Princípios de Design e Padrões de Projeto

Robert C. Martin

Tradução de Ricardo Pereira

(http://www.ricardopdias.com.br)

Índice

Introdução

O que é arquitetura de software? A resposta é multicamada. No nível mais alto, existem os padrões de arquitetura que definem a forma geral e a estrutura dos aplicativos de software[1]. Um nível abaixo está a arquitetura relacionada especificamente 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 projeto[2], pacotes, componentes e classes. É neste nível que nos interessaremos neste capítulo.

Nosso escopo neste capítulo é bastante limitado. Há muito mais a ser dito sobre os princípios e padrões expostos aqui. Os leitores interessados leiam MARTIN (1999).

1 Shaw96
2 GOF96

1. Arquitetura e Dependências

O que há de errado com o software? O design de muitos aplicativos de software começa como uma imagem vital nas mentes de seus projetistas. Nesta fase, ele é limpo, elegante e atraente. Tem uma beleza simples que faz os projetistas e implementadores desejarem vê-lo funcionando. Algumas dessas aplicações conseguem manter essa pureza de design passando pelo desenvolvimento inicial e na primeira versão.

Mas então algo começa a acontecer. O software começa a apodrecer. No começo não é tão ruim assim. Uma verruga feia aqui, um hack desajeitado ali, mas a beleza do design ainda transparece. No entanto, com o tempo, à medida que a podridão continua, as feias feridas e as infamações se acumulam até dominarem o design da aplicação. O programa se torna uma massa purulenta de código que os desenvolvedores acham cada vez mais difícil de manter. Eventualmente, o esforço necessário para fazer até mesmo as mais simples mudanças no aplicativo torna-se tão alto que os engenheiros e gerentes de linha de frente clamam por um projeto de reformulação.

Tais redesigns raramente são bem sucedidos. Embora os designers iniciem com boas intenções, eles acham que estão atirando em um alvo em movimento. O sistema antigo continua evoluindo e mudando, e o novo design deve continuar. As verrugas e úlceras se acumulam no novo design antes que ele chegue ao seu primeiro lançamento. Naquele dia fatídico, geralmente muito depois do planejado, o emaranhado de problemas no novo design pode ser tão ruim que os designers já estão implorando por outro redesign.

Sintomas de Design Podre

Existem quatro sintomas primários que revelam o estado de apodrecimento de nossos projetos. 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 um software ser difícil de mudar, mesmo de formas simples. Cada alteração causa uma cascata de alterações subsequentes em módulos dependentes. O que começa como uma simples mudança de dois dias para um módulo cresce em uma maratona de mudanças de várias semanas em módulo após módulo, à medida que os engenheiros perseguem o segmento da mudança através do aplicativo.

Quando o software se comporta dessa maneira, os gerentes tem receio de permitir que os engenheiros consertem problemas não críticos. Essa relutância deriva do fato de que eles não sabem, com alguma confiabilidade, quando os engenheiros estarão prontos. Se os gerentes soltarem os engenheiros em tais problemas, eles poderão desaparecer por longos períodos de tempo. O design do software começa a assumir algumas características de um *Roach Motel* (*Hotel de Baratas)* [3] - os engenheiros fazem o check-in, mas não conseguem fazer o check-out.

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

Fragilidade

Intimamente relacionado à rigidez está a fragilidade. Fragilidade é a tendência do software quebrar em muitos lugares toda vez que ele é alterado. Muitas vezes, a quebra ocorre em áreas que não têm relação conceitual com a área que foi alterada. Tais erros preenchem os corações dos gerentes com pressentimento. Toda vez que eles autorizam uma correção, eles temem que o software seja interrompido de alguma forma inesperada.

Com o tempo, à medida que a fragilidade se torna pior, a probabilidade de quebra aumenta gradativamente. Tal software é impossível de manter. Cada correção piora, introduzindo mais problemas do que soluções.

Tal software faz com que gerentes e clientes suspeitem que os desenvolvedores tenham perdido o controle de seu software. A desconfiança reina e a credibilidade é perdida.

Imobilidade

Imobilidade é a incapacidade de reutilizar software de outros projetos ou de partes do mesmo projeto. Muitas vezes acontece que um engenheiro descobrirá que precisa de um módulo semelhante ao que outro engenheiro escreveu. No entanto, também acontece frequentemente que o módulo em questão tenha muita bagagem das quais ele 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.

3 Roach Motel (ou “Hotel de Baratas”, em português) é uma marca de dispositivo para capturar baratas. O termo passou a ser usado como referência para todas as armadilhas que usam um perfume ou outra forma de isca para atrair baratas até um compartimento no qual uma substância pegajosa faz com que elas fiquem presas.

Viscosidade

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 formas preservam o design, outras não (por exemplo, hacks). Quando os métodos de preservação de design são mais difíceis de empregar do que os hacks, a viscosidade do design é alta. É fácil fazer a coisa errada, mas é 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 forçam grandes recompilações, mesmo que essas alterações não sejam ideais do ponto de vista do design. Se o sistema de controle de código-fonte precisar de horas para registrar apenas alguns arquivos, os engenheiros ficarão tentados a fazer alterações que exigem o menor número possível de checagens, independentemente de o design ser preservado.

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

Mudando Requisitos

A causa imediata da degradação do design é bem compreendida. Os requisitos foram alterados de maneiras que o design inicial não previa. Muitas vezes, essas mudanças precisam ser feitas rapidamente e podem ser feitas por engenheiros que não estão familiarizados com a filosofia original do projeto. Então, embora a mudança para o design funcione, isso de alguma forma viola o design original. Pouco a pouco, conforme as mudanças continuam a chegar, essas violações se acumulam até que a malignidade se instale.

No entanto, não podemos culpar o desvio dos requisitos por degradar 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 constante chuva de mudanças nos requisitos, são nossos projetos que estão em falta. Precisamos, de alguma forma, encontrar uma maneira de tornar nossos projetos resilientes a essas mudanças e protegê-las de apodrecer.

Gestão de Dependências

Que tipo de alterações fazem com que os projetos apodreçam? Alterações que introduzem dependências novas e não planejadas. Cada um dos quatro sintomas mencionados acima é

direta ou indiretamente causado por dependências impróprias entre os módulos do software. É a arquitetura de dependências que está degradando e, com ela, a capacidade do software de ser mantido.

Para evitar a degradação da arquitetura de dependências, deve-se gerenciar as dependências entre os módulos em um aplicativo. Esse gerenciamento consiste na criação de *firewalls* de dependência. Através desses *firewalls*, as dependências não se propagam.

O *Design Orientado a Objetos* é repleto de princípios e técnicas para construir esses *firewalls* e 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 dependências de um aplicativo.

2. Princípios do Design de Classes Orientadas a Objetos

The Open Closed Principle (OCP)

O Princípio do Aberto Fechado [4]

> *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 Meyer[5]. Significa simplesmente isto: Devemos escrever nossos módulos para que eles possam ser estendidos, sem exigir que eles sejam modificados. Em outras palavras, queremos poder alterar o que os módulos fazem, sem alterar o código-fonte dos módulos.

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

Polimorfismo Dinâmico

Considere a Listagem 1. A função `LogOn` deve ser alterada sempre que um novo tipo de modem for adicionado ao software. Pior, como cada tipo diferente de modem depende da enumeração `Modem::Type`, cada modem deve ser recompilado toda vez que um novo tipo de modem for adicionado.

Listagem 1
Logon, deve ser modificado para ser estendido.

```
struct Modem
{
    enum Type {hayes, courrier, ernie} type;
};
```

4 OCP97
5 OOSC98

```
struct Hayes
{
    Modem::Type type;
    // elementos relacionados a Hayes
};

struct Courrier
{
    Modem::Type type;
    // elementos relacionados a Courrier
};

struct Ernie
{
    Modem::Type type;
    // elementos relacionados a Ernie
};

void LogOn(Modem& m,
           string& pno, string& user, 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)
        DialErnie((Ernie&)m, pno)
        // ... você entendeu a ideia
}
```

É claro que essa não é a pior característica desse tipo de design. Programas que são projetados dessa maneira tendem a estar cheios de instruções `if/else` ou `switch` semelhantes. Toda vez que qualquer coisa precisar ser feita para o modem, uma cadeia de declarações `if/else` precisará selecionar as funções adequadas para usar. Quando novos modems são adicionados ou a política do modem for alterada, o código deverá ser varrido com a finalidade de encontrar todas essas instruções de seleção, e cada uma deverá ser modificada apropriadamente.

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

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

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

Como um exemplo do *OCP*, considere a Figura 1. 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. Veja a Listagem 2.

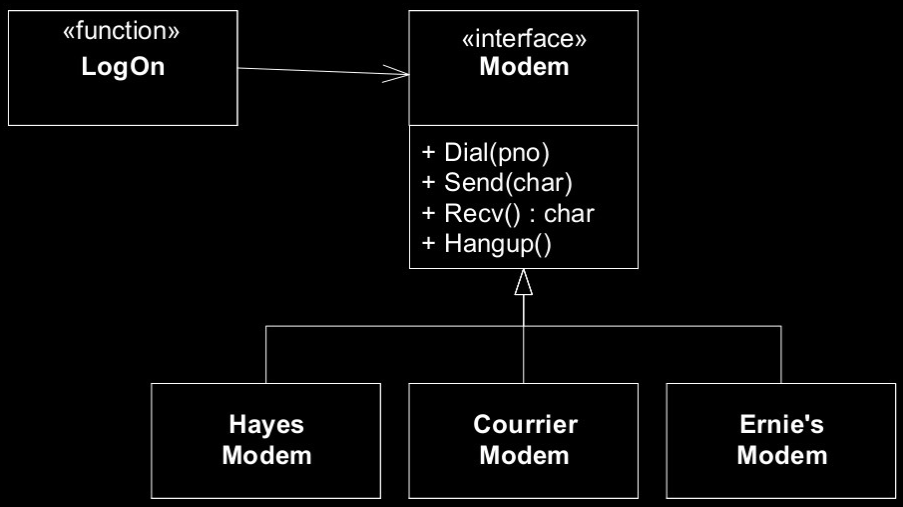

Figura 1- Esquema OCP

Listagem 2

`LogOn` foi fechado para modificação.

```
class Modem
{
    public:
    virtual void Dial(const string& pno) = 0;
    virtual void Send(char) = 0;
```

```
    virtual char Recv() = 0;
    virtual void Hangup() = 0;
};

void LogOn(Modem& m,
           string& pno, string& user, string& pw)
{
    m.Dial(pno);
    // você entendeu a ideia.
}
```

Polimorfismo Estático

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

Listagem 3
`LogOn` é fechado para modificação através de polimorfismo estático.

```
template <typename MODEM>
void LogOn(MODEM& m,
string& pno, string& user, string& pw)
{
    m.Dial(pno);
    // você entendeu a ideia.
}
```

Objetivos da arquitetura do OCP

Usando essas técnicas para se adequar ao *OCP*, podemos criar módulos que sejam extensíveis sem serem alterados. Isso significa que, com um pouco de previsão, podemos adicionar novos recursos ao código existente, sem alterá-lo e apenas adicionando novo código. Este é um ideal que pode ser difícil de alcançar, mas você verá como isso foi alcançado, várias vezes, nos estudos de caso mais adiante neste livro.

Mesmo se o *OCP* não puder ser totalmente alcançado, ainda que seja uma conformidade parcial do *OCP*, isso pode melhorar consideravelmente a estrutura de um aplicativo. É sempre melhor se as alterações não se propagarem para o código existente que já funciona. Se você não precisar alterar o código em operação, provavelmente não o quebrará.

The Liskov Substitution Principle (LSP)

O Princípio da Substituição de Liskov [6]

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

Este princípio foi cunhado por Barbara Liskov[7] em seu trabalho sobre abstração de dados e teoria de tipos. Também deriva do conceito de *Design by Contract (DBC)*, de Bertrand Meyer[8].

O conceito, como dito acima, é mostrado na Figura 2. Classes derivadas devem ser substituíveis por suas classes base. Ou seja, um utilizador de uma classe base deve continuar a funcionar corretamente se uma derivada dessa classe base for passada para ele.

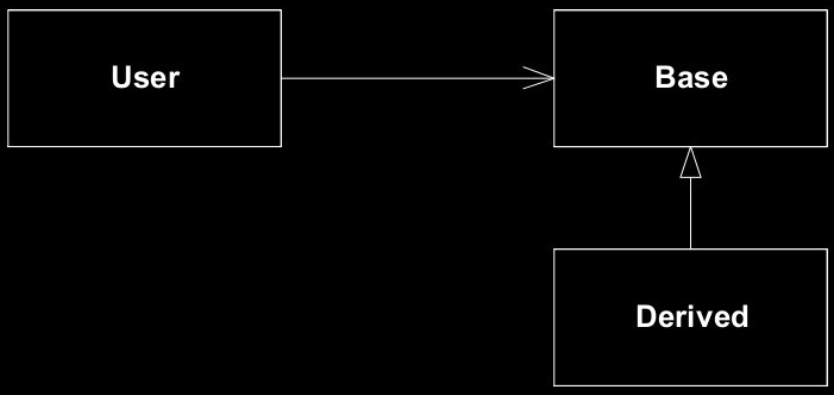

Figura 2: Esquema LSP

Em outras palavras, se uma função `User` tiver um argumento do tipo `Base`, como mostra a Listagem 4, deve ser correto passar uma instância de `Derived` para essa função.

Listagem 4

Exemplo de User, Based, Derived.

```
void User(Base& b);
Derived d;
User(d);
```

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

6 LSP97
7 Liksov88
8 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. Este é um relacionamento que no causa a tentação de modelar círculos e elipses usando herança, como mostrado na Figura 3.

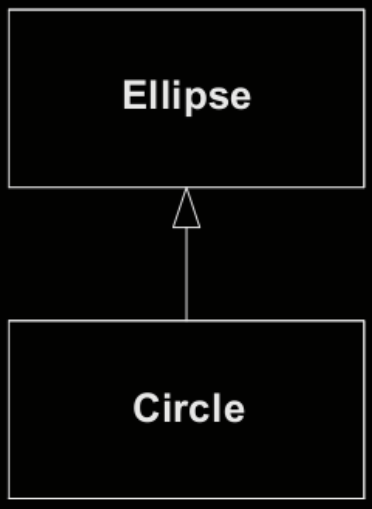

Figura 3: Dilema do Círculo/Elipse

Enquanto isso satisfaz nosso modelo conceitual, existem certas dificuldades. Um olhar mais atento à declaração de `Ellipse` na Figura 4 começa a expô-los. Observe que o `Ellipse` possui 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.

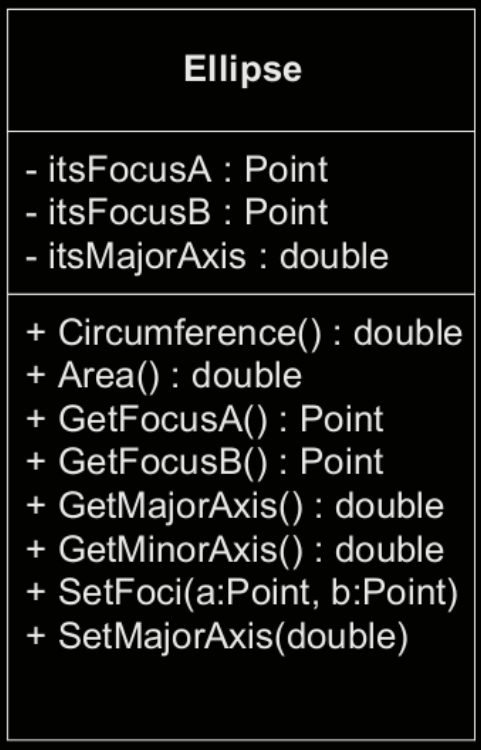

Figura 4: Declaração da Elipse

Isso é lamentável, já que o `Circle` precisa, de fato, apenas de dois elementos de dados, um ponto central e um raio.

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

Listagem 5
Keeping the Circle Foci coincident.

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

Clientes arruínam tudo

Certamente, o modelo que criamos é auto-consistente. Um exemplo de `Circle` obedecerá a todas as regras de um círculo. Não há nada que você possa fazer para violar essas regras. Então também para `Ellipse`. As duas classes formam um modelo bem consistente, mesmo se o `Circle` tiver muitos elementos de dados.

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

```
void f(Ellipse& e)
{
    Point a(-1,0);
    Point 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 para trabalhar com uma elipse. Dessa forma, espera poder definir os focos e o eixo principal e, em seguida, verificar se eles foram definidos corretamente. Se passarmos uma instância de `Ellipse` para essa função, tudo correrá bem. No entanto, se passarmos uma instância de `Circle` para a função, ela irá falhar bastante.

Se fôssemos explicitar o contrato de `Ellipse`, veríamos uma pós-condição no `SetFoci` que garantiria que os valores de entrada foram copiados para as variáveis de membro, e que a variável de eixo maior foi deixada inalterada. Claramente `Circle` viola esta garantia porque ignora a segunda variável de entrada de `SetFoci`.

Design por Contrato

Reafirmando o *LSP*, podemos dizer que, para ser substituível, o contrato da classe base deve ser honrado pela classe derivada. Como o `Circle` não honra o contrato implícito da `Ellipse`, ele 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 a cada invocação. Aqueles de nós que não estão usando *Eiffel*, têm que se contentar com simples *asserções* e *comentários*.

Para declarar o contrato de um método, definimos 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 serão indefinidos e o método não deverá ser chamado. Também definimos o que o método garante ser verdadeiro depois de concluído. Isso é chamado de pós-condição. Um método que falhar sua pós-condição não deve retornar.

Reafirmando o LSP novamente, 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, métodos derivados não devem *esperar mais e não fornecer menos*.

Repercussões da Violação do LSP

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

Se o design for usado maciçamente, o custo de reparar a violação do *LSP* poderá ser muito grande para suportar. Pode não ser econômico voltar e alterar o design e, em seguida, recriar e retestar todos os utilizadores existentes. Portanto, a solução provavelmente será colocar uma instrução `if/else` no utilizador que descobriu a violação. Essa instrução `if/else` verifica se a elipse é, na verdade, uma elipse e não um círculo. Veja a Listagem 6.

Listagem 6

Correção feia para violação de LSP

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

```
        assert(e.GetMajorAxis() == 3);
    }
    else
        throw NotAnEllipse(e);
}
```

Um exame cuidadoso da Listagem 6 mostrará que é uma violação do *OCP*. Agora, sempre que alguma nova derivada de `Ellipse` for criada, esta função terá que ser verificada para ver se deve ser permitido operar nela. Assim, as violações do *LSP* são violações latentes do *OCP*.

The Dependency Inversion Principle (DIP)

O Princípio da Inversão de Dependência [9]

> *Dependa de abstrações. Não dependa de concreções.*

Se o *OCP* indica o objetivo da arquitetura OO, o *DIP* indica o mecanismo principal. Inversão de dependência é a estratégia de depender de interfaces ou funções e classes abstratas, ao invés de funções e classes concretas. Este princípio é a força motriz por detrás do design de componentes, COM, CORBA, EJB, etc.

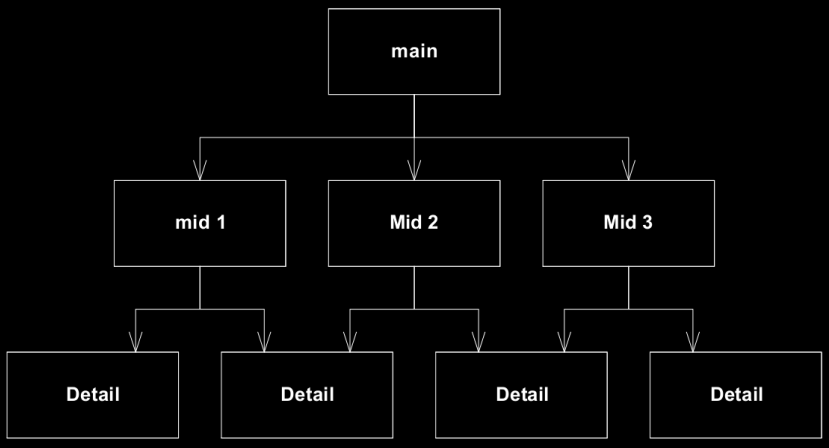

Figura 5: Estrutura de dependências de uma arquitetura procedural

Projetos procedurais exibem um tipo particular de estrutura de dependência. Como mostra a Figura 5, 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 mais baixo, etc.

9 DIP97

Um pequeno pensamento deve expor essa estrutura de dependência como intrinsecamente fraca. Os módulos de alto nível lidam com as políticas de alto nível do aplicativo. Essas políticas geralmente pouco se importam com os detalhes que as implementam. Por que, então, esses módulos de alto nível dependem 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. No entanto, os módulos que contêm implementação detalhada não são mais dependentes, e sim dependem de abstrações. Assim, a dependência deles foi invertida. Veja a Figura 6.

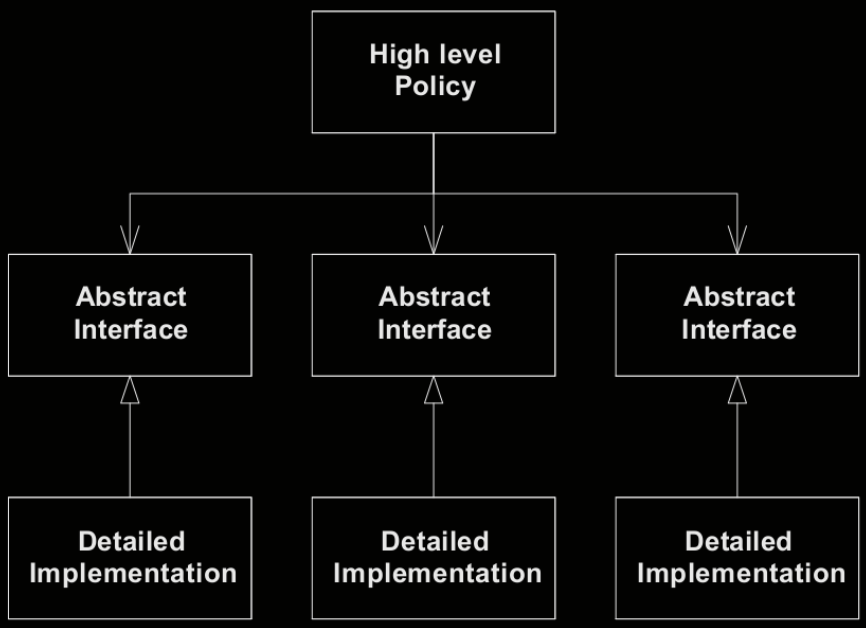

Figura 6: Estrutura de dependências de uma arquitetura orientada a objetos

Dependendo de abstrações

A implicação desse princípio é bem simples. No design, toda dependência deve ter como alvo uma interface ou uma classe abstrata. Nenhuma dependência deve ter como alvo uma classe concreta.

Claramente tal restrição é bem severa, e existem circunstâncias atenuantes que iremos explorá-la de forma provisória. Mas, tanto quanto for viável, o princípio deve ser seguido. A razão é simples, coisas concretas mudam muito, coisas abstratas mudam com muito menos frequência. Além disso, as abstrações são "pontos de articulação", eles representam os locais onde o projeto pode dobrar ou ser estendido, sem que sejam modificados (*OCP*).

Substratos como *COM* reforçam esse 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 atenuantes

Uma motivação por trás do *DIP* é impedir que você dependa de módulos voláteis. O *DIP* faz a suposição de que qualquer coisa concreta é volátil. Embora isso seja frequente, especialmente no desenvolvimento inicial, há exceções. Por exemplo, a biblioteca padrão *string.h*, da linguagem *C*, é muito concreta, mas não é volátil. Dependendo disso em um ambiente de sequência de caracteres ANSI não é prejudicial. Da mesma forma, se você já experimentou e certificou que os módulos são concretos de fato, mas não voláteis, depender deles não é tão ruim. Uma vez que eles não são susceptíveis de mudança, eles não são susceptíveis de injetar volatilidade em seu design.

Tome cuidado no entanto. Uma dependência de *string.h* poderia 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 Objeto

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 em toda a arquitetura do design. Assim, pode parecer que não há escapatória e que toda a arquitetura estará repleta de dependências de classes concretas. No entanto, há uma solução elegante para esse problema chamada ABSTRACT FACTORY[10] - um padrão de projeto que analisaremos com mais detalhes no próximo capítulo.

The Interface Segregation Principle (ISP)

O Princípio da Segregação de Interface[11]

> *Muitas interfaces específicas do utilizador são melhores que uma interface de propósito geral.*

O *ISP* é outra das tecnologias de ativação que suportam substratos de componentes, como o *COM*. Sem isso, componentes e classes seriam muito menos úteis e portáveis.

10 GOF96
11 ISP97

A essência do princípio é bem simples. Se você tiver uma classe que tenha vários utilizadores, em vez de carregar a classe com todos os métodos que os utilizadores precisam, crie interfaces específicas para cada utilizador e multiplique-os para herdá-los na classe.

A Figura 7 mostra uma classe com muitos utilizadores e uma grande interface para atendê-los. Observe que sempre que uma alteração for feita em um dos métodos que o `ClientA` chama, `ClientB` e `ClientC` podem ser afetados. Pode ser necessário recompilar e reimplementá-los. Isto é um infortúnio.

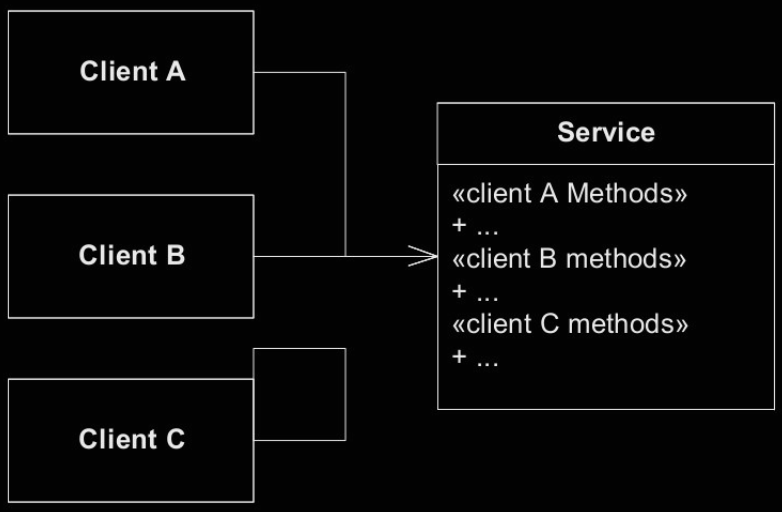

Figura 7: Serviço inchado com interfaces integradas

Uma técnica melhor é mostrada na Figura 8. Os métodos necessários para cada utilizador são colocados em interfaces especiais específicas desse utilizador. Essas interfaces são herdadas de forma múltipla pela classe `Service` e implementadas lá.

Se a interface do `ClientA` precisar ser alterada, o `ClientB` e o `ClientC` permanecerão inalterados. Eles não terão que ser recompilados ou reimplantados.

O que significa utilizador específico?

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 fosse esse o caso, o serviço dependeria de cada utilizador de maneira bizarra e nociva. Em vez disso, os utilizadores devem ser categorizados por seu tipo e as interfaces para cada tipo de utilizador devem ser criadas.

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

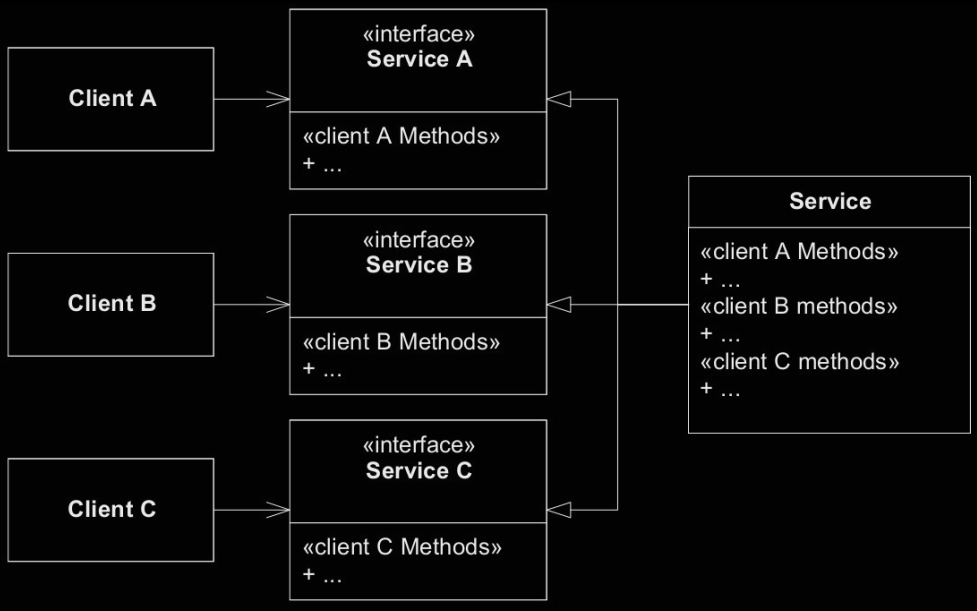

Figura 8: Interfaces segregadas

Alterando Interfaces

Quando a manutenção em aplicações orientadas a objetos é efetuada, as interfaces para classes e componentes existentes frequentemente 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 atenuado pela adição de novas interfaces a objetos existentes, em vez de alterar a interface existente. Os utilizadores da interface antiga que desejam acessar métodos da nova interface podem consultar o objeto para essa interface, conforme mostrado no código a seguir.

```
void Client(Service* s)
{
    if (NewService* ns = dynamic_cast<NewService*>(s))
    {
        // use a nova interface de serviço
    }
}
```

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

Principles of Package Architecture

As classes são um meio necessário, mas insuficiente, de organizar um design. Uma maior granularidade dos pacotes é necessária para ajudar na organização. Mas como escolhemos quais classes pertencem a quais pacotes? Abaixo estão três princípios conhecidos como *Princípios de Coesão do Pacote*, que tentam ajudar o arquiteto de software.

The Release Reuse Equivalency Principle (REP)

O Princípio de Equivalência de Reutilização de Release[12]

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

Um elemento reutilizável, seja ele um componente, uma classe ou um grupo de classes, não pode ser reutilizado a menos que seja gerenciado por um sistema de *Release* de algum tipo. Os usuários não estarão dispostos a usar o elemento se forem forçados a atualizar cada vez que o autor o alterar. Portanto. mesmo que o autor tenha lançado uma nova versão de seu elemento reutilizável, ele deve estar disposto a da suporte e manter versões mais antigas, enquanto seus clientes lidam com o lento processo de se preparar para a atualização. Assim, os clientes recusarão a reutilização de um elemento, a menos que o autor prometa acompanhar os números de versão e manter versões antigas por algum tempo.

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

The Common Closure Principle (CCP)

O Princípio do Fechamento Comum [13]

> *Classes que mudam juntas, permanecem 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 qualquer *Release*, maior será o trabalho para reconstruir, testar e implantar o *Release*. Portanto, gostaríamos de minimizar o número de pacotes que são alterados em qualquer ciclo de lançamento do produto.

12 Ganularity97
13 Ganularity97

Para conseguir isso, devemos agrupar as classes que achamos que mudarão juntas. Isso requer uma certa quantidade de previsibilidade, 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 no *Release* pacote será minimizado.

The Common Reuse Principle (CRP)

O Princípio Comum de Reutilização [14]

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

Depender de um pacote é depender de tudo o que tem dentro dele. Quando um pacote é alterado e seu número de *Release* é coincidente, todos os utilizadores desse pacote devem verificar se funcionam com o novo pacote - mesmo que nada que eles tenham usado no pacote tenha sido alterado.

Nós experimentamos isso com frequência quando nosso fornecedor de SO libera um novo sistema operacional. Temos que atualizar mais cedo ou mais tarde, porque o fornecedor não suportará a versão antiga para sempre. Então, mesmo que nada de interesse para nós tenha mudado na nova versão, devemos passar pelo esforço de atualização e revalidação.

O mesmo pode acontecer com os pacotes se as classes que não são usadas juntas forem agrupadas. Alterações em uma classe que não me interessa ainda forçarão uma nova versão do pacote e ainda me farão passar pelo esforço de atualizar e revalidar.

Tensão entre os Princípios de Coesão do Pacote

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 *PCC* facilita a vida dos mantenedores. O *PCC* esforça-se para tornar os pacotes o mais grandes possíveis (afinal, se todas as classes residirem em apenas um pacote, apenas um pacote irá mudar). O *CRP*, no entanto, tenta tornar os pacotes muito pequenos.

Felizmente, os pacotes não são feitos em pedra. De fato, é da natureza dos pacotes mudar e se instabilizar durante o curso do desenvolvimento. No início de um projeto, os arquitetos podem configurar a estrutura de pacotes de modo que o *PCC* domine e o desenvolvimento juntamente com a manutenção sejam auxiliados. Mais tarde, conforme a arquitetura se estabilizar, os arquitetos poderão refatorar a estrutura dos pacotes para maximizar o *REP* e o *CRP* para os reutilizadores externos.

14 Ganularity97

Os Princípios do Acoplamento de Pacotes

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

The Acyclic Dependencies Principle (ADP)

O princípio das dependências acíclicas [15]

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

Como os pacotes são o grânulo de *Release*, eles também tendem a concentrar a mão de obra. Os engenheiros normalmente trabalham em um único pacote, em vez de trabalhar em dezenas. Essa inclinação é ampliada pelos princípios de coesão do pacote, pois tendem a agrupar as classes que estão relacionadas. Assim, os engenheiros descobrirão que suas mudanças são direcionadas para apenas alguns pacotes. Depois que essas alterações forem feitas, elas poderão liberar esses pacotes para o restante do projeto.

Antes que eles possam fazer esse *Release*, eles devem testar se o pacote funciona. Para fazer isso, eles devem compilá-lo e compilá-lo com todos os pacotes dos quais ele depende. Espero que este número seja pequeno.

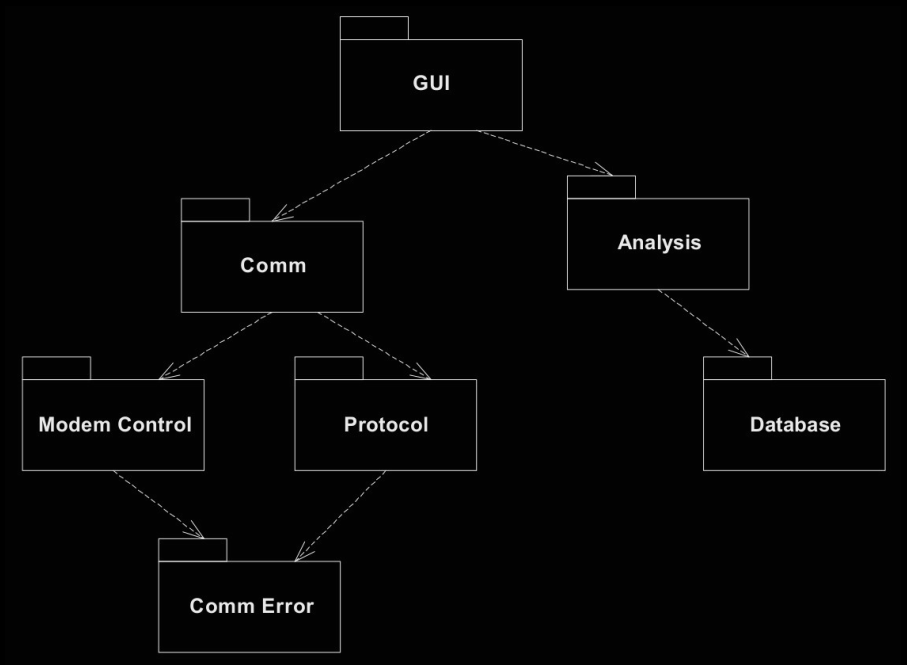

Figura 9: Rede acíclica de pacote

15 Ganularity97

Considere a Figura 9. Os leitores astutos reconhecerão que há várias falhas na arquitetura. O *DIP* parece ter sido abandonado e, junto com ele, o *OCP*. A *GUI* depende diretamente do pacote de comunicações e, aparentemente, é responsável por transportar dados para o pacote de análise. :(

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

Um ciclo se arrasta

Mas agora vamos dizer que sou o engenheiro trabalhando no pacote `CommError`. Eu 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 colocar minha mensagem na tela. Isso significa que tornei `CommError` dependente da `GUI`. Veja a Figura 10.

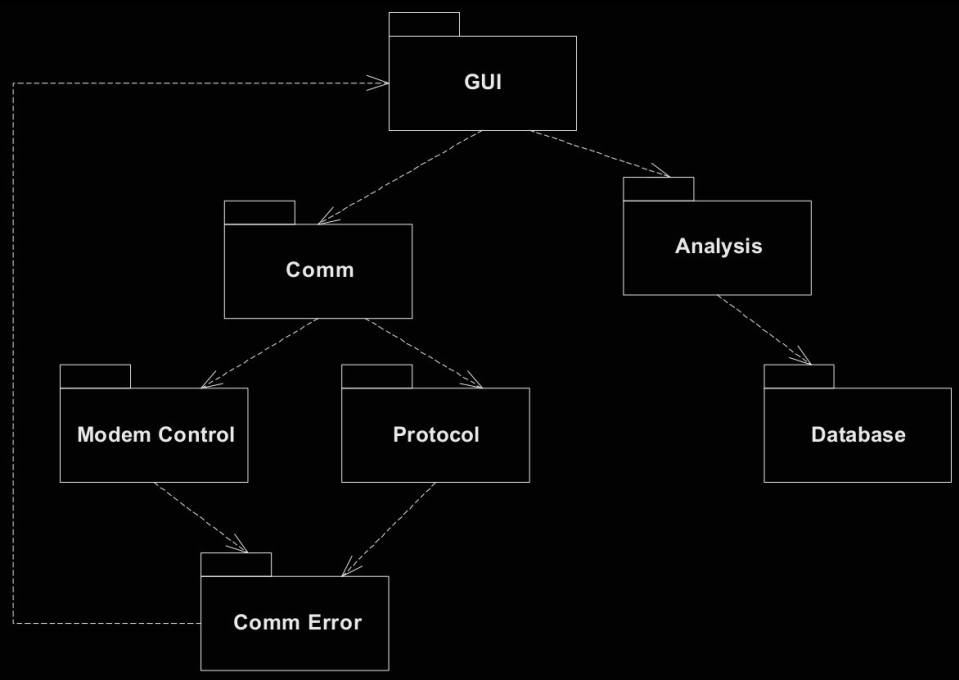

Figura 10: Um ciclo foi adicionado

Agora o que acontece quando os caras que estão trabalhando no protocolo querem liberar o pacote. Eles têm que construir sua suíte de testes com `CommError`, `GUI`, `Comm`, `ModemControl`, `Analysis` e `Database`! Isso é claramente desastroso. A carga de trabalho

dos engenheiros foi aumentada por uma quantidade repugnante, devido a uma pequena dependência que saiu do controle.

Isso significa que alguém precisa estar observando a estrutura de dependências do pacote com regularidade e interrompendo os 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 o ciclo

Os ciclos podem ser quebrados de duas maneiras. O primeiro envolve a criação de um novo pacote e o segundo faz uso do *DIP* e do *ISP*. A Figura 11 mostra como quebrar o ciclo adicionando um novo pacote. As classes que o `CommError` precisava são extraídas da `GUI` e colocadas em um novo pacote denominado `MessageManager`. Tanto a `GUI` quanto o `CommError` dependem do novo pacote.

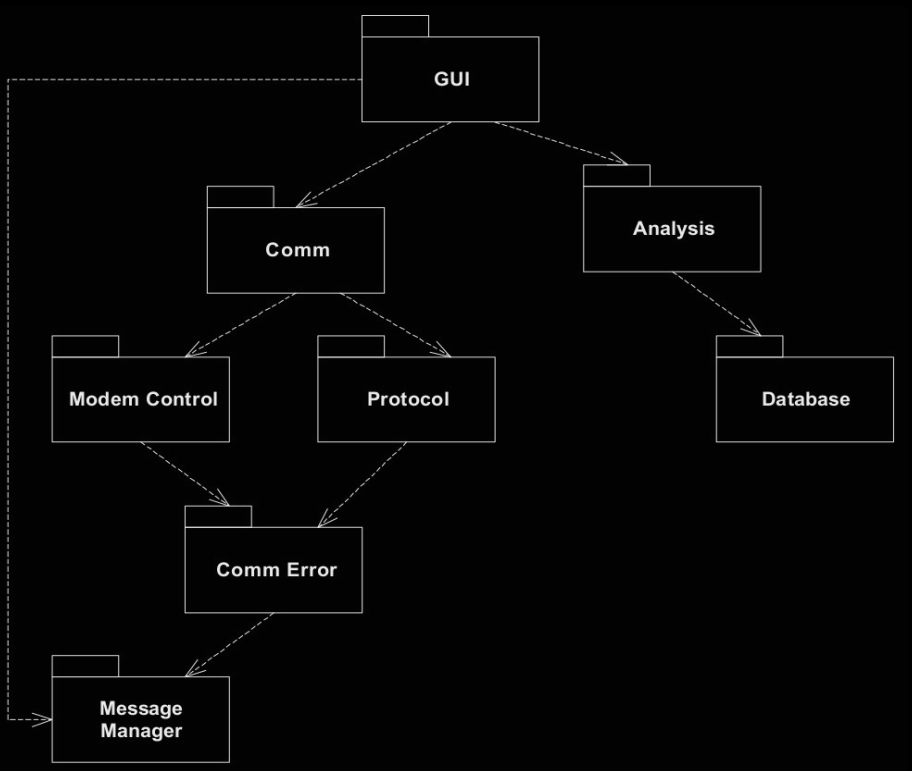

Figura 11- Quebrando o ciclo de pacotes

Este é um exemplo de como a estrutura do pacote tende a se instabilizar e mudar durante o desenvolvimento. Um novo pacote surgiu e as classes passaram do pacote antigo para novos pacotes, para ajudar nos ciclos de quebra. A Figura 12 mostra uma imagem antes e depois da outra técnica para quebrar ciclos. Aqui vemos dois pacotes que estão ligados por um ciclo. A

classe A depende da classe X, e a classe Y depende da classe B. Nós 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 necessidades. Y usa essa interface e B a implementa.

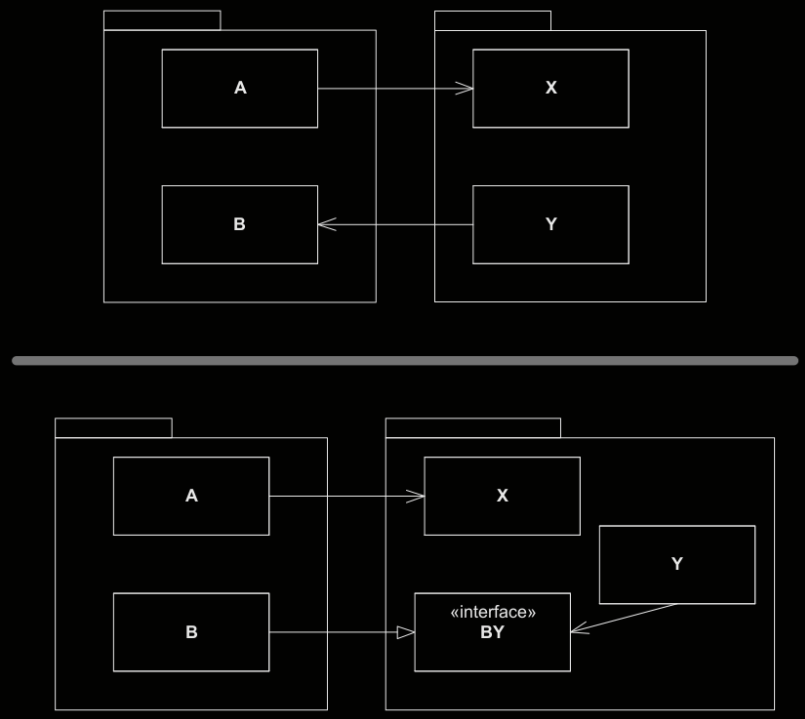

Figura 12- Quebrando o ciclo de pacotes com contrato

Observe o posicionamento de BY. 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 das idades dos pacotes. As interfaces são frequentemente incluídas no pacote que as utiliza, e não no pacote que as implementa.

The Stable Dependencies Principle (SDP)

O Princípio das Dependências Estáveis [16]

Dependa na direção da estabilidade.

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

16 Stability97

Estabilidade

O que se entende por estabilidade? Coloque uma moeda no seu ombro. É estável essa posição? Provavelmente você diria que não. No entanto, a menos que se desequilibre, permanecerá nessa posição por muito, muito tempo. Assim, a estabilidade não tem nada a ver com frequência de mudança. A moeda não está mudando, mas é difícil pensar nela como estável.

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

Como isso se relaciona com o software? Existem muitos fatores que dificultam a alteração de um pacote de software. Seu tamanho, complexidade, clareza, etc. Vamos ignorar todos esses fatores e nos concentrar em algo diferente. Uma maneira certa de tornar um pacote de software difícil de mudar 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 mudanças com todos os pacotes dependentes.

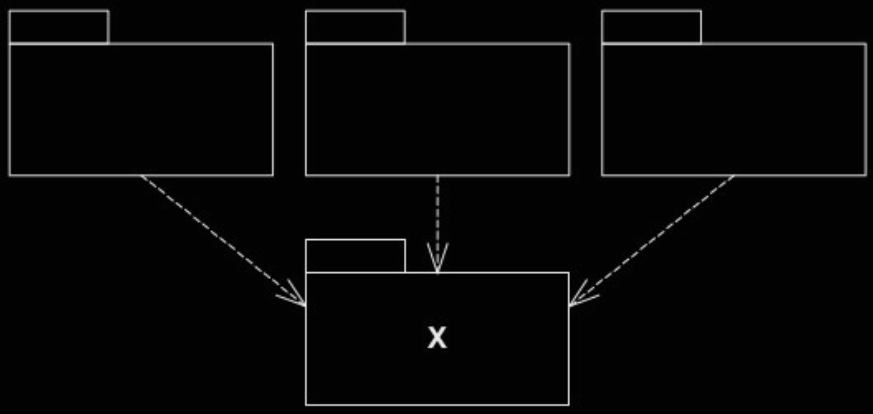

Figura 13: X é um pacote estável

A Figura 13 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, então não tem influência externa para fazê-lo mudar. Nós dizemos que é independente.

A Figura 14, por outro lado, mostra um pacote muito instável. Y não tem outros pacotes dependendo dele; Dizemos que não tem responsabilidades. Y também tem três pacotes dos quais depende, portanto, as alterações podem vir de três fontes externas. Nós dizemos que Y é dependente.

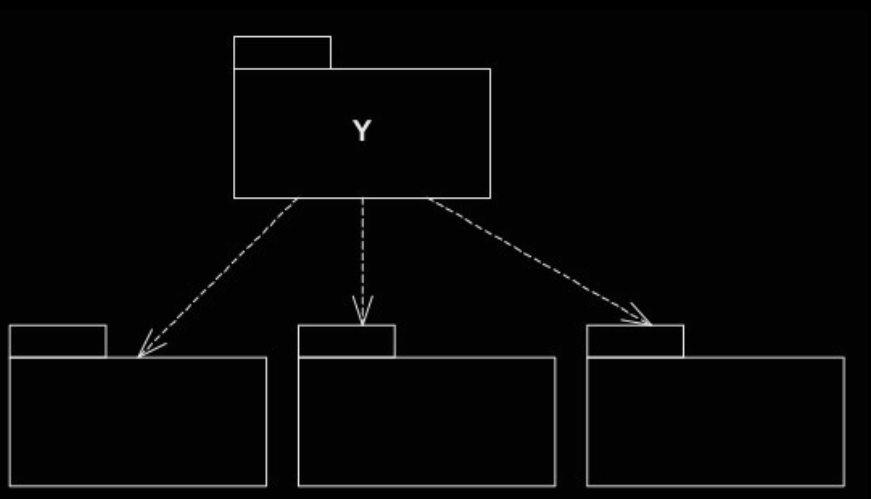

Figura 14: Y é instável

Métricas de estabilidade

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

- Aa: Acoplamento Aferente, ou seja, o número de classes fora do pacote que dependem de classes dentro do pacote. (isto é, dependências de entrada);
- Ae: Acoplamento Eferente, ou seja, o número de classes fora do pacote que as classes dentro do pacote dependem (isto é, dependências de saída);
- I: Instabilidade, uma métrica que tem o intervalo de “zero” a “um”: [0,1].

Fórmula: I = Ae / Aa + Ae

Se não houver dependências de saída, o resultado será “zero” e o pacote será estável. Se não houver dependências de entrada, será “um” e o pacote será instável.

Agora podemos reformular o *SDP* da seguinte forma: "*Depender de pacotes cuja métrica é menor que a sua*".

Justificativa

Todos os softwares devem ser estáveis? Um dos atributos mais importantes do software bem projetado é a facilidade de mudança. O software flexível na presença de requisitos variáveis é bem pensado. No entanto, esse software é instável pela nossa definição. De fato, desejamos muito que partes de nosso software sejam instáveis. Queremos que certos módulos sejam fáceis de mudar, de modo que, quando os requisitos se desviarem, o design possa responder com facilidade.

A Figura 15 mostra como o *SDP* pode ser violado. `Flexible` é um pacote que pretendemos tornar fácil de alterar. Queremos que o `Flexible` seja instável. No entanto, alguns engenheiros, trabalhando no pacote chamado `Stable`, colocaram uma dependência no `Flexible`. Isso viola o *SDP*, pois a métrica *"I"* para `Stable` é muito menor do que a métrica *"I"* para `Flexible`. Como resultado, o `Flexible` não será mais fácil de alterar. Uma mudança para o `Flexible` nos forçará a lidar com o `Stable` e todos os seus dependentes.

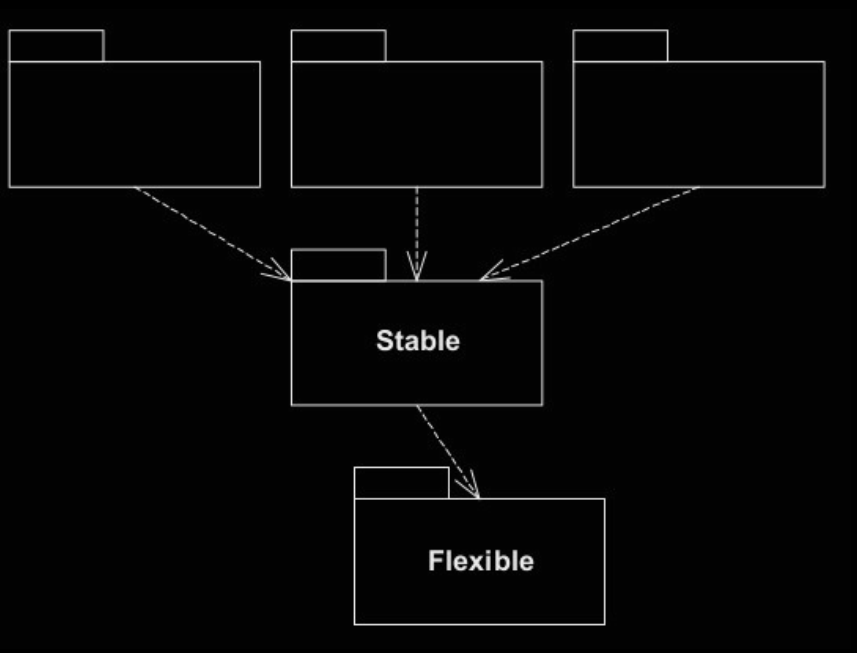

Figura 15: Violação do SDP

The Stable Abstractions Principle (SAP)

O Princípio das Abstrações Estáveis [17]

> *Pacotes estáveis devem ser pacotes abstratos.*

Podemos visualizar a estrutura de pacotes de nossa aplicação como um conjunto de pacotes interconectados com pacotes instáveis no topo 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 base são muito difíceis de mudar. E isso nos leva a um dilema: queremos pacotes em nosso projeto que sejam difíceis de mudar?

17 Stability97

Claramente, quanto mais pacotes difíceis de mudar, menos flexível será o nosso design geral. No entanto, há uma lacuna que podemos rastrear. Os pacotes altamente estáveis na parte inferior da rede de dependência podem ser muito difíceis de alterar, 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 nosso aplicativo 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 reafirmação do *DIP*. Afirma que os pacotes que são os mais dependentes (isto é, 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.

- 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;
- A: Nível de Abstração, uma métrica que tem o intervalo de zero a um: [0,1].

Fórmula: A = Na / Nc

A métrica A (Abstração) tem um intervalo de "zero" a "um", assim como a métrica I (Instabilidade). Um valor "zero" significa que o pacote não contém classes abstratas. Um valor "um" significa que o pacote só contém classes abstratas.

O gráfico I vs A.

O *SAP* agora pode ser atualizado em termos das métricas I e A: I deve aumentar conforme A diminuir. Isto é, pacotes concretos devem ser instáveis enquanto pacotes abstratos devem ser estáveis. Podemos traçar isso graficamente no gráfico A vs I. Veja a Figura 16.

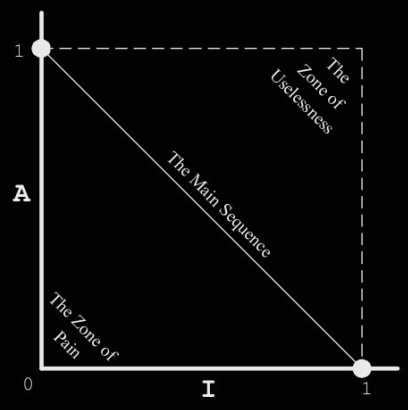

Figura 16: Gráfico A vs I

Parece claro que os pacotes devem aparecer em qualquer um dos dois pontos pretos na Figura 16. Aqueles no canto superior esquerdo são completamente abstratos e muito estáveis. Aqueles no canto inferior direito são completamente concretos

e muito instáveis. É assim que gostamos. No entanto, e o pacote X na Figura 17? Para onde deveria ir?

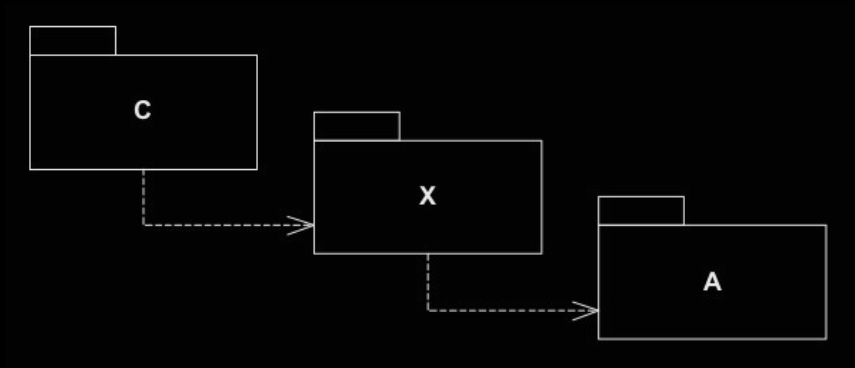

Figura 17: Onde colocamos X no gráfico A vs I?

Podemos determinar onde queremos o pacote X, olhando para onde não queremos que ele esteja. O canto superior direito do *Gráfico A vs I* representa pacotes altamente abstratos, dos quais ninguém depende. Esta é a zona de inutilidade. Certamente não queremos que X viva lá. Por outro lado, o ponto inferior esquerdo do *Gráfico A vs I* representa pacotes que são concretos e têm muitas dependências de entrada. Este ponto representa o pior caso para um pacote. Como os elementos são concretos, eles não podem ser estendidos da maneira como as entidades abstratas podem; e como eles têm muitas dependências de entrada, a mudança será muito dolorosa. Esta é a zona da dor, e certamente não queremos que nosso pacote fique lá.

Maximizar a distância entre essas duas zonas nos dá uma linha chamada de *Sequência Principal*. Queremos que nossos pacotes fiquem nessa linha, se possível. Uma posição nesta linha significa que o pacote é abstrato em proporção às suas dependências de entrada e é concreto em proporção às 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 até que ponto esse pacote é da *Sequência Principal*.

- **D: Distância.** Isso varia de [0,~0.707].
- **D': Distância Normalizada.** Essa métrica é muito mais conveniente que D, pois varia de [0,1].

Fórmula: $D' = A + I - 1 / \sqrt{2}$.

O valor “zero” indica que o pacote está diretamente na *Sequência Principal*. O valor “um” indica que o pacote está o mais longe 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 foram usados para ajudar a medir a estrutura de dependência de um aplicativo.

3. Padrões de Arquitetura Orientada a Objetos

Ao seguir os princípios descritos acima para criar arquiteturas orientadas a objetos, descobre-se que são repetidas as mesmas estruturas diversas vezes. Essas estruturas repetitivas de design e arquitetura são conhecidas como padrões de projeto[18].

A definição essencial de um padrão de projeto é uma boa solução, bem usada e conhecida para um problema comum. Padrões de projeto definitivamente não são novos. Em vez disso, são técnicas antigas que mostraram sua utilidade durante um período de muitos anos.

Alguns padrões de projeto comuns são descritos abaixo. Estes são os padrões que você encontrará durante a leitura dos estudos de caso mais adiante no livro.

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

Abstract Server (Servidor Abstrato)

Quando um cliente depende diretamente de um servidor, o *DIP* é violado. As alterações no servidor serão propagadas para o utilizador, e o utilizador não poderá usar facilmente servidores semelhantes. Isso pode ser corrigido inserindo-se uma interface abstrata entre o utilizador e o servidor, conforme mostrado na Figura 18.

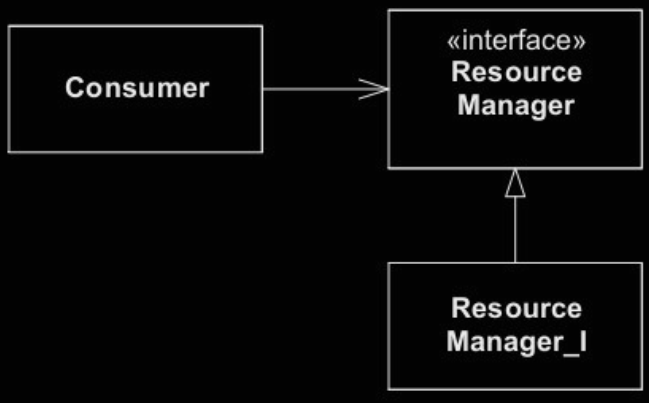

Figura 18: Abstract Server

18 GOF96

A interface abstrata se torna um "ponto de articulação" no qual o design pode se flexionar. Diferentes implementações do servidor podem ser vinculadas a um utilizador desavisado.

Adapter (Adaptador)

Quando inserir uma interface abstrata é inviável, porque o servidor é um software de terceiros, ou é tão fortemente dependente de que não pode ser facilmente alterado, um ADAPTER pode ser usado para vincular a interface abstrata ao servidor. Veja a Figura 19.

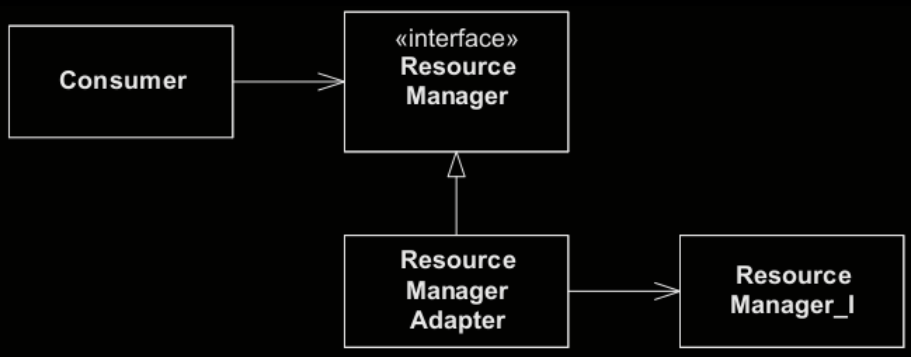

Figura 19: Adapter

O adaptador é um objeto que implementa a interface abstrata para delegar ao servidor. Todos os métodos do ADAPTER simplesmente traduzem e depois delegam.

Observer (Observador)

Muitas vezes ocorre que um elemento de um design precisa tomar alguma forma de ação quando outro elemento no design 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 tratar essa situação com um OBSERVER, veja a Figura 20. O `Sensor` deriva de uma classe chamada `Subject`, e o `Meter`

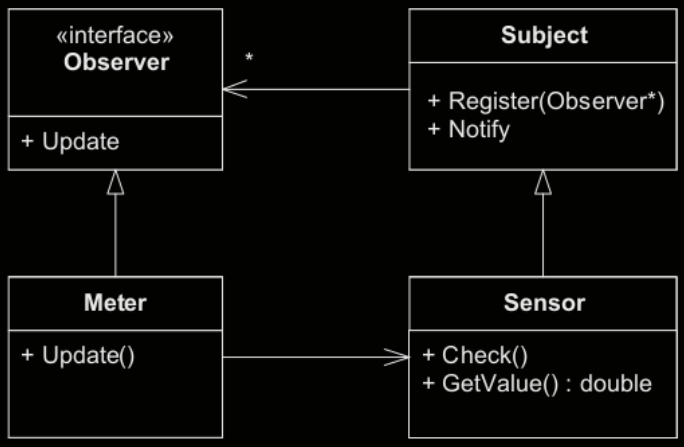

Figura 20: Estrutura do observer

deriva de uma interface chamada `Observer`. `Subject` contém uma lista de *Observadores*. Essa lista é carregada pelo método `Register` do `Subject`. Para ser informado dos eventos, o nosso `Meter` deve se registrar com `Subject`, a classe base do `Sensor`.

A Figura 21 descreve a dinâmica da colaboração. Algumas entidades passam o controle para o `Sensor`, que determina que sua leitura foi alterada. O `Sensor` chama o `Notify` do `Subject`. O `Subject`, em seguida, percorre todos os observadores que foram registrados, chamando o `Update` em cada um. A mensagem de `Update` é capturada pelo `Meter` que a utiliza para ler o novo valor do `Sensor` e exibi-lo.

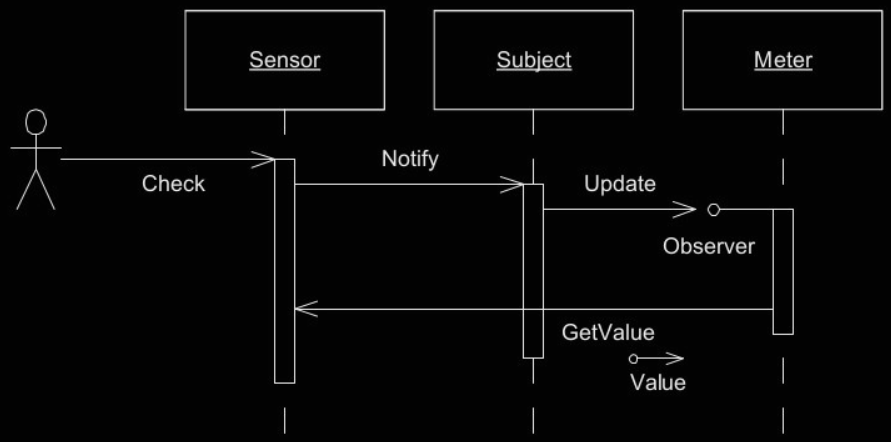

Figura 21

Bridge (Ponte)

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

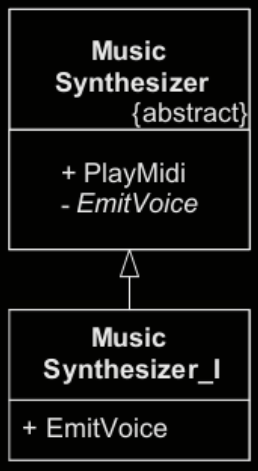

Figura 22: Hierarquia mal acoplada

Por exemplo, considere uma classe de sintetizador de música. A classe base converte a entrada MIDI em um conjunto de chamadas `EmitVoice` primitivas que são implementadas por uma classe derivada.

Observe que a função `EmitVoice` da classe derivada seria útil, por si só. Infelizmente está intrinsecamente ligada à classe `MusicSynthesizer` e à função `PlayMidi`. Não há como chegar ao método `EmitVoice` sem arrastar a classe base. Além disso, não

há como criar diferentes implementações da função `PlayMidi` que usam a mesma função `EmitVoice`. Em suma, a hierarquia é apenas para o acoplamento.

O padrão BRIDGE resolve esse problema criando uma forte separação entre a interface e a implementação. A Figura 23 mostra como isso funciona. A classe `MusicSynthesizer` contém uma função abstrata `PlayMidi` que é implementada pelo `MusicSynthesizer_I`. Ele chama a função `EmitVoice` que é implementada no `MusicSynthesizer` para delegar à interface `VoiceEmitter`. Essa interface é implementada pelo `VoiceEmitter_I` que emite os sons necessários.

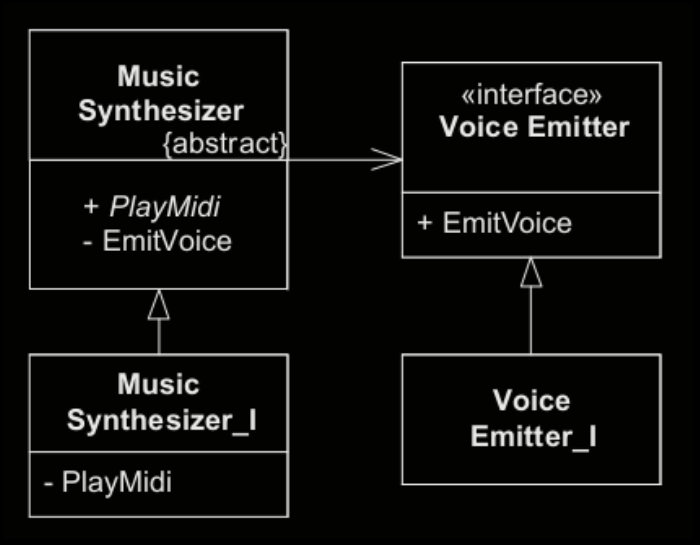

Figura 23: Hierarquia desacoplada com Bridge

Agora é possível implementar o `EmitVoice` e o `PlayMidi` separadamente um do outro. As duas funções foram desacopladas. 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 a mesma função `EmitVoice`.

Abstract Factory (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. ABSTRACT FACTORY é um padrão que permite que a dependência da classe concreta exista em um, e apenas um, lugar.

A Figura 24 mostra como isso é feito para o exemplo do `Modem`. Todos os usuários que desejam criar modems usam uma interface chamada `ModemFactory`. Um ponteiro para essa interface é mantido em uma variável global chamada `GtheFactory`. Os usuários chamam a

função `Make` passando em uma string que define exclusivamente a subclasse particular do `Modem` que eles desejam. A função `Make` retorna um ponteiro para uma interface de `Modem`.

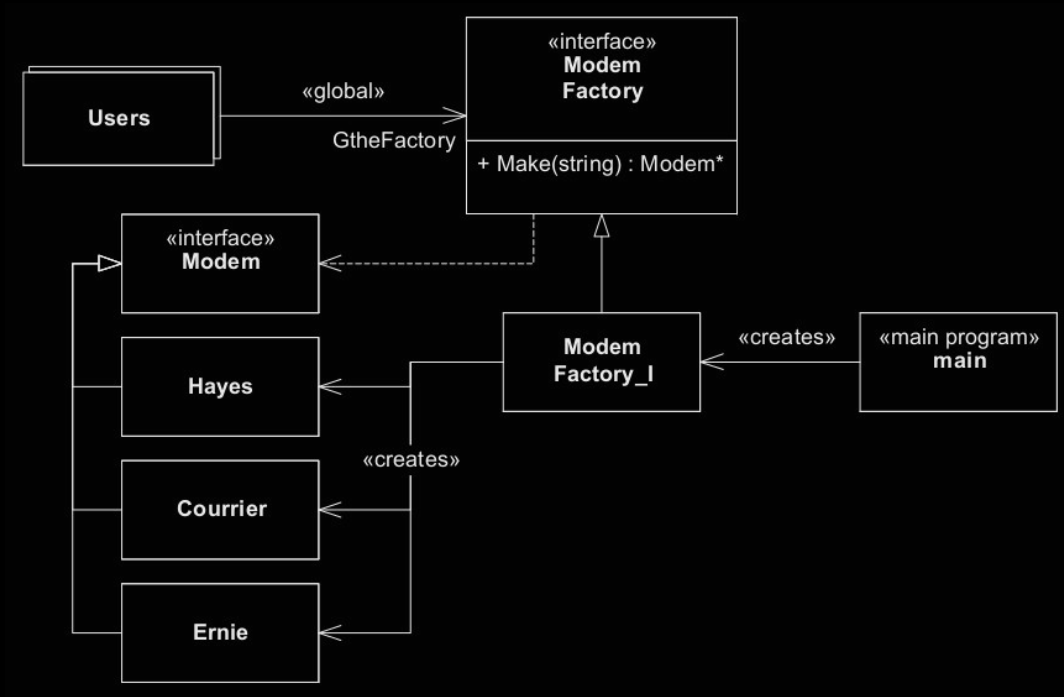

Figura 24: Abstract Factory

A interface `ModemFactory` foi implementada pelo `ModemFactory_I`. Esta classe é criada por `main`, e um ponteiro para ela é carregado no global da `GtheFactory`. Portanto, nenhum módulo no sistema conhece as classes de `Modem` concretas, exceto o `ModemFactory_I`, e nenhum módulo conhece o `ModemFactory_I`, exceto o `main`.

4. Conclusão

Este conteúdo introduziu o conceito de arquitetura orientada a objetos e definiu-a como a estrutura de classes e pacotes que mantém a aplicação de software flexível, robusta, reutilizável e desenvolvível. Os princípios e padrões apresentados aqui suportam 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 conteúdo, na verdade, ao encurtar o tópico tanto, corremos o risco de fazer um desserviço ao leitor. Já foi dito que um pouco de conhecimento é uma coisa perigosa, e este conteúdo forneceu um pequeno conhecimento. Nós pedimos que você procure os livros e artigos nas citações deste capítulo para aprender mais.

5. Bibliografia

[Shaw96]: Patterns of Software Architecture (???), Garlan and Shaw, ...

[GOF96]: Design Patterns...

[OOSC98]: OOSC…

[OCP97]: The Open Closed Principle, Robert C. Martin...

[LSP97]: The Liskov Substitution Principle, Robert C. Martin

[DIP97]: The Dependency Inversion Principle, Robert C. Martin

[ISP97]: The Interface Segregation Principle, Robert C. Martin

[Granularity97]: Granularity, Robert C. Martin

[Stability97]: Stability, Robert C. Martin

[Liksov88]: Data Abstraction and Hierarchy…

[Martin99]: Designing Object Oriented Applications using UML, 2d. ed., Robert C. Martin, Prentice Hall, 1999.