O Programador Pragmático:

De Aprendiz a Mestre

•••

Capítulo 2: Uma Abordagem Pragmática

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Os Males da Duplicação

Infelizmente, as informações são instáveis e mudam a todo momento. Requisitos, legislação, a vontade do cliente, tudo muda muito rapidamente.

Grande parte do tempo de desenvolvimento é gasto com manutenção, que não é feita apenas após o lançamento do sistema, mas em todo o processo.

Embora exista muita duplicação nas especificações de um software, se essa duplicação se refletir no código, a manutenção se tornará um pesadelo.

Logo, se o código possui duplicação, todo o processo de desenvolvimento se torna doloroso, muito antes do lançamento do produto.

Dicas

DICA 11:

NRS - Não Se Repita

Cada bloco de informações deve ter uma representação oficial, exclusiva e sem ambiguidades dentro de um sistema.

Como surge a duplicação?

Grande parte dos casos de duplicação que encontramos se enquadra em uma das categorias a seguir:

- Duplicação imposta: Os desenvolvedores acham que não têm escolha, o ambiente parece pedir a duplicação.
- Duplicação inadvertida: Os desenvolvedores não percebem que estão duplicando informações.
- Duplicação impaciente: Os desenvolvedores ficam com preguiça e duplicam porque parece mais fácil.

Duplicação imposta

A seguir são fornecidas algumas dicas para lidar com duplicações que nos são impostas por diferentes motivos:

- Várias representações das informações: Se uma especificação precisa ser representada em linguagens distintas, trabalhe com linguagens intermediárias e gere os códigos-alvo.
- Documentação no código: Se um código muda, seu comentário também precisará mudar. Mantenha os detalhes de baixo nível no código, comente apenas o porquê, e em alto nível.
- Documentação e código: Há duplicação entre a especificação e o código. Por isso, tente fazer com que os testes de aceitação sejam gerados automaticamente a partir da especificação.

Duplicação inadvertida

Às vezes, a duplicação ocorre como resultado de erros no projeto. Suponha um exemplo proveniente da indústria de distribuição, com o seguinte design:

- Um Caminhão possui um tipo, um número de licença e um motorista.
- Uma RotaDeDistribuição é a combinação de uma rota, um caminhão e um motorista.

Agora responda às seguintes perguntas:

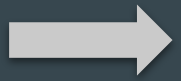
- O que aconteceria se Sally ficasse doente e tivéssemos de mudar os motoristas?
- Tanto *Caminhão* quanto *RotaDeDistribuição* têm um motorista. Qual alteraríamos?

"Independente da solução, evite dados não normalizados!"

Duplicação inadvertida: outro exemplo

A classe a seguir representa uma linha. Você consegue perceber a falta de normalização e a consequente repetição?

```
public class Line {
    public Point start;
    public Point end;
    double length;
}
```



```
public class Line {
    public Point start;
    public Point end;

    public double length() {
        return start.distanceTo(end);
    }
}
```

Duplicação inadvertida: exceção à regra

Em alguns casos, é possível abrir mão do princípio NSR, por exemplo, por questão de desempenho:

```
public class Line {
   private Point start, end;
   private double length;
   private boolean changed;
   public void setStart(Point start) {this.start = start; changed = true;}
   public void setEnd(Point end) {this.end = end; changed = true;}
   public Point getStart() {return start;}
   public Point getEnd() {return end;}

   double length() {
       if(changed){ length = start.distanceTo(end); changed = false; }
       return length;
   }
}
```

Duplicação impaciente

Todo projeto sofre pressões de tempo, forças essas que podem levar os melhores desenvolvedores a tomar atalhos.

Quando um desenvolvedor apressado precisa de uma rotina semelhante a uma já escrita, ele fica tentado a copiar a original e fazer algumas alterações.

Sempre que se sentir tentado a fazer isso, lembre do ditado popular que diz: *"atalhos causam grandes atrasos"*.

Você já ouviu falar do fiasco do bug do milênio? Ele não seria um problema se desenvolvedores tivessem criado serviços centralizados de parametrização de datas.

Duplicação entre desenvolvedores

O tipo de duplicação mais difícil de detectar e manipular é o que acontece entre códigos desenvolvidos por diferentes programadores de uma equipe.

Para evitar esse tipo de duplicação, é importante:

- Ter um líder exigente e técnico
- Estabelecer um boa divisão de responsabilidades no projeto
- Promover uma comunicação ativa entre os desenvolvedores
- Estabelecer um repositório de códigos utilitários de interesse comum
- Explorar e conhecer o código e a documentação alheia
- Facilitar a exploração do seu código e documentação

Dicas

DICA 12:

Facilite a reutilização

Cultive um ambiente em que seja mais fácil encontrar e reutilizar coisas existentes do que criá-las novamente.

O que é ortogonalidade?

Na Matemática, a ortogonalidade indica que duas linhas são independentes, não se cruzam. Na Computação, duas ou mais coisas são ortogonais quando alterações em uma não afetam as outras.

Em um sistema bem projetado, o código do bancos de dados será ortogonal à interface do usuário. Mudar uma tela não implica em mudar uma tabela, e vice-versa.

Sistemas não ortogonais são inerentemente mais difíceis de mudar e controlar. Quando componentes do sistema são altamente interdependentes, não existe o conceito de correção local.

Dicas

DICA 13:

Elimine efeitos entre elementos não relacionados

Projetar componentes autossuficientes e independentes é fundamental para evitar problemas no sistema como um todo.

Vantagens da ortogonalidade: ganho de produtividade

Independência: é mais fácil criar componentes autônomos pequenos do que grandes módulos. Tais componentes podem ser desenvolvidos, testados isoladamente e esquecidos.

Reusabilidade: componentes coesos e bem delimitados podem ser utilizados em composições não previstas inicialmente. Quanto menores forem as dependências, maior será o reúso.

Capacidade de composição: componentes com funcionalidades bem delimitadas não se sobrepõem. Por isso, é possível criar novos componentes com os já existentes, sem repetição.

Vantagens da ortogonalidade: redução de risco

Manutenibilidade: Se um módulo estiver danificado, terá menos probabilidades de espalhar os efeitos para o resto do sistema. Também será mais fácil substituí-lo por outro íntegro.

Isolamento: Manutenções pequenas em um módulo ficarão restritas apenas ao módulo. O sistema resultante se torna menos frágil.

Testabilidade: Um sistema ortogonal permite que as partes sejam testadas de forma independente. Logo, os testes serão mais fáceis de projetar e executar, além de mais precisos.

Baixa dependência: O sistema se tornará menos dependente de plataformas, fornecedores e frameworks, pois os detalhes de terceiros ficarão restritos a módulos pequenos e isolados.

Equipes de Projetos

Algumas equipes são eficientes, todos sabem suas responsabilidades e tudo flui bem. Já outras equipes vivem atrasadas, brigando e atrapalhando uns aos outros.

Esse é um problema de ortogonalidade. Quando equipes são organizadas com sobreposição, os membros ficam confusos.

Em equipes não ortogonais, toda pequena modificação precisa de uma reunião, pois qualquer um dos membros pode ser afetado. Separe e crie subtimes por responsabilidades (Slack ??).

Uma medida de ortogonalidade é a quantidade de pessoas que precisam ser avisadas a cada alteração. Quanto maior o número, menos ortogonal é o grupo.

Projetos de Sistemas

Os sistemas devem ser compostos por um conjunto de módulos em cooperação, cada um implementando uma funcionalidade independente das outras.

Por isso, adote uma organização por funcionalidades e camadas. Uma camada fornece funcionalidades a outra.

> *"Se eu alterar dramaticamente os requisitos existentes por trás de uma função específica, quantos módulos serão afetados?"*

Em um sistema ortogonal, a resposta para essa pergunta é: apenas um!

Kits de ferramentas e bibliotecas

Desenvolvedores devem tomar cuidado com a ausência de ortogonalidade imposta por frameworks e bibliotecas externas.

Quando usar funcionalidades externas (ou mesmo uma biblioteca de outros membros de sua equipe), pergunte a si mesmo se elas impõem a seu código mudanças que não deveriam existir.

Manter esses detalhes isolados de seu código trará o benefício adicional de facilitar mudanças de fornecedor no futuro.

Para se proteger de dependências não ortogonais, adote interfaces e camadas de anticorrupção de lógica de negócio.

Codificando

Há várias técnicas que você pode usar para manter a ortogonalidade no código:

- Mantenha seu código desvinculado: Escreva códigos cautelosos, que não revelem nada desnecessário ou dependam de implementações de outros módulos. Peça, não faça.
- Evite dados globais: Se um módulo referenciar dados globais, ele ficará vinculado aos outros que compartilham do estado dessa variável. Adote estados e contextos locais.
- Evite funções semelhantes: É comum escrever funções que são parecidas, têm o mesmo começo e fim, mas variam um pouco. Repetição é um problema, estude o padrão Strategy.

Adquira o hábito de ser constantemente crítico com seu código.

Reversibilidade

"Nada é para sempre. Se você conta seguramente com um fato, pode quase garantir que ele mudará."

Decisões críticas devem ser facilmente reversíveis. Em vez de tomar decisões esculpidas em pedra, considere-as mais como se tivessem sido escritas na areia da praia.

Códigos estão cheios de chamadas diretas a bancos de dados específicos. O que acontecerá com o sistema e os desenvolvedores quando a Direção pedir para trocar do Banco A para o Banco B?

Dicas

DICA 14:

Não há decisões definitivas

Projete o seu sistema para acomodar mudanças importantes, pois não há nada mais constante que o fato de que uma mudança ocorrerá.

Projéteis Luminosos

Os projéteis luminosos funcionam porque operam no mesmo ambiente e sob as mesmas restrições das balas reais, chegam a seu alvo rapidamente e fornecem ao atirador um feedback imediato.

Em processos de software, o desenvolvimento incremental pode ser utilizado como uma forma de reduzir incertezas, como projéteis luminosos que ajudam a encontrar um alvo.

Essa estratégia é oposta ao desenvolvimento convencional, de engenharia pesada, no qual o sistema é dividido em módulos que serão projetados, desenvolvidos e só apenas no final montados.

O desenvolvimento de projéteis luminosos é ágil. Cada funcionalidade é construída de forma simplificada de uma ponta a outra, e depois refinada de acordo com o resultado.

Código que brilha no escuro

A abordagem do código rastreador tem muitas vantagens:

- Os usuários podem ver algo funcionando antes
- Os desenvolvedores constroem uma estrutura na qual podem trabalhar
- Você terá uma plataforma de integração
- Você terá algo para demonstrar
- Você sentirá melhor o progresso

Dicas

DICA 15:

Use projéteis luminosos para encontrar o alvo

Adote uma estratégia de desenvolvimento que o leve de um requisito até algum aspecto do sistema final de forma rápida e visível, repetidamente.

Nem sempre os projéteis luminosos acertam seu alvo

Os projéteis luminosos mostram o que está sendo atingido. Nem sempre o alvo é atingido.

Você usará a técnica em situações em que não tiver 100% de certeza de para onde está indo.

Um pequeno corpo de código apresenta pouca inércia – é fácil e rápido alterá-lo.

O feedback permitirá gerar versões mais precisas, rápidas e baratas do que com outros métodos.

Usuários poderão ter certeza de que o que estão vendo foi baseado na realidade e não em uma especificação por escrito.

Protótipos e notas Post-it

Muitos segmentos diferentes da indústria usam protótipos para testar ideias específicas: a criação de protótipos é muito mais barata do que a produção em tamanho natural.

Protótipos de software são construídos para responder a apenas algumas questões, expor o risco e oferecer chances de correção a um custo bastante reduzido.

Como os fabricantes de carros, podemos construir protótipos com diferentes materiais:

- Notas post-it são ótimas para coisas dinâmicas como o fluxo de trabalho e a lógica do aplicativo.
- Desenhos em um quadro branco podem representar interface de usuários.
- Ferramentas de prototipação de interface podem ser usadas para exemplos mais detalhados.

Coisas que devem ter um protótipo

Devemos prototipar qualquer coisa que seja arriscada, não tenha sido testada antes ou seja absolutamente crítica para o sistema final. É possível criar protótipos de:

- Arquitetura
- Nova funcionalidade em um sistema existente
- Estrutura ou conteúdo de dados externos
- Ferramentas ou componentes de terceiros
- Questões de desempenho
- Projeto de interface de usuário

Dicas

DICA 16:

Crie protótipos para aprender

A criação de protótipos é uma experiência de aprendizado. Seu valor não está no código produzido, mas nas lições aprendidas.

Como usar protótipos

Na construção de um protótipo, que detalhes você pode ignorar?

- Precisão: podem ser utilizados dados fictícios onde e quando for apropriado
- Integralidade: o protótipo pode funcionar apenas em um aspecto muito limitado
- Robustez: a verificação de erros pode estar incompleta ou nem mesmo existir
- Estilo: código não precisa de muitos comentários ou documentação

Linguagens de Domínio

As linguagens têm uma influência significativa em como abordamos um problema ou lidamos com a comunicação. Tentamos sempre escrever código usando o vocabulário do domínio do aplicativo.

Em alguns casos, podemos passar para o próximo nível e realmente programar usando o vocabulário, a sintaxe e a semântica – a linguagem – do domínio.

Em algumas situações, deve-se considerar a criação de uma mini-linguagem para fazer seu projeto se aproximar do domínio do problema.

Uma mini-linguagem não precisa ser usada diretamente pelo aplicativo para ser útil. Podemos usá-la para criar artefatos que sejam compilados, lidos ou usados pelo próprio programa.

Dicas

DICA 17:

Programe em um nível próximo ao domínio do problema

Codificar em um nível mais alto de abstração ajuda a ficar livre para se concentrar em problemas do domínio, ignorando detalhes triviais da implementação.

Estimando

Responda rápido:

Quanto tempo um bloco de 1.000 bytes leva para passar por um roteador?

Quanto espaço em disco você precisará para um milhão de nomes e endereços?

Quantos meses serão necessários para a distribuição de seu projeto?

Dicas

DICA 18:

Estime para evitar surpresas

Desenvolver a habilidade de estimar e ter uma percepção intuitiva da dimensão das coisas dá uma aptidão aparentemente mágica para determinar sua viabilidade.

Qual nível de exatidão é suficientemente exato?

"*Todas as respostas são estimativas, o que ocorre é que algumas são mais precisas do que as outras.*"

Qual é o valor de π?

Se você estiver considerando quanto de arame comprar para colocar ao redor de um canteiro de flores circular, então “3”.

Se estiver na escola, talvez “22/7” seja uma boa aproximação.

Se estiver na NASA, 12 casas decimais servirão.

De onde vêm as estimativas?

Todas as estimativas são baseadas em modelos do problema. As seguintes dicas podem ser utilizadas na criação de seus modelos:

- Entenda o que está sendo pedido
- Construa um modelo do sistema
- Divida o modelo em componentes
- Dê a cada parâmetro um valor
- Calcule as respostas
- Acompanhe sua habilidade em estimar

Estimando cronogramas de projetos

As regras comuns para o cálculo de estimativas podem ser inúteis diante das complexidades e peculiaridades do desenvolvimento de um sistema de tamanho considerável.

A única maneira de determinar o cronograma de um projeto é ganhando experiência nesse projeto, por isso dê foco em um desenvolvimento incremental: planeje, execute, verifique, repita.

Com o desenvolvimento incremental, as suposições ficam mais precisas à medida que o projeto é conduzido. A confiança também cresce na mesma medida.

Não dê estimativas de forma intempestiva. Quando pedirem um valor, responda:

“Dou um retorno depois”.

Dicas

DICA 19:

Reexamine o cronograma junto ao código

Não tente estimar antes mesmo de o projeto começar, ajude a gerência a entender que a equipe, sua produtividade e o ambiente determinarão o cronograma.

Sobre os autores

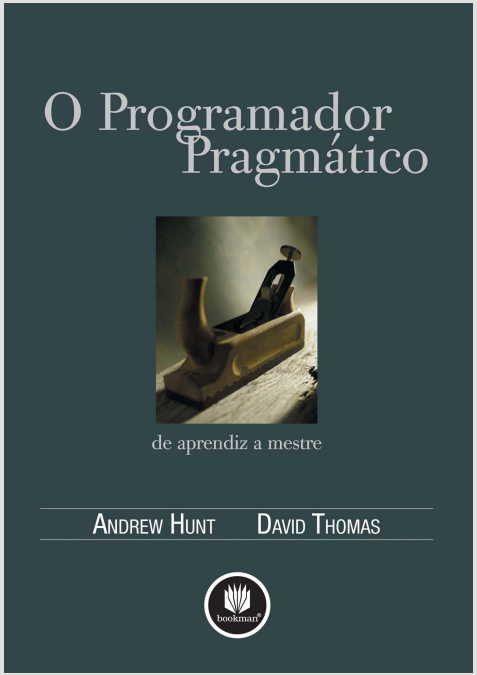
Andrew Hunt trabalhou em diversas áreas, como telecomunicações, serviços financeiros, artes gráficas etc. Hunt se especializou em combinar técnicas já consolidadas com tecnologias de ponta, criando soluções novas e práticas. Ele administra sua empresa de consultoria em Raleigh, Carolina do Norte.

Em 1994 , David Thomas, fundou na Inglaterra uma empresa de criação de software certificada pela ISO9001 que distribuiu mundialmente projetos sofisticados e personalizados. Hoje, Thomas é consultor independente e vive em Dallas, Texas.

Atualmente, David e Andrew trabalham juntos em The Pragmatic Programmers, L.L.C

Este conjunto de slides foi elaborado a partir da obra:

HUNT, Andrew ; THOMAS, David. **O Programador Pragmático:** de aprendiz a mestre.Bookman, 2010.



A equipe

Lucas Sigoli, Autor

Aluno e bolsista do PET/ADS desde abril de 2023
LinkedIn

Lucas Oliveira, Revisor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn