4. Principais Decisões de Construção

Após você ter se certificado de que uma base apropriada foi estabelecida para a construção, a preparação então se volta para as decisões específicas dessa construção. Este tópico focaliza as preparações pelas quais os programadores individuais e os líderes técnicos são responsáveis, direta ou indiretamente.

4.1 Escolha da linguagem de programação

> *Liberando-se a mente de todo trabalho desnecessário, uma boa notação a deixa livre para concentrar-se em problemas mais avançados e, com efeito, aumenta o poder mental da raça. Antes da introdução da notação arábica, a multiplicação era difícil e a divisão - mesmo de números inteiros - exigia as mais altas faculdades matemáticas. Provavelmente nada no mundo moderno teria espantado mais um matemático grego do que saber que... uma enorme proporção da população da Europa Ocidental poderia efetuar uma operação de divisão com os maiores números. Esse fato representaria para ele uma impossibilidade total... Nosso poder de cálculo moderno, com frações decimais, é o resultado quase milagroso da descoberta gradual de uma notação perfeita.*
>
> *-Alfred North Whitchcad*

A linguagem de programação na qual o sistema será implementado deve ser de grande interesse, pois você estará imerso nela do início ao fim da construção.

Estudos têm mostrado que a escolha da linguagem de programação afeta a produtividade e a qualidade do código de diversas maneiras.

Os programadores são mais produtivos usando uma linguagem familiar do que usando uma linguagem desconhecida. Dados mostram que os programadores que trabalham com uma linguagem que já usam há três anos ou mais são cerca de 30% mais produtivos do que aqueles com experiência equivalente, mas iniciantes em uma linguagem.

Os programadores que trabalham com linguagens de alto nível alcançam maior produtividade do que aqueles que trabalham com linguagens menos abstratas. Linguagens como C++, Java, Smalltalk e Visual Basic têm a reputação de otimizar a produtividade, a confiabilidade, a simplicidade e a compreensibilidade por fatores de 5 a 15 em relação às linguagens menos abstratas, como *assembly* e C. Você economiza tempo quando não precisa ter uma cerimônia de comemoração sempre que um comando em C faz o que deveria fazer. Além disso, as linguagens de nível mais alto são mais expressivas do que as linguagens de nível mais baixo. Cada linha de código expressa mais elementos.

Algumas linguagens expressam conceitos de programação melhor do que outras. Você pode traçar um paralelo entre linguagens naturais, como o inglês, e linguagens de programação, como Java e C++. No caso das linguagens naturais, os linguistas Sapir e Whorf formularam hipóteses de um relacionamento entre o poder expressivo de uma linguagem e a capacidade de ter certos pensamentos. A hipótese de Sapir-Whorf diz que sua capacidade de ter certos pensamentos depende do conhecimento de palavras capazes de expressar o pensamento. Se você não conhece as palavras, não pode expressar o pensamento e nem mesmo poderia ser capaz de formulá-lo (Whorf 1956).

Analogamente, os programadores podem ser influenciados por suas linguagens. As palavras disponíveis em uma linguagem de programação para expressar seus pensamentos de programação certamente determinam o modo como você expressa suas ideias e poderiam até determinar quais delas pode expressar.

A evidência do efeito das linguagens de programação sobre o pensamento dos

programadores é comum. Atente para esta história típica: “Estávamos escrevendo um novo sistema em C++, mas a maioria de nossos programadores não tinha muita experiência nessa linguagem. A experiência deles era com Fortran. Eles escreviam o código que compilava em C++, mas na verdade estavam escrevendo em Fortran disfarçado. Eles forçavam a linguagem C+ + para simular os maus recursos da linguagem Fortran (como instruções “go to” e variáveis globais) e ignoravam o rico conjunto de recursos orientados a objetos da linguagem C+ +".

4.2 Convenções de programação

Em *software* de alta qualidade, você pode ver um relacionamento entre a integridade conceitual da arquitetura e sua implementação de baixo nível. A implementação deve ser compatível com a arquitetura que a governa e compatível internamente. Esse é o objetivo das diretrizes de construção para nomes de variável, nomes de classe, nomes de rotina, convenções de formatação e convenções para comentários.

Em um programa complexo, as **diretrizes arquitetônicas** fornecem ao programa uma **estabilidade estrutural** e as **diretrizes de construção** proporcionam **harmonia de baixo nível**, articulando cada classe como uma parte exata de um projeto completo. Qualquer programa de grande porte exige uma estrutura de controle que unifique os detalhes de sua linguagem de programação. Parte da beleza de uma grande estrutura é a maneira pela qual suas partes detalhadas corroboram as implicações de sua arquitetura. Sem uma disciplina unificadora, sua criação será uma mistura desorganizada de variações de estilo. Tais variações sobrecarregam seu cérebro - e somente com a finalidade de entender as diferenças de estilo de codificação que são basicamente arbitrárias. Um dos segredos para obter-se êxito na programação é evitar variações arbitrárias, para que seu cérebro possa ficar livre para focalizar as variações realmente necessárias.

E se você tivesse um excelente projeto para uma pintura, mas uma parte fosse clássica, outra impressionista e outra cubista? Ele não teria integridade conceitual, independentemente do quanto você seguisse seu projeto principal. Ele se pareceria com uma colagem. Um programa também precisa de integridade de baixo nível.

Antes que a construção comece, explique claramente as convenções de programação que você vai usar. Os detalhes da convenção de codificação estão em tal nível de precisão que é quase impossível alterá-los no *software* após ele ser escrito.

4.3 Sua posição na onda tecnológica

Ao longo de minha carreira, vi a estrela do PC em ascensão, enquanto a estrela do computador de grande porte desaparecia no horizonte. Vi programas de GUI substituírem os programas baseados em caracteres. E vi a Web ascender, enquanto o Windows declinava. Só posso supor que alguma tecnologia nova estará em ascendência e a programação da Web estará quase no fim. Esses ciclos ou ondas da tecnologia subentendem diferentes práticas de programação, dependendo de onde você se encontrar na onda.

Em ambientes tecnologicamente maduros - os que estão no fim da onda, como no caso da programação para a Web em meados do ano 2000 –, tiramos proveito de uma rica infraestrutura de desenvolvimento de *software.* Os ambientes que encontram-se no fim da onda oferecem numerosas opções de linguagem de programação, verificação de erros abrangente para código escrito nessas linguagens, ferramentas de depuração poderosas e otimização de desempenho automática e confiável. Os compiladores são praticamente isentos de erros. As ferramentas são bem documentadas na literatura do fornecedor, em livros e em artigos de outros fornecedores, e em amplos recursos da Web. As

ferramentas são integradas; portanto, você pode trabalhar na interface com o usuário, banco de dados, relatórios e lógica comercial dentro de um único ambiente. Se tiver problemas, você pode encontrar prontamente dicas de ferramentas descritas em FAQs. Muitos consultores e aulas de treinamento também estão disponíveis.

Em ambientes que estão no início da onda - como no caso da programação para a Web em meados dos anos 90, por exemplo - a situação é oposta. Poucas opções de linguagem de programação estão disponíveis e essas linguagens tendem a estar repletas de erros e mal documentadas. Os programadores passam um período de tempo significativo simplesmente tentando descobrir como a linguagem funciona, em vez de escrever um código novo. Eles também gastam incontáveis horas contornando erros nos produtos da linguagem, no sistema operacional subjacente e em outras ferramentas. As ferramentas de programação nos ambientes que estão no início da onda tendem a ser primitivas. Depuradores podem nem mesmo existir e otimizadores de compilador ainda são apenas um vislumbre aos olhos de algum programador. Os fornecedores revisam sua versão do compilador frequentemente e parece que cada nova versão estraga partes significativas de seu código. As ferramentas não são integradas e, assim, você tende a trabalhar com diferentes ferramentas para interface com o usuário, banco de dados, relatórios e lógica comercial. As ferramentas tendem a não ser muito compatíveis e você pode consumir uma quantidade significativa de seu trabalho apenas para manter a funcionalidade existente funcionando contra o ataque furioso dos lançamentos de compiladores e bibliotecas. Se você tiver problemas, existirá na Web uma literatura de referência de alguma forma, mas ela nem sempre será confiável e, se a literatura disponível for um guia, sempre que você encontrar um problema parecerá que é o primeiro a descobri-lo.

Esses comentários podem parecer uma recomendação para evitar programação no início da onda, mas não é esse o objetivo. Alguns dos aplicativos mais inovadores surgem de programas do início da onda. A questão é que o modo como você gasta seus dias de programação dependerá de onde está na onda tecnológica. Se você estiver na parte final da onda, poderá planejar passar a maior parte do seu dia firmemente escrevendo nova funcionalidade. Se você estiver na parte inicial da onda, poderá supor que vai gastar uma parte considerável de seu tempo tentando descobrir os recursos não documentados de sua linguagem de programação, depurando erros que revelam-se ser defeitos no código da biblioteca, revisando código para que funcione com um novo lançamento da biblioteca de algum fornecedor e etc.

Quando você se encontrar trabalhando em um ambiente primitivo, compreenda que as práticas de programação podem ajudá-lo até mais do que em ambientes amadurecidos. Conforme David Gries salientou, suas ferramentas de programação não precisam determinar o modo como você pensa sobre a programação (1981). Gries faz uma distinção entre programação *em* uma linguagem e programação *para* uma linguagem. Os programadores que programam "em" uma linguagem limitam suas ideias às construções que a linguagem suporta diretamente. Se as ferramentas da linguagem forem primitivas, as ideias do programador também serão primitivas.

Os programadores que programam "para" uma linguagem, primeiro decidem quais ideias desejam expressar e depois determinam como irão expressar essas ideias usando as ferramentas fornecidas por suas linguagens específicas.

Exemplo de programação para uma linguagem

Nos primeiros dias do Visual Basic, eu estava frustrado porque queria manter a lógica de negócio, a interface com o usuário e o banco de dados separados no produto que estava desenvolvendo, mas não havia nenhuma maneira incorporada nessa linguagem para fazer isso. Eu sabia que, se não tivesse cuidado, com o passar do tempo alguns de meus *"forms"* em Visual Basic acabariam contendo lógica de negócio, outros conteriam

código de banco de dados e alguns não conteriam nem um nem outro - eu acabaria não conseguindo nunca me lembrar qual código estaria localizado em qual lugar. Eu tinha acabado de concluir um projeto em C++ cujo trabalho de separação desses temas tinha sido malfeito e não queria passar pelo *déjà vu* daquelas dores de cabeça em uma linguagem diferente.

Consequentemente, adotei uma convenção de projeto que impunha que o arquivo de formulário só poderia recuperar dados do banco de dados e armazená-los de volta no banco de dados. Ele não poderia transmitir esses dados diretamente para outras partes do programa. Cada formulário mantinha uma rotina *IsFormCompleted( ),* usada por meio de uma chamada para determinar se o formulário ativado tinha salvo seus dados. *IsFormCompleted( )* era a única rotina pública que os formulários poderiam ter. Os formulários também não poderiam conter nenhuma regra de negócio. Qualquer outro código deveria estar contido em um arquivo.bas associado, incluindo as verificações de validação das entradas feitas no formulário.

O Visual Basic não encorajava esse tipo de estratégia. Ele estimulava os programadores a colocar o máximo de código possível no arquivo.frm e não tornava fácil para o arquivo.frm chamar um arquivo.bas associado.

Essa convenção era muito simples, mas à medida que eu me aprofundada em meu projeto, descobria que ela me ajudava a evitar numerosos casos nos quais, sem ela, teria que escrever um código complexo. Eu teria que ter muitos formulários, mas mantendo-os ocultos, para que pudesse chamar as rotinas de verificação de validação dos dados dentro deles, ou teria que copiar o código dos formulários em outros locais e depois manter código paralelo nesses locais. A convenção da rotina *IsFormCompleted()* também manteve as coisas simples. Como cada formulário funcionava exatamente da mesma maneira, eu nunca precisava pensar uma segunda vez na semântica da rotina *IsFormCompleted( )* - seu significado era o mesmo, sempre que ela era usada.

O Visual Basic não suportava essa convenção diretamente, mas meu uso de uma convenção de programação simples - programação *para* a linguagem - compensou a falta de estrutura da linguagem naquela época e ajudou a manter o projeto intelectualmente tratável.

Entender a distinção entre programação “em” uma linguagem e programação “para” uma linguagem é fundamental. A maioria dos princípios de programação importantes depende não de linguagens específicas, mas da maneira de usá-las. Se sua linguagem não possui os elementos de construção que você deseja usar ou é propensa a outros tipos de problemas, tente compensá-los. Invente suas próprias convenções de codificação, padrões, bibliotecas de classe e outros acréscimos.

4.4 Escolha das principais práticas de construção

Parte da preparação para a construção é decidir quais das muitas boas práticas disponíveis você acentuará. Alguns projetos usam programação em pares e desenvolvimento *”test-first”*, enquanto outros usam desenvolvimento individual e inspeções formais. Qualquer combinação de técnicas pode funcionar bem, dependendo das circunstâncias específicas do projeto.

A lista de verificação a seguir resume as práticas específicas que você deve optar conscientemente por incluir ou excluir durante a construção.

Lista de verificação: principais práticas de construção

Codificação

□Você definiu que proporção do projeto de *software* será realizada antecipadamente e quanto será feito no teclado, enquanto o código estiver sendo escrito?

□Você definiu convenções de codificação para nomes, comentários e apresentação?

□Você definiu práticas de codificação específicas subentendidas pela arquitetura, como a maneira pela qual as condições de erro serão tratadas, como a segurança será tratada, quais convenções serão usadas para interfaces de classe, quais padrões se aplicarão em código reutilizado, o quanto o desempenho será levado em consideração durante a codificação, etc.?

□Você identificou sua posição na onda tecnológica e ajustou sua estratégia correspondente? Se necessário, você identificou como programará *para* a linguagem, em vez de ser limitado pela programação *em* uma determinada linguagem?

Trabalho de equipe

□Você definiu um procedimento de integração - isto é, você definiu os passos específicos que um programador deve dar antes de incluir seu código nos arquivos-fonte mestres do controle de versões?

□Os programadores programarão em pares, individualmente ou em alguma combinação dos dois?

Controle de qualidade

□Os programadores escreverão casos de teste para seus códigos antes de escreverem o código em si?

□Os programadores escreverão testes unitários para seus códigos, independentemente de os escreverem antes ou depois dos programas?

□Os programadores passarão seus códigos pelo depurador, antes de registrá-los no controle de versões?

□Os programadores farão teste de integração de seus códigos, antes de registrá-los no controle de versões?

□Os programadores examinarão ou inspecionarão os códigos uns dos outros?

Ferramentas

□Você escolheu uma ferramenta de controle de revisões?

□Você escolheu uma linguagem e uma versão de linguagem ou compilador?

□Você escolheu uma infraestrutura, ou optou explicitamente por não usar uma infraestrutura?

□Você decidiu se permitirá o uso de recursos não-padronizados da linguagem?

□Você identificou e adquiriu outras ferramentas que irá usar - editor, ferramenta de *refactoring,* depurador, estrutura de teste, corretor de sintaxe, etc.?

Pontos-chave

❖ Toda linguagem de programação tem vantagens e desvantagens. Conheça as vantagens e desvantagens da linguagem que você está usando.

❖ Estabeleça convenções de programação antes de iniciar a programação. É quase impossível alterar o código para atendê-las posteriormente.

❖ Existem mais práticas de construção do que você pode usar em um projeto. Escolha conscientemente as que mais convêm ao seu projeto.

❖ Verifique se as práticas de programação que você está usando são uma resposta para a linguagem de programação que está sendo utilizada ou são controladas por ela. Lembre de programar *para* a linguagem, em vez de programar *em* uma linguagem.

❖ Sua posição na onda tecnológica determina quais estratégias serão eficientes - ou mesmo possíveis. Identifique onde você está na onda tecnológica e ajuste seus planos e expectativas adequadamente.