de raciocínio está correta…

7. **sergiotaborda**, em July 27th, 2011 às 20:50 disse:

   Hum… qual é o CNPJ do Estado de São Paulo ? E o da União ? E o da perfeitura de são paulo (cidade) ? É obvio que bancos têm cnpj porque são empresas, mas perfeituras não são empresas. Não me parece que tenham. Mas basta que você encontra um exemplo …
   O desempenho do banco é irrelevante ao modelo. Não se preocupe com isso. Preocupe-se com o modelo de objetos. Agora, o modelo de objetos java é diferente do modelo de objetos de modelagem. O de modelagem é muito próximo do negócio, o de java é muito próximo do codigo. O do java tem que ter uma performance aceitável, mas hoje em dia, desde que vc não invente alguma maluquice, ferramentas como o hibernate já lhe dão uma boa performance. Contudo, modelo de banco não é o que caracteriza um modelo de sistema. Ou seja, não é nisso que tem que pensar primeiro. O que tem que pensar primeiro é no modelo de negocio e depois no modelo de código. Se tiver , como lhe sugeri, um campo "tipo de pessoa" é trivial fazer o count. O meu ponto é que vc não deve usar os identificadores humanos (CPF, CNPJ, Inscrição estadual, etc..) para saber o tipo de pessoa. Esse campo tem que ser tão irrelevante quanto o nome ou o endereço da sede. É meramente informativo ao usuário; não ao sistema.

8. **Nilson**, em July 28th, 2011 às 10:20 disse:

   Sérgio,

   Comecei a modelar meu DER pensando que da forma que eu colocasse no modelo eu iria aproveitar pra gerar as classes pelo hibernate(classe de entidade + jsf), por isso a preocupação em criar tabelas bem estruturadas, para evitar correções muito bruscas no código…mas vou tentar me organizar conforme está orientando.
   Vou te passar 2 cnpj's.
   00394585000171 / 05903125000145
   Primeiro fiz um Doc. de visão + Diag de casos de uso, e agora passei pro DER onde estou parado pra entender melhor os conceitos.

9. **sergiotaborda**, em July 28th, 2011 às 23:22 disse:

   Governo do Estado de Rondônia de e Prefeitura de Porto Velho. Interessante. Não sabia disso.

10. **Nilson**, em July 29th, 2011 às 00:16 disse:

    Sérgio,
    Agora como te falei esses(e outros) órgãos podem ter identificações adicionais e baseado nisso é que vem na idéia modelar a classe pois como são identificadores exclusivos muitas vezes não conseguimos desatrelar nosso raciocínio…imagina eu que preciso fechar meu sistema para apresentar na banca final de novembro…perdi meu orientador(pediu demissão) e as aulas só voltam semana que vem…tô meio perdido nessa modelagem.

11. **Nilson**, em August 6th, 2011 às 13:09 disse:

    Sérgio,
    Você citou…"O mais simples é simplesmente criar um entidade pessoa, que tem um identificador fiscal e tem um tipo (juridica ou fisica) e um numero de indentificação fiscal (NIF)."…são 2 ou 3 atributos ? me lembrei também que podemos usar o atributo NIF para guardar o CEI(cadastro de empregador individual)é usado para obras temporárias da construção civil para recolhimento do inss e nesse caso não é necessário ter cnpj ou cpf.

12. **sergiotaborda**, em August 6th, 2011 às 22:55 disse:

    São apenas 2. Eu repeti o identificador fiscal
    Acho que o CEI seria usado apenas se fizer sentido no contexto. Por exemplo , no contexto de obras.
    O ponto é que podemos ter muitas caracteristicas para a pessoa/empresa/entidade, mas nenhuma delas serve para distinguir entre o tipo de pessoa/entidade. Por exemplo se estivessemos falando de clientes, todos eles podem ser clientes. Os diferentes tipos terão diferentes tratamentos no sistema em vários aspetos, desde enviar email promocional até calcular impostos. E para isso precisamos de um campo especifico para distinguir entre eles.

13. **Nilson**, em August 7th, 2011 às 19:17 disse:

    Sérgio,
    Então se eu criar as classes:
    Entidade(idEntidade,tipo,nif)
    Tipo_Entidade(pf,pj,orgao_publico,idEntidade(pk),tipo(pk))
    Criar e relacionar com o Tipo_Entidade:
    PFisica(atributos da pessoa fisica)
    PJuridica(atributos da pessoa juridica)
    OrgaoPublico(atributos do orgao publico)
    Mas na última explicação vc fala em criar um campo específico para distinguir entre eles..não entendi…tô no rumo ou bastante por fora ?

14. **sergiotaborda**, em August 10th, 2011 às 13:58 disse:

    Não sei se entendi certo, mas parece-me que vc quer criar uma classe para cada tipo de pessoa.
    A Pessoa teria então um id, um nif e um tipo. O tipo seria um enum, não uma outra entidade. por exemplo 0 = fisica, 1 = juridica , 2 = orgão publico.
    Se bem que vc mesmo argumentou que todos estes seria a mesma coisa.
    Você pode usar uma tabela suplementar para cada tipo ou uma tabela com todos os tipos. O hibernate suporta os dois. Eu usaria uma tabela para todos os tipos, mas dependendo do banco de dados, ele pode ser burrinho e ocupar espaço mesmo quando o campo é null. Ai cai num assunto de eficiencia e não de modelo.

    Sendo que todo o mundo está na mesma tabela, então é todo o mundo da mesma classe de dados. Mas não da mesma classe de modelo. É aqui que entra o conceito da interface. Você pode criar 3 interfaces que são proxies de um mesmo tipo de objeto. Vc controla quais métodos existe e não existem no tipo de pessoa pela interface e não pelo banco de dados ou pela entidade persistida.

    Então, por exemplo, se seu objeto de modelo de dados é Pessoa, vc teria métodos como
    Pessoa pessoa = …
    PessoaFisica pf = pessoa.asPessoaFisica();

Este método poderia dar exceção se o tipo do objeto não fosse o certo (no caso se pessoa.tipo não fosse 0).

Assim parece meio sem graça, mas se vc tiver um objeto que é uma pessoa como Cliente ou Fornecedor este modelo é mais util.

Cliente cliente = …
PessoaFisica pf = cliente.asPessoaFisica();

Neste caso usar herança é mais complexo porque Cliente também terá uma herança dele (poderia ter) e não ha como misturar as duas. Mas com a interface vc pode chavear entre as duas "views' quando quiser e fazer por exemplo:

Cliente cliente = …
PessoaFisica pf = cliente.asPessoaFisica();
Cliente cliente = pf.asCliente();

Este não é o modelo ORM padrão do Hibernate. Isto é um modelo util para as logicas de negocio. Usando hibernate provavelmente vc irá ter um serviço

Cliente cliente = …
Pessoa pessoa = servico.obtemPessoaPorCliente(cliente)
Cliente cliente = servico.obtemClientePorPessoa(pessoa)

Em que a mágica acontece no serviço e vc pode relacionar o cliente e a pessoa práticamente como quiser.

Pessoas sempre cumprem um papel no sistema (às vezes mais do que um : um cliente pode tb ser um fornecedor) e é esse papel que é modelado, não necessáriamente a estrutura de "pessoa" do mundo real.

15. **Nilson**, em August 28th, 2011 às 12:01 disse:

    Sérgio,

    Seguindo o assunto da modelagem, gostaria de lhe enviar meu modelo de classes com seus relacionamentos para que você, se possível, realizar suas considerações mas não tem como anexar aqui no post…

16. **sergiotaborda**, em August 31st, 2011 às 09:19 disse:

    Vamos continuar isto em outro post : Modelando do Zero. Lá tem o modelo que me enviou.

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