Metáforas para um Melhor Entendimento do Desenvolvimento de Software

A ciência da computação tem uma das linguagens mais ricas entre as áreas do conhecimento. Em que outra área você poderia caminhar em uma “sala estéril”, cuidadosamente controlada a 20°C e encontrar “vírus”, “cavalos de Tróia”, “worms”, “bugs”, “bombas”, “crashes”, “incêndios” e “erros fatais”?

Essas metáforas visuais descrevem fenômenos de software específicos. Metáforas igualmente vívidas descrevem fenômenos mais amplos e você pode usá-las para melhorar seu entendimento do processo de desenvolvimento de software.

2.1 A importância das metáforas

Importantes desenvolvimentos frequentemente surgem a partir de analogias. Comparando um assunto do qual você pouco entende com outro assunto semelhante, do qual tem maior domínio, poderá ter ideias que resultem em uma melhor compreensão dessa matérias que lhe é menos familiar. **O uso da metáfora é denominado “modelagem”.**

Em geral, o **poder dos modelos** é que eles são vívidos e podem ser assimilados como **conjuntos conceituais**. Eles sugerem **propriedades, relacionamentos** e **áreas adicionais de questionamento**.

Conforme poderíamos esperar, algumas metáforas são mais eficazes do que outras. Uma boa metáfora é simples, relaciona-se bem com outras metáforas relevantes e explica grande parte da evidência experimental e outros fenômenos observados.

Considere o exemplo de uma pedra pesada que balança presa por um barbante. Antes de Galileu, um aristotélico observando a pedra balançando pensaria que um objeto pesado se movia naturalmente de uma posição mais alta para um estado de repouso em uma posição mais baixa. Segundo esse aristotélico, o que a pedra estava realmente fazendo era cair com dificuldade. Galileu, no entanto, ao observar a pedra balançando, visualizou um pêndulo. Ele interpretou que o que a pedra estava realmente fazendo era repetir o mesmo movimento ininterruptamente, de modo quase perfeito.

Os poderes sugestivos dos dois modelos são bastante diferentes. O aristotélico que viu a pedra balançando como um objeto em queda observaria o peso da pedra, a altura para a qual ela teria sido elevada e o tempo que demorava para ficar em repouso. Para o modelo do pêndulo de Galileu, os fatores proeminentes eram diferentes. Galileu observou o peso da pedra, o raio de oscilação do pêndulo, o deslocamento angular e o tempo de cada oscilação. Na verdade, Galileu descobriu leis que os aristotélicos não poderiam identificar, pois o modelo deles os levava a ver fenômenos diferentes e a fazer perguntas também diferentes.

As **metáforas contribuem** para um melhor **entendimento dos problemas de desenvolvimento de software** da mesma maneira que contribuem para uma melhor compreensão das questões científicas. Em seu discurso ao receber o prêmio *Turing Award* de 1973, Charles Bachman descreveu a mudança da visão geocêntrica do universo, predominante durante muito tempo, para uma visão heliocêntrica. O modelo geocêntrico de Ptolomeu durou 1.400 anos, sem contestação lógica. Então, em 1543, Copérnico apresentou a teoria heliocêntrica, segundo a qual o sol, e não a Terra, seria o centro do universo. Basicamente, essa mudança nos modelos mentais levou à descoberta de novos planetas, à reclassificação da lua como um satélite, em vez de um planeta, e a um entendimento diferente do lugar da espécie humana no universo.

Bachman comparou a mudança do pensamento ptolomaico para o copernicano na astronomia com a mudança na programação de computadores no início dos anos 70. Quando ele estabeleceu essa comparação, em 1973, o processamento de dados estava mudando a visão sobre os sistemas de informação, antes centralizada no computador;- para uma visão centralizada nos bancos de dados. Bachman mostrou que os patriarcas do

processamento de dados queriam ver todos os dados como um fluxo sequencial de cartões passando por um computador (a visão centralizada no computador). A mudança foi deslocar o enfoque para um *pool* de dados, no qual o computador estivesse atuando (uma visão orientada para banco de dados).

> "O valor das metáforas não deve ser subestimado. As metáforas têm como virtude um comportamento esperado que é compreendido por todos. A comunicação desnecessária e os mal-entendidos são reduzidos.
>
> O aprendizado e a educação são mais rápidos. Com efeito, as metáforas são uma maneira de interiorizar e abstrair conceitos, permitindo que o pensamento de alguém esteja em um plano mais alto e que os enganos de baixo nível sejam evitados." (Fernando J.Corbató)

Hoje é difícil imaginar que alguém ainda acredite que o sol gira em torno da Terra. Analogamente, é difícil imaginar que nos dias atuais um programador ainda creia que todos os dados poderiam ser vistos como um fluxo sequencial de cartões. Nos dois casos, descartada a teoria antiga, parece incrível que alguém já tenha acreditado nela. Ainda mais fantástico do que isso: os adeptos da teoria antiga julgavam a nova tão ridícula, na época, quanto você acredita que a antiga o é agora.

A visão geocêntrica do universo impediu o progresso dos astrônomos que se apegaram a ela, depois que uma teoria melhor estava disponível. Da mesma forma, a visão centralizada no computador, do universo da computação, impediu o progresso dos cientistas dessa área que a ela se agarraram, depois que a teoria centralizada no banco de dados estava disponível.

É bastante tentador tornar superficial o poder das metáforas. Para cada um dos exemplos anteriores, a resposta natural é dizer: "Bem, é claro que a metáfora correta é mais útil. A outra estava errada!" Embora essa seja uma reação natural, ela é simplista. A história da ciência não é uma série de trocas da metáfora "errada" para a "certa". É uma série de mudanças de metáforas "piores" para "melhores", da menos abrangente para a mais abrangente, da sugestiva em uma área para a sugestiva em outra.

O desenvolvimento de software é um **campo mais novo** do que a maioria das outras ciências. Ele ainda não está maduro o suficiente para possuir um conjunto de metáforas padronizadas. Consequentemente, ele tem uma profusão de metáforas complementares e conflitantes. Algumas são melhores do que outras. **A sua capacidade de entender as metáforas determina o quanto assimilará o desenvolvimento de software**.

2.2 Como utilizar metáforas de software

Uma metáfora de software é mais parecida com uma lanterna do que com um mapa. Ela não indica a você onde encontrar a resposta; ela ensina como procurá-la. **Uma metáfora serve mais como uma heurística do que como um algoritmo**.

Algoritmo é um **conjunto de instruções** bem-definidas para executar uma determinada tarefa. Um algoritmo é previsível, determinístico e não está sujeito a mudança. Um algoritmo mostra a você como ir do ponto A para o ponto B sem mudanças de direção, sem desvios para os pontos D, E e F, e sem parada para tomar uma xícara de café.

Heurística é uma técnica que o ajudará a **procurar uma resposta**. Os resultados obtidos a partir dela estão sujeitos a mudança, pois uma heurística indica apenas como procurar e não o que encontrar. Ela não informa como ir diretamente do ponto A para o ponto B; ela pode nem mesmo saber onde estão esses pontos. Poderíamos dizer que uma heurística é um algoritmo em trajes de palhaço. Ela é menos previsível, é mais divertida e vem sem garantia de 30 dias ou seu dinheiro de volta.

Eis aqui um algoritmo para ir de carro até a casa de alguém: pegue a BR-153,

direção sul, para Brasília. Depois, pegue a saída Jardim Guanabara e percorra 7 quilômetros. Vire à direita no sinal junto ao supermercado e, em seguida, dobre a primeira rua à esquerda. Entre na garagem da casa amarela à esquerda, na quadra 42.

Aqui está uma heurística para chegar na casa de alguém: encontre a última carta que enviamos para você. Vá até a cidade que consta no endereço do remetente. Quando sobre como usar chegar na cidade, pergunte a alguém onde é nossa casa. Todo mundo nos conhece. Caso não encontre ninguém, ligue-nos de um telefone público e iremos até você.

A **diferença entre um algoritmo e uma heurística** é sutil e os dois termos possuem suas coincidências. A principal diferença entre os dois é o **nível de procedimento indireto da solução**. Um algoritmo fornece as instruções diretamente. Uma heurística mostra como descobrir as instruções por conta própria ou, pelo menos, onde procurá-las.

Possuir instruções que indicassem a você exatamente como resolver seus problemas de programação certamente tornaria a tarefa de programar mais fácil e os resultados mais previsíveis. Mas a ciência da programação ainda não está tão avançada assim e talvez nunca esteja. A parte mais desafiadora da programação é conceitualizar o problema; muitos erros de programação são erros conceituais. Como cada programa é conceitualmente único, é difícil ou até impossível criar um conjunto geral de instruções que leve a uma solução para cada caso. Assim, **saber como encarar os problemas** em geral é no mínimo tão valioso quanto conhecer soluções específicas para problemas específicos.

Como você pode usar metáforas de software? Use-as para **ter ideias** a respeito de seus problemas e processos de programação. Use-as também para ajudar a refletir sobre suas atividades de programação e para descobrir formas mais eficazes de fazer as coisas. Você não poderá examinar uma linha de código e dizer que ela viola uma das metáforas. Com o passar do tempo, contudo, a pessoa que usa metáforas para ilustrar o processo de desenvolvimento de software será vista como alguém que possui um maior entendimento a respeito de programação e produz um código melhor mais rapidamente do que quem não as usa.

2.3 Metáforas de software comuns

Muitas metáforas confusas têm surgido em torno do desenvolvimento de software. David Gries afirma que **escrever software é uma ciência** (1981). Donald Knuth ressalta que **é uma arte** (1998). Watts Humphrey conclui que **é um processo** (1989). P. J. Plauger e Kent Beck esclarecem que **é como dirigir um carro**, embora tirem conclusões praticamente opostas (Plauger 1993, Beck 2000). Já na visão de Alistair Cockburn, **é um jogo** (2002). Para Eric Raymond, compara-se a **um bazar** (2000). Segundo Andy Hunt e Dave Thomas **é como jardinagem**. Paul Heckel garante que é como **filmar Branca de Neve e os Sete Anões** (1994). E Fred Brooks entende que **é como trabalhar na agricultura, caçar lobisomens ou submergir com dinossauros em uma poça de alcatrão** (1995). Quais são as melhores metáforas?

Caligrafia de software: escrita de código

A metáfora mais primitiva para desenvolvimento de software é proveniente da expressão "**escrever código**". A metáfora da escrita sugere que desenvolver um programa é como escrever uma carta descontraída - você se senta com uma caneta na mão e uma folha de papel à frente, e a escreve do início ao fim. Ela não exige nenhum planejamento formal e você decide o que quer relatar à medida que vai escrevendo.

Muitas ideias derivam da metáfora da escrita. Jon Bentley afirma que você deve

ser capaz de sentar-se junto à fogueira com um copo de conhaque, um bom charuto e seu cão de caça predileto para apreciar um “programa culto” da maneira como faria com um bom romance. Os programadores frequentemente falam sobre "legibilidade do programa".

Em relação ao trabalho de uma pessoa ou a projetos de pequena escala, a metáfora da escrita da carta funciona adequadamente, mas para outros propósitos ela deixa a festa antes da hora - não descreve o desenvolvimento de software de forma completa nem adequada. Normalmente, a escrita é uma atividade realizada por uma única pessoa, enquanto um projeto de software em geral envolverá muitas, com muitas responsabilidades diferentes. Quando termina de escrever uma carta, você a coloca em um envelope e a envia pelo correio. Você não pode mais alterá-la e, para todos os efeitos, ela está concluída. Software não é tão difícil de se alterar e dificilmente está concluído. Até 90% do trabalho de desenvolvimento em um sistema de software típico se dá após o seu lançamento inicial (Pigoski 1997). Na escrita, a originalidade é muito importante. Na construção de software, tentar criar um trabalho verdadeiramente original é quase sempre menos eficaz do que concentrar-se na reutilização de ideias de projeto, código e casos de teste de trabalhos anteriores. Em resumo, a metáfora da escrita envolve um processo de desenvolvimento de software simples e rígido demais para ser saudável.

Infelizmente, a metáfora da escrita da carta foi perpetuada por um dos livros sobre software mais populares do planeta, *The Mythical Man-Month*, de Fred Brooks (Brooks 1995). Brooks diz, “Planeje jogar um fora; você fará isso de qualquer maneira". Isso evoca a imagem de um grande número de rascunhos, escritos pela metade, jogados no lixo, como se vê na Figura 2-1.

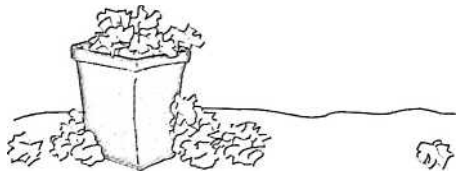

Figura 2-1 A metáfora da escrita da carta sugere que o processo de software baseia-se na dispendiosa estratégia de tentativa e erro, em vez de em um planejamento e design criteriosos.

Cultivo de software: criação de sistemas

Em contraste com a rígida metáfora da escrita, alguns desenvolvedores de software dizem que você deve imaginar a criação de software como algo parecido com o plantio de sementes e o desenvolvimento da lavoura. Você **planeja uma parte, codifica-a, testa-a e insere-a no sistema aos poucos**. Assim, avançando em pequenos passos, você minimiza a dificuldade que pode ter em dado momento.

A ideia de avançar um pouquinho de cada vez pode ter certa semelhança com a maneira como a plantação cresce na lavoura, mas a analogia do cultivo é fraca e não é informativa. É difícil estender a metáfora do cultivo além da simples ideia de se fazer as coisas aos poucos. Se você adotar a metáfora do cultivo, poderá se ver falando sobre a fertilização do plano do sistema, podar o projeto detalhado, aumentar o rendimento do código por meio do tratamento eficiente da terra e colher o código em si. Você falará sobre alternância em uma safra de C++, em vez de cevada, sobre permitir que a terra descanse por um ano, para aumentar o suprimento de nitrogênio no disco rígido.

O ponto fraco da metáfora do cultivo de software é a sugestão de que **você não tem nenhum controle direto sobre o modo como o software se desenvolve**. Você planta as sementes de código na primavera. Se o Almanaque do Produtor Rural assim o permitir, você terá uma abundante safra de código no outono.

Cultura de ostras de software: incremento de sistemas

Às vezes as pessoas falam sobre desenvolvimento de software, quando na verdade querem se referir a incremento de software. As duas metáforas estão intimamente relacionadas, mas crescimento de software é a imagem mais incisiva. "Incremento", no caso de você não ter um dicionário à mão, significa qualquer desenvolvimento ou aumento no tamanho por meio de um acréscimo ou inclusão gradual externa. O incremento descreve, por exemplo, a maneira como uma ostra produz uma pérola, acrescentando gradualmente pequenas quantidades de carbonato de cálcio.

Isso não quer dizer que você precisa aprender a fazer código a partir de sedimento transportado pela água; significa, isto sim, que você precisa aprender a **aumentar seus sistemas de software gradativamente**, uma pequena quantidade por vez. Outros termos intimamente relacionados a incremento são "incrementar, "iterativo", "adaptável" e "evolutivo". Projeto, construção e teste incrementais são alguns dos mais poderosos conceitos de desenvolvimento de software disponíveis.

No desenvolvimento incremental, você primeiramente cria a versão mais simples possível do sistema que irá executar. Ele não precisa aceitar entrada real, não precisa realizar manipulações reais nos dados, não precisa produzir saída real - basta que possua um esqueleto suficientemente forte para conter o sistema real enquanto é desenvolvido. Ele poderia chamar classes fictícias para cada uma das funções básicas que você tiver identificado. Esse início básico é exatamente como uma ostra começa a fazer uma pérola: com um pequeno grão de areia.

Após ter formado o esqueleto, pouco a pouco você vai acrescentando músculos e pele. Você então transforma cada uma das classes fictícias em classes reais. Em vez de seu programa fazer de conta que aceita entrada, você insere código que aceita entrada real. Em vez de seu programa fazer de conta que produz saída, você insere código que produz saída real. E assim vai acrescentando um pouco de código por vez. até construir um sistema totalmente funcional.

A vantagem da metáfora incremental é que ela não promete demais. Ela é mais difícil de se estender inadequadamente do que a metáfora do cultivo. A imagem de uma ostra formando uma pérola é uma boa maneira de visualizar o desenvolvimento incremental, ou incremento.

Construção de software: edificação de software

A imagem da "edificação" de software é mais útil do que a da "escrita" ou do "cultivo" de software. Ela é compatível com a ideia do incremento de software e oferece uma orientação mais detalhada. A edificação de Software **envolve vários estágios de planejamento, preparação e execução, que variam no tipo e no grau**, dependendo do que estiver sendo construído. Quando você examina a metáfora, acaba encontrando várias outras paralelas.

Construir uma pequena torre usando latinhas de cerveja exige uma mão firme, uma superfície plana e 10 latas em perfeito estado. No entanto, construir uma torre 100 vezes maior não exige apenas 100 vezes o número de latinhas de cerveja. Exige um tipo completamente diferente de planejamento e construção.

Se você estiver construindo uma estrutura simples - como uma casinha para seu cachorro Fido –, pode ir até a madeireira e comprar madeira e pregos. No final da tarde, você terá uma casa nova para o Fido. Mas caso você esqueça de fazer a porta, ou cometer algum outro equívoco, isso não será um grande problema: basta corrigir o erro ou mesmo recomeçar o trabalho desde o início. Você terá perdido apenas parte de uma tarde. Essa estratégia livre também se aplica a pequenos projetos de software. Se você usar o projeto errado para 1.000 linhas de código, poderá refazer ou começar

tudo de novo, sem maiores perdas.

Já no caso de você estar construindo uma casa, o processo de edificação é mais complicado e o mesmo ocorre em relação às consequências de um projeto malfeito. Primeiramente, você precisa decidir que tipo de casa deseja construir - no desenvolvimento de software, análogo à **definição do problema**. Em seguida, você e um arquiteto precisam propor um projeto geral e aprová-lo. Isso é semelhante ao **projeto arquitetônico do software**. Você desenha plantas detalhadas e contrata um empreiteiro. Isso é semelhante ao **projeto detalhado do software**. Você prepara o terreno da construção, assenta os alicerces, constrói a armação básica, levanta as paredes, firma o teto e prepara as conexões hidráulica e elétrica. Isso se assemelha à **construção do software**. Quando a maior parte da casa estiver pronta, vêm os paisagistas, pintores e decoradores para melhorar as condições de sua propriedade, a casa que você construiu. Isso se assemelha à **otimização do software**. Ao longo de todo o processo de construção da sua residência, vários fiscais se apresentam para verificar o local, a fundação, a armação, a instalação elétrica e outros itens passíveis de inspeção. Isso é semelhante às **análises e inspeções de software**.

A maior complexidade e tamanho implicam em consequências mais graves nos dois tipos de atividades. Na construção de uma casa, os materiais costumam ser caros, mas a principal despesa está na mão-de-obra. Desprender uma parede (sendo a casa de madeira) e movê-la um metro se torna caro não porque você desperdiça muitos pregos, mas porque precisará pagar aos trabalhadores na obra o tempo extra envolvido nessa atividade. Portanto, seu projeto deverá ser o mais completo possível, para que não haja perda de tempo na correção de erros que poderiam ter sido evitados. Na construção de um produto de software, os materiais são mais baratos, mas os custos de mão-de-obra também são caros. Mudar o formato de um relatório é tão caro quanto mover uma parede em uma casa, pois o principal componente do custo em ambos os casos é o **tempo das pessoas** envolvidas na atividade.

Que outros paralelos as duas atividades compartilham? Na construção de uma casa, você não tentaria construir coisas que pode comprar prontas. Você comprará então uma máquina de lavar roupas e uma secadora, uma lava-louças, um refrigerador e um freezer. Você também comprará armários pré-fabricados, balcões, janelas, portas e acessórios para o banheiro. Ao construir um sistema de software, fará a mesma coisa. Você fará uso extensivo de recursos de linguagem de alto nível, em vez de escrever seu próprio código em nível de sistema operacional. Você também poderá usar bibliotecas prontas de classes contêineres, funções científicas, classes de interface com o usuário e classes de manipulação de banco de dados. Não faria sentido codificar coisas que você pode comprar prontas.

Contudo, no caso de estar construindo uma residência elegante, com móveis de primeira linha, poderá ter armários personalizados. Você poderia ter uma máquina lava-louças, um refrigerador e um freezer embutidos, para ficarem harmonicamente no mesmo padrão que os armários. Você também poderia colocar janelas exclusivas, em formatos e tamanhos incomuns. Essa personalização tem paralelos no desenvolvimento de software. Ao construir um produto de software de primeira classe, poderá criar suas próprias funções científicas para obter uma melhor velocidade ou precisão. Poderá construir suas próprias classes contêineres, classes de interface com o usuário e classes de banco de dados, para proporcionar ao seu sistema uma aparência e um comportamento transparente e perfeitamente compatível.

Tanto a construção de uma casa como a construção de software se beneficiam de níveis apropriados de planejamento. Se você construir o software na ordem errada, será difícil codificar, testar e depurar. Ele poderá demorar mais tempo para ser concluído ou o projeto poderá desintegrar-se, pois o trabalho de todo mundo é complexo demais e, portanto, confuso demais, quando tudo é combinado.

Planejamento criterioso não significa necessariamente planejamento exaustivo ou excessivo. No caso da sua nova residência, pode planejar os suportes estruturais e posteriormente decidir se colocará piso de madeira de lei ou carpete, a cor que irá pintar as paredes, o tipo de cobertura para o telhado, etc. Um projeto com planejamento bem-feito aumenta sua capacidade de mudar de ideia a respeito dos detalhes, mais tarde. Quanto mais experiência você tiver com o tipo de software que estiver construindo, com mais alternativas poderá contar. Você simplesmente precisa ter certeza de que planejou o suficiente para que problemas não ocorram posteriormente.

A analogia da construção também ajuda a explicar por que diferentes projetos de *software* se beneficiam de diferentes estratégias de desenvolvimento. Na obra da sua casa, caso você estivesse construindo uma despensa ou um galpão para guardar ferramentas, por certo usaria níveis de planejamento, *design* e controle de qualidade diferentes do que se estivesse construindo um centro médico ou um reator nuclear. Igualmente seriam outras as estratégias no caso de construir uma escola, um arranha-céu ou uma casa de três quartos. Do mesmo mod, no *software* geralmente você pode usar estratégias de desenvolvimento flexíveis e leves, mas às vezes precisará de estratégias rígidas e pesadas para atingir objetivos de segurança e outros.

Fazer **alterações no *software*** apresenta um outro paralelo com a construção civil. Mover uma parede por uma distância de um metro custará mais se ela for de sustentação do que se for apenas uma divisória entre compartimentos. Analogamente, fazer alterações estruturais em um programa custa mais do que adicionar ou excluir recursos periféricos.

Por fim, a analogia da construção proporciona compreensão em projetos de *software* extremamente grandes. Como a pena por uma falha em uma estrutura de grandes proporções é severa, a estrutura precisa ser muito bem-feita. Os construtores criam e inspecionam seus planos cuidadosamente. Eles constroem usando margens de segurança: é melhor pagar 10% a mais por um material que garante maior resistência do que ver um arranha-céu desmoronando. O controle dos prazos é um aspecto que tem importância fundamental. Quando o Empire State Building foi construído, cada caminhão de entrega tinha o tempo cronometrado de 15 minutos para descarregar. Qualquer caminhão que não estivesse no seu respectivo lugar, no momento certo, provocaria o atraso do projeto inteiro.

Do mesmo modo, para projetos de *software* excepcionalmente grandes é necessário um planejamento maior do que para projetos apenas grandes. Capers Jones relata que um sistema de *software* com um milhão de linhas de código exige em média 69 tipos de documentação (1998). Normalmente, a especificação de requisitos de tal sistema teria cerca de 4.000 a 5.000 páginas e a documentação do projeto do *software* pode ter facilmente de duas a três vezes o volume dos requisitos. É bastante improvável que uma pessoa consiga entender o *design* completo de um projeto desse porte - ou mesmo o leia integralmente. Um maior grau de preparação é apropriado.

Economicamente falando, nós construímos projetos de *software* comparáveis ao Empire State Building e são necessários controles técnicos e gerenciais de envergadura semelhante.

Aplicação de técnicas de software: a caixa de ferramentas intelectual

Profissionais eficientes no desenvolvimento de *software* de alta qualidade passaram anos acumulando dezenas de técnicas, truques e encantamentos mágicos. As técnicas não são regras; elas são ferramentas analíticas. Um bom artista conhece a ferramenta certa para o trabalho e sabe como usá-la corretamente. O mesmo acontece com os programadores. Quanto mais você aprende sobre programação, mais preenche sua

caixa de ferramentas mental com ferramentas analíticas e com o conhecimento de quando e como usá-las de forma correta.

Em se tratando de *software,* às vezes os consultores dizem para você adotar certos métodos de desenvolvimento de *software,* excluindo outros. Isso é lamentável, porque se você adotar uma única metodologia, verá o mundo inteiro em termos dessa metodologia. Em alguns casos, você perderá oportunidades de usar outros métodos mais convenientes para seu problema atual. A metáfora da caixa de ferramentas ajuda a manter todos os métodos, técnicas e dicas em perspectiva - prontos para uso. quando apropriado.

Combinando metáforas

Como as metáforas são heurísticas e não algoritmos, elas não são mutuamente exclusivas. Você pode usar as metáforas do incremento e da construção. Pode usar a da escrita, se quiser, e combiná-la com as metáforas do ato de dirigir, da caçada de lobisomens ou do mergulho na poça de alcatrão com dinossauros. Use a metáfora, ou a combinação de metáforas, que estimule seu próprio pensamento ou que se comunique bem com as outras pessoas de sua equipe.

O uso de metáforas é um negócio impreciso. Você precisa estendê-las para tirar proveito das idéias heurísticas que elas fornecem. Mas se você estendê-las demais ou na direção errada, elas o enganarão. Assim como você pode usar mal qualquer ferramenta poderosa, também pode usar mal as metáforas, mas o poder delas as tornam elementos valiosos de sua caixa de ferramentas intelectual.

Pontos-chave

- As metáforas são heurísticas e não algoritmos. Assim, elas tendem a ser um pouco descuidadas.
- As metáforas o ajudam a entender o processo de desenvolvimento de *software* relacionando-o com outras atividades que você já conhece.
- Algumas metáforas são melhores do que outras.
- Tratar a construção de *software* como se trata a construção civil sugere a necessidade de uma preparação cuidadosa e ilustra a diferença entre grandes e pequenos projetos.
- Pensar nas práticas de desenvolvimento de *software* como ferramentas de uma caixa de ferramentas intelectual sugere, ainda mais, que todo programador possui muitas ferramentas e que nenhuma delas se adapta a todos os trabalhos. Escolher a ferramenta certa para cada tipo de problema é um dos segredos para você se tornar um programador eficiente.
- As metáforas não são mutuamente exclusivas. Use a combinação de metáforas que funcionar melhor para seu caso.