

Programação Orientada a Objetos

Padrões de Projeto

Refatoração

Guinther Pauli

Arquiteto de Software - MVP

Borland®
Delphi™ 7
Advanced Certified

Borland®
Delphi™ 6
PRODUCT P CERTIFIED

Borland®
Delphi™
WEB Development Certified

Borland®
Kylix™
PRODUCT P CERTIFIED

Borland®
Delphi™ 2005
PRODUCT P CERTIFIED

Borland®
Delphi™ 2006
PRODUCT P CERTIFIED

Microsoft®
CERTIFIED
Professional

Microsoft®
CERTIFIED
Application Developer

Microsoft®
CERTIFIED
Solution Developer

Microsoft®
CERTIFIED
Professional
Developer

Microsoft®
CERTIFIED
Technology
Specialist

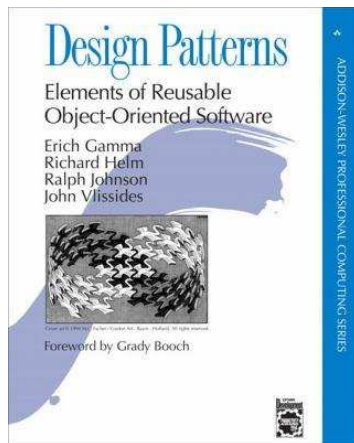
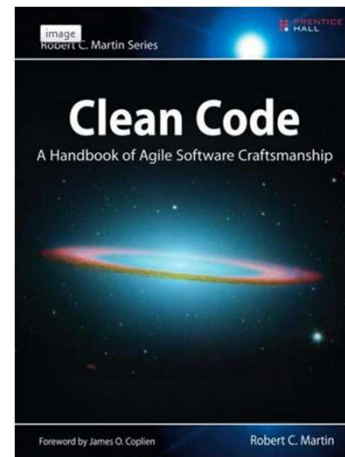
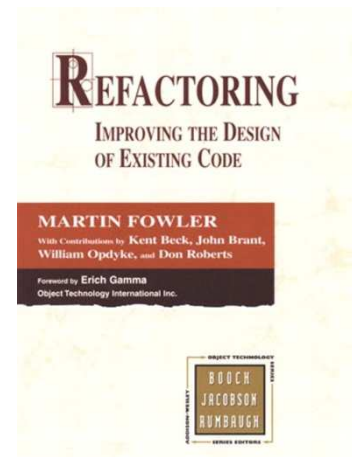
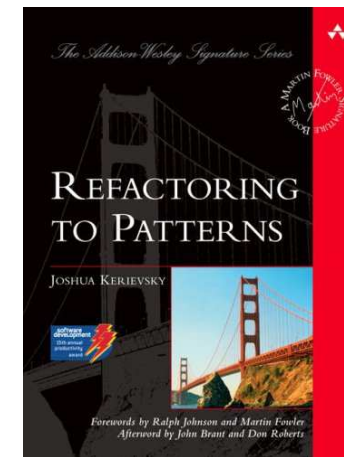


Sobre o autor



Gunther Pauli - Arquiteto de Software certificado pela Microsoft, Professor de Pós-Graduação, Membro Editorial das Revistas .NET Magazine Brasil e ClubeDelphi, as maiores publicações do país em seus segmentos - Certificado oficial MCP, MCAD, MCSD.NET, MCTS, MCPD (ASP.NET C# - 4.0) - Certificado Delphi: 3, 5, 6, 7, 2005, 2006, Delphi para Web, Delphi para Linux. MVP – Most Valuable Professional. Foi arquiteto e desenvolvedor do sistema de doações on-line do projeto Criança Esperança da Rede Globo / Unesco. Beta Test de várias versões do Delphi, palestrante em 5 edições da Borland Conference, maior evento de desenvolvedores do país. Ministra treinamentos corporativos para empresas e universidades, já tendo treinado mais de 5 mil profissionais em todo Brasil e exterior, incluindo grandes empresas como SONY, UNICAMP e KBR. Autor de mais de 1000 vídeo-aulas de ensino a distância, publicou mais de 130 artigos, além do livro “Delphi – Programação para Banco de Dados e Web”. Fez a cobertura de grandes eventos no exterior, como o TechEd da Microsoft e a BorCon da Borland americana. Desenvolve em Delphi há 17 anos e 12 anos em .NET / C#. Bacharel em Sistemas de Informação e Mestrando do Programa de Pós-Graduação em Informática na UFSM - Universidade Federal de Santa Maria - RS. Áreas de interesse e pesquisa: Engenharia de Software, Programação Orientada a Objetos, Padrões de Projeto e Refatoração.

Bibliografia



Roteiro

- * Bad Smells (Code Smells)
- * Código Limpo (Code Clean)
- * Padrões de Projeto (Design Patterns)
- * Refatoração (Refactoring)
- * Refatoração para Padrões

Bad Smells (Code Smells)

```
if (flag == true)
    opcao = 0;
else
    opcao = 1;
double t = SldCC();
string ts = w44(t.ToString());
```



Bad Smells

- * De acordo com Kent Beck, *bad smells* (“maus cheiros”) são estruturas no código que sugerem a possibilidade de refatoração. Alguns exemplos de problemas em estruturas de código são: código duplicado.

Bad Smells

- * Rigidez – uma alteração em um ponto do código requer alterações em cascata (dependências);
- * Fragilidade – A modificação de um ponto do código quebra outras funcionalidades;
- * Complexidade – Arquitetura muito complexa pois foi preparada para manipular qualquer tipo de possibilidade;
- * Duplicação – Código redundante, duplicado;
- * Legibilidade – Código difícil de compreender e entender.

Código Duplicado (Duplicated Code)

- * Este provavelmente é um dos problemas mais graves no código. Podem ser duas expressões idênticas em dois métodos na mesma classe, ou duas classes não associadas. Essa duplicação pode ser eliminada aplicando-se a refatoração *Extrair Método (Extract Method)*. Nesse caso, o código é implementado uma única vez e chamado no local de origem onde estava o código duplicado. Se o código duplicado encontra-se em duas classes distintas, pode-se aplicar a refatoração *Extrair Classe (Extract Class)*.

Método Longo (Long Method)

- * Consiste de um método com uma implementação de lógica com várias, dezenas ou até centenas de linhas de código. O ideal seria criar métodos com no máximo dez linhas de código, podendo ser inclusive de uma a cinco. Vários problemas podem ser originados da implementação de métodos longos. Primeiro, pode existir código duplicado. Segundo, será difícil reaproveitar parte do código em outro local. A solução é dividir o método longo em métodos menores, usando-se a refatoração *Extrair Método (Extract Method)*.

Classe Larga (Large Class)

- * Uma boa prática da programação orientada a objetos sugere que uma classe deve ter uma única responsabilidade (princípio da responsabilidade única). Quando uma classe possui muitas instâncias no código, isso sugere que ela está fazendo muito, ou pelo menos realizando trabalho que deveria ser feito por outras classes. Algumas refatorações podem ajudar a reduzir classes longas, como por exemplo, *Extrair Classe (Extract Class)*, *Extrair Subclasse (Extract subclass)*, entre outras.

Lista de Parâmetros Longa (Long Parameter List)

- * Um método que recebe muitos parâmetros sugere que classes possam estar compartilhando muita informação, ou ainda, dados globais. Se um método com uma longa lista de parâmetros está sendo chamado em vários locais do código, uma mudança em um deles pode desencadear uma cadeia muito grande de alterações em vários locais do código. Algumas refatorações podem ajudar a resolver esse problema, por exemplo, a *Substituir Parâmetro por Método* (*Replace Parameter with Method*) (Fowler, 1999) e *Introduzir Parâmetro Objeto* (*Introduce Parameter Object*) (Fowler, 1999). Neste último caso, por exemplo, uma longa lista de parâmetros se resume a uma única estrutura (classe, por exemplo), que internamente define atributos.

Mudança Divergente (Divergent Change)

- * Ocorre quando uma classe é alterada geralmente de diferentes formas por diferentes motivos. Normalmente, uma mudança requer a alteração de várias classes. Nesse caso, uma boa solução seria isolar ou encapsular código passível de mudança para que comporte a mudança em um único local, quando necessária.

Tiro Cirúrgico (Shotgun Surgery)

- * Esta situação é o oposto de *Mudança Divergente* (*Divergent Change*). Em uma pequena mudança de código, é necessário realizar várias pequenas mudanças em várias classes.

Recurso Inveioso (Feature Envy)

- * Ocorre quando um método realiza a maioria das suas funções consultando outros métodos ou mesmo outros objetos, a fim de obter dados, realizar cálculos, lógica etc.

Dados Agregados (Data Clumps)

- * Ocorre normalmente quando dados costumam aparecer juntos em várias situações: campos em classes, em lista de parâmetros em assinaturas de métodos etc. Uma solução seria aplicar a refatoração *Introduzir Parâmetro Objeto* (*Introduce Parameter Object*).

Obsessão Primitiva (Primitive Obsession)

- * Existem basicamente dois tipos de dados na maioria das linguagens: os primitivos e as estruturas (normalmente classes, formadas a partir da composição de outros tipos primitivos). A obsessão primitiva ocorre normalmente com desenvolvedores com pouca experiência técnica em orientação a objetos. Refatorações como *Substituir Dado por Objeto* (*Replace Data with Object*) e *Substituir Código de Tipo por Estado / Estratégia* (*Replace Type Code with State / Strategy*) podem ajudar a solucionar problemas com obsessão primitiva.

Sentenças switch (Switch Statements)

- * Testar várias possibilidades com sentenças switch (*case*) tende a duplicar código. Se um novo teste for necessário na sentença, para comportar uma nova possibilidade, e esse teste for feito em vários locais, então várias alterações serão necessárias. Muitas vezes os testes podem ser substituídos por polimorfismo, usando a refatoração *Substituir Condicional por Polimorfismo* (*Replace Conditional with Polymorphism*) (Fowler, 1999).

Hierarquia de Herança Paralela (Parallel Inheritance Hierarchies)

- * Nesse caso, cada vez que uma classe se torna subclasse de outra, também é necessário criar uma subclasse de outra classe (pois existem hierarquias paralelas). As refatorações *Mover Método* (*Move Method*) e *Mover Campo* (*Move Field*) ajudam a minimizar problemas relacionados a este bad smell.

Classe Preguiçosa (Lazy Class)

- * Cada classe criada no sistema de software tem um custo para ser mantida e evoluída. Classes que não estão fazendo o suficiente para pagarem seu próprio custo devem ser eliminadas. A refatoração *Recolher Hierarquia* (*Collapse Hierarchy*) pode minimizar problemas associados a classes preguiçosas.

Generalidade especulativa (Speculative Generality)

- * Este problema ocorre quando se projeta uma solução genérica para uma possível situação que possivelmente nunca ocorrerá, tornando objetos flexíveis o suficiente para tratar qualquer tipo de situação, por exemplo, criando-se classes abstratas que não realizam muito. A refatoração *Recolher Hierarquia* (*Collapse Hierarchy*) pode minimizar problemas associados à generalidade.

Campo Temporário (Temporary Field)

- * Muitas vezes um objeto tem sua variável de instância configurada somente sobre certas circunstâncias. Tal código é difícil de entender. Um exemplo ocorre quando um algoritmo depende de uma série de variáveis. A refatoração *Extrair Classe* (*Extract Class*) pode minimizar problemas associados á essas variáveis “pobres”.

Cadeia de Mensagens (Message Chains)

- * Ocorre quando um cliente consulta um objeto a partir de outro objeto, que por sua vez repassa a consulta a outro objeto, que consulta outro, e assim por diante, causando uma cadeia de chamadas. Qualquer mudança nos relacionamentos intermediários pode ocasionar grandes problemas. As refatorações *Extrair Método* (*Extract Method*) (Fowler, 1999) e *Esconder Delegação* (*Hide Delegate*) (Fowler, 1999) minimizam problemas desse tipo.

Homem do Meio (*Middle Man*)

- * O encapsulamento é um dos principais recursos da programação orientada a objetos, a habilidade de esconder detalhes internos do resto do mundo. A delegação ocorre quando uma classe recebe chamadas e utiliza um objeto interno para realizar a maioria de suas funções (delegação). A refatoração *Remover Homem do Meio* (*Remove Middle Man*) (Fowler, 1999) permite reduzir este problema.

Intimidade inadequada (Inappropriate Intimacy)

- * Muitas vezes classes se tornam demasiadamente íntimas e perdem muito tempo investigando partes privadas da própria classe. As refatorações *Mover Método* (*Move Method*) e *Mover Campo* (*Move Field*) ajudam a separar partes para reduzir essa intimidade.

Classes Alternativas com Diferentes Interfaces (Alternative Classes with Different Interfaces)

- * Ocorre com métodos que fazem a mesma coisa, mas com diferentes assinaturas. As refatorações *Mover Método* (*Move Method*) e *Extrair Superclasse* (*Extract Super Class*) ajudam a resolver esse tipo de problema.

Biblioteca de Classes Incompleta (Incomplete Library Class)

- * Objetos são o princípio básico da reutilização de código. Bibliotecas de classes realizam boa parte de tarefas comuns usando esse conceito. Porém, normalmente não se pode alterar uma biblioteca base de classes. A refatoração *Introduzir Método Estrangeiro* (*Introduce Foreign Method*) permite simular a inclusão de um método em uma classe que não pode ser estendida nem modificada.

Classe de Dados (Data Class)

- * Classes de dados normalmente contêm muitos campos públicos para representarem seus dados, e nada mais. Esse tipo de classe frequentemente trabalha como container de dados que são utilizados e repassados para outras classes. A refatoração *Encapsular Campo* (*Encapsulate Field*) permite encapsular o acesso a esses campos.

Legado Recusado (Refused Bequest)

- * Classes herdam métodos e dados de classes superiores, porém muitas vezes não precisam de todos os dados e comportamentos herdados para funcionar. Provavelmente há um problema na hierarquia. As refatorações *Descer Método / Campo* (*Push down Method / Field*) (Fowler, 1999) permite deslocar métodos e campos para classes mais específicas, mantendo nas classes superiores somente o que realmente é comum a toda hierarquia.

Comentários (Comments)

- * Comentários são bons, porém, um código com muitos comentários é um forte indício que ele está difícil de entender. Talvez seja mais apropriado dar nomes mais sugestivos a métodos, deixando mais claras suas responsabilidades, usando a refatoração *Renomear Método* (*Rename Method*) (Fowler, 1999).

Code Clean



Code Clean

- * O que é um código limpo:
 - * Simples;
 - * Eficiente;
 - * Direto;
 - * Elegante;
 - * Sem duplicidade;
 - * Feito com cuidado;
 - * Que faz uma coisa;

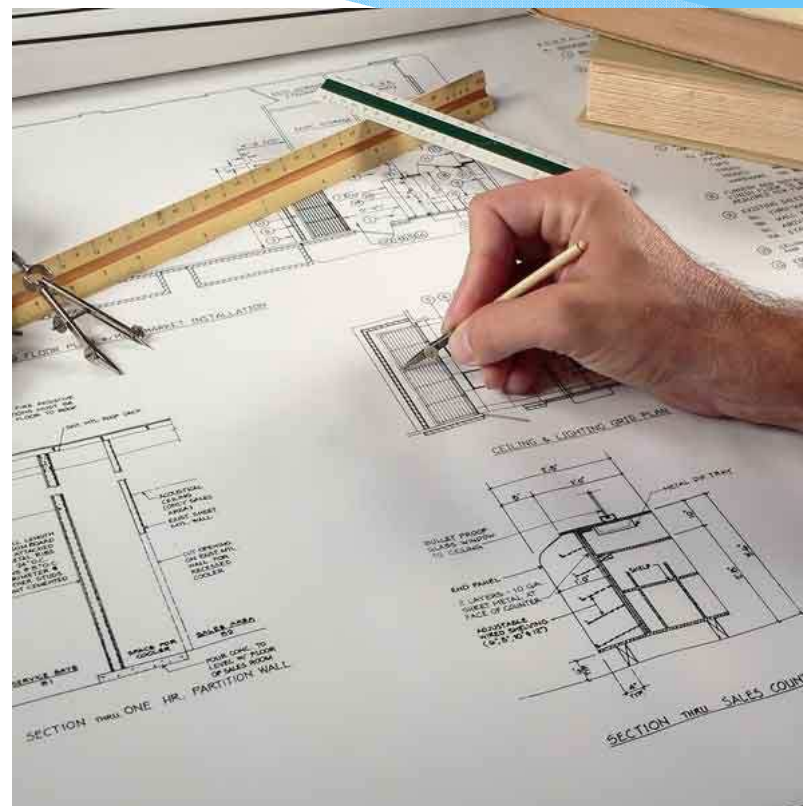
Code Clean

- * Nomes significativos, que revelam sua intenção;
- * Expressa o que faz, por que existe, como é usado;
- * Nomes pronunciáveis;
- * Notação húngara, tipo não precisa estar no nome;
- * Nomes de classes são substantivos;
- * Nomes de métodos são verbos;
- * Assessores devem ter “get” e “set”;

Code Clean

- * Métodos pequenos (5 ou 10 linhas);
- * Linhas de até 100 chars;
- * Métodos devem realizar uma única tarefa;
- * Cuidar com lista longa de parâmetros;
- * Classes pequenas (200 linhas);
- * Seguir o princípio da responsabilidade única;
- * Cuidar o acoplamento entre classes (dependências);
- * Refatore ao invés de comentar;

Padrões de Projeto



Introdução

- * Chrisptopher Alexander, um arquiteto, professor e comentarista social, afirma: “cada padrão descreve um problema no nosso ambiente e o cerne de sua solução, de tal forma que você possa usar essa solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira”.

Introdução

- * Usar padrões de projeto torna mais fácil reutilizar projetos e arquiteturas bem-sucedidas. Expressar técnicas testadas e aprovadas as torna mais acessíveis para os desenvolvedores de novos sistemas. Os padrões de projeto ajudam a escolher alternativas de projeto que tornam um sistema reutilizável e evitar alternativas que comprometam a reutilização. Os padrões de projeto podem melhorar a documentação e a manutenção de sistemas ao fornecer uma especificação explícita de interações de classes e objetos e o seu objeto subjacente. Em suma, ajudam um projetista a obter mais rapidamente um projeto adequado.

Introdução

- * Padrões ajudam a solucionar problemas de projeto, por exemplo, quando estamos procurando objetos apropriados em um sistema orientado a objetos. Um objeto empacota tanto os dados quanto os procedimentos que operam sobre eles. Os procedimentos são chamados de métodos ou operações. Um objeto executa uma operação quando ele recebe uma solicitação (ou mensagem) de um cliente.

Introdução

- * As solicitações são a única maneira de conseguir que um objeto execute uma operação. As operações são a única maneira de mudar os dados internos de um objeto. Por causa dessas restrições, diz-se que o estado interno de um objeto está encapsulado; ele não pode ser acessado diretamente e sua representação é invisível do exterior do objeto.

Introdução

- * A parte difícil sobre projeto orientado a objetos é a decomposição de um sistema em objetos. A tarefa é difícil porque muitos fatores entram em jogo: encapsulamento, granularidade, dependência, flexibilidade, desempenho, evolução, reutilização, e assim por diante. Todos influenciam a decomposição, e frequentemente, formas conflitantes (Gamma et al., 1995).

Introdução

- * As metodologias de projeto orientado a objetos favorecem muitas abordagens diferentes. Você pode escrever uma descrição de um problema, separar os substantivos e os verbos e criar classes e operações correspondentes. Ou você pode se concentrar sobre as colaborações e responsabilidades no seu sistema. Ou ainda, poderá modelar o mundo real e, na fase de projeto, traduzir os objetos encontrados durante a análise. Sempre haverá desacordo sobre qual é a melhor abordagem.

Introdução

- * Muitos objetos num projeto provêm do modelo de análise. Porém, nem todos estes objetos acabam tendo uma representação simples como no mundo real. São representadas de mais baixo nível. Os padrões de projeto ajudam a identificar abstrações menos óbvias bem como os objetos que podem capturá-las. Por exemplo, objetos que processam um algoritmo não existem na natureza, mas são cruciais em projetos flexíveis.

Introdução

- * Além disso, os objetos podem variar em tamanho e número e podem representar qualquer coisa. Como decidir o que deve ser um objeto? Os padrões de projeto também ajudam nesse sentido. Por exemplo, o padrão *Façade* descreve como representar subsistemas inteiros como objetos. Outros padrões descrevem maneiras específicas de como decompor um objeto em objetos menores. O *Abstract Factory* e *Builder* fornecem objetos cujas únicas responsabilidades são criar outros objetos (Gamma et al., 1995).

Introdução

- * Padrões de Projeto também ajudam a especificar interfaces de objetos. As interfaces são fundamentais em sistemas orientados a objetos. Não existe nenhuma maneira de saber algo sobre um objeto ou de pedir que faça algo sem intermédio de sua interface. A interface nada diz sobre a implementação de um objeto, com isso objetos estão livres para implementar as solicitações de diferentes maneiras, ao mesmo tempo em que podem ser totalmente substituídos por outros objetos que implementam a mesma interface, realizando as mesmas operações, de forma diferente (técnica conhecida como polimorfismo). Isso reduz significativamente o acoplamento entre classes, promovendo a reutilização. Os padrões de projeto ajudam a definir interfaces para identificação de seus elementos-chave e pelos tipos de dados que são enviados através de uma interface.

Introdução

- * Padrões de projeto também ajudam a especificar a implementação de objetos. Uma implementação é definida por uma classe. Ela representa os dados que o objeto deve conter bem como suas operações. Novas classes podem ser criadas a partir da *herança* de outras classes. Nesse caso, diz-se que as classes descendentes são as subclasses, e a classe base, a superclasse. Quando uma classe superior apenas especifica uma interface comum para todas as suas subclasses, ela é chamada de *classe abstrata*. As classes que herdam dela e efetivamente realizam as operações são chamadas de *classes concretas*.

Introdução

- * Nesse ponto é importante diferir entre herança de classe versus herança de interface. Ao herdar de uma interface, uma classe apenas assume que irá implementar as operações por elas definida. Diferentemente, uma classe abstrata pode também conter um comportamento padrão para ser herdado por suas subclasses, o que não pode ser feito somente com interfaces.

Introdução

- * Isso remete a uma das principais boas práticas da programação orientada a objetos, a programação para interfaces, e não para uma implementação. Padrões de projeto ajudam a seguir esta prática, ao promoverem o baixo acoplamento entre classes usando soluções baseadas em interfaces. Clientes com isso conhecem pouco sobre tipos específicos que usam.

Introdução

- * Padrões de projeto também ajudam a identificar quando utilizar herança ou composição, um dos princípios fundamentais da programação orientada a objetos para reaproveitamento de código. A composição é uma alternativa à herança, conhecida como *herança de caixa preta*, já que os detalhes do objeto herdado não ficam expostos ao mundo externo. Já a herança é mais rígida, a classe que herda estará dependendo da hierarquia de tipos definida, tendo pouca flexibilidade. Nesse caso, é preferível usar composição à herança.

Introdução

- * A chave para a maximização da reutilização está na antecipação de novos requisitos e mudanças nos requisitos existentes e em projetar sistemas de modo que eles possam evoluir de acordo. Padrões ajudam a projetar sistemas de software melhor preparados para mudanças.

Introdução

- * A chave para a maximização da reutilização está na antecipação de novos requisitos e mudanças nos requisitos existentes e em projetar sistemas de modo que eles possam evoluir de acordo. Padrões ajudam a projetar sistemas de software melhor preparados para mudanças.

Padrões GoF

- * Os padrões "GoF" são organizados em 3 famílias :
 - * Padrões de criação : relacionados à criação de objetos;
 - * Padrões estruturais : tratam das associações entre classes e objetos;
 - * Padrões comportamentais : tratam das interações e divisões de responsabilidades entre as classes ou objetos;

Padrões de Criação

- * Abstract Factory
- * Builder
- * Factory Method
- * Prototype
- * Singleton

Padrões Estruturais

- * Adapter
- * Bridge
- * Composite
- * Decorator
- * Facade
- * Flyweight
- * Proxy

Padrões Comportamentais

- * Chain of Responsibility
- * Command
- * Iterator
- * Mediator
- * Memento
- * Observer
- * State
- * Strategy
- * Template Method
- * Visitor

Excesso de engenharia

- * Um sistema que é projetado o mais flexível ou sofisticado quanto deveria ser, é feito em demasia. Programadores projetam em excesso hoje, normalmente usando dezenas de padrões de projeto, pois acreditam conhecer os futuros requisitos do sistema, tornando-os mais fáceis de acomodar amanhã. A menos que programadores sejam videntes, as previsões erradas podem resultar em perda de tempo e dinheiro preciosos. Normalmente programadores tentam antecipar necessidades que nunca se materializam. Novos programadores vão ser obrigados a trabalhar em uma estrutura de código complexa, mais complicada do que deveria ser.

Projeto antecipado com padrões

- * Um grande projeto antecipado ocorre quando um sistema de software é projetado em um primeiro momento para um grande conjunto de requisitos que podem futuramente mudar, serem descartados ou postergados para uma futura versão do sistema. Muito tempo é gasto pensando em um projeto que é mais elegante ou sofisticado do que precisaria ser.

Escassez de engenharia

- * Projeto em escassez é mais comum que projeto em excesso. Softwares mal projetados podem ocorrer por diversas razões:
 - * Falta de tempo;
 - * Falta de conhecimento sobre um bom projeto de software;
 - * Necessidade de se adicionar rapidamente funcionalidades a sistemas existentes;
 - * Projetos sendo executados em paralelo.
- * Em curto prazo projetar em escassez reduz custos. Ao longo do tempo, torna-se caro, muito caro. O código se torna difícil de manter, muitas vezes uma “bagunça” que nem é possível de ser mantida, quem dirá evoluída. Muitas vezes, gerentes sem conhecimento técnico nestes problemas, explicam que a organização será mais competitiva e bem sucedida se “não consertar o que não está quebrado”.

Escassez de engenharia

- * Projeto em escassez de forma contínua leva a um ritmo de desenvolvimento de software que pode ser descrito como “rápido, normal, lento, mais lento, extremamente lento”, algo como:
 - * A versão 1.0 do sistema é entregue rapidamente;
 - * A versão 2.0 do sistema é entregue, o código ruim retarda o desenvolvimento;
 - * À medida que se entregam novas versões, mais tempo é necessário para implementar novas funcionalidades ou corrigir bugs existentes, pois o código ruim se multiplica, as pessoas começam a perder fé no sistema, nos programadores, e até mesmo no processo que levou todo mundo a esta posição; Neste ponto é muito comum haver uma rotatividade de profissionais na equipe, o que piora ainda mais a situação;
 - * Em algum momento, depois da versão 4.0, a equipe percebe que não existe mais uma solução, o caos já está generalizado, então começa a estudar a opção de uma reescrita total do sistema.

Débito de Projeto (Design Debt)

- * A linguagem técnica de refatoração não comunica efetivamente com a maioria das gerências. Melhorar um projeto existente sem modificar o comportamento externo observável, o que envolve custo e tempo, não será visto com bons olhos por gerentes. A metáfora financeira do débito de projeto é uma boa solução. O débito de projeto ocorre quando o desenvolvedor não realiza consistentemente três coisas: remover duplicação, simplificar código e deixar clara a sua intenção. Humanos não são perfeitos, não escrevem corretamente da primeira vez, e naturalmente um débito de projeto é acumulado. A questão é, quando esse débito será pago?

Débito de Projeto

- * Devido à ignorância ou o comprometimento a “não consertar aquilo que não está quebrado”, muitos programadores e equipes gastam pouco tempo para pagar esse débito, como resultado, criam uma grande confusão, que leva a sistemas mal projetados, de difícil compreensão e manutenção. Em termos financeiros, quando um débito não é pago, taxas vão se acumulando na forma de juros, pior eles se tornam e mais difícil será o pagamento. Juros compostos se acumulam e, à medida que o tempo passa, livra-se dessa dívida torna-se um sonho impossível. Assim é também com o débito de projeto.

Dilema

- * Excesso de engenharia é um problema, pois geralmente não se pode aplicar padrões de forma antecipada;
- * Escassez de engenharia é a falta da aplicação de padrões;
- * Então, como e quando aplicar padrões de projeto?



Refatoração



Refatoração

- * Uma premissa básica de qualquer sistema de software do mundo real é a necessidade de evoluir. À medida que estes são evoluídos, modificados e adaptados de acordo com novas exigências, o código e outros artefatos envolvidos se tornam mais complexos e se afastam de seu objetivo original, podendo diminuir sua qualidade. Em muitos projetos, boa parte do custo é dedicada à sua manutenção. Boas metodologias de desenvolvimento de software e ferramentas não resolvem este problema, pois sua utilização normalmente auxilia na aceleração no desenvolvimento, o que faz com que as empresas consequentemente aumentem o número de requisitos a serem implementados dentro do mesmo espaço de tempo, tornando os sistemas de software ainda mais complexos.

Refatoração

- * Para lidar com esse alto índice incremental de complexidade, que geram custos, existe a necessidade de se usar técnicas que facilitem a manutenibilidade e legibilidade, melhorando a qualidade interna do sistema de software. Uma das técnicas usadas para solucionar esse problema é conhecida como reestruturação, ou no caso específico da programação orientada a objetos, Refatoração.

Refatoração

- * Refatoração é o processo de modificar um sistema de software sem alterar o comportamento externo do código, melhorando sua estrutura interna. É uma forma disciplinada de limpar minimizando as chances de introduzir erros. Em essência, quando você refatora, você está melhorando o design do código depois de ter sido escrito. Um bom projeto deveria vir em primeiro lugar, a seguir a sua codificação. Com a evolução do sistema de software, mudanças ocorrerão, e a sua estrutura e integridade original diminuem.

Refatoração

- * Essa reestruturação também é definida como a transformação de uma forma de representação para outra, no mesmo nível relativo de abstração, enquanto preserva o comportamento externo do sistema (funcionalidade e semântica). Essas transformações podem melhorar a estrutura do projeto. Embora a reestruturação crie novas versões que implementam ou propõem mudanças no objetivo final do sistema, normalmente não envolvem modificações referentes a novos requisitos. No entanto, podem conduzir a observar melhor o objetivo do sistema, já que sugerem mudanças que poderiam melhorar seus aspectos.

Refatoração

- * Nesse sentido, é necessário identificar locais nos artefatos de software que precisam ter sua estrutura transformada, seja para melhorar sua legibilidade, melhorar o projeto, tornar mais simples de evoluir e de manter. Tais estruturas passíveis de refatoração são também conhecidas como “Bad Smells”, ou problemas. Um *bad smell* sugere que existem deficiências ou limitações nos artefatos que podem ser sanadas ou melhoradas, porém é necessário que a aplicação continue apresentando o mesmo comportamento externo após a transformação interna.

Refatoração

- * Outro ponto a ser considerado no estudo da programação orientada a objetos é o uso de padrões de projeto, amplamente utilizados no projeto de arquiteturas de software, pois representam soluções gerais, reutilizáveis, testadas e documentadas, resolvendo problemas recorrentes de programação orientada a objetos. Muitas vezes, no entanto, padrões de projeto tendem a ser utilizados de forma antecipada em um projeto.

Refatoração

- * Refatoração é processo inverso desta prática. Você parte de um mau design (um verdadeiro caos), refatora e obtém um código bem projetado. Cada refatoração consiste de um passo simples, pequeno. Você move um campo para outra classe, move um método para uma classe superior na hierarquia. Cada pequeno passo melhora o design do código, mantendo o comportamento externo.

Refatoração

- * Existem várias motivações para o uso de refatorações. Primeiro, elas tornam mais fácil a adição de novo código. Quando se inclui uma nova funcionalidade em um sistema de software, existem basicamente duas opções: programar rapidamente sem se preocupar o quão bem se encaixa no projeto existente, ou pode-se modificar o projeto existente a fim de melhor acomodar a funcionalidade. No primeiro caso, ocorre um débito de projeto, o qual pode ser pago mais tarde com refatoração.

Refatoração

- * Segundo, as refatorações melhoraram um projeto de código existente, no momento que tornam o código mais simples, claro e mais fácil de trabalhar, manter e evoluir. A refatoração contínua envolve procurar constantemente por problemas e removê-los imediatamente. Habituar-se ao higiênico hábito de refatorar continuamente, resultará em código sempre limpo.

Refatoração

- * E finalmente, refatorações ajudam a se obter um melhor entendimento do código. Se um código está difícil de ser compreendido, programadores normalmente comentam este código para tornar mais clara sua intenção. Se o código não está claro, o problema precisa ser removido através de refatoração.

Refatoração - Fundamentos

- * Pode-se então definir o termo refatorar (verbo) como reestruturar um sistema de software aplicando-se uma série de refatorações sem modificar seu comportamento observável. Nesse caso, refatoração (nome), é uma mudança feita na estrutura interna de um sistema de software para torná-lo mais fácil de entender e modificar sem alterar o comportamento externo observável.

Refatoração - Fundamentos

- * Um bom contraste com refatoração é performance e otimização. Assim como refatoração, melhorar a performance e otimização de um sistema de software não afeta o comportamento observável. No entanto, têm propósitos diferentes, pois melhorar a performance pode tornar o código mais fácil de entender. Para o usuário final, um sistema de software refatorado terá as mesmas funções anteriores (Fowler, 1999).

Refatoração - Fundamentos

- * A refatoração ajuda no processo de identificação de *bugs*, pois clarifica a estrutura do programa e torna mais fácil a localização de problemas. Refatoração ajuda a programar de forma mais rápida, apesar de isso parecer estranho. Melhorar o projeto e legibilidade não reduz a velocidade do desenvolvimento? Em longo prazo, esses fatores vão tornar mais fáceis as adições de novas funcionalidades e correções de bugs, que serão feitas de forma mais rápida. A refatoração não é uma atividade que deve ser “agendada”. Ela deve ser feita continuamente, em pequenos passos.

Refatoração - Fundamentos

- * Um momento oportuno para refatorar é quando uma nova funcionalidade deve ser adicionada. A refatoração nesse caso pode ajudar a tornar o código mais fácil de entender imediatamente. Melhorando o projeto, a nova funcionalidade pode ser adicionada de forma mais simples e rápida, sem introduzir novos bugs. A refatoração vai ajudar a encontrar *bugs* mais facilmente, porque o código se torna mais legível, mais claro.

Refatoração - Fundamentos

- * Outro bom momento para refatorar é quando são feitas revisões no código. Grandes organizações realizam constantes revisões no código a fim de garantir a qualidade. Isso também ajuda a desenvolvedores mais experientes a transmitirem seu conhecimento aos menos experientes. Ajudam a sugerir ideias. As refatorações nesse caso são uma forma mais concreta de se obter resultados de uma revisão de código.

Gerentes e Refatorações

- * Programas possuem dois tipos de valores: o que eles podem fazer hoje e o que eles podem fazer amanhã. Na maior parte do tempo, a programação foca no que o programa deve (ou pelo menos deveria) fazer hoje. A refatoração é uma forma de ligar o hoje com o amanhã. Quando uma decisão não faz sentido hoje, muda-se a decisão, então o trabalho pode ser feito.

Gerentes e Refatorações

- * O que fazem os programas difíceis de serem trabalhados:
 - * Programas difíceis de ler são difíceis de modificar;
 - * Programas que têm a lógica duplicada são difíceis de modificar;
 - * Programas que requerem comportamento adicional que precisam modificar seu código de execução são difíceis de alterar;
 - * Programas com lógica complexa são difíceis de modificar.

Gerentes e Refatorações

- * Então, programas devem ser fáceis de ler, devem ter a lógica especificada em apenas um único lugar, que permitam ser modificadas sem alterar o comportamento externo observável, e que permita que lógicas condicionais possam ser expressas de forma simples.

Gerentes e Refatorações

- * Se refatorações são importantes, porém não modificam o comportamento externo observável, como explicar a sua necessidade para gerentes, que normalmente não têm experiência técnica? Qualidade é um aspecto chave, este termo pode convencer a maioria dos gerentes do valor das revisões do código.

Gerentes e Refatorações

- * A refatoração tende a quebrar grandes problemas em problemas menores (por exemplo, objetos). Porém, mais objetos precisam ser gerenciados. E objetos nesse caso normalmente costumam delegar tarefas a outros objetos, indiretamente. Por exemplo, ao colocar a lógica de uma tarefa em um método chamado por vários objetos, utiliza-se uma boa prática de compartilhamento. Além disso, a intenção do método e sua implementação são feitas separadamente. Com isso, isola-se algo que pode mudar, evitando replicação ao mesmo tempo em que deixa o código limpo. Tudo isso, no entanto, não irá modificar o comportamento externo, porém o programa terá mais valor, pois tem qualidades que serão mais apreciadas amanhã.

Problemas das Refatorações

- * A refatoração, no entanto, também tem seus problemas. Muitas aplicações de banco de dados são fortemente acopladas com esquemas de banco de dados. Uma solução é considerar a utilização de uma camada intermediária de comunicação com o banco de dados, que isole muitas das responsabilidades, permitindo flexibilidade. Outro problema diz respeito à modificação de interfaces de objetos.

Problemas das Refatorações

- * Bons projetistas modelam classes (interfaces) e relações, de forma que uma modificação possa ser feita internamente na implementação sem modificar suas interfaces públicas. Porém, muitas refatorações modificam essas interfaces, por exemplo, a *Renomear Método* (*Rename Method*). Isso desencadeará uma série de modificações em todos os locais onde o método é invocado. Uma boa solução é manter uma versão de “transição”, fazendo que a interface antiga ainda possa ser usada até que a nova a substitua completamente. Uma boa dica é nunca publicar interfaces prematuramente, pois interfaces, segundo os fundamentos da programação orientada a objetos, devem (ou deveriam) ser imutáveis.

Problemas das Refatorações

- * E quando não se deve refatorar? A refatoração, por definição, não muda o comportamento observável. Não deve ser usada em princípio para corrigir bugs, mas como uma forma de ajudar a identificá-los. Se um sistema de software está repleto de bugs e está difícil de estabilizá-lo, talvez uma melhor opção seria reescrevê-lo do zero, o que não é uma decisão fácil.

Refatoração e Projeto

- * A refatoração serve como uma alternativa ao chamado *big design up-front* (Grande Projeto Antecipado, ou G.P.A.). Nesse cenário não se cria inicialmente um grande projeto, que tente prever todas as possibilidades de mudanças que possam ocorrer no sistema de software. Ao invés disso, criam-se pequenas porções do software bem desenhadas, como sugere a XP (*Extreme Programming*). A refatoração nesse caso dá ênfase nas mudanças. Utiliza-se a abordagem *up-front*, porém não se tenta achar a solução para tudo. Ao desenvolver de forma evolutiva e incremental, o conceito de solução pode mudar, ou melhorar, e com a refatoração isso não se torna um problema.

Refatoração e Performance

- * Ao tornar o sistema de software mais fácil de entender, com mudanças no código, frequentemente fará o mesmo rodar mais lento. No entanto, um código mais limpo facilitará os testes de performance (*tuning*). A maioria dos problemas de performance em sistemas de software se concentra em pequenas porções isoladas do código. A preocupação com performance pode se dar apenas no final de um projeto, o qual deve ser desenvolvido com constante refatoração. Existe uma fase de otimização específica para este fim.

1 - Composing Methods

- * Extract Method
- * Inline Method
- * Inline Temp
- * Replace Temp with Query
- * Introduce Explaining Variable
- * Split Temporary Variable
- * Remove Assignments to Parameters
- * Replace Method with Method Object
- * Substitute Algorithm

2 - Moving Features between Objects

- * Move Method
- * Move Field
- * Extract Class
- * Inline Class
- * Hide Delegate
- * Remove Middle Man
- * Introduce Foreign Method
- * Introduce Local Extension

3 – Organizing Data

- * Self Encapsulate Field
- * Replace Data Value with Object
- * Change Value to Reference
- * Change Reference to Value
- * Replace Array with Object
- * Duplicate Observed Data
- * Change Unidirectional Association to Bidirectional
- * Change Bidirectional Association to Unidirectional
- * Replace Magic Number with Symbolic Constant
- * Encapsulate Field
- * Encapsulate Collection
- * Replace Record with Data Class
- * Replace Type Code with Class
- * Replace Type Code with Subclasses
- * Replace Type Code with State/Strategy
- * Replace Subclass with Fields

4 – Simplifying Conditional Expressions

- * Decompose Conditional
- * Consolidate Conditional Expression
- * Consolidate Duplicate Conditional Fragments
- * Remove Control Flag
- * Replace Nested Conditional with Guard Clauses
- * Replace Conditional with Polymorphism
- * Introduce Assertion

5 – Making Methods Calls Simpler

- * Rename Method
- * Add Parameter
- * Remove Parameter
- * Separate Query from Modifier
- * Parameterize Method
- * Replace Parameter with Explicit Methods
- * Preserve Whole Object
- * Replace Parameter with Method
- * Introduce Parameter Object
- * Remove Setting Method
- * Hide Method
- * Replace Constructor with Factory Method
- * Encapsulate Downcast
- * Replace Error Code with Exception
- * Replace Exception with Test

6 – Dealing with Generalization

- * Pull Up Field
- * Pull Up Method
- * Pull Up Constructor Body
- * Push Down Method
- * Push Down Field
- * Extract Subclass
- * Extract Superclass
- * Extract Interface
- * Collapse Hierarchy
- * Form Template Method
- * Replace Inheritance with Delegation
- * Replace Delegation with Inheritance

Conclusão

- * Padrões de Projeto são soluções reutilizáveis para problemas recorrentes no desenvolvimento orientado a objetos. No entanto, aplicar padrões antecipadamente em um projeto requer mais tempo e aumenta custos, ao mesmo tempo em que aumenta sua complexidade. Requisitos são modificados, criados ou mesmo eliminados conforme projetos evoluem. Tentar aplicar padrões antecipadamente para prever essas mudanças é um problema conhecido como excesso de engenharia. Já projeto em escassez reduz custos, pois o sistema de software é criado rapidamente devido à falta de tempo, conhecimento ou necessidade de se adicionar funcionalidades em curto prazo. Ao longo do tempo, armazena-se um débito de projeto, que torna o sistema de software muito difícil de ser mantido e evoluído. É nesse contexto que entram as refatorações, como uma forma de melhorar o projeto existente sem alterar seu comportamento externo observável. Refatoração é o aspecto-chave para um projeto evolutivo, e é na evolução que se encontra a verdadeira sabedoria.

Referências Bibliográficas

- * FOWLER, M. Refactoring: Improving the Design of Existing Code. Addison-Wesley, 1999. Object Technologies Series.
- * KERIEVSKY, J. Refatoração para Padrões. Bookman, 2008.
- * GAMMA, E.; HELM, R.; JOHSON, R.; VLISSIDES, JOHN. Padrões de Projeto: Soluções reutilizáveis de software orientado a objetos. Addison-Wesley, 1995).
- * BECK, K. Extreme Programming Explained: Embrace Change. Boston, USA:
Addison-Wesley, 1999.
- * BECK. K. Test-Driven Development. Boston, MA: Addison-Wesley, 2002.

Obrigado! Dúvidas? Contato:



<http://www.facebook.com/guintherpauli>



<http://www.twitter.com/guintherpauli>



<http://br.linkedin.com/in/guintherpauli>



<http://www.gpauli.com>



guintherpauli@gmail.com



gpauli@gpauli.com



[guinther.pauli](https://www.skype.com/user/guinther.pauli)



guintherpauli@gmail.com