# Princípios de design e Padrões de design

Robert C. Martin
www.objectmentor.com

O que é arquitetura de software? A resposta é multicamada. No nível mais alto, estão os padrões de arquitetura que definem a forma e a estrutura geral dos aplicativos de software1 . Abaixo de um nível está a arquitetura que está especificamente relacionada ao propósito do aplicativo de software. Ainda outro nível abaixo reside a arquitetura dos módulos e suas interconexões. Este é o domínio dos padrões de projeto2, pacotes, componentes e classes. É com esse nível que nos ocuparemos neste capítulo.

Nosso escopo neste capítulo é bastante limitado. Há muito mais a ser dito sobre os princípios e padrões aqui expostos. Os leitores interessados devem consultar [Martin99].

## Arquitetura e dependências

O que há de errado com o software? O projeto de muitos aplicativos de software começa como uma imagem vital na mente de seus projetistas. Nesta fase, é limpo, elegante e atraente. Tem uma beleza simples que faz com que os designers e implementadores coçam para vê-lo funcionando. Alguns desses aplicativos conseguem manter essa pureza de design durante o desenvolvimento inicial e na primeira versão.

Mas então algo começa a acontecer. O software começa a apodrecer. No começo não é tão ruim. Uma verruga feia aqui, um corte desajeitado ali, mas a beleza do design ainda aparece. No entanto, com o tempo, à medida que a podridão continua, as feias feridas purulentas e furúnculos se acumulam até dominar o design do aplicativo. O programa torna-se uma massa de código purulenta que os desenvolvedores acham cada vez mais difícil de manter. Evento

1. [Shaw96]
2. [GOF96]

Aliado, o grande esforço necessário para fazer até mesmo as mudanças mais simples no aplicativo torna-se tão grande que os engenheiros e gerentes de linha de frente clamam por um projeto de redesenho.

Esses redesenhos raramente são bem-sucedidos. Embora os designers comecem com boas intenções, eles descobrem que estão atirando em um alvo em movimento. O sistema antigo continua a evoluir e mudar, e o novo design deve acompanhar. As verrugas e úlceras se acumulam no novo design antes mesmo de chegar ao seu primeiro lançamento. Nesse dia fatídico, geralmente muito mais tarde do que o planejado, o pântano de problemas no novo design pode ser tão ruim que os designers já estão clamando por outro redesenho.

## Sintomas de design podre

Existem quatro sintomas principais que nos dizem que nossos projetos estão apodrecendo. Eles não são ortogonais, mas estão relacionados entre si de maneiras que se tornarão óbvias. são eles: rigidez, fragilidade, imobilidade e viscosidade.

**Rigidez.** Rigidez é a tendência de o software ser difícil de mudar, mesmo de maneiras simples. Cada mudança causa uma cascata de mudanças subsequentes em módulos dependentes. O que começa como uma simples mudança de dois dias para um módulo se transforma em uma maratona de várias semanas de mudança em módulo após módulo, à medida que os engenheiros perseguem o fio da mudança através do aplicativo.

Quando o software se comporta dessa maneira, os gerentes temem permitir que os engenheiros corrijam problemas não críticos. Essa relutância deriva do fato de que eles não sabem, com certeza, quando os engenheiros vão terminar. Se os gerentes soltarem os engenheiros em relação a esses problemas, eles podem desaparecer por longos períodos de tempo. O design do software começa a assumir algumas características de um motel de baratas - os engenheiros fazem check-in, mas não fazem check-out.

Quando os medos do gerente se tornam tão agudos que eles se recusam a permitir mudanças no software, a rigidez oficial se instala. Assim, o que começa como uma deficiência de projeto, acaba sendo uma política de gestão adversa.

**Fragilidade.** Intimamente relacionada à rigidez está a fragilidade. A fragilidade é a tendência do software de quebrar em muitos lugares toda vez que é alterado. Muitas vezes a quebra ocorre em áreas que não possuem relação conceitual com a área que foi alterada. Tais erros enchem os corações dos gerentes com pressentimentos. Toda vez que autorizam uma correção, eles temem que o software falhe de alguma forma inesperada.

À medida que a fragilidade piora, a probabilidade de quebra aumenta com o tempo, aproximando-se assintoticamente de 1. Esse software é impossível de manter. Cada correção torna pior, introduzindo mais problemas do que são resolvidos.

Esse software faz com que gerentes e clientes suspeitem que os desenvolvedores perderam o controle de seu software. A desconfiança reina e a credibilidade se perde.

**Imobilidade.** Imobilidade é a incapacidade de reutilizar software de outros projetos ou de partes do mesmo projeto. Muitas vezes acontece de um engenheiro descobrir que precisa de um módulo semelhante a um que outro engenheiro escreveu. No entanto, também acontece frequentemente que o módulo em questão tem muita bagagem da qual depende.
Depois de muito trabalho, os engenheiros descobrem que o trabalho e o risco necessários para separar as partes desejáveis do software das partes indesejáveis são grandes demais para serem tolerados. E assim o software é simplesmente reescrito em vez de reutilizado.

**Viscosidade.** A viscosidade vem em duas formas: viscosidade do design e viscosidade do ambiente. Quando confrontados com uma mudança, os engenheiros geralmente encontram mais de uma maneira de fazer a mudança. Algumas das maneiras preservam o design, outras não (ou seja, são hacks). Quando os métodos de preservação do design são mais difíceis de empregar do que os hacks, então a viscosidade do design é alta. É fácil fazer a coisa errada, mas difícil fazer a coisa certa.

A viscosidade do ambiente ocorre quando o ambiente de desenvolvimento é lento e ineficiente. Por exemplo, se os tempos de compilação forem muito longos, os engenheiros ficarão tentados a fazer alterações que não forcem grandes recompilações, mesmo que essas alterações não sejam ideais do ponto de vista do projeto. Se o sistema de controle de código-fonte exigir horas para fazer check-in de apenas alguns arquivos, os engenheiros ficarão tentados a fazer alterações que exijam o mínimo de check-ins possível, independentemente de o design ser preservado.

Esses quatro sintomas são os sinais indicadores de uma arquitetura pobre. Qualquer aplicativo que os exiba está sofrendo de um design que está apodrecendo de dentro para fora. Mas o que causa essa podridão?

## Mudando os Requisitos

A causa imediata da degradação do design é bem compreendida. Os requisitos estão mudando de maneiras que o projeto inicial não previa. Muitas vezes, essas mudanças precisam ser feitas rapidamente e podem ser feitas por engenheiros que não estão familiarizados com a filosofia de projeto original. Portanto, embora a mudança no design funcione, de alguma forma viola o design original. Pouco a pouco, à medida que as mudanças continuam a surgir, essas violações se acumulam até que a malignidade se instale.

No entanto, não podemos culpar a deriva dos requisitos pela degradação do design. Nós, como engenheiros de software, sabemos muito bem que os requisitos mudam. De fato, a maioria de nós percebe que o documento de requisitos é o documento mais volátil do

projeto. Se nossos projetos estão falhando devido à chuva constante de mudanças nos requisitos, são nossos projetos que estão com defeito. Devemos de alguma forma encontrar uma maneira de tornar nossos projetos resistentes a essas mudanças e protegê-los do apodrecimento.

### Gerenciamento de Dependências

Que tipo de mudanças fazem com que os designs apodreçam? Alterações que introduzem dependências novas e não planejadas. Cada um dos quatro sintomas mencionados acima é causado direta ou indiretamente por dependências impróprias entre os módulos do software. É a arquitetura de dependência que está se degradando e, com ela, a capacidade do software de ser mantido.

Para evitar a degradação da arquitetura de dependências, as dependências entre os módulos em um aplicativo devem ser gerenciadas. Esta gestão consiste na criação de firewalls de dependência. Através desses firewalls, as dependências não se propagam.

O Projeto Orientado a Objetos está repleto de princípios e técnicas para construir tais firewalls e para gerenciar dependências de módulos. São esses princípios e técnicas que serão discutidos no restante deste capítulo. Primeiro, examinaremos os princípios e, em seguida, as técnicas, ou padrões de projeto, que ajudam a manter a arquitetura de dependência de um aplicativo.

## Princípios do Design de Classes Orientado a Objetos

### O Princípio Aberto Fechado (OCP)1

*Um módulo deve estar aberto para extensão, mas fechado para modificação.*

De todos os princípios do design orientado a objetos, este é o mais importante. Originou-se do trabalho de Bertrand Meyer2 . Significa simplesmente isto: devemos escrever nossos módulos para que possam ser estendidos, sem exigir que sejam modificados. Em outras palavras, queremos poder alterar o que os módulos fazem, sem alterar o código-fonte dos módulos.

1. [OCP97]
2. [OOSC98]

Isso pode parecer contraditório, mas existem várias técnicas para alcançar o OCP em larga escala. Todas essas técnicas são baseadas em abstração. De fato, *a abstração é a chave para o OCP.* Várias dessas técnicas são descritas abaixo.

**Polimorfismo Dinâmico.** Considere a Listagem 2-1. a função LogOn deve ser alterada toda vez que um novo tipo de modem é adicionado ao software. Pior, uma vez que cada tipo diferente de modem depende da enumeração Modem::Type, cada modem deve ser recompilado toda vez que um novo tipo de modem é adicionado.

**Listagem 2-1**
Logon, deve ser modificado para ser estendido.

```
struct Modem {

    enum Tipo {hayes, Courrier, ernie) tipo;
};

estrutura Hayes
{
    Modem::Tipo de tipo;
    // Coisas relacionadas a Hayes
};

struct Courrier
{
    Modem::Tipo de tipo;
    // Coisas relacionadas ao correio
};

struct Ernie
{
    Modem::Tipo de tipo;
    // Coisas relacionadas ao Ernie
};

void LogOn(Modem& m, string&
                    pno, string& usuário, string& pw)
{
    if (m.type == Modem::hayes)
        DialHayes((Hayes&)m, pno);
    else if (m.type == Modem::courrier)
        DialCourrier((Courrier&)m, pno);
    else if (m.type == Modem::ernie)
        Disque Ernie((Ernie&)m, pno)
    // ...Você entendeu a ideia
}
```

Claro que este não é o pior atributo deste tipo de design. Programas que são projetados dessa maneira tendem a estar cheios de instruções if/else ou switch semelhantes. Toda vez que algo precisa ser feito no modem, uma instrução switch if/else chain precisará selecionar as funções apropriadas a serem usadas. Quando novos modems são adicionados ou a política do modem é alterada, o código deve ser varrido para todas essas instruções de seleção e cada uma deve ser modificada adequadamente.

Pior ainda, os programadores podem usar otimizações locais que ocultam a estrutura das instruções de seleção. Por exemplo, pode ser que a função seja exatamente a mesma para os modems Hayes e Courrier. Assim, podemos ver um código como este:

```
if (modem.type == Modem::ernie)
    SendErnie((Ernie&)modem, c);
senão
    SendHayes((Hayes&)modem, c);
```

Claramente, tais estruturas tornam o sistema muito mais difícil de manter e são muito propensos a erros.

Como exemplo do OCP, considere a Figura 2-13. Aqui a função LogOn depende apenas da interface do Modem. Modems adicionais não farão com que a função LogOn seja alterada. Assim, criamos um módulo que pode ser estendido, com novos modems, sem necessidade de modificação. Consulte a Listagem 2-2.

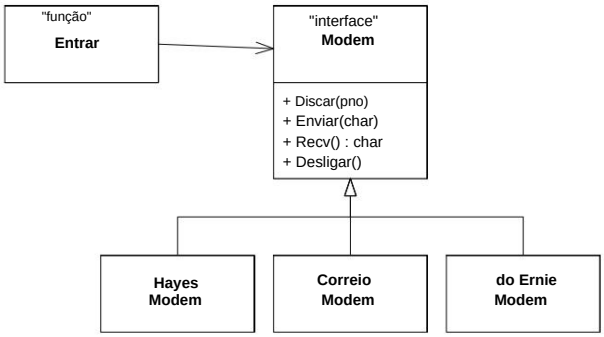


**Figura 2-13**

**Listagem 2-2**
O LogOn foi fechado para modificação

```
classe Modem
{
público:
```

**Listagem 2-2**
O LogOn foi fechado para modificação

```
    virtual void Dial(const string& pno) = 0;
    virtual void Send(char) = 0;
    caractere virtual Recv() = 0;
    virtual void Hangup() = 0;
};

void LogOn(Modem& m,
                string& pno, string& usuário, string& pw)
{
    m.Disque(pno);
    // Você entendeu a ideia.
}
```

**Polimorfismo Estático.** Outra técnica para se adequar ao OCP é através do uso de templates ou genéricos. A Listagem 2-3 mostra como isso é feito. A função LogOn pode ser estendida com muitos tipos diferentes de modems sem a necessidade de modificação.

**Listagem 2-3**
O logon está fechado para modificação por polimorfismo estático

```
modelo <typename MODEM>
void LogOn(MODEM& m,
                string& pno, string& usuário, string& pw)
{
    m.Disque(pno);
    // Você entendeu a ideia.
}
```

**Objetivos Arquitetônicos do OCP.** Usando essas técnicas para estar em conformidade com o OCP, podemos criar módulos que são extensíveis, sem serem alterados. Isso significa que, com um pouco de premeditação, podemos adicionar novos recursos ao código existente, sem alterar o código existente e apenas adicionando novo código. Este é um ideal que pode ser difícil de alcançar, mas você o verá alcançado, várias vezes, nos estudos de caso mais adiante neste livro.

Mesmo que o OCP não possa ser totalmente alcançado, mesmo o cumprimento parcial do OCP pode trazer melhorias dramáticas na estrutura de um aplicativo. É sempre melhor se as alterações não se propagarem no código existente que já funciona. Se você não precisar alterar o código de trabalho, provavelmente não irá quebrá-lo.

## O Princípio da Substituição de Liskov (LSP)1

*As subclasses devem ser substituíveis por suas classes base.*

Este princípio foi cunhado por Barbar Liskov2 em seu trabalho sobre abstração de dados e teoria de tipos. Também deriva do conceito de Design by Contract (DBC) de Bertrand Meyer3 .

O conceito, conforme declarado acima, é representado na Figura 2-14. As classes derivadas devem ser substituíveis por suas classes base. Ou seja, um usuário de uma classe base deve continuar funcionando corretamente se uma derivada dessa classe base for passada para ele.

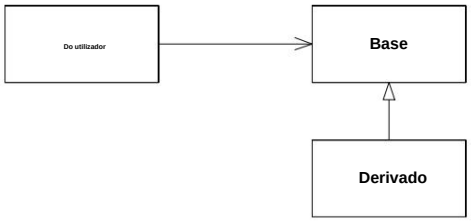


**Figura 2-14**
esquema LSP.

Em outras palavras, se alguma função User recebe um argumento do tipo Base, como mostrado na Listagem 2-4, deve ser legal passar uma instância de Derived para essa função.

**Listagem 2-4**
Usuário, Baseado, Derivado, exemplo.

```
void Usuário(Base& b);

d derivado;
Usuário(d);
```

Isso pode parecer óbvio, mas há sutilezas que precisam ser consideradas. O exemplo canônico é o dilema Círculo/Elipse.

1. [LSP97]
2. [Liksov88]
3. [OOSC98]

**O Dilema Círculo/Elipse.** A maioria de nós aprende, na matemática do ensino médio, que um círculo é apenas uma forma degenerada de uma elipse. Todos os círculos são elipses com focos coincidentes. Esse relacionamento é-um nos tenta a modelar círculos e elipses usando herança como mostrado na Figura 2-15.

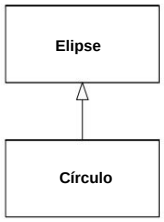


**Figura 2-15**
Dilema do círculo/elipse

Embora isso satisfaça nosso modelo conceitual, existem algumas dificuldades. Um olhar mais atento à declaração de Ellipse na Figura 2-16 começa a expô-los. Observe que Ellipse tem três elementos de dados. Os dois primeiros são os focos e o último é o comprimento do eixo maior. Se Circle herdar de Ellipse, ele herdará essas variáveis de dados. Isso é lamentável, pois Circle realmente precisa apenas de dois elementos de dados, um ponto central e um raio.

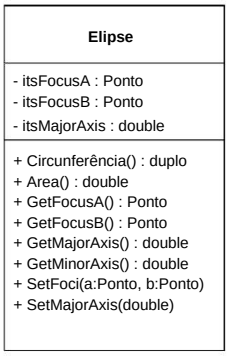


**Figura 2-16**
Declaração de elipse

Ainda assim, se ignorarmos a ligeira sobrecarga no espaço, podemos fazer com que Circle se comporte adequadamente substituindo seu método SetFoci para garantir que ambos os focos sejam mantidos no mesmo valor. Consulte a Listagem 2-5. Assim, qualquer foco atuará como o centro do círculo, e o eixo maior será seu diâmetro.

**Listagem 2-5**
Mantendo os Focos do Círculo coincidentes.

```
void Circle::SetFoci(const Point& a, const Point& b)
{
    seuFocoA = a;
    seuFocoB = a;
}
```

**Os clientes estragam tudo.** Certamente o modelo que criamos é auto-consistente. Uma instância de Circle obedece a todas as regras de um círculo. Não há nada que você possa fazer para fazê-lo violar essas regras. Assim também para Elipse. As duas classes formam um modelo bem consistente, mesmo que Circle tenha muitos elementos de dados.

No entanto, Circle e Ellipse não vivem sozinhos em um universo.
Eles coabitam esse universo com muitas outras entidades e fornecem suas interfaces públicas para essas entidades. Essas interfaces implicam um contrato. O contrato pode não ser explicitamente declarado, mas está lá, no entanto. Por exemplo, os usuários do Ellipse têm o direito de esperar que o seguinte fragmento de código seja bem-sucedido:

```
void f(Elipse& e)
{
    Ponto a(-1,0);
    Ponto b(1,0);
    e.SetFoci(a,b);
    e.SetMajorAxis(3);
    assert(e.GetFocusA() == a);
    assert(e.GetFocusB() == b);
    assert(e.GetMajorAxis() == 3);
}
```

Nesse caso, a função espera trabalhar com uma elipse. Dessa forma, espera-se poder definir os focos e o eixo principal e, em seguida, verificar se eles foram definidos corretamente. Se passarmos uma instância de Ellipse para esta função, ela ficará bem feliz.
No entanto, se passarmos uma instância de Circle para a função, ela falhará bastante.

Se fôssemos tornar explícito o contrato de Ellipse, veríamos uma pós-condição no SetFoci que garantia que os valores de entrada fossem copiados para as variáveis de membro e que a variável do eixo principal fosse deixada inalterada. Claramente Circle viola essa garantia porque ignora a segunda variável de entrada de SetFoci.

**Projeto por contrato.** Reafirmando o LSP, podemos dizer que, para ser substituível, o contrato da classe base deve ser honrado pela classe derivada. Desde

A Circle não honra o contrato implícito da Ellipse, não é substituível e viola o LSP.

Tornar o contrato explícito é uma via de pesquisa seguida por Bertrand Meyer. Ele inventou uma linguagem chamada Eiffel na qual os contratos são explicitamente declarados para cada método e explicitamente verificados em cada invocação. Aqueles de nós que não estão usando Eiffel, têm que se contentar com simples afirmações e comentários.

Para declarar o contrato de um método, declaramos o que deve ser verdadeiro antes que o método seja chamado. Isso é chamado de pré-condição. Se a pré-condição falhar, os resultados do método são indefinidos e o método não deve ser chamado. Também declaramos o que o método garante que será verdadeiro depois de concluído. Isso é chamado de pós-condição.
Um método que falha em sua pós-condição não deve retornar.

Reafirmando o LSP mais uma vez, desta vez em termos de contratos, uma classe derivada é substituível por sua classe base se:

**1.** Suas pré-condições não são mais fortes que o método da classe base.

**2.** Suas pós-condições não são mais fracas que o método da classe base.

Ou, em outras palavras, os métodos derivados não devem *esperar mais nem fornecer menos.*

**Repercussões da violação de LSP.** Infelizmente, as violações de LSP são difíceis de detectar até que seja tarde demais. No caso Círculo/Elipse, tudo funcionou bem até que algum cliente apareceu e descobriu que o contrato implícito havia sido violado.

Se o design for muito usado, o custo de reparar a violação do LSP pode ser muito alto para suportar. Pode não ser econômico voltar e alterar o design e, em seguida, reconstruir e testar novamente todos os clientes existentes. Portanto, a solução provavelmente será colocar em um if/else no cliente que descobriu a violação. Esta instrução if/else verifica se a Elipse é realmente uma Elipse e não um Círculo. Consulte a Listagem 2-6.

**Listagem 2-6**

Correção feia para violação de LSP

```
void f(Elipse& e)
{
    if (typeid(e) == typeid(Elipse))
    {
        Ponto a(-1,0);
        Ponto b(1,0);
        e.SetFoci(a,b);
        e.SetMajorAxis(3);
        assert(e.GetFocusA() == a);
        assert(e.GetFocusB() == b);
```

**Listagem 2-6**
Correção feia para violação de LSP

```
        assert(e.GetMajorAxis() == 3);
    }
    senão
        throw NotAnElipse(e);
}
```

O exame cuidadoso da Listagem 2-6 mostrará que ela é uma violação do OCP. Agora, sempre que alguma nova derivada de Ellipse for criada, essa função terá que ser verificada para ver se deve ser permitida a operação sobre ela. *Assim, as violações do LSP são violações latentes do OCP.*

## O Princípio de Inversão de Dependência (DIP)1

*Depender de abstrações. Não dependa de concreções.*

Se o OCP declara o objetivo da arquitetura OO, o DIP indica o mecanismo principal.
A inversão de dependência é a estratégia de depender de interfaces ou funções e classes abstratas, em vez de funções e classes concretas. Este princípio é a força por trás do design de componentes, COM, CORBA, EJB, etc.

Os designs procedurais exibem um tipo particular de estrutura de dependência. Como mostra a Figura 2-17, essa estrutura começa no topo e aponta para os detalhes. Módulos de alto nível dependem de módulos de nível inferior, que dependem de módulos de nível ainda mais baixo, etc.

Um pouco de reflexão deve expor essa estrutura de dependência como intrinsecamente fraca. Os módulos de alto nível lidam com as políticas de alto nível da aplicação. Essas políticas geralmente se preocupam pouco com os detalhes que as implementam. Por que, então, esses módulos de alto nível devem depender diretamente desses módulos de implementação?

Uma arquitetura orientada a objetos mostra uma estrutura de dependência muito diferente, na qual a maioria das dependências aponta para abstrações. Além disso, os módulos que contêm implementação detalhada não são mais dependentes, mas sim *de* abstrações. Assim, a dependência deles foi *invertida.* Consulte a Figura 2-18.

**Dependendo das abstrações.** A implicação deste princípio é bastante simples. Cada dependência no design deve ter como alvo uma interface ou uma classe abstrata.
Nenhuma dependência deve ter como alvo uma classe concreta.

1. [DIP97]

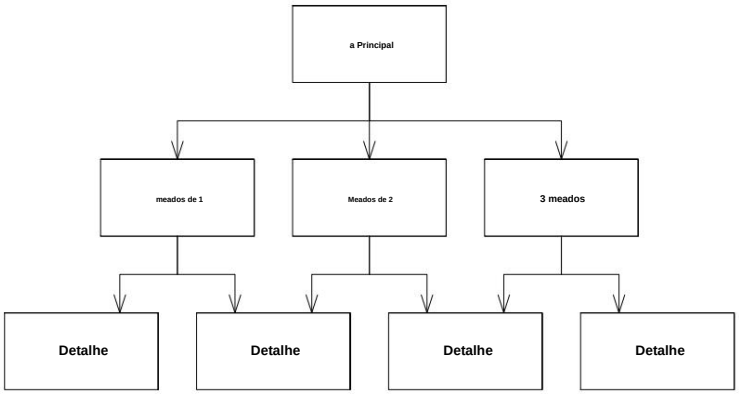


**Figura 2-17**
Estrutura de Dependência de uma Arquitetura Processual

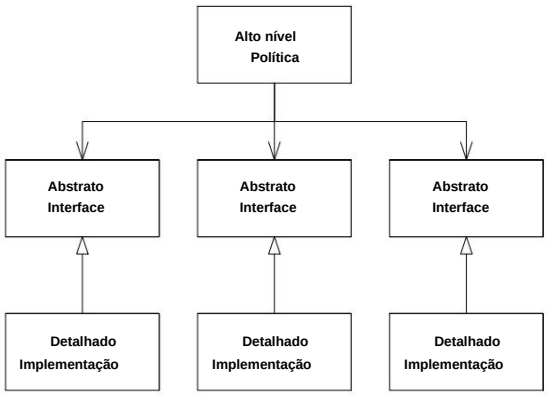


**Figura 2-18**
Estrutura de Dependência de uma Arquitetura Orientada a Objetos

Claramente tal restrição é draconiana, e existem circunstâncias atenuantes que iremos explorar momentaneamente. Mas, na medida do possível, o princípio deve ser seguido. A razão é simples, as coisas concretas mudam muito, as coisas abstratas mudam com muito menos frequência. Além disso, as abstrações são “pontos de articulação”, representam os lugares onde o projeto pode dobrar ou ser estendido, sem que eles mesmos sejam modificados (OCP).

Substratos como COM reforçam este princípio, pelo menos entre componentes. A única parte visível de um componente COM é sua interface abstrata. Assim, em COM, há pouca fuga do DIP.

**Forças Mitigantes.** Uma motivação por trás do DIP é evitar que você dependa de módulos voláteis. O DIP assume que qualquer coisa de concreto é volátil. Embora isso seja frequentemente assim, especialmente no desenvolvimento inicial, há exceções. Por exemplo, a biblioteca C padrão string.h é muito concreta, mas não é nada volátil. Depender dele em um ambiente de string ANSI não é prejudicial.
Da mesma forma, se você experimentou e testou módulos que são concretos, mas não voláteis, depender deles não é tão ruim. Como eles provavelmente não mudarão, eles provavelmente não injetarão volatilidade em seu design.

Tome cuidado no entanto. Uma dependência de string.h pode ficar muito feia quando os requisitos para o projeto forçarem você a mudar para caracteres UNICODE. A não volatilidade não substitui a substituibilidade de uma interface abstrata.

**Criação de Objetos.** Um dos lugares mais comuns em que os projetos dependem de classes concretas é quando esses projetos criam instâncias. Por definição, você não pode criar instâncias de classes abstratas. Assim, para criar uma instância, você deve depender de uma classe concreta.

A criação de instâncias pode acontecer por toda a arquitetura do projeto. Assim, pode parecer que não há escapatória e que toda a arquitetura estará repleta de dependências de classes concretas. No entanto, existe uma solução elegante para este problema chamada ABSTRACTORY [1] -- um padrão de projeto que examinaremos com mais detalhes no final deste capítulo.

## O Princípio de Segregação de Interface (ISP)[2]

*Muitas interfaces específicas do cliente são melhores do que uma interface de uso geral*

O ISP é outra das tecnologias que permitem suporte a substratos de componentes como COM. Sem ele, componentes e classes seriam muito menos úteis e portáveis.

A essência do princípio é bastante simples. Se você tem uma classe que tem vários clientes, ao invés de carregar a classe com todos os métodos que os clientes precisam, crie interfaces específicas para cada cliente e multiplique-as herdando-as na classe.

1. [GOF96] p??
2. [ISP97]

A Figura 2-19 mostra uma classe com muitos clientes e uma grande interface para atender a todos eles. Observe que sempre que uma alteração é feita em um dos métodos que ClientA chama, ClientB e ClientC podem ser afetados. Pode ser necessário recompilá-los e reimplantá-los. Isto é um infortúnio.

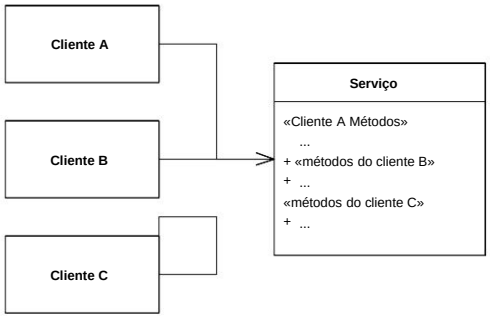


**Figura 2-19**
Fat Service com interfaces integradas

Uma técnica melhor é mostrada na Figura 2-20. Os métodos necessários para cada cliente são colocados em interfaces especiais que são específicas para aquele cliente. Essas interfaces são multiplamente herdadas pela classe Service e implementadas lá.

Se a interface para ClientA precisar ser alterada, ClientB e ClientC permanecerão inalterados. Eles não precisarão ser recompilados ou reimplantados.

**O que significa Específico do Cliente?** O ISP não recomenda que cada classe que usa um serviço tenha sua própria classe de interface especial da qual o serviço deve herdar. Se assim fosse, o serviço dependeria de cada cliente de uma forma bizarra e insalubre. Em vez disso, os clientes devem ser categorizados por tipo e as interfaces para cada tipo de cliente devem ser criadas.

Se dois ou mais tipos de clientes diferentes precisarem do mesmo método, o método deverá ser adicionado a ambas as interfaces. Isso não é prejudicial nem confuso para o cliente.

**Alterando Interfaces.** Quando os aplicativos orientados a objetos são mantidos, as interfaces para classes e componentes existentes geralmente mudam. Há momentos em que essas mudanças têm um impacto enorme e forçam a recompilação e a redistribuição de uma parte muito grande do design. Esse impacto pode ser mitigado adicionando novas interfaces a objetos existentes, em vez de alterar a interface existente. Clientes do antigo inter

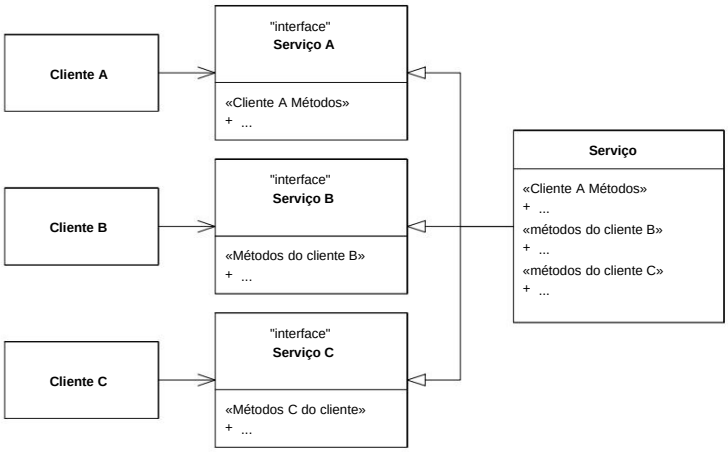


**Figura 2-20**
Interfaces Segregadas

face que desejam acessar os métodos da nova interface, podem consultar o objeto dessa interface conforme mostrado no código a seguir.

```
void Cliente(Serviço* s)
{
    if (NewService* ns = dynamic_cast<NewService*>(s))
    {
        // usa a nova interface de serviço
    }
}
```

Como acontece com todos os princípios, deve-se tomar cuidado para não exagerar. O espectro de uma classe com centenas de interfaces diferentes, algumas segregadas por cliente e outras segregadas por versão, seria realmente assustador.

# Princípios da Arquitetura de Pacotes

As aulas são um meio necessário, mas insuficiente, de organizar um projeto. A maior granularidade dos pacotes é necessária para ajudar a trazer ordem. Mas como escolhemos quais classes pertencem a quais pacotes. Abaixo estão três princípios conhecidos como *Princípios de Coesão de Pacotes,* que tentam ajudar o arquiteto de software.

### O Princípio de Equivalência de Reutilização de Liberação (REP)1

*O grânulo de reutilização é o grânulo de liberação.*

Um elemento reutilizável, seja um componente, uma classe ou um cluster de classes, não pode ser reutilizado a menos que seja gerenciado por algum tipo de sistema de liberação. Os usuários não estarão dispostos a usar o elemento se forem forçados a atualizar toda vez que o autor o alterar. Desta forma. mesmo que o autor tenha lançado uma nova versão de seu elemento reutilizável, ele deve estar disposto a oferecer suporte e manter versões mais antigas enquanto seus clientes se preparam lentamente para atualizar. Assim, os clientes se recusarão a reutilizar um elemento a menos que o autor prometa manter o controle dos números de versão e manter as versões antigas por algum tempo.

Portanto, um critério para agrupar classes em pacotes é a reutilização. Como os pacotes são a unidade de liberação, eles também são a unidade de reutilização. Portanto, os arquitetos fariam bem em agrupar classes reutilizáveis em pacotes.

### O Princípio de Fechamento Comum (PCC)2

*Classes que mudam juntas, pertencem juntas.*

Um grande projeto de desenvolvimento é subdividido em uma grande rede de pacotes inter-relacionados. O trabalho para gerenciar, testar e liberar esses pacotes não é trivial. Quanto mais pacotes forem alterados em uma determinada versão, maior será o trabalho para reconstruir, testar e implantar a versão. Portanto, gostaríamos de minimizar o número de pacotes que são alterados em qualquer ciclo de lançamento do produto.

Para isso, agrupamos turmas que achamos que vão mudar juntas. Isso requer uma certa quantidade de precisão, uma vez que devemos antecipar os tipos de mudanças que são prováveis. Ainda assim, quando agrupamos classes que mudam juntas nos mesmos pacotes, o impacto do pacote de lançamento para lançamento será minimizado.

### O Princípio de Reutilização Comum (PCR)3

*As classes que não são reutilizadas juntas não devem ser agrupadas.*

Uma dependência de um pacote é uma dependência de tudo dentro do pacote.
Quando um pacote muda e seu número de lançamento é aumentado, todos os clientes desse pacote

1. [Granularidade 97]
2. [Granularidade 97]
3. [Granularidade 97]

age deve verificar se eles funcionam com o novo pacote -- mesmo se nada que eles usaram no pacote realmente tenha mudado.

Frequentemente experimentamos isso quando nosso fornecedor de SO lança um novo sistema operacional. Temos que atualizar mais cedo ou mais tarde, porque o fornecedor não suportará a versão antiga para sempre. Portanto, embora nada de nosso interesse tenha mudado na nova versão, devemos passar pelo esforço de atualização e revalidação.

O mesmo pode acontecer com pacotes se classes que não são usadas juntas forem agrupadas. Alterações em uma classe com a qual não me importo ainda forçarão um novo lançamento do pacote e ainda me farão passar pelo esforço de atualização e revalidação.

## Tensão entre os Princípios de Coesão da Embalagem

Esses três princípios são mutuamente exclusivos. Eles não podem ser satisfeitos simultaneamente. Isso porque cada princípio beneficia um grupo diferente de pessoas. O REP e o CRP facilitam a vida dos reutilizadores, enquanto o CCP facilita a vida dos mantenedores.
O CCP se esforça para fazer os pacotes tão grandes quanto possível (afinal, se todas as classes estiverem em apenas um pacote, então apenas um pacote mudará). O CRP, no entanto, tenta fazer pacotes muito pequenos.

Felizmente, os pacotes não são fixos em pedra. De fato, é da natureza dos pacotes mudar e tremer durante o curso do desenvolvimento. No início de um projeto, os arquitetos podem configurar a estrutura do pacote de forma que o CCP domine e o desenvolvimento e a manutenção sejam auxiliados. Mais tarde, à medida que a arquitetura se estabiliza, os arquitetos podem refatorar a estrutura do pacote para maximizar REP e CRP para os reutilizadores externos.

## Os Princípios de Acoplamento de Pacotes.

Os próximos três pacotes governam as inter-relações entre os pacotes. As aplicações tendem a ser grandes redes de pacotes interligados. As regras que governam essas inter-relações são algumas das regras mais importantes na arquitetura orientada a objetos.

## O Princípio das Dependências Acíclicas (ADP)1

*As dependências entre os pacotes não devem formar ciclos.*

Como os pacotes são o grânulo da liberação, eles também tendem a concentrar a mão de obra. Os engenheiros normalmente trabalham dentro de um único pacote em vez de trabalhar em dezenas. Essa tendência é amplificada pelos princípios de coesão do pacote, uma vez que tendem a agrupar

1. [Granularidade 97]

juntas as classes que estão relacionadas. Assim, os engenheiros descobrirão que suas mudanças são direcionadas para apenas alguns pacotes. Depois que essas alterações forem feitas, eles poderão liberar esses pacotes para o restante do projeto.

Antes que eles possam fazer este lançamento, no entanto, eles devem testar se o pacote funciona. Para fazer isso, eles devem compilá-lo e construí-lo com todos os pacotes dos quais depende.
Espero que esse número seja pequeno.

Considere a Figura 2-21. Leitores astutos reconhecerão que há uma série de falhas na arquitetura. O DIP parece ter sido abandonado, e junto com ele o OCP. A GUI depende diretamente do pacote de comunicação e aparentemente é responsável por transportar os dados para o pacote de análise. Eca.

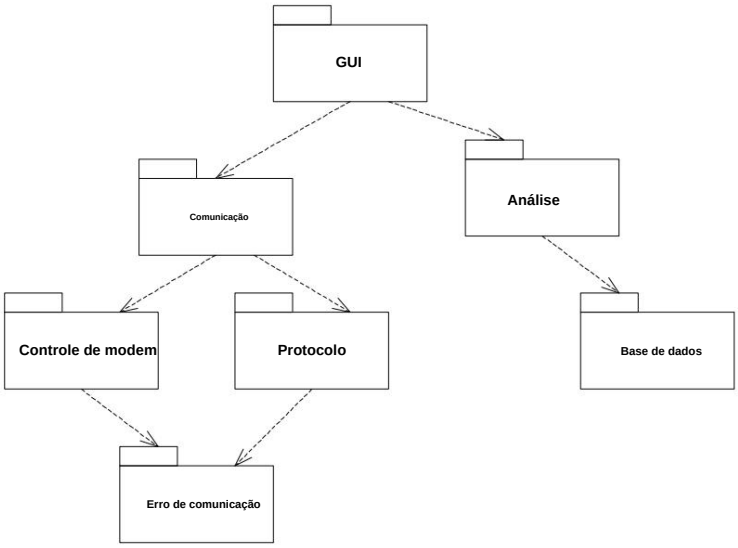


**Figura 2-21**
Rede de Pacotes Acíclicos

Ainda assim, vamos usar essa estrutura bastante feia para alguns exemplos. Considere o que seria necessário para liberar o pacote Protocol. Os engenheiros teriam que construí-lo com a versão mais recente do pacote CommError e executar seus testes. Protocolo
não tem outras dependências, então nenhum outro pacote é necessário. Isso é legal. Podemos testar e liberar com uma quantidade mínima de trabalho.

**Um ciclo se aproxima.** Mas agora vamos dizer que eu sou um engenheiro trabalhando no pacote CommError. Decidi que preciso exibir uma mensagem na tela. Como a tela é controlada pela GUI, envio uma mensagem para um dos objetos da GUI para que minha mensagem apareça na tela. Isso significa que eu fiz CommError dependente da GUI. Consulte a Figura 2-22.

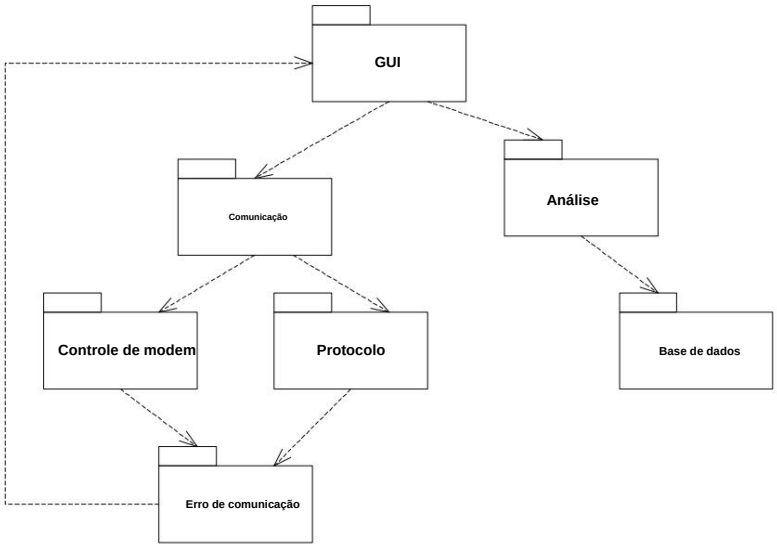


**Figura 2-22**
Um ciclo foi adicionado.

Agora o que acontece quando os caras que estão trabalhando no protocolo querem liberar seu pacote. Eles precisam construir seu conjunto de testes com CommError, GUI, Comm, ModemControl, Analysis e Database! Isso é claramente desastroso. A carga de trabalho dos engenheiros foi aumentada em uma quantidade abominável, devido a uma única pequena dependência que ficou fora de controle.

Isso significa que alguém precisa observar a estrutura de dependência do pacote com regularidade e quebrar ciclos onde quer que eles apareçam. Caso contrário, as dependências transitivas entre os módulos farão com que cada módulo dependa de todos os outros módulos.

**Quebrando um Ciclo.** Os ciclos podem ser quebrados de duas maneiras. A primeira envolve a criação de um novo pacote e a segunda faz uso do DIP e do ISP.

A Figura 2-23 mostra como quebrar o ciclo adicionando um novo pacote. As classes que o CommError precisava são retiradas da GUI e colocadas em um novo pacote chamado MessageManager. Tanto a GUI quanto o CommError são feitos para depender deste novo pacote.

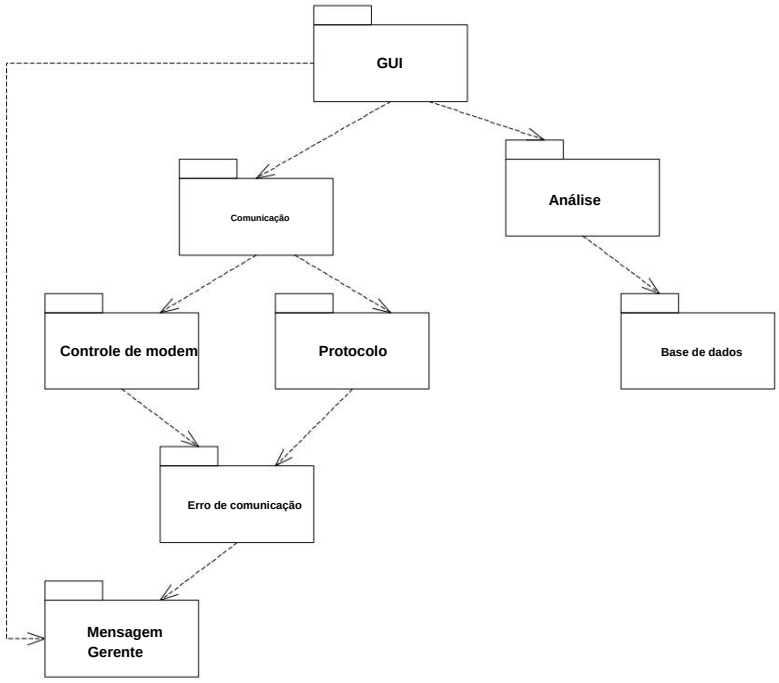


**Figura 2-23**

Este é um exemplo de como a estrutura do pacote tende a oscilar e mudar durante o desenvolvimento. Novos pacotes passam a existir e as classes passam de pacotes antigos para novos pacotes, para ajudar a quebrar os ciclos.

A Figura 2-24 mostra uma imagem antes e depois da outra técnica para quebrar ciclos. Aqui vemos dois pacotes que estão vinculados por um ciclo. A classe A depende da classe X e a classe Y depende da classe B. Quebramos o ciclo invertendo a dependência entre Y e B. Isso é feito adicionando uma nova interface, BY, a B. Essa interface tem todos os métodos que Y precisa. Y usa essa interface e B a implementa.

Observe a colocação de BY. Ele é colocado no pacote com a classe que o utiliza. Este é um padrão que você verá repetido ao longo dos estudos de caso que tratam de pacotes

idades. As interfaces são frequentemente incluídas no pacote que as utiliza, e não no pacote que as implementa.

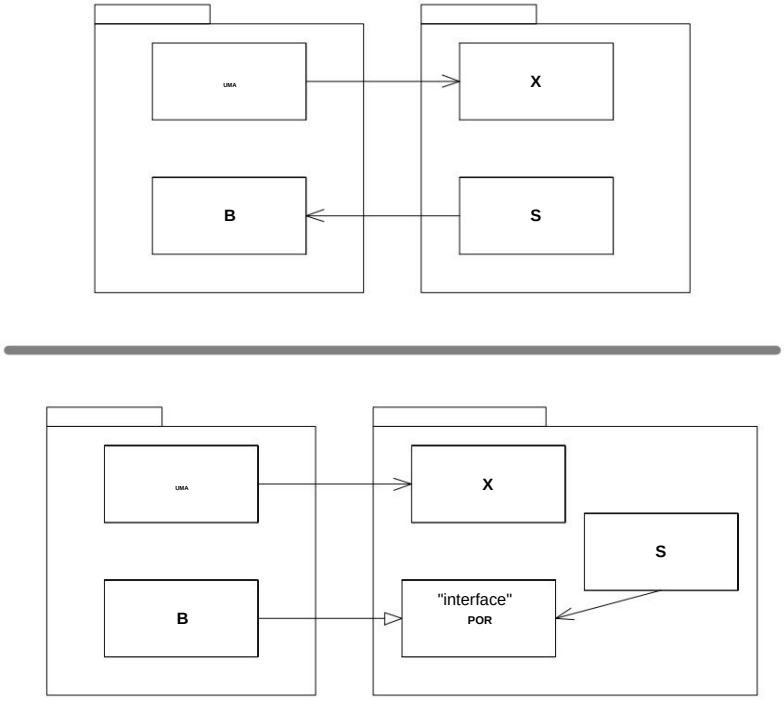


**Figura 2-24**

## O Princípio de Dependências Estáveis (SDP)1

*Dependem na direção da estabilidade.*

Embora este pareça ser um princípio óbvio, há muito que podemos dizer sobre isso.
A estabilidade nem sempre é bem compreendida.

**Estabilidade.** O que se entende por estabilidade? Coloque um centavo de lado. Está estável nessa posição?
Provavelmente você diria que não. No entanto, a menos que seja perturbado, ele permanecerá nessa posição

1. [Estabilidade 97]

durante muito, muito tempo. Assim, a estabilidade não tem nada a ver diretamente com a frequência da mudança. O centavo não está mudando, mas é difícil pensar nele como estável.

A estabilidade está relacionada à quantidade de trabalho necessária para fazer uma mudança. A moeda não é estável porque requer muito pouco trabalho para derrubá-la. Por outro lado, uma mesa é muito estável porque exige um esforço considerável para virá-la.

Como isso se relaciona com o software? Há muitos fatores que tornam um pacote de software difícil de mudar. Seu tamanho, complexidade, clareza, etc. Vamos ignorar todos esses fatores e focar em algo diferente. Uma maneira segura de dificultar a mudança de um pacote de software é fazer com que muitos outros pacotes de software dependam dele. Um pacote com muitas dependências de entrada é muito estável porque requer muito trabalho para reconciliar quaisquer alterações com todos os pacotes dependentes.

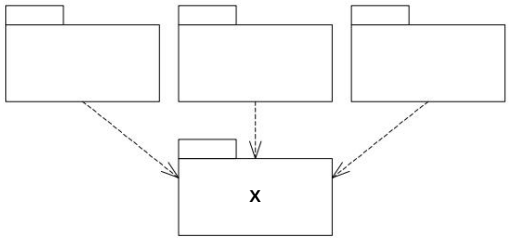


**Figura 2-25**
X é um pacote estável

A Figura 2-25 mostra X: um pacote estável. Este pacote tem três pacotes dependendo dele e, portanto, tem três boas razões para não mudar. Dizemos que é *responsável* por esses três pacotes. Por outro lado, X não depende de nada, portanto não tem influência externa para fazê-lo mudar. Dizemos que é *independente.*

A Figura 2-26, por outro lado, mostra um pacote muito instável. Y não tem outros pacotes que dependam dele; dizemos que é irresponsável. Y também possui três pacotes dos quais depende, portanto, as alterações podem vir de três fontes externas. Dizemos que Y é dependente.

**Métricas de Estabilidade.** Podemos calcular a estabilidade de um pacote usando um trio de métricas simples.

*Ca* Acoplamento Aferente. O número de classes fora do pacote que dependem das classes dentro do pacote. (ou seja, dependências de entrada)

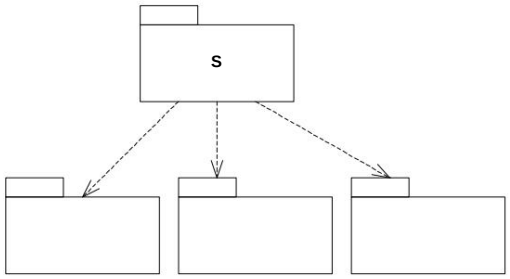


**Figura 2-26**
Y é instável.

Acoplamento Eferente *Ce* . O número de classes fora do pacote das quais as classes dentro do pacote dependem. (ou seja, dependências de saída)

$I_{EU}$ Instabilidade. $I_{EU} = \frac{Ce}{Ca + Ce}$ . Esta é uma métrica que tem o intervalo: [0,1].

Se não houver dependências de saída, eu serei zero e o pacote será estável. Se não houver dependências de entrada, eu serei uma e o pacote será instável.

Agora podemos reformular o SDP da seguinte forma: “Depende de pacotes cuja métrica I seja menor que a sua”.

**Justificativa.** Todos os softwares devem ser estáveis? Um dos atributos mais importantes de um software bem projetado é a facilidade de mudança. O software que é flexível na presença de mudanças de requisitos é bem pensado. No entanto, esse software é instável por nossa definição. Na verdade, desejamos muito que partes de nosso software sejam instáveis. Queremos que certos módulos sejam fáceis de mudar para que, quando os requisitos mudarem, o projeto possa responder com facilidade.

A Figura 2-27 mostra como o SDP pode ser violado. Flexível é um pacote que pretendemos que seja fácil de mudar. Queremos que o Flexível seja instável. No entanto, algum engenheiro, trabalhando no pacote chamado Stable, pendurou uma dependência no Flexible. Isso viola o SDP, pois a métrica I para Estável é muito menor que a métrica I para Flexível. Como resultado, o Flexível não será mais fácil de mudar. Uma mudança para Flexível nos forçará a lidar com a Estável e todos os seus dependentes.

## O Princípio de Abstrações Estáveis (SAP)1

*Pacotes estáveis devem ser pacotes abstratos.*

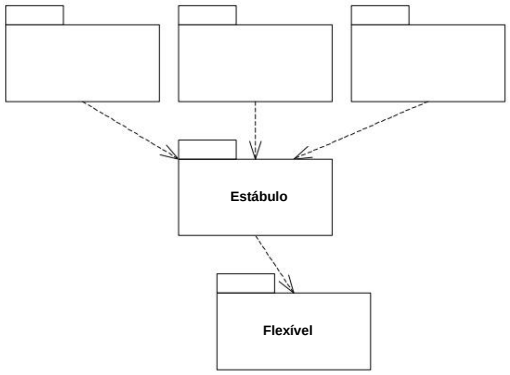


**Figura 2-27**
Violação do SDP.

Podemos imaginar a estrutura de pacotes do nosso aplicativo como um conjunto de pacotes interconectados com pacotes instáveis na parte superior e pacotes estáveis na parte inferior. Nesta visão, todas as dependências apontam para baixo.

Esses pacotes no topo são instáveis e flexíveis. Mas aqueles na parte inferior são muito difíceis de mudar. E isso nos leva a um dilema: queremos embalagens em nosso design que sejam difíceis de mudar?

Claramente, quanto mais pacotes são difíceis de mudar, menos flexível será nosso design geral. No entanto, há uma brecha pela qual podemos rastejar. Os pacotes altamente estáveis na parte inferior da rede de dependências podem ser muito difíceis de mudar, mas de acordo com o OCP eles não precisam ser difíceis de estender!

Se os pacotes estáveis na parte inferior também forem altamente abstratos, eles poderão ser facilmente estendidos. Isso significa que é possível compor nossa aplicação a partir de pacotes instáveis que são fáceis de alterar e pacotes estáveis que são fáceis de estender. Isto é uma coisa boa.

Assim, o SAP é apenas uma reformulação do DIP. Ele afirma que os pacotes que são mais dependentes (ou seja, estáveis) também devem ser os mais abstratos. Mas como medimos a abstração?

**As Métricas de Abstração.** Podemos derivar outro trio de métricas para nos ajudar a calcular a abstração.

1. [Estabilidade 97]

*Nc* Número de classes no pacote.

*Na* Número de classes abstratas no pacote. Lembre-se, uma classe abstrata é uma classe com pelo menos uma interface pura e não pode ser instanciada.

*Uma* Abstração. $_{UMA} = \frac{N / D}{Nc}$

A métrica A tem um intervalo de [0,1], assim como a métrica I. Um valor zero significa que o pacote não contém classes abstratas. Um valor de um significa que o pacote não contém nada além de classes abstratas.

**O gráfico I vs A.** O SAP agora pode ser reapresentado em termos das métricas I e A: devo aumentar à medida que A diminui. Ou seja, pacotes concretos devem ser instáveis enquanto pacotes abstratos devem ser estáveis. Podemos plotar isso graficamente no gráfico A vs I.
Consulte a Figura 2-28.

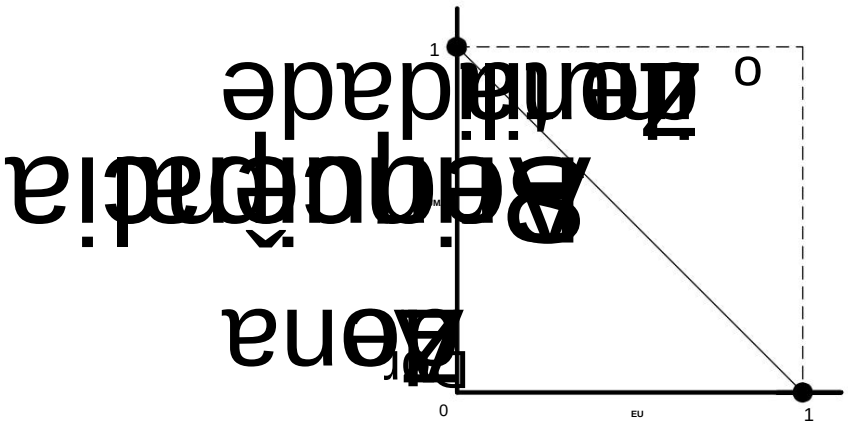


**Figura 2-28**
O gráfico A vs I.

Parece claro que os pacotes devem aparecer em qualquer um dos dois pontos pretos na Figura 2-28. Os do canto superior esquerdo são completamente abstratos e muito estáveis. Os do canto inferior direito são completamente concretos e muito instáveis. Esta é apenas a maneira como gostamos. E quanto ao pacote X na Figura 2-29? Onde deve ir?

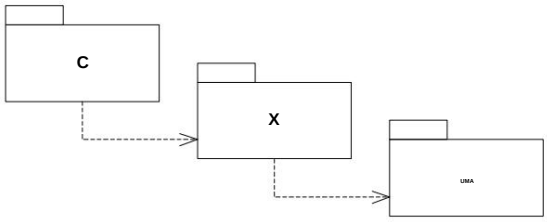


**Figura 2-29**
O que colocamos X no gráfico A vs I?

Podemos determinar onde queremos o Pacote X, olhando para onde não queremos que ele vá. O canto superior direito do gráfico AI representa pacotes que são altamente abstratos e dos quais ninguém depende. Esta é a zona de inutilidade. Certamente não queremos que X more lá. Por outro lado, o ponto inferior esquerdo do gráfico AI representa pacotes que são concretos e têm muitas dependências de entrada. Este ponto representa o pior caso para um pacote. Uma vez que os elementos ali são concretos, eles não podem ser estendidos da mesma forma que as entidades abstratas podem; e como eles têm muitas dependências de entrada, a mudança será muito dolorosa. Esta é a zona de dor, e certamente não queremos que nosso pacote more lá.

Maximizar a distância entre essas duas zonas nos dá uma linha chamada de *sequência principal.* Gostaríamos que nossos pacotes ficassem nesta linha, se possível. Uma posição nesta linha significa que o pacote é abstrato na proporção de suas dependências de entrada e é concreto na proporção de suas dependências de saída. Em outras palavras, as classes em tal pacote estão em conformidade com o DIP.

**Métricas de distância.** Isso nos deixa mais um conjunto de métricas para examinar. Dados os valores A e I de qualquer pacote, gostaríamos de saber a que distância esse pacote está da sequência principal.

*D* Distância. $D = \frac{|A + I - 1|}{\sqrt{2}}$. Isso varia de [0,~0,707].

D' Distância Normalizada. $D' = |A + I - 1|$. Essa métrica é muito mais conveniente que D, pois varia de [0,1]. Zero indica que o pacote está diretamente na sequência principal. Um indica que o pacote está o mais distante possível da sequência principal.

Essas métricas medem a arquitetura orientada a objetos. Eles são imperfeitos, e confiar neles como o único indicador de uma arquitetura robusta seria imprudente. No entanto, eles podem ser, e têm sido, usados para ajudar a medir a estrutura de dependência de um aplicativo.

# Padrões de Arquitetura Orientada a Objetos

Ao seguir os princípios descritos acima para criar arquiteturas orientadas a objetos, descobre-se , que se repete as mesmas estruturas repetidamente. Esses repetindo
estruturas de design e arquitetura são conhecidas como padrões de design1 .

A definição essencial de um padrão de projeto é uma boa solução bem conhecida e usada para um problema comum. Os padrões de design definitivamente não são novos. Pelo contrário, são técnicas antigas que mostraram sua utilidade por um período de muitos anos.

Alguns padrões de projeto comuns são descritos abaixo. Esses são os padrões que você encontrará ao ler os estudos de caso mais adiante no livro.

Deve-se notar que o tópico de Design Patterns não pode ser abordado adequadamente em um único capítulo de um único livro. Os leitores interessados são fortemente encorajados a ler [GOF96].

## Servidor abstrato

Quando um cliente depende diretamente de um servidor, o DIP é violado. As alterações no servidor se propagarão para o cliente, e o cliente não poderá usar facilmente servidores semelhantes. Isso pode ser corrigido inserindo uma interface abstrata entre o cliente e o servidor, conforme mostrado na Figura 2-30.

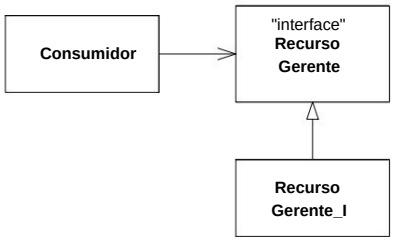


**Figura 2-30**
Servidor abstrato

A interface abstrata torna-se um “ponto de articulação” sobre o qual o design pode se flexionar. Diferentes implementações do servidor podem ser vinculadas a um cliente desavisado.

1. [GOF96]

## Adaptador

Quando inserir uma interface abstrata é inviável porque o servidor é um software de terceiros, ou é tão dependente que não pode ser alterado facilmente, um ADAPTADOR pode ser usado para vincular a interface abstrata ao servidor. Consulte a Figura 2-31.

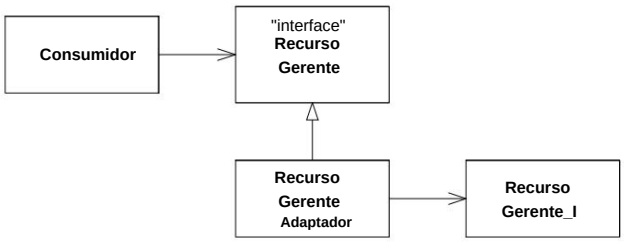


**Figura 2-31**
Adaptador

O adaptador é um objeto que implementa a interface abstrata para delegar ao servidor. Cada método do adaptador simplesmente traduz e depois delega.

## Observador

Muitas vezes ocorre que um elemento de um projeto precisa tomar alguma forma de ação quando outro elemento do projeto descobre que um evento ocorreu. No entanto, frequentemente não queremos que o detector saiba sobre o ator.

Considere o caso de um medidor que mostra o status de um sensor. Toda vez que o sensor muda sua leitura, queremos que o medidor exiba o novo valor. No entanto, não queremos que o sensor saiba nada sobre o medidor.

Podemos resolver esta situação com um OBSERVER, veja a Figura 2-32. O Sensor deriva de uma classe chamada Subject e Meter deriva de uma interface chamada Observer. O assunto contém uma lista de Observadores. Esta lista é carregada pelo método Regsiter de Subject. Para ser informado dos eventos, nosso Medidor deve se registrar com a classe base Subject do Sensor.

A Figura 2-33 descreve a dinâmica da colaboração. Alguma entidade passa o controle para o Sensor que determina que sua leitura foi alterada. O Sensor chama Notificar em seu Assunto. O Sujeito então percorre todos os Observadores

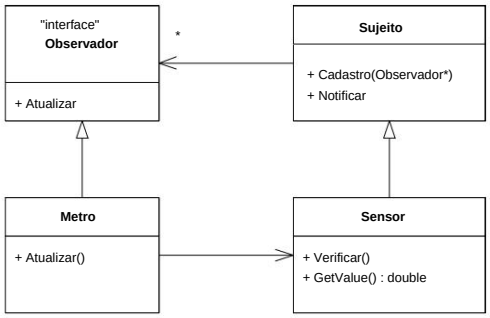


**Figura 2-32**
Estrutura do Observador

que foram registrados, chamando Update em cada um. A mensagem de atualização é capturada pelo Medidor que a utiliza para ler o novo valor do Sensor e exibi-lo.

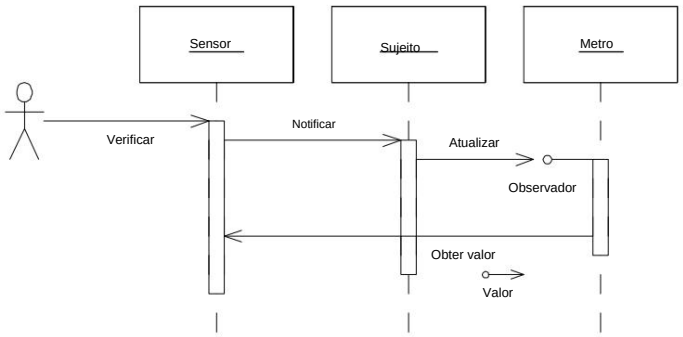


**Figura 2-33**

# Ponte

Um dos problemas com a implementação de uma classe abstrata com herança é que a classe derivada é fortemente acoplada à classe base. Isso pode levar a problemas quando outros clientes desejam usar as funções de classe derivadas sem arrastar o medidor de bagagem da hierarquia base.

Por exemplo, considere uma classe de sintetizador de música. A classe base converte a entrada MIDI em um conjunto de chamadas primitivas do EmitVoice que são implementadas por uma classe derivada. Observe que a função EmitVoice da classe derivada seria útil por si só. Infelizmente, está inextricavelmente ligado à classe MusicSynthesizer e à função PlayMidi. Não há como acessar o método PlayVoice sem arrastar a classe base com ele. Além disso, não há como criar diferentes implementações da função PlayMidi que usem a mesma função EmitVoice. Em suma, a hierarquia é apenas acoplada.

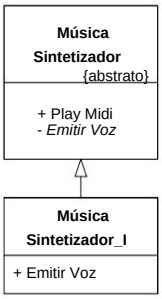


**Figura 2-34**
Hierarquia mal acoplada

O padrão BRIDGE resolve esse problema criando uma forte separação entre a interface e a implementação. A Figura 2-35 mostra como isso funciona. A classe MusicSynthesizer contém uma função abstrata PlayMidi que é implementada por MusicSynthesizer_I. Ele chama a função EmitVoice que é implementada no MusicSynthesizer para delegar à interface VoiceEmitter. Esta interface é implementada por VoiceEmitter_I e emite os sons necessários.

Agora é possível implementar o EmitVoice e o PlayMidi separadamente um do outro. As duas funções foram dissociadas. O EmitVoice pode ser chamado sem trazer toda a bagagem do MusicSynthesizer, e o PlayMidi pode ser implementado de várias maneiras diferentes, enquanto ainda usa o mesmo EmitVoice função.

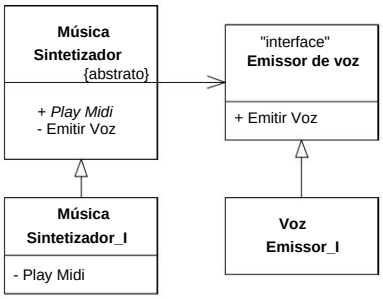


**Figura 2-35**
Hierarquia desacoplada com Bridge

## Fábrica Abstrata

O DIP recomenda fortemente que os módulos não dependam de classes concretas. No entanto, para criar uma instância de uma classe, você deve depender da classe concreta. ABSTRACTFACTORY é um padrão que permite que a dependência da classe concreta exista em um e apenas um lugar.

A Figura 2-36 mostra como isso é feito para o exemplo do Modem. Todos os usuários que desejam criar modems utilizam uma interface chamada ModemFactory. Um ponteiro para essa interface é mantido em uma variável global chamada GtheFactory. Os usuários chamam a função Make passando uma string que define exclusivamente a subclasse particular de Modem que eles desejam. A função Make retorna um ponteiro para uma interface de Modem.

A interface ModemFactory implementada por ModemFactory_I. Esta classe é criada por main, e um ponteiro para ela é carregado no GtheFactory global. Assim, nenhum módulo no sistema sabe sobre as classes concretas do modem, exceto ModemFactory_I, e nenhum módulo sabe sobre ModemFactory_I, exceto para
a Principal.

# Conclusão

Este capítulo introduziu o conceito de arquitetura orientada a objetos e a definiu como a estrutura de classes e pacotes que mantém o aplicativo de software flexível, robusto, reutilizável e desenvolvível. Os princípios e padrões aqui apresentados suportam

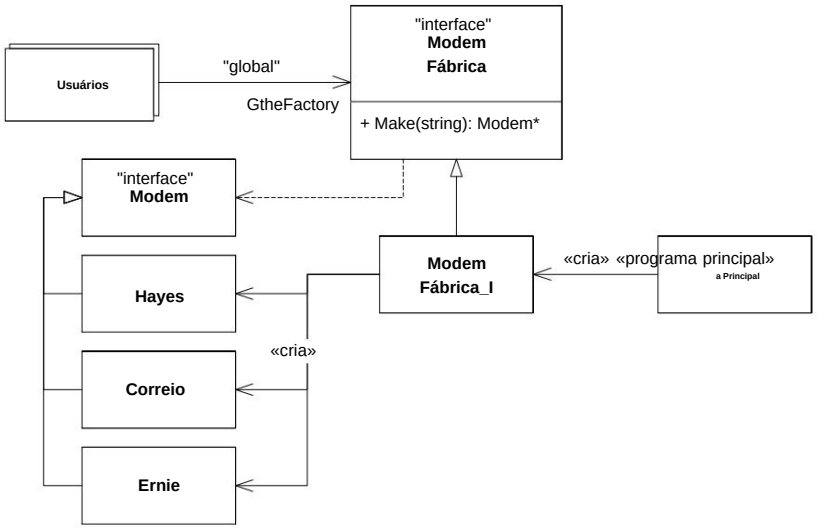


**Figura 2-36**
Fábrica Abstrata

tais arquiteturas, e foram comprovados ao longo do tempo como auxiliares poderosos na arquitetura de software.

Esta foi uma visão geral. Há muito mais a ser dito sobre o tópico da arquitetura OO do que pode ser dito nas poucas páginas deste capítulo, de fato, ao encurtar tanto o tópico, corremos o risco de prestar um desserviço ao leitor. Foi dito que um pouco de conhecimento é uma coisa perigosa, e este capítulo forneceu um pouco de conhecimento. Recomendamos fortemente que você procure os livros e artigos nas citações deste capítulo para saber mais.

# Bibliografia

**[Shaw96]:** Padrões de Arquitetura de Software (???), Garlan e Shaw, ...

**[GOF96]:** Padrões de Design...

**[OOSC98]:** OOSC...

**[OCP97]:** O Princípio Aberto Fechado, Robert C. Martin...

**[LSP97]:** O Princípio da Substituição de Liskov, Robert C. Martin

**[DIP97]:** O Princípio da Inversão de Dependência, Robert C. Martin

**[ISP97]:** O Princípio de Segregação de Interface, Robert C. Martin

**[Granularidade97]:** Granularidade, Robert C. Martin

**[Estabilidade97]:** Estabilidade, Robert C. Martin

**[Liksov88]:** Abstração de dados e hierarquia...

**[Martin99]:** *Projetando Aplicativos Orientados a Objetos usando UML,* 2d. ed., Roberto C. Martin, Prentice Hall, 1999.