O Programador Pragmático:

De Aprendiz a Mestre

•••

Capítulo 5: Seja Flexível

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

Introdução

A vida não para!

Os códigos que escrevemos também não podem parar ...

Para acompanhar a quase frenética velocidade da mudança, deve ser feito todo o possível para que os códigos sejam flexíveis.

Caso contrário, o código ficará rapidamente desatualizado, ou frágil demais para ser corrigido, e será deixado para trás na louca corrida em direção ao futuro.

A desvinculação e a Lei de Deméter

A construção de um código “cauteloso” é benéfica. Isso funciona de duas maneiras: não se revelar para os outros e não interagir com muitas pessoas.

Deve-se organizar o código em módulos e limitar a interação entre eles.

Sistemas com muitas dependências desnecessárias são difíceis (e caros) de editar e tendem a ser altamente instáveis.

Para manter as dependências em um nível mínimo, use a **Lei de Deméter** para projetar métodos e funções.

Dicas

DICA 36: Reduza a vinculação entre módulos

A Lei de Deméter

Os objetos devem evitar acessar os dados e funções internas de outros objetos. Um objeto deve apenas interagir com suas dependências imediatas.

Um objeto deve usar somentes funções que são:

- dele próprio
- de parâmetros de uma de suas funções
- de objetos criados internamente
- de objetos criados no escopo de uma função

A Lei de Deméter

Exemplo da Lei de Deméter para funções

```
public class Demeter {
   private A a = new A();

   public int func() {
        // ...
   }

   public void example(B b) {
      C c;
      int f = func(); //função dele próprio
      b.invert(); //de um parâmetro
      a.setActive(); //de um objeto interno
      c.print(); //de um objeto no escopo do método
   }
}
```

A Lei de Deméter - Custos

A Lei de Deméter torna o código mais adaptável e robusto, mas com o custo de delegar e gerenciar qualquer módulo que ele venha a terceirizar suas funcionalidades.

Na prática, com a Lei de Deméter, classes se tornam contêineres de métodos encapsuladores, para os quais solicitações serão encaminhadas.

O ato de criar classes adicionais para delegação implica no aumento do tempo de desenvolvimento e de execução, o que precisa ser levado em conta.

Metaprogramação

“Não há nível de genialidade que supere a preocupação com os detalhes.”

Oitava Lei de Levy

Detalhes excessivos podem atrapalhar em códigos que precisam ser alterados com frequência. Sempre que ocorrem alterações no código, há risco de se introduzir novos erros no sistema.

Portanto, têm-se como lema **“fora com os detalhes!”**. Tire-os do código. Ao fazer isso, o código se torna altamente configurável e “leve” – isto é, facilmente adaptável às mudanças.

Dicas

DICA 37: Configure, não integre.

Metadados

Metadados são dados sobre dados. Provavelmente, o exemplo mais comum seja um esquema de banco de dados ou um dicionário de dados.

Metadado é qualquer dado que descreva o aplicativo – como ele deve ser executado, que recursos deve usar e assim por diante.

Normalmente, os metadados são acessados e usados em tempo de execução e não em tempo de compilação.

Use **metadados** para descrever as opções de configuração de um aplicativo: parâmetros de ajuste, preferências do usuário, o diretório de instalação e assim por diante.

Aplicativos baseados em metadados

Deve-se especificar o ***que*** deve ser feito e não ***como.*** Programe pensando no cenário como um todo e deixe os detalhes fora da base de código compilada.

Com isso, obtêm-se diversas vantagens:

- Maior desvinculação do projeto, o que resulta em um programa mais flexível e adaptável.
- Criação de um projeto mais robusto e abstrato, por deixar os detalhes para depois.
- Personalização do aplicativo sem necessidade de compilá-lo novamente.

Vinculação temporal

Pense no tempo como um elemento do projeto do próprio software.

Há dois aspectos do tempo que são importantes em projetos de sistemas: concorrência (coisas ocorrendo ao mesmo tempo) e ordem (as posições relativas das coisas no tempo).

Deve-se permitir a concorrência e tentar desvincular qualquer dependência de tempo ou ordem.

Ao fazer isso, aceleramos várias partes do processo de desenvolvimento que podem acontecer simultaneamente: análise de fluxo de trabalho, arquitetura, projeto e implantação.

Fluxo de trabalho

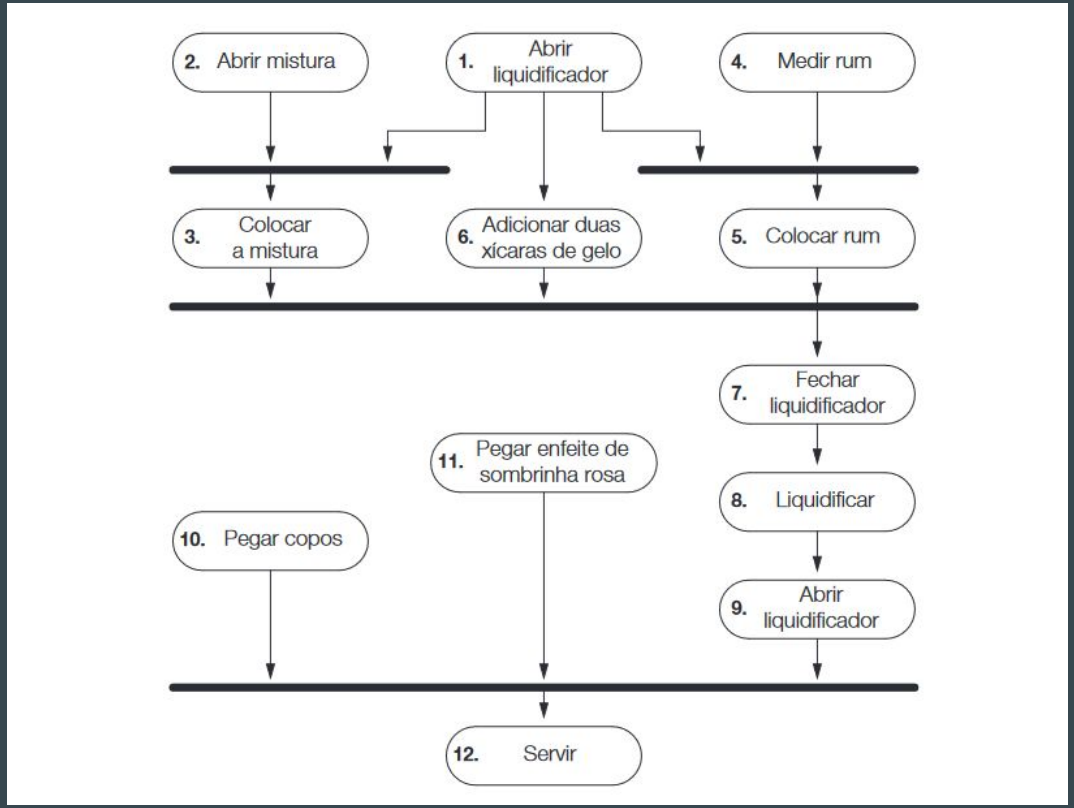
Em muitos projetos, é necessário modelar e analisar os fluxos de trabalho dos usuários como parte da análise de requisitos.

Deve-se descobrir o que pode ocorrer ao mesmo tempo e o que deve ocorrer em uma ordem rigorosa.

Uma maneira de fazer isso é capturar a descrição do fluxo de trabalho usando uma notação como o Diagrama de Atividades UML.

Fluxo de trabalho

É possível usar Diagramas de Atividades para maximizar o paralelismo, identificando atividades que poderiam ser executadas em paralelo, mas não são.



Dicas

DICA 40: Projete usando serviços

Projete pensando na concorrência

Em vez de componentes, crie serviços – objetos independentes e concorrentes por trás de interfaces consistentes e bem definidas.

Com um código linear, é fácil fazer suposições que levem a uma programação malfeita. Mas a concorrência o forçará a ponderar as coisas um pouco mais cuidadosamente.

Já que agora as ações podem ocorrer ao “mesmo tempo”, subitamente pode-se encontrar algumas dependências baseadas no tempo.

Dependências baseadas em tempo

Qualquer variável global ou estática deve ser protegida do acesso concorrente. Variáveis globais, em específico, precisam ser tratadas com cautela.

Além disso, as informações de estado devem ser apresentadas de modo consistente, independentemente da ordem das chamadas que tratam delas.

Deve-se assegurar que um objeto esteja em um estado válido a qualquer momento em que ele possa ser chamado.

Apenas um modo de ver

Cada módulo tem suas próprias responsabilidades; uma boa definição de um módulo (ou classe) é que ele deve ter apenas uma responsabilidade, bem estabelecida.

No tempo de execução, os objetos se comunicam uns com os outros, e cabe ao desenvolvedor gerenciar as dependências lógicas entre eles.

É preciso sincronizar alterações no estado (ou atualizações em valores de dados) nesses diferentes objetos.

Isso tem de ser feito de uma maneira limpa e flexível – não queremos que eles saibam muito uns sobre os outros.

Eventos

Um evento é uma mensagem especial que diz “algo interessante acabou de ocorrer”.

Use eventos para sinalizar alterações que ocorreram em um objeto, mas que outros objetos também possam estar interessados.

O uso de eventos reduz a vinculação entre esses objetos – o emitente do evento não precisa ter nenhum conhecimento explícito sobre o destinatário.

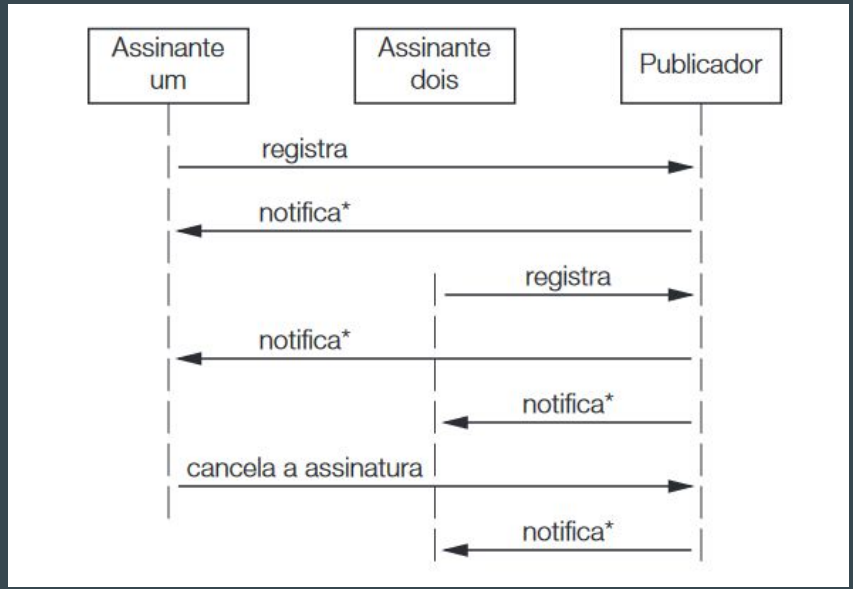
Publicação/Assinatura

É preciso evitar o recebimento de todos os eventos por meio da mesma rotina. Essa rotina passa a conhecer detalhadamente as interações entre muitos objetos, violando o encapsulamento.

Além disso, se a vinculação aumenta, os próprios objetos terão de conhecer esses eventos, violando a ortogonalidade.

Publicação / Assinatura

Os objetos devem poder se registrar para receber só os eventos que precisam e nunca devem receber eventos que não precisam.



Publicação / Assinatura

Os objetos Subscriber interessados devem se inscrever em um Publisher para receber seus eventos.

Quando o Publisher gera um evento de interesse, ele chama cada Subscriber sequencialmente na ordem em que eles se inscreveram e os notifica que o evento ocorreu.

Assim que um Subscriber não tiver mais a necessidade de receber eventos do Publisher, ele pode se desinscrever e parar de receber notificações.

Model-View-Controller

O MVC prega a separação do modelo de sua representação gráfica. Ao adotá-lo, é possível:

- Dar suporte a várias visualizações do mesmo modelo de dados.
- Usar os mesmos visualizadores em muitos modelos de dados diferentes.
- Dar suporte a vários controladores para fornecer mecanismos de entrada não tradicionais.

Ao afrouxar a vinculação entre o modelo e a visualização/controlador, obtém-se grande flexibilidade a baixo custo.

Além das GUIs

Embora o MVC seja ensinado no contexto do desenvolvimento de GUIs, trata-se de uma técnica de programação de uso geral.

- **Modelo:** o modelo de dados abstrato que representa o objeto-alvo. O modelo não tem conhecimento direto de nenhuma visualização ou controlador.
- **Visualização:** uma maneira de representar o modelo. Ela se adapta a alterações no modelo e a eventos lógicos provenientes do controlador.
- **Controlador:** uma maneira de controlar a visualização e fornecer novos dados para o modelo. Ele publica eventos tanto para o modelo quanto para a visualização.

Sobre os autores

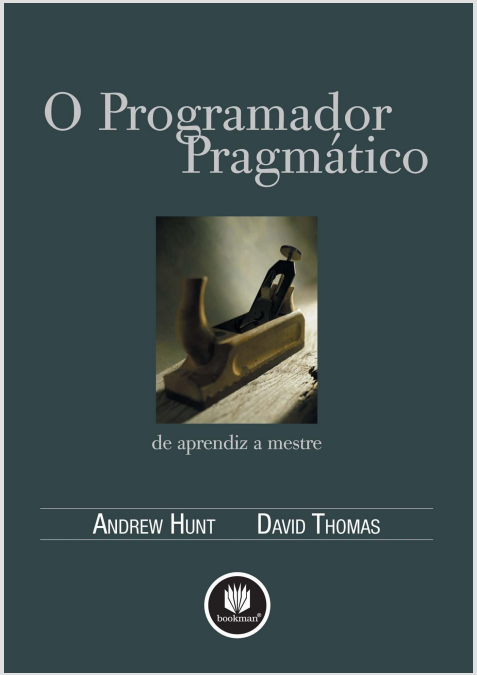
Andrew Hunt trabalhou em diversas áreas, como telecomunicações, serviços financeiros, artes gráficas etc. Hunt se especializou em combinar técnicas já consolidadas com tecnologias de ponta, criando soluções novas e práticas. Ele administra sua empresa de consultoria em Raleigh, Carolina do Norte.

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

Gabriel Z Manhani, Autor

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

Lucas Oliveira, Revisor

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