O'REILLY®

Software C++ Projeto

Princípios e padrões de design para

Software de alta qualidade

Klaus Iglberger

O'REILLY®

Design de software C++

Um bom design de software é essencial para o sucesso do seu projeto, mas projetar software é uma tarefa difícil. Você precisa ter um profundo conhecimento das consequências das decisões de design e uma boa visão geral das alternativas de design disponíveis. Com este livro, desenvolvedores C++ experientes obterão uma visão geral completa, prática e incomparável do design de software com esta linguagem moderna.

O instrutor e consultor de C++ Klaus Iglberger explica como gerenciar dependências e abstrações, melhorar a mutabilidade e a extensibilidade de entidades de software e aplicar e implementar padrões de design modernos para ajudar você a aproveitar as possibilidades atuais. O design de software é o aspecto mais essencial de um projeto de software, pois impacta suas propriedades mais importantes: manutenibilidade, mutabilidade e extensibilidade.

- Aprenda a avaliar seu código em relação ao software projeto
- Entender o que é design de software, incluindo objetivos de design como mutabilidade e extensibilidade
- Explore as vantagens e desvantagens de cada design abordagem
- Aprenda como os padrões de design ajudam a resolver problemas e expressar intenção
- Escolha a forma correta de um padrão de design para aproveitar ao máximo suas vantagens

> Este livro elevará o nível de praticamente qualquer programador C++. Ele está repleto de padrões de design práticos e ideias fascinantes. Aprendi muito mais com este livro do que eu jamais esperava.
>
> —Mark Summerfield
> Proprietário da Qtrac Ltd

Klaus Iglberger é instrutor e consultor freelancer de C++. Ele compartilha seus 15 anos de experiência em C++ em cursos de treinamento populares ao redor do mundo e é palestrante frequente em conferências de C++. Desde que concluiu seu doutorado em 2010, ele se concentra em design de software em larga escala e na melhoria da manutenibilidade de software.

PROGRAMAÇÃO

US$ 79,99 CAN$ 99,99
ISBN: 978-1-098-11316-2

Twitter: @oreillymedia
linkedin.com/company/oreilly-media
youtube.com/oreillymedia

Elogios ao design de software C++

Mesmo depois de conhecer padrões de design há muito tempo, este livro me mostrou um número surpreendentemente grande de novos aspectos sobre como usá-los corretamente no contexto de C++ e dos princípios SOLID.

—Matthias Dörfel, CTO da INCHRON AG

Gostei muito de ler o livro! Estudar as diretrizes me fez reconsiderar meu código e aprimorá-lo aplicando-as. Você poderia pedir mais?

—Daniela Engert, engenheira de software sênior da GMH Prüftechnik GmbH

Um dos livros sobre design de software mais divertidos e úteis que li nos últimos tempos.

—Patrice Roy, professor, Collège Lionel-Groulx

Já se passaram mais de 25 anos desde que os Padrões de Design da Gang of Four mudaram a forma como os programadores pensam sobre software. Este livro mudou a minha forma de pensar sobre Padrões de Design.

—Stephan Weller, Software da Siemens Digital Industries

Design de software C++

Princípios e padrões de design para software de alta qualidade

Klaus Iglberger

Pequim • Boston • Farnham • Sebastopol • Tóquio O'REILLY®

Design de software
C++ por Klaus Iglberger

Impresso nos Estados Unidos da América.

Publicado pela O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Os livros da O'Reilly podem ser adquiridos para uso educacional, comercial ou promocional. Edições online também estão disponíveis para a maioria dos títulos (http://oreilly.com). Para mais informações, entre em contato com nosso departamento de vendas corporativas/institucionais: 800-998-9938 ou corporate@oreilly.com.

Editora de Aquisições: Amanda Quinn
Editora de desenvolvimento: Shira Evans
Editora de produção: Kate Galloway
Editora de texto: Sonia Saruba
Revisor: Piper Editorial Consulting, LLC

Indexador: Judith McConville
Designer de interiores: David Futato
Designer de capa: Karen Montgomery
Ilustradora: Kate Dullea

Setembro de 2022: Primeira edição

Histórico de revisão da primeira edição
21/09/2022: Primeiro lançamento
2023-12-08: Segundo lançamento

Veja http://oreilly.com/catalog/errata.csp?isbn=9781098113162 para detalhes do lançamento.

978-1-098-11316-2

[LSI]

Índice

Prefácio

Em suas mãos, você tem o livro de C++ que eu gostaria de ter tido há muitos anos. Não como um dos meus primeiros livros, não, mas como um livro avançado, depois que eu já havia assimilado a mecânica da linguagem e conseguido pensar além da sintaxe C++. Sim, este livro definitivamente me ajudaria a entender melhor os aspectos fundamentais da manutenção de software, e tenho certeza de que ajudará você também.

Por que eu escrevi este livro

Quando comecei a realmente me aprofundar na linguagem (alguns anos depois do lançamento do primeiro padrão C++), eu já tinha lido praticamente todos os livros de C++ que existiam. Mas, apesar do fato de que muitos desses livros eram ótimos e definitivamente abriram caminho para minha carreira atual como instrutor e consultor de C++, eles estavam muito focados nos pequenos detalhes e nas especificidades da implementação, e muito distantes do panorama geral do software sustentável.

Na época, poucos livros realmente focavam no panorama geral, lidando com o desenvolvimento de grandes sistemas de software. Entre eles estavam "Large Scale C++ Software Design", de John Lakos, uma ótima[1], mas literalmente pesada, introdução ao gerenciamento de dependências, e o chamado livro "Gang of Four", que é o livro clássico sobre padrões de projeto de software.[2] Infelizmente, ao longo dos anos, essa situação não mudou muito: a maioria dos livros, palestras, blogs, etc., foca principalmente na mecânica e nos recursos da linguagem — os pequenos detalhes e especificidades. Pouquíssimos, e na minha opinião pouquíssimos, novos lançamentos focam em software sustentável, mutabilidade, extensibilidade e testabilidade. E se tentam, infelizmente, rapidamente caem no hábito comum de explicar a mecânica da linguagem e demonstrar recursos.

1 John Lakos, Projeto de software C++ em larga escala (Addison-Wesley, 1996).

2 Erich Gamma et al., Padrões de projeto: elementos de software orientado a objetos reutilizável (Addison-Wesley, 1994).

É por isso que escrevi este livro. Um livro que, ao contrário da maioria dos outros, não se dedica à mecânica ou aos muitos recursos da linguagem, mas se concentra principalmente na mutabilidade, extensibilidade e testabilidade do software em geral. Um livro que não pretende que o uso de novos padrões ou recursos de C++ fará a diferença entre um software bom ou ruim, mas, em vez disso, mostra claramente que é o gerenciamento de dependências que é decisivo, que as dependências em nosso código decidem se ele é bom ou ruim. Como tal, é um tipo raro de livro no mundo de C++, pois se concentra no panorama geral: o design de software.

Sobre o que é este livro

Design de Software

Do meu ponto de vista, um bom design de software é a essência de todo projeto de software bem-sucedido. No entanto, apesar de seu papel fundamental, há muito pouca literatura sobre o assunto e pouquíssimos conselhos sobre o que fazer e como fazer as coisas corretamente. Por quê? Bem, porque é difícil. Muito difícil. Provavelmente a faceta mais difícil da escrita de software que temos que enfrentar. E isso porque não existe uma única solução "certa", nenhum conselho "de ouro" para passar adiante entre as gerações de desenvolvedores de software. Sempre depende.

Apesar dessa limitação, darei dicas sobre como projetar um software de boa qualidade. Apresentarei princípios, diretrizes e padrões de design que ajudarão você a entender melhor como gerenciar dependências e transformar seu software em algo com o qual você possa trabalhar por décadas. Como mencionado anteriormente, não existe um conselho "de ouro" e este livro não contém nenhuma solução definitiva ou perfeita. Em vez disso, tento mostrar os aspectos mais fundamentais de um bom software, os detalhes mais importantes, a diversidade e os prós e contras de diferentes designs. Também formularei objetivos intrínsecos de design e demonstrarei como alcançá-los com C++ moderno.

C++ moderno

Por mais de uma década, celebramos o advento do C++ Moderno, aplaudindo os muitos novos recursos e extensões da linguagem e, com isso, criando a impressão de que o C++ Moderno nos ajudará a resolver todos os problemas relacionados a software. Mas não é o caso neste livro. Este livro não pretende que aplicar algumas dicas inteligentes ao código o tornará "moderno" ou resultará automaticamente em um bom design. Além disso, este livro não mostrará o C++ Moderno como uma coleção de novos recursos. Em vez disso, mostrará como a filosofia da linguagem evoluiu e a maneira como implementamos soluções em C++ hoje.

Mas é claro que também veremos código. Muito código. E, claro, este livro fará uso dos recursos dos padrões C++ mais recentes (incluindo C++20). No entanto, também se esforçará para enfatizar que o design é independente dos detalhes de implementação e dos recursos utilizados. Novos recursos não alteram as regras sobre o que é um bom ou mau design; eles apenas mudam a maneira como implementamos um bom design. Eles facilitam a implementação de um bom design. Portanto, este livro mostra e discute detalhes de implementação, mas (espero) não se perde neles e sempre se mantém focado no panorama geral: design de software e padrões de design.

Padrões de Projeto

Assim que você começa a mencionar padrões de projeto, inadvertidamente evoca a expectativa de programação orientada a objetos e hierarquias de herança. Sim, este livro mostrará a origem orientada a objetos de muitos padrões de projeto. No entanto, ele enfatizará fortemente o fato de que não há apenas uma maneira de fazer bom uso de um padrão de projeto. Demonstrarei como a implementação de padrões de projeto evoluiu e se diversificou, fazendo uso de muitos paradigmas diferentes, incluindo programação orientada a objetos, programação genérica e programação funcional. Este livro reconhece a realidade de que não existe um paradigma verdadeiro e não pretende que haja apenas uma única abordagem, uma solução sempre funcional para todos os problemas. Em vez disso, ele tenta mostrar o C++ moderno como ele realmente é: a oportunidade de combinar todos os paradigmas, entrelaçá-los em uma rede forte e durável e criar um design de software que durará por décadas.

Espero que este livro seja a peça que faltava na literatura sobre C++. Espero que ele ajude você tanto quanto me ajudou. Espero que ele contenha algumas respostas que você estava procurando e que lhe forneça alguns insights importantes que você estava perdendo. Espero também que este livro mantenha você entretido e motivado a ler tudo. Mais importante ainda, espero que este livro lhe mostre a importância do design de software e o papel que os padrões de design desempenham. Porque, como você verá, padrões de design estão em toda parte!

Para quem é este livro

Este livro é valioso para todos os desenvolvedores C++. Em particular, é para todos os desenvolvedores C++ interessados em entender os problemas comuns de software sustentável e aprender sobre soluções comuns para esses problemas (e presumo que isso se aplique a todos os desenvolvedores C++). No entanto, este livro não é um livro para iniciantes em C++. Na verdade, a maioria das diretrizes neste livro exige alguma experiência com desenvolvimento de software em geral e C++ em particular. Por exemplo, presumo que você tenha um sólido domínio da mecânica da linguagem de hierarquias de herança e alguma experiência com modelos. Assim, posso recorrer aos recursos correspondentes sempre que necessário e apropriado. De vez em quando, até recorrerei a alguns recursos do C++20 (em parti

Conceitos de C++20). No entanto, como o foco é o design de software, raramente me deterei na explicação de um recurso específico; portanto, se você não conhece um recurso, consulte sua referência favorita da linguagem C++. Apenas ocasionalmente adicionarei alguns lembretes, principalmente sobre expressões idiomáticas comuns de C++ (como a Regra de 5).

Como este livro é estruturado

Este livro está organizado em capítulos, cada um contendo diversas diretrizes. Cada diretriz se concentra em um aspecto-chave da manutenção de software ou em um padrão de design específico. Portanto, as diretrizes representam os principais pontos, os aspectos que espero que tragam mais valor para você. Elas foram escritas de forma que você possa lê-las todas do início ao fim, mas, como são apenas vagamente acopladas, permitem que você também comece com a diretriz que lhe chamar a atenção. Ainda assim, elas não são independentes. Portanto, cada diretriz contém as referências cruzadas necessárias para outras diretrizes para mostrar que tudo está conectado.

Convenções usadas neste livro

As seguintes convenções tipográficas são usadas neste livro:

itálico

Indica novos termos, URLs, endereços de e-mail, nomes de arquivos e extensões de arquivo.

Largura constante

Usada para listagens de programas, bem como dentro de parágrafos para se referir a elementos do programa, como nomes de variáveis ou funções, bancos de dados, tipos de dados, variáveis de ambiente, instruções e palavras-chave.

Largura constante em

negrito Mostra comandos ou outro texto que deve ser digitado literalmente pelo usuário.

Itálico de largura constante

Mostra texto que deve ser substituído por valores fornecidos pelo usuário ou por valores determinados pelo contexto.

Este elemento significa uma dica ou sugestão.

Este elemento significa uma nota geral.

Usando exemplos de código

Material suplementar (exemplos de código, exercícios, etc.) está disponível para download em https://github.com/igl42/cpp_software_design.

Se você tiver alguma dúvida técnica ou problema ao usar os exemplos de código, envie um e-mail para bookquestions@oreilly.com.

Este livro está aqui para ajudar você a realizar seu trabalho. Em geral, se um código de exemplo for oferecido com este livro, você poderá usá-lo em seus programas e documentação. Não é necessário entrar em contato conosco para obter permissão, a menos que esteja reproduzindo uma parte significativa do código. Por exemplo, escrever um programa que use vários trechos de código deste livro não requer permissão. Vender ou distribuir exemplos de livros da O'Reilly requer permissão. Responder a uma pergunta citando este livro e citando um código de exemplo não requer permissão. Incorporar uma quantidade significativa de código de exemplo deste livro na documentação do seu produto requer permissão.

Agradecemos, mas geralmente não exigimos, a atribuição. Uma atribuição geralmente inclui o título, autor, editora e ISBN. Por exemplo: “C++ Software Design by Klaus Iglberger (O'Reilly). Copyright 2022 Klaus Iglberger, 978-1-098-11316-2.”

Se você acha que o uso de exemplos de código não se enquadra no uso justo ou na permissão dada acima, sinta-se à vontade para entrar em contato conosco pelo e-mail permissions@oreilly.com.

Aprendizagem on-line da O'Reilly

Por mais de 40 anos, a O'Reilly Media forneceu tecnologia e treinamento empresarial, conhecimento e insights para ajudar empresas a ter sucesso.

Nossa rede única de especialistas e inovadores compartilha seu conhecimento e expertise por meio de livros, artigos e nossa plataforma de aprendizagem online. A plataforma de aprendizagem online da O'Reilly oferece acesso sob demanda a cursos de treinamento ao vivo, trilhas de aprendizagem aprofundadas, ambientes de codificação interativos e uma vasta coleção de textos e vídeos da O'Reilly e de mais de 200 outras editoras. Para mais informações, visite http://oreilly.com.

Como entrar em contato conosco

Por favor, envie comentários e perguntas sobre este livro à editora:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (nos Estados Unidos ou Canadá)
707-829-0515 (internacional ou local)
707-829-0104 (fax)

Temos uma página web para este livro, onde listamos erratas, exemplos e quaisquer informações adicionais. Você pode acessá-la em https://oreil.ly/c-plus-plus.

E-mail bookquestions@oreilly.com para comentar ou fazer perguntas técnicas sobre este livro.

Para notícias e informações sobre nossos livros e cursos, visite http://oreilly.com.

Encontre-nos no LinkedIn: https://linkedin.com/company/oreilly-media.

Siga-nos no Twitter: http://twitter.com/oreillymedia.

Assista-nos no YouTube: http://youtube.com/oreillymedia.

Agradecimentos

Um livro como este nunca é obra de uma única pessoa. Pelo contrário, tenho que agradecer explicitamente a muitas pessoas que me ajudaram de diferentes maneiras a torná-lo realidade. Em primeiro lugar, quero expressar minha profunda gratidão à minha esposa, Steffi, que leu o livro inteiro sem nem mesmo saber C++. E que cuidou dos nossos dois filhos para me dar a calma necessária para colocar todas essas informações no papel (ainda não sei qual dos dois foi o maior sacrifício).

Um agradecimento especial aos meus revisores, Daniela Engert, Patrice Roy, Stefan Weller, Mark Summerfield e Jacob Bandes-Storch, por investirem seu valioso tempo para tornar este livro melhor, desafiando constantemente minhas explicações e exemplos.

Um grande agradecimento também a Arthur O'Dwyer, Eduardo Madrid e Julian Schmidt por suas contribuições e feedback sobre o padrão de design Type Erasure, e a Johannes Gutekunst pelas discussões sobre arquitetura e documentação de software.

Além disso, quero agradecer aos meus dois leitores frios, Matthias Dörfel e Vittorio Romeo, que ajudaram a detectar muitos erros de última hora (e de fato detectaram).

Por último, mas não menos importante, um grande agradecimento à minha editora, Shira Evans, que passou muitas horas dando conselhos inestimáveis sobre como tornar o livro mais consistente e mais divertido de ler.

CAPÍTULO 1

A Arte do Design de Software

O que é design de software? E por que você deveria se importar com isso? Neste capítulo, apresentarei o contexto para este livro sobre design de software. Explicarei o design de software em geral, ajudarei você a entender por que ele é vital para o sucesso de um projeto e por que é a única coisa que você deve acertar. Mas você também verá que o design de software é complicado. Muito complicado. Na verdade, é a parte mais complexa do desenvolvimento de software. Portanto, também explicarei vários princípios de design de software que ajudarão você a se manter no caminho certo.

Na “Diretriz 1: Entendendo a Importância do Design de Software”, na página 2, focarei no panorama geral e explicarei que se espera que o software mude. Consequentemente, o software deve ser capaz de lidar com mudanças. No entanto, isso é muito mais fácil de dizer do que de fazer, já que, na realidade, acoplamentos e dependências tornam nossa vida como desenvolvedores muito mais difícil. Esse problema é abordado pelo design de software. Apresentarei o design de software como a arte de gerenciar dependências e abstrações — uma parte essencial da engenharia de software.

Na “Diretriz 2: Design para Mudanças”, na página 11, abordarei explicitamente acoplamentos e dependências e ajudarei você a entender como projetar para mudanças e como tornar o software mais adaptável. Para isso, apresentarei o Princípio da Responsabilidade Única (SRP) e o princípio Não se Repita (DRY), que ajudarão você a atingir esse objetivo.

Na “Diretriz 3: Interfaces Separadas para Evitar Acoplamento Artificial” na página 24, expandirei a discussão sobre acoplamento e abordarei especificamente o acoplamento por meio de interfaces. Também apresentarei o Princípio de Segregação de Interface (ISP) como um meio de reduzir o acoplamento artificial induzido por interfaces.

Na “Diretriz 4: Design para Testabilidade”, na página 28, abordarei questões de testabilidade que surgem como resultado do acoplamento artificial. Em particular, levantarei a questão de como testar uma função-membro privada e demonstrarei que a única solução verdadeira é uma aplicação consequente da separação de responsabilidades.

Na “Diretriz 5: Design para Extensão”, na página 35, abordarei um tipo importante de mudança: extensões. Assim como o código deve ser fácil de alterar, ele também deve ser fácil de estender. Darei uma ideia de como atingir esse objetivo e demonstrarei o valor do Princípio Aberto-Fechado (OCP).

Diretriz 1: Compreender a importância de Design de software

Se eu lhe perguntasse quais propriedades de código são mais importantes para você, você, depois de pensar um pouco, provavelmente diria coisas como legibilidade, testabilidade, manutenibilidade, extensibilidade, reusabilidade e escalabilidade. E eu concordaria plenamente. Mas agora, se eu lhe perguntasse como atingir esses objetivos, há uma boa chance de você começar a listar alguns recursos do C++: RAII, algoritmos, lambdas, módulos e assim por diante.

Recursos não são design de software .

Sim, C++ oferece muitos recursos. Muitos! Aproximadamente metade das quase 2.000 páginas do padrão C++ impresso são dedicadas a explicar a mecânica e os recursos da linguagem.[1] E desde o lançamento do C++11, há a promessa explícita de que haverá mais: a cada três anos, o comitê de padronização C++ nos abençoa com um novo padrão C++ que vem com recursos adicionais e totalmente novos. Sabendo disso, não é nenhuma surpresa que na comunidade C++ haja uma ênfase tão forte em recursos e na mecânica da linguagem. A maioria dos livros, palestras e blogs se concentra em recursos, novas bibliotecas e detalhes da linguagem.[2] Quase parece que os recursos são a coisa mais importante sobre

programação em C++ e cruciais para o sucesso de um projeto C++. Mas, honestamente, não são. Nem o conhecimento sobre todos os recursos nem a escolha do padrão C++ são responsáveis pelo sucesso de um projeto. Não, você não deve esperar que os recursos salvem seu projeto. Pelo contrário: um projeto pode ser muito bem-sucedido mesmo que use uma linguagem C++ mais antiga

1 Mas é claro que você nunca tentaria imprimir o padrão C++ atual. Você usaria um PDF do padrão C++ oficial ou use o rascunho de trabalho atual. No entanto, para a maior parte do seu trabalho diário, talvez você queira consultar o site de referência do C++.

2 Infelizmente, não posso apresentar números, pois dificilmente posso dizer que tenho uma visão geral completa do vasto universo do C++. Pelo contrário, posso nem ter uma visão geral completa das fontes que conheço! Portanto, considere isso como minha impressão pessoal e a forma como percebo a comunidade C++. Você pode ter uma impressão diferente.

padrão, e mesmo que apenas um subconjunto dos recursos disponíveis seja usado. Deixando de lado os aspectos humanos do desenvolvimento de software, muito mais importante para a questão sobre o sucesso ou fracasso de um projeto é a estrutura geral do software. É a estrutura que é, em última análise, responsável pela manutenibilidade: quão fácil é alterar, estender e testar código? Sem a capacidade de alterar facilmente o código, adicionar novas funcionalidades e ter confiança em sua correção devido aos testes, um projeto está no fim de seu ciclo de vida. A estrutura também é responsável pela escalabilidade de um projeto: quão grande o projeto pode crescer antes de entrar em colapso sob seu próprio peso? Quantas pessoas podem trabalhar na realização da visão do projeto antes de pisarem nos calos umas das outras?

A estrutura geral é o design de um projeto. O design desempenha um papel muito mais central no sucesso de um projeto do que qualquer funcionalidade. Um bom software não se resume principalmente ao uso adequado de qualquer funcionalidade; em vez disso, trata-se de arquitetura e design sólidos. Um bom design de software pode tolerar algumas decisões de implementação ruins, mas um design de software ruim não pode ser salvo apenas pelo uso heroico de funcionalidades (antigas ou novas).

Design de Software: A Arte de Gerenciar Dependências e Abstrações

Por que o design de software é tão importante para a qualidade de um projeto? Bem, supondo que tudo funcione perfeitamente agora, desde que nada mude no seu software e nada precise ser adicionado, você está bem. No entanto, esse estado provavelmente não durará muito. É razoável esperar que algo mude. Afinal, a única constante no desenvolvimento de software é a mudança. A mudança é a força motriz por trás de todos os nossos problemas (e também da maioria das nossas soluções). É por isso que o software é chamado de software: porque, em comparação com o hardware, ele é flexível e maleável. Sim, espera-se que o software seja facilmente adaptado aos requisitos em constante mudança. Mas, como você deve saber, na realidade, essa expectativa pode nem sempre ser verdadeira.

Para ilustrar esse ponto, imagine que você seleciona um problema do seu sistema de rastreamento de problemas que a equipe classificou com um esforço esperado de 2. Seja qual for o significado de 2 em seu(s) projeto(s), certamente não parece uma tarefa grande, então você está confiante de que será concluído rapidamente. De boa-fé, você primeiro dedica algum tempo para entender o que é esperado e, em seguida, começa fazendo uma alteração em alguma entidade A.
Devido ao feedback imediato dos seus testes (você tem sorte de ter testes!), você é rapidamente lembrado de que também precisa resolver o problema na entidade B. Isso é surpreendente! Você não esperava que B estivesse envolvido. Mesmo assim, você prossegue e adapta B mesmo assim.
No entanto, novamente inesperadamente, a compilação noturna revela que isso faz com que C e D parem de funcionar. Antes de continuar, você investiga o problema um pouco mais a fundo e descobre que as raízes do problema estão espalhadas por uma grande parte da base de código. A pequena tarefa, inicialmente aparentemente inocente, evoluiu para um código grande e potencialmente arriscado.

modificação.3 Sua confiança em resolver o problema rapidamente se foi. E seus planos para o resto da semana também.

Talvez esta história lhe pareça familiar. Talvez você possa até contribuir com algumas histórias de guerra próprias. De fato, a maioria dos desenvolvedores tem experiências semelhantes. E a maioria dessas experiências tem a mesma fonte de problemas. Geralmente, o problema pode ser reduzido a uma única palavra: dependências. Como Kent Beck expressou em seu livro sobre desenvolvimento orientado a testes:4

> Dependência é o principal problema no desenvolvimento de software em todas as escalas.

Dependências são a ruína da existência de todo desenvolvedor de software. "Mas é claro que existem dependências", você argumenta. "Sempre haverá dependências. De que outra forma diferentes partes do código deveriam funcionar juntas?" E, claro, você está certo. Diferentes partes do código precisam funcionar juntas, e essa interação sempre criará alguma forma de acoplamento. No entanto, embora existam dependências necessárias e inevitáveis, também existem dependências artificiais que introduzimos acidentalmente porque não entendemos o problema subjacente, não temos uma ideia clara do panorama geral ou simplesmente não prestamos atenção suficiente. Nem é preciso dizer que essas dependências artificiais prejudicam. Elas dificultam a compreensão do nosso software, a alteração de software, a adição de novos recursos e a escrita de testes. Portanto, uma das principais tarefas, se não a principal, de um desenvolvedor de software é manter as dependências artificiais no mínimo.

Essa minimização de dependências é o objetivo da arquitetura e do design de software.
Para expressá-lo nas palavras de Robert C. Martin:5

> O objetivo da arquitetura de software é minimizar os recursos humanos necessários para construir e manter o sistema requerido.

Arquitetura e design são as ferramentas necessárias para minimizar o esforço de trabalho em qualquer projeto. Eles lidam com dependências e reduzem a complexidade por meio de abstrações. Em minhas próprias palavras:6

> O design de software é a arte de gerenciar interdependências entre componentes de software. Seu objetivo é minimizar dependências artificiais (técnicas) e introduzir as abstrações e concessões necessárias.

---

3 O risco ou não da modificação do código pode depender muito da sua cobertura de teste. Uma boa cobertura de teste A erage pode realmente absorver alguns dos danos que um projeto de software ruim pode causar.

4 Kent Beck, Desenvolvimento orientado a testes: por exemplo (Addison-Wesley, 2002).

5 Robert C. Martin, Arquitetura Limpa (Addison-Wesley, 2017).

6 Estas são, de fato, minhas próprias palavras, visto que não existe uma definição única e comum de design de software. Consequentemente, você pode ter sua própria definição do que o design de software envolve, e isso é perfeitamente aceitável. No entanto, observe que este livro, incluindo a discussão sobre padrões de design, é baseado na minha definição.

Sim, design de software é uma arte. Não é uma ciência e não vem com um conjunto de respostas fáceis e claras.7 Muitas vezes, o panorama geral do design nos escapa e somos sobrecarregados pelas complexas interdependências das entidades de software. Mas estamos tentando lidar com essa complexidade e reduzi-la introduzindo o tipo certo de abstrações.
Dessa forma, mantemos o nível de detalhes em um nível razoável. No entanto, muitas vezes, os desenvolvedores individuais da equipe podem ter uma ideia diferente da arquitetura e do design. Podemos não conseguir implementar nossa própria visão de design e ser forçados a fazer concessões para seguir em frente.

O termo abstração é usado em diferentes contextos. É usado para a organização de funcionalidades e itens de dados em tipos de dados e funções. Mas também é usado para descrever a modelagem de comportamento comum e a representação de um conjunto de requisitos e expectativas. Neste livro sobre design de software, usarei o termo principalmente para este último (veja em particular o Capítulo 2).

Observe que as palavras arquitetura e design podem ser intercambiáveis nas citações anteriores, pois são muito semelhantes e compartilham os mesmos objetivos. No entanto, não são a mesma coisa. As semelhanças, mas também as diferenças, ficam claras se você observar os três níveis de desenvolvimento de software.

Os Três Níveis de Desenvolvimento de Software Arquitetura

de Software e Design de Software são apenas dois dos três níveis de desenvolvimento de software. Eles são complementados pelo nível de Detalhes de Implementação.
A Figura 1-1 fornece uma visão geral desses três níveis.

Para lhe dar uma ideia desses três níveis, vamos começar com um exemplo real da relação entre arquitetura, design e detalhes de implementação. Considere-se no papel de um arquiteto. E não, por favor, não se imagine em uma cadeira confortável em frente a um computador com um café quente ao seu lado, mas imagine-se do lado de fora, em um canteiro de obras. Sim, estou falando de um arquiteto de edifícios. 8 Como arquiteto, você seria responsável por todas as propriedades importantes de uma casa: sua integração com o bairro, sua integridade estrutural, a disposição dos cômodos, o encanamento, etc. Você também cuidaria de uma aparência agradável e das qualidades funcionais — talvez uma sala de estar ampla, fácil acesso entre a cozinha e a sala de jantar, e assim por diante. Em outras palavras, você cuidaria do conjunto geral

7 Só para esclarecer: a ciência da computação é uma ciência (está no nome). A engenharia de software parece ser uma ciência híbrida forma de ciência, artesanato e arte. E um aspecto desta última é o design de software.

8 Com essa metáfora, não estou tentando dizer que arquitetos de edifícios trabalham no canteiro de obras o dia todo. É bem provável que esse arquiteto passe tanto tempo em uma cadeira confortável e em frente ao computador quanto pessoas como você e eu. Mas acho que você entendeu.

arquitetura, os aspectos que seriam difíceis de mudar posteriormente, mas você também lidaria com os aspectos menores do design referentes ao edifício. No entanto, é difícil diferenciar os dois: a fronteira entre arquitetura e design parece fluida e não está claramente separada.

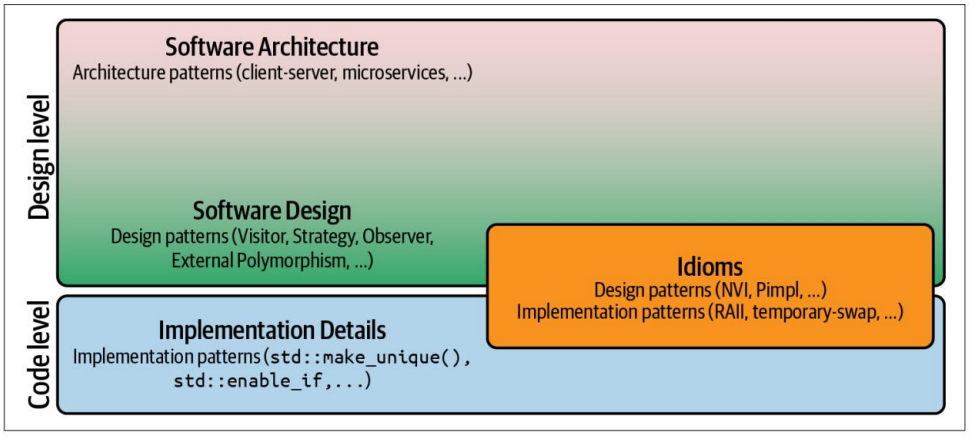

Figura 1-1. Os três níveis de desenvolvimento de software: Arquitetura de Software, Design de Software e Detalhes de Implementação. Expressões idiomáticas podem ser padrões de design ou de implementação.

No entanto, essas decisões seriam o fim da sua responsabilidade. Como arquiteto, você não se preocuparia com onde colocar a geladeira, a TV ou outros móveis.
Você não lidaria com todos os detalhes interessantes sobre onde colocar quadros e outras peças de decoração. Em outras palavras, você não cuidaria dos detalhes; apenas garantiria que o proprietário tivesse a estrutura necessária para viver bem.

Os móveis e outros "detalhes interessantes" nesta metáfora correspondem ao nível mais baixo e concreto do desenvolvimento de software: os detalhes de implementação. Este nível trata de como uma solução é implementada. Você escolhe o necessário (e disponível)
Padrão C++ ou qualquer subconjunto dele, bem como os recursos apropriados, palavras-chave e especificações de linguagem a serem usadas, e lidar com aspectos como aquisição de memória, segurança de exceção, desempenho, etc. Este também é o nível de padrões de implementação, como std::make_unique() como uma função de fábrica, std::enable_if como uma solução recorrente para se beneficiar explicitamente do SFINAE, etc.9

9 Substitution Failure Is Not An Error (SFINAE) é um mecanismo básico de template comumente usado como substituto para conceitos do C++20 para restringir templates. Para uma explicação sobre SFINAE e std::enable_if em particular, consulte seu livro favorito sobre templates C++. Se você não tiver nenhum, uma ótima opção é a bíblia de templates C++: David Vandevoorde, Nicolai Josuttis e Douglas Gregor's C++ Templates: The Complete Guide (Addison-Wesley).

No design de software, você começa a se concentrar no panorama geral. Questões sobre manutenibilidade, mutabilidade, extensibilidade, testabilidade e escalabilidade são mais pronunciadas neste nível. O design de software lida principalmente com a interação de entidades de software, que na metáfora anterior são representadas pela disposição de salas, portas, canos e cabos. Neste nível, você lida com as dependências físicas e lógicas dos componentes (classes, funções, etc.).10 É o nível de padrões de design como Visitor, Strategy e Decorator que define uma estrutura de dependência entre entidades de software, conforme explicado no Capítulo 3. Esses padrões, que geralmente são transferíveis de uma linguagem para outra, ajudam a decompor coisas complexas em partes digeríveis.

Arquitetura de software é o mais confuso dos três níveis, o mais difícil de colocar em palavras.
Isso ocorre porque não existe uma definição comum e universalmente aceita de arquitetura de software. Embora possa haver muitas visões diferentes sobre o que exatamente é uma arquitetura, há um aspecto com o qual todos parecem concordar: a arquitetura geralmente envolve as grandes decisões, os aspectos do seu software que estão entre os mais difíceis de mudar no futuro:

> Arquitetura são as decisões que você gostaria de tomar logo no início de um projeto, mas que não necessariamente são mais propensas a tomar do que qualquer outra.11
>
> —Ralph Johnson

Na Arquitetura de Software, são utilizados padrões de arquitetura, como arquitetura cliente-servidor, microsserviços e assim por diante.12 Esses padrões também abordam a questão de como projetar sistemas, onde é possível alterar uma parte sem afetar as demais do software. Semelhante aos padrões de design de software, eles definem e abordam a estrutura e as interdependências entre as entidades de software. Em contraste com os padrões de design, porém, eles geralmente lidam com os principais participantes, as grandes entidades do software (por exemplo, módulos e componentes em vez de classes e funções).

Dessa perspectiva, a Arquitetura de Software representa a estratégia geral da sua abordagem de software, enquanto o Design de Software é a tática para fazer a estratégia funcionar.
O problema com essa imagem é que não há uma definição de “grande”. Especialmente com o advento dos microsserviços, torna-se cada vez mais difícil traçar uma linha clara entre entidades pequenas e grandes.13

---

10 Para obter mais informações sobre gerenciamento de dependências físicas e lógicas, consulte o livro “dam” de John Lakos, Desenvolvimento de software C++ em larga escala: processo e arquitetura (Addison-Wesley).

11 Martin Fowler, “Quem precisa de um arquiteto?” IEEE Software, 20, n.º 5 (2003), 11–13, https://doi.org/10.1109/MS.2003.1231144.

12 Uma ótima introdução aos microsserviços pode ser encontrada no livro de Sam Newman , Building Microservices: Designing Fine-Grained Systems, 2ª edição (O'Reilly).

13 Mark Richards e Neal Ford, Fundamentos da Arquitetura de Software: Uma Abordagem de Engenharia (O'Reilly, 2020).

Assim, a arquitetura é frequentemente descrita como o que os desenvolvedores especialistas em um projeto percebem como decisões-chave.

O que torna a separação entre arquitetura, design e detalhes um pouco mais difícil é o conceito de idioma. Um idioma é uma solução comumente usada, mas específica da linguagem, para um problema recorrente. Como tal, um idioma também representa um padrão, mas pode ser um padrão de implementação ou um padrão de design.[14] Em termos mais gerais, as expressões idiomáticas C++ são as melhores práticas da comunidade C++, tanto para design quanto para implementação. Em C++, a maioria das expressões idiomáticas se enquadra na categoria de detalhes de implementação. Por exemplo, existe a expressão idiomática de copiar e trocar. que você pode saber a partir da implementação de um operador de atribuição de cópia e do idioma RAII (Aquisição de Recursos é Inicialização — você definitivamente deve estar familiarizado com isso; caso contrário, consulte seu segundo livro favorito de C++15). Nenhuma dessas expressões introduz uma abstração e nenhuma delas ajuda a dissociar. Ainda assim, são indispensáveis para implementar um bom código C++.

Você pergunta: "Poderia ser um pouco mais específico, por favor? O RAII também não oferece alguma forma de desacoplamento? Ele não desvincula o gerenciamento de recursos da lógica de negócios?" Você está certo: o RAII separa o gerenciamento de recursos da lógica de negócios.
No entanto, isso não é alcançado por meio de desacoplamento, ou seja, abstração, mas sim por meio de encapsulamento. Tanto a abstração quanto o encapsulamento ajudam a tornar sistemas complexos mais fáceis de entender e modificar, mas enquanto a abstração resolve os problemas e questões que surgem no nível de Design de Software, o encapsulamento resolve os problemas e questões que surgem no nível de Detalhes de Implementação. Para citar a Wikipédia:

> As vantagens do RAII como técnica de gerenciamento de recursos são que ele fornece encapsulamento, segurança de exceções […] e localidade […]. O encapsulamento é fornecido porque a lógica de gerenciamento de recursos é definida uma vez na classe, não em cada local de chamada.

Embora a maioria das expressões idiomáticas se enquadre na categoria de Detalhes de Implementação, também existem expressões idiomáticas que se enquadram na categoria de Design de Software. Dois exemplos são a linguagem Non-Virtual Interface (NVI) e a linguagem Pimpl. Essas duas expressões idiomáticas são baseadas em dois padrões de design clássicos: o padrão de design Template Method e o Bridge.

---

14 O termo padrão de implementação foi usado pela primeira vez no livro Implementation Patterns (Padrões de Implementação) de Kent Beck (Addison-Wesley). Neste livro, uso esse termo para fazer uma distinção clara do termo padrão de design, já que o termo idioma pode se referir a um padrão no nível de Design de Software ou no nível de Detalhes de Implementação. Usarei o termo consistentemente para me referir a soluções comumente usadas no nível de Detalhes de Implementação.

15 O segundo favorito depois deste, claro. Se este for o seu único livro, você pode consultar o clássico Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3ª ed., de Scott Meyers (Addison-Wesley).

padrão de design, respectivamente.16 Eles introduzem uma abstração e ajudam a desacoplar e projetar para mudanças e extensões.

O foco nos recursos

Se a arquitetura e o design de software são tão importantes, por que nós, da comunidade C++, focamos tanto em recursos? Por que criamos a ilusão de que os padrões, a mecânica da linguagem e os recursos do C++ são decisivos para um projeto?
Acredito que há três fortes razões para isso. Primeiro, como há tantos recursos, às vezes com detalhes complexos, precisamos dedicar bastante tempo para discutir como usar todos eles corretamente. Precisamos criar um entendimento comum sobre qual uso é bom e qual é ruim. Nós, como comunidade, precisamos desenvolver um senso de C++ idiomático.

O segundo motivo é que podemos ter expectativas equivocadas sobre os recursos. Como exemplo, consideremos os módulos do C++20. Sem entrar em detalhes, esse recurso pode de fato ser considerado a maior revolução técnica desde o início do C++.
Os módulos podem finalmente pôr fim à prática questionável e complicada de incluir arquivos de cabeçalho em arquivos de origem.

Devido a esse potencial, as expectativas para esse recurso são enormes. Algumas pessoas até esperam que os módulos salvem seus projetos corrigindo seus problemas estruturais. Infelizmente, os módulos terão dificuldade em atender a essas expectativas: eles não melhoram a estrutura ou o design do seu código, mas podem apenas representar a estrutura e o design atuais. Os módulos não corrigem seus problemas de design, mas podem tornar as falhas visíveis. Portanto, os módulos simplesmente não podem salvar seu projeto. Portanto, podemos estar colocando expectativas demais ou erradas nos recursos.

E por último, mas não menos importante, o terceiro motivo é que, apesar da enorme quantidade de recursos e de sua complexidade, em comparação com a complexidade do design de software, a complexidade dos recursos do C++ é pequena. É muito mais fácil explicar um determinado conjunto de regras para recursos, independentemente de quantos casos especiais eles contenham, do que explicar a melhor maneira de desacoplar entidades de software.

Embora geralmente haja uma boa resposta para todas as perguntas relacionadas a recursos, a resposta comum no design de software é "Depende". Essa resposta pode nem ser evidência de inexperiência, mas da percepção de que a melhor maneira de tornar o código mais sustentável, mutável, extensível, testável e escalável depende muito de muitos projetos.

---

16 Os padrões de projeto Template Method e Bridge são dois dos 23 padrões de projeto clássicos introduzidos no livro Gang of Four (GoF), de Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software. Não entrarei em detalhes sobre o Template Method neste livro, mas você encontrará boas explicações em vários livros didáticos, incluindo o próprio livro GoF. No entanto, explicarei o padrão de projeto Bridge na “Diretriz 28: Construir Pontes para Remover Dependências Físicas”, na página 250.

fatores específicos. A dissociação da complexa interação entre muitas entidades pode, de fato, ser um dos empreendimentos mais desafiadores que a humanidade já enfrentou:

> Design e programação são atividades humanas; esqueça isso e tudo estará perdido.17

Para mim, uma combinação desses três motivos é o motivo pelo qual focamos tanto em recursos. Mas, por favor, não me entenda mal. Isso não quer dizer que recursos não sejam importantes. Pelo contrário, recursos são importantes. E sim, é necessário falar sobre recursos e aprender a usá-los corretamente, mas, mais uma vez, eles sozinhos não salvam seu projeto.

O Foco no Design de Software e nos Princípios de Design Embora os recursos sejam importantes, e embora seja claro que seja bom falar sobre eles, o design de software é mais importante. O design de software é essencial. Eu até diria que é a base do sucesso dos nossos projetos. Portanto, neste livro, tentarei realmente focar no design de software e nos princípios de design em vez dos recursos. É claro que ainda mostrarei código C++ bom e atualizado, mas não forçarei o uso das últimas e melhores adições da linguagem.18 Usarei alguns novos recursos quando for razoável e benéfico, como os conceitos do C++20, mas não darei atenção ao noexcept ou usarei constexpr em todos os lugares.19 Em vez disso, tentarei abordar os aspectos difíceis do software. Vou, na maior parte, me concentrar no design de software, na lógica por trás das decisões de design, nos princípios de design, no gerenciamento de dependências e no tratamento de abstrações.

Em resumo, o design de software é a parte crítica da escrita de software. Desenvolvedores de software devem ter um bom conhecimento de design de software para escrever software de qualidade e de fácil manutenção. Afinal, um bom software tem baixo custo, e um software ruim é caro.

Diretriz 1: Entenda a importância do design de software

- Tratar o design de software como uma parte essencial da escrita de software.
- Concentre-se menos nos detalhes da linguagem C++ e mais no design de software.
- Evite acoplamentos e dependências desnecessários para tornar o software mais adaptável a mudanças frequentes.
- Entenda o design de software como a arte de gerenciar dependências e abstrações.

17 Bjarne Stroustrup, A linguagem de programação C++, 3ª ed. (Addison-Wesley, 2000).

18 Parabéns a John Lakos, que argumenta de forma semelhante e usa C++98 em seu livro, Large-Scale C++ Software Developÿ ment: Process and Architecture (Addison-Wesley).

19 Sim, Ben e Jason, vocês leram corretamente. Eu não vou construir TODAS as coisas. Veja Ben Deane e Jason Turner, “construir TODAS as coisas”. CppCon 2017.

- Considere a fronteira entre o design de software e a arquitetura de software como fluido.

Diretriz 2: Design para Mudança

Uma das expectativas essenciais para um bom software é sua capacidade de mudar facilmente. Essa expectativa faz parte até mesmo da palavra software. O software, diferentemente do hardware, deve ser capaz de se adaptar facilmente às mudanças de requisitos (veja também “Diretriz 1: Entendendo a Importância do Design de Software” na página 2). No entanto, com base na sua própria experiência, você pode perceber que, muitas vezes, não é fácil alterar o código. Pelo contrário, às vezes uma mudança aparentemente simples acaba sendo um esforço de uma semana.

Separação de Responsabilidades:

Uma das melhores e comprovadas soluções para reduzir dependências artificiais e simplificar mudanças é separar responsabilidades. O cerne da ideia é dividir, segregar ou extrair partes da funcionalidade:20

> Sistemas divididos em partes pequenas, bem nomeadas e compreensíveis permitem um trabalho mais rápido.

A intenção por trás da separação de interesses é compreender e gerenciar melhor a complexidade e, assim, projetar softwares mais modulares. Essa ideia é provavelmente tão antiga quanto o próprio software e, por isso, recebeu muitos nomes diferentes. Por exemplo, a mesma ideia é chamada de ortogonalidade pelos programadores pragmáticos. 21 Eles recomendam separar os aspectos ortogonais do software. Tom DeMarco chama isso de coesão: 22

> Coesão é uma medida da força de associação dos elementos dentro de um módulo. Um módulo altamente coeso é um conjunto de declarações e itens de dados que devem ser tratados como um todo, pois estão intimamente relacionados. Qualquer tentativa de separá-los resultaria apenas em maior acoplamento e menor legibilidade.

Nos princípios SOLID,23 um dos conjuntos de princípios de design mais estabelecidos, a ideia é conhecida como Princípio da Responsabilidade Única (SRP):

---

20 Michael Feathers, Trabalhando efetivamente com código legado (Addison-Wesley, 2013).

21 David Thomas e Andrew Hunt, The Pragmatic Programmer: Your Journey to Mastery, edição de 20º aniversário (Addison-Wesley, 2019).

22 Tom DeMarco, Análise Estruturada e Especificação de Sistemas (Prentice Hall, 1979).

23 SOLID é uma sigla de siglas, uma abreviação dos cinco princípios descritos nas próximas diretrizes: SRP, OCP, LSP, ISP e DIP.

> Uma classe deve ter apenas um motivo para mudar.24

Embora o conceito seja antigo e comumente conhecido por diversos nomes, muitas tentativas de explicar a separação de responsabilidades levantam mais perguntas do que respostas. Isso é particularmente verdadeiro para o SRP. O próprio nome deste princípio de design já levanta questões: o que é uma responsabilidade? E o que é uma responsabilidade única? Uma tentativa comum de esclarecer a imprecisão sobre o SRP é a seguinte:

> Tudo deve fazer apenas uma coisa.

Infelizmente, essa explicação é difícil de superar em termos de imprecisão. Assim como a palavra responsabilidade não carrega muito significado, uma única coisa não ajuda a esclarecê-la.

Independentemente do nome, a ideia é sempre a mesma: agrupar apenas as coisas que realmente pertencem juntas e separar tudo o que não pertence estritamente. Ou, em outras palavras: separar as coisas que mudam por motivos diferentes. Ao fazer isso, você reduz o acoplamento artificial entre diferentes aspectos do seu código e ajuda a tornar seu software mais adaptável a mudanças. Na melhor das hipóteses, você pode alterar um aspecto específico do seu software em um só lugar.

Um Exemplo de Acoplamento Artificial

Vamos esclarecer a separação de interesses por meio de um exemplo de código. E eu tenho um ótimo exemplo: apresento a vocês a classe abstrata Document :

```
// #include <some_json_library.h> // Possível dependência física

classe Documento

{ público: // ...
    virtual
    ~Documento() = padrão;

    vazio virtual exportToJSON( /*...*/ ) const = 0; vazio virtual ❶
    serialize( ByteStream&, /*...*/ ) const = 0; // ... ❷

};
```

Esta parece ser uma classe base muito útil para todos os tipos de documentos, não é? Primeiro, temos a função exportToJSON() ❶. Todas as classes derivadas precisarão implementar a função exportToJSON() para produzir um arquivo JSON. do documento. Isso será muito útil: sem precisar saber sobre um tipo específico de

---

24 O primeiro livro sobre os princípios SOLID foi Agile Software Development: Principles, Patterns, and Practices, de Robert C. Martin (Pearson). Uma alternativa mais recente e muito mais barata é Clean Architecture, também de Robert C. Martin (Addison-Wesley).

documento (e podemos imaginar que eventualmente teremos documentos PDF, documentos do Word e muitos outros), sempre podemos exportar no formato JSON. Ótimo! Em segundo lugar, existe uma função serialize() ( ). Essa função permite transformar um Documento em bytes por meio de um ByteStream. Você pode armazenar esses bytes em algum sistema persistente, como um arquivo ou um banco de dados. E, claro, podemos esperar que existam muitas outras funções úteis disponíveis que nos permitirão usar esse documento para praticamente tudo.

No entanto, posso ver a carranca em seu rosto. Não, você não parece particularmente convencido de que este é um bom design de software. Pode ser porque você está apenas muito desconfiado deste exemplo (ele simplesmente parece bom demais para ser verdade). Ou pode ser que você tenha aprendido da maneira mais difícil que esse tipo de design eventualmente leva a problemas. Você pode ter experimentado que usar o princípio comum de design orientado a objetos para agrupar os dados e as funções que operam sobre eles pode facilmente levar a um acoplamento infeliz. E eu concordo: apesar do fato de que esta classe base parece um ótimo pacote tudo-em-um, e até parece ter tudo o que podemos precisar, este design logo levará a problemas.

Este é um design ruim porque contém muitas dependências. É claro que existem as dependências óbvias e diretas, como, por exemplo, a dependência da classe ByteStream . No entanto, este design também favorece a introdução de dependências artificiais, o que dificultará alterações subsequentes. Neste caso, existem três tipos de dependências artificiais. Duas delas são introduzidas pela função exportToJSON() e uma pela função serialize() .

Primeiro, exportToJSON() precisa ser implementado nas classes derivadas. E sim, não há escolha, pois é uma função virtual pura. (denotado pela sequência = 0, o chamado especificador puro). Como as classes derivadas provavelmente não desejarão arcar com o fardo de implementar exportações JSON manualmente, elas dependerão de uma biblioteca JSON externa de terceiros: json, rapidjson, ou simdjson. Independentemente da biblioteca escolhida para esse propósito, devido à função-membro exportToJSON() , os documentos derivados dependeriam repentinamente dessa biblioteca. E, muito provavelmente, todas as classes derivadas dependeriam da mesma biblioteca, apenas por questões de consistência. Portanto, as classes derivadas não são realmente independentes; elas são artificialmente acopladas a uma decisão de design específica.25 Além disso, a dependência de uma biblioteca JSON específica definitivamente limitaria a reutilização da hierarquia, pois ela não seria mais leve. E mudar para outra biblioteca causaria uma grande mudança, pois todas as classes derivadas teriam que ser adaptadas.26

25 Não se esqueça de que as decisões de design tomadas por essa biblioteca externa podem impactar seu próprio design, o que obviamente aumentaria o acoplamento.

26 Isso inclui as classes que outras pessoas podem ter escrito, ou seja, classes que você não controla. E não, o Outras pessoas não ficarão felizes com a mudança. Portanto, a mudança pode ser realmente difícil.

É claro que o mesmo tipo de dependência artificial é introduzido pela função serialize() . É provável que serialize() também seja implementado em termos de uma biblioteca de terceiros, como protobuf. ou Boost.serialization. Isso piora consideravelmente a situação de dependência, pois introduz um acoplamento entre dois aspectos de design ortogonais e não relacionados (por exemplo, exportação JSON e serialização). Uma alteração em um aspecto pode resultar em alterações no outro.

Na pior das hipóteses, a função exportToJSON() pode introduzir uma segunda dependência. Os argumentos esperados na chamada exportToJSON() podem refletir acidentalmente alguns detalhes de implementação da biblioteca JSON escolhida. Nesse caso, a eventual troca para outra biblioteca pode resultar em uma alteração na assinatura da função exportToJSON() , o que, consequentemente, causaria alterações em todos os chamadores.
Assim, a dependência da biblioteca JSON escolhida pode acidentalmente ser muito mais disseminada do que o pretendido.

O terceiro tipo de dependência é introduzido pela função serialize() . Devido a essa função, as classes derivadas de Document dependem de decisões globais sobre como os documentos são serializados. Qual formato usamos? Usamos little endian ou big endian? Precisamos adicionar a informação de que os bytes representam um arquivo PDF ou um arquivo do Word? Em caso afirmativo (e presumo que seja muito provável), como representamos tal documento? Por meio de um valor inteiro? Por exemplo, poderíamos usar uma enumeração para esse propósito:27

```
classe enum DocumentType
{
    pdf,

    word, // ... Potencialmente muitos outros tipos de documentos
};
```

Essa abordagem é muito comum para serialização. No entanto, se essa representação de documento de baixo nível for usada nas implementações das classes Document , acoplaríamos acidentalmente todos os diferentes tipos de documentos. Cada classe derivada saberia implicitamente sobre todos os outros tipos de Document . Como resultado, adicionar um novo tipo de documento afetaria diretamente todos os tipos de documentos existentes. Isso seria uma falha grave de projeto, pois, novamente, dificultaria a alteração.

Infelizmente, a classe Document promove muitos tipos diferentes de acoplamento. Portanto, não, a classe Document não é um ótimo exemplo de bom design de classe, pois não é fácil de alterar. Pelo contrário, é difícil de alterar e, portanto, um ótimo exemplo de violação do SRP: as classes derivadas de Document e os usuários da classe Document mudam por vários motivos, pois criamos um forte acoplamento entre várias classes.

---

27 Uma enumeração parece ser uma escolha óbvia, mas é claro que existem outras opções. No final, precisamos de um conjunto de valores acordado que represente os diferentes formatos de documento na representação em bytes.

Aspectos ortogonais e não relacionados. Resumindo, a derivação de classes e usuários de documentos pode mudar por qualquer um dos seguintes motivos:

- Os detalhes de implementação da função exportToJSON() mudam devido a um dependência direta da biblioteca JSON usada
- A assinatura da função exportToJSON() muda porque a implementação subjacente muda • A classe Document e a função

serialize() mudam devido a uma dependência direta da classe ByteStream • Os detalhes de implementação da função serialize()

mudam devido a uma
dependência direta dos detalhes de implementação

- Todos os tipos de documentos mudam devido à dependência direta do Documento Enumeração de tipos

Obviamente, esse design promove mais mudanças, e cada mudança seria mais difícil. E, claro, no caso geral, existe o perigo de que aspectos ortogonais adicionais sejam acoplados artificialmente dentro dos documentos, o que aumentaria ainda mais a complexidade de fazer uma mudança. Além disso, algumas dessas mudanças definitivamente não se restringem a um único lugar na base de código. Em particular, mudanças nos detalhes de implementação de exportToJSON() e serialize() não se restringiriam a apenas uma classe, mas provavelmente a todos os tipos de documentos (PDF, Word e assim por diante). Portanto, uma mudança afetaria um número significativo de lugares em toda a base de código, o que representa um risco de manutenção.

Acoplamento lógico versus físico O

acoplamento não se limita ao acoplamento lógico, mas também se estende ao acoplamento físico. A Figura 1-2 ilustra esse acoplamento. Vamos supor que exista uma classe Usuário no nível mais baixo da nossa arquitetura que precise usar documentos que residem em um nível mais alto da arquitetura. É claro que a classe Usuário depende diretamente da classe Documento , o que é uma dependência necessária — uma dependência intrínseca do problema em questão.
Portanto, isso não deve ser uma preocupação para nós. No entanto, a (potencial) dependência física de Document na biblioteca JSON selecionada e a dependência direta na classe Byte Stream causam uma dependência indireta e transitiva de User na biblioteca JSON e ByteStream, que residem no nível mais alto da nossa arquitetura. Na pior das hipóteses, isso significa que alterações na biblioteca JSON ou na classe ByteStream afetam User. Esperamos que seja fácil perceber que se trata de uma dependência artificial, não intencional: um User não deveria depender de JSON ou serialização.

Devo declarar explicitamente que há uma potencial dependência física do Document na biblioteca JSON selecionada. Se o arquivo de cabeçalho <Document.h> incluir qualquer cabeçalho da biblioteca JSON escolhida (conforme indicado no trecho de código no início de "Um Exemplo de Acoplamento Artificial" na página 12), por exemplo, porque a função export ToJSON() espera alguns argumentos baseados nessa biblioteca, então há uma dependência clara dessa biblioteca. No entanto, se a interface puder abstrair adequadamente esses detalhes e o cabeçalho <Document.h> não incluir nada da biblioteca JSON, a dependência física pode ser evitada. Portanto, depende de quão bem as dependências podem ser (e são) abstraídas.

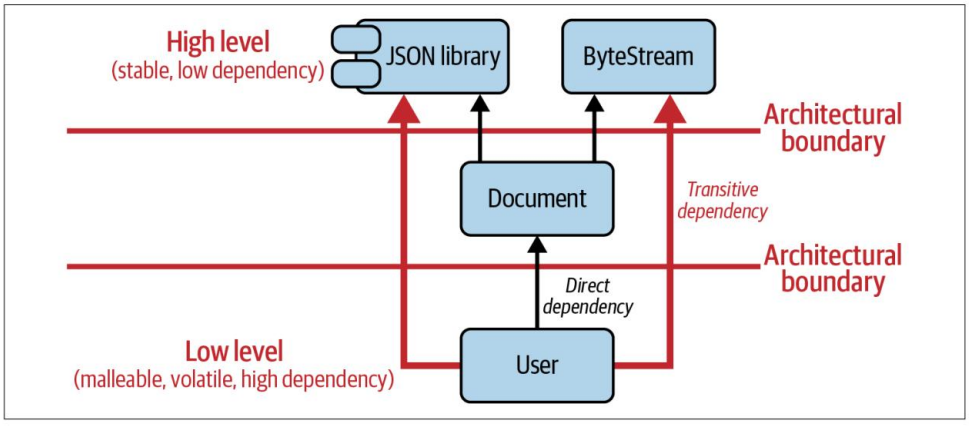

Figura 1-2. O forte acoplamento físico transitivo entre *o Usuário* e aspectos ortogonais como JSON e serialização.

"Alto nível, baixo nível — agora estou confuso", você reclama. Sim, eu sei que esses dois termos costumam causar alguma confusão. Então, antes de prosseguirmos, vamos concordar sobre a terminologia para alto nível e baixo nível. A origem desses dois termos está relacionada à maneira como desenhamos diagramas na Linguagem de Modelagem Unificada (UML): A funcionalidade que consideramos estável aparece no topo, em um nível alto. A funcionalidade que muda com mais frequência e, portanto, é considerada volátil ou maleável, aparece na parte inferior, o nível baixo. Infelizmente, quando desenhamos arquiteturas, muitas vezes tentamos mostrar como as coisas se desenvolvem umas sobre as outras, então as partes mais estáveis aparecem na parte inferior de uma arquitetura. Isso, é claro, causa alguma confusão. Independentemente de como as coisas são desenhadas, lembre-se destes termos: alto nível se refere às partes estáveis da sua arquitetura, e baixo nível se refere aos aspectos que mudam com mais frequência ou têm maior probabilidade de mudar.

Voltando ao problema: o SRP recomenda que separemos as preocupações e as coisas que não pertencem verdadeiramente, ou seja, as coisas não coesas (adesivas). Em outras palavras, recomenda-se separar as coisas que mudam por diferentes razões em variações.

pontos. A Figura 1-3 mostra a situação de acoplamento se isolarmos os aspectos JSON e de serialização em preocupações separadas.

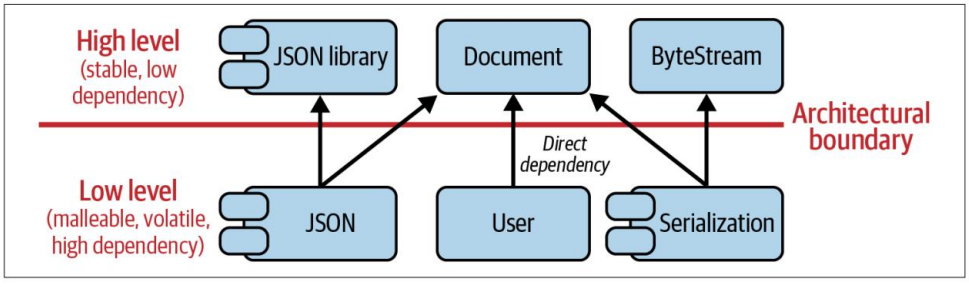

Figura 1-3. A adesão ao SRP resolve o acoplamento artificial entre *Usuário* e JSON e serialização.

Com base nesse conselho, a classe Document é refatorada da seguinte maneira:

```
classe Documento

{ público: // ... virtual ~Documento() = padrão;

    // Não há mais funções 'exportToJSON()' e 'serialize()'.
    // Somente as operações básicas do documento, que não // causam
    acoplamento forte, permanecem. // ... };
```

Os aspectos JSON e de serialização não fazem parte das funcionalidades fundamentais de uma classe Document . A classe Document deve representar apenas as operações mais básicas de diferentes tipos de documentos. Todos os aspectos ortogonais devem ser separados. Isso tornará as alterações consideravelmente mais fáceis. Por exemplo, ao isolar o aspecto JSON em um ponto de variação separado e no novo componente JSON , a troca de uma biblioteca JSON para outra afetará apenas esse componente. A alteração poderia ser feita exatamente em um único lugar e ocorreria isoladamente de todos os outros aspectos ortogonais. Também seria mais fácil oferecer suporte ao formato JSON por meio de várias bibliotecas JSON. Além disso, qualquer alteração na forma como os documentos são serializados afetaria apenas um componente no código: o novo componente Serialization . Além disso, a Serialization atuaria como um ponto de variação que permite alterações isoladas e fáceis. Essa seria a situação ideal.

Após sua decepção inicial com o exemplo do Documento , vejo que você parece mais feliz novamente. Talvez até haja um sorriso de "Eu sabia!" em seu rosto. No entanto, você ainda não está totalmente satisfeito: "Sim, concordo com a ideia geral de separar as preocupações. Mas como preciso estruturar meu software para separar as preocupações? O que preciso fazer para que funcione?" Essa é uma excelente pergunta, mas com muitas respostas que abordarei nos próximos capítulos. O primeiro e mais importante ponto, no entanto, é a identificação de um ponto de variação, ou seja, algum aspecto em seu código onde mudanças são esperadas. Esses pontos de variação devem ser extraídos, isolados e encapsulados, de modo que não haja mais dependências nessas variações. Isso, em última análise, ajudará a facilitar as mudanças.

“Mas isso ainda é apenas um conselho superficial!”, ouço você dizer. E você está certo. Infelizmente, não há uma resposta única e não há uma resposta simples. Depende. Mas prometo dar muitas respostas concretas sobre como separar interesses nos próximos capítulos. Afinal, este é um livro sobre design de software, ou seja, um livro sobre gerenciamento de dependências. Como um pequeno teaser, no Capítulo 3 apresentarei uma abordagem geral e prática para este problema: padrões de design. Com essa ideia geral em mente, mostrarei como separar interesses usando diferentes padrões de design. Por exemplo, os padrões de design Visitor, Strategy e External Polymorphism me vêm à mente. Todos esses padrões têm diferentes pontos fortes e fracos, mas compartilham a propriedade de introduzir algum tipo de abstração para ajudar a reduzir dependências. Além disso, prometo analisar detalhadamente como implementar esses padrões de design em C++ moderno.

Apresentarei o padrão de design Visitor na “Diretriz 16: Use Visitor para estender operações” na página 112, e o padrão de design Strategy na “Diretriz 19: Use Strategy para isolar como as coisas são feitas” na página 140. O padrão de design External Polymorphism será o tópico da “Diretriz 31: Use External Polymorphism para Nonintrusive Runtime Polymorphism” na página 279.

Não se repita. Há um segundo

aspecto importante na mutabilidade. Para explicá-lo, apresentarei outro exemplo: uma hierarquia de itens. A Figura 1-4 dá uma ideia dessa hierarquia.

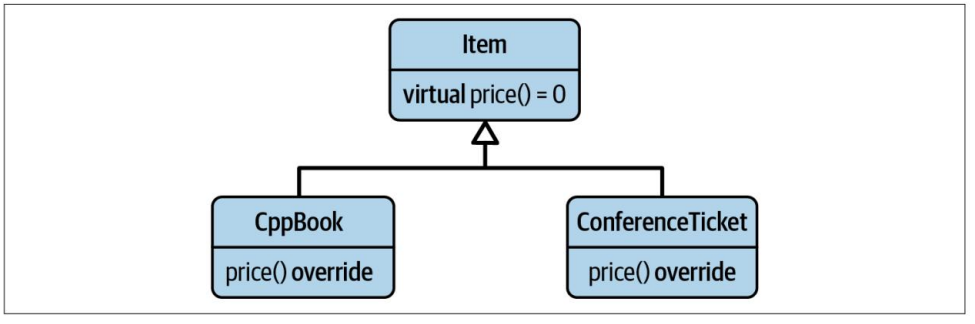

Figura 1-4. A hierarquia de classes *Item* .

No topo dessa hierarquia está a classe base Item :

```
//---- <Dinheiro.h> ----------

classe Dinheiro { /*...*/ };

Operador monetário *( Moeda dinheiro, fator duplo );
Operador monetário +( Moeda lhs, Dinheiro rhs );


//---- <Item.h> ----------

#incluir <Dinheiro.h>

classe Item

{ público:
    virtual ~Item() = padrão; dinheiro
    virtual preço() const = 0;
};
```

A classe base Item representa uma abstração para qualquer tipo de item que tenha um preço (representado pela classe Money). Por meio da função price() , você pode consultar esse preço. É claro que existem muitos itens possíveis, mas, para fins ilustrativos, nos restringiremos a CppBook e ConferenceTicket:

```
//---- <CppBook.h> ----------

#incluir <Item.h> #incluir
<Dinheiro.h> #incluir
<string>

classe CppBook : item público

{ público:
    CppBook explícito ( std::string título, std::string autor, preço em dinheiro ) :                ❸
        título_( std::mover(título) ) ,
        autor_( std::mover(autor) )
```

```
        , priceWithTax_( preço * 1,15 ) // taxa de imposto de 15%
    {}

    std::string const& título() const { retornar título_; } std::string const& autor()
    const { retornar autor_; }

    Preço monetário() const override { return priceWithTax_; }

  privado:
    std::string título_; std::string
    autor_;
    Preço do dinheiroComImposto_;
};
```

O construtor da classe CppBook espera um título e um autor na forma de strings e um preço na forma de Money ( ).28 Além disso, ele só permite acessar o título, o autor e o preço com as funções title(), author() e price() (e). No entanto, a função price() é um pouco especial: obviamente, livros estão sujeitos a impostos. Portanto, o preço original do livro precisa ser adaptado de acordo com uma determinada alíquota de imposto. Neste exemplo, assumo uma alíquota de imposto imaginária de 15%.

A classe ConferenceTicket é o segundo exemplo de um Item:

```
//---- <ConferenceTicket.h> ----------

#incluir <Item.h> #incluir
<Dinheiro.h> #incluir
<string>

class ConferenceTicket : public Item { public:

explicit
    ConferenceTicket( std::string name, Money price ) : name_( std::move(name) ) ,
        priceWithTax_( price * 1.15 ) // taxa
        de imposto de 15%
    {}

    std::string const& nome() const { return nome_; }

    Preço monetário() const override { return priceWithTax_; }

  privado:
    std::string nome_;
```

28 Você pode estar se perguntando sobre o uso explícito da palavra-chave explicit para este construtor. Então você também pode estar ciente de que a Diretriz Básica C.46 recomenda usar explicit por padrão para construtores de argumento único. Este é um conselho muito bom e altamente recomendado, pois previne conversões não intencionais e potencialmente indesejadas. Embora não seja tão valioso, o mesmo conselho também é válido para todos os outros construtores, exceto para os construtores de cópia e movimentação, que não realizam conversões. Pelo menos não faz mal.

```
    Preço do dinheiroComImposto_;
};
```

ConferenceTicket é muito semelhante à classe CppBook , mas espera apenas o nome da conferência e o preço no construtor ( ). É claro que você pode acessar o nome e o preço com as funções name() e price() , respectivamente. Mais importante ainda, o preço de uma conferência em C++ também está sujeito a impostos. Portanto, adaptamos novamente o preço original de acordo com a alíquota imaginária de 15%.

Com essa funcionalidade disponível, podemos prosseguir e criar alguns itens na função main() :

```
#include <CppBook.h>
#include <ConferenceTicket.h> #include
<algoritmo> #include <cstdlib>
#include < memória>
#include <vetor>


int principal()
{
    std::vector<std::unique_ptr<Item>> itens{};


    items.emplace_back( std::make_unique<CppBook>(" C++ eficaz", "Meyers",
    19,99) );
        items.emplace_back( std::make_unique<CppBook>(" Modelos C++", "Josuttis", 49,99) );

    items.emplace_back( std::make_unique<ConferenceTicket>("CppCon", 999.0) );
    items.emplace_back( std::make_unique<ConferenceTicket>("Reunião C++", 699.0) );
    items.emplace_back( std::make_unique<ConferenceTicket>("C++ no Mar", 499.0) );

    Dinheiro const preço_total =
        std::accumulate( begin(itens), end(itens), Dinheiro{},
            []( Dinheiro acumula, auto const& item )
                { return accu + item->price(); } );


    // ...

    retornar EXIT_SUCCESS;
}
```

Em main(), criamos alguns itens (dois livros e três conferências) e calculamos o preço total de todos os itens.29 O preço total, é claro, incluirá a taxa de imposto imaginária de 15%.

Parece um bom design. Separamos os tipos específicos de itens e conseguimos alterar a forma como o preço de cada item é calculado isoladamente. Parece que cumprimos o SRP e extraímos e isolamos os pontos de variação. E, claro, há mais itens. Muitos mais. E todos eles garantirão que a alíquota de imposto aplicável seja devidamente considerada. Ótimo! Embora essa hierarquia de itens nos deixe felizes por algum tempo, o design infelizmente tem uma falha significativa. Podemos não perceber hoje, mas sempre há uma sombra pairando à distância, a nêmesis dos problemas em software: a mudança.

O que acontece se, por algum motivo, a alíquota do imposto mudar? E se a alíquota de 15% for reduzida para 12%? Ou aumentada para 16%? Ainda ouço os argumentos do dia em que o design inicial foi incluído na base de código: "Não, isso nunca vai acontecer!" Bem, até mesmo as coisas mais inesperadas podem acontecer. Por exemplo, na Alemanha, a alíquota do imposto foi reduzida de 19% para 16% por meio semestre em 2021. Isso, é claro, significaria que teríamos que alterar a alíquota do imposto em nossa base de código. Onde aplicamos a alteração?
Na situação atual, a mudança afetaria praticamente todas as classes derivadas da classe Item . A mudança se espalharia por toda a base de código!

Assim como o SRP recomenda separar os pontos de variação, devemos tomar cuidado para não duplicar informações em toda a base de código. Assim como tudo deve ter uma única responsabilidade (um único motivo para a alteração), cada responsabilidade deve existir apenas uma vez no sistema. Essa ideia é comumente chamada de princípio Don't Repeat Yourself (DRY). Este princípio nos aconselha a não duplicar algumas informações importantes em muitos lugares, mas sim a projetar o sistema de forma que possamos fazer a alteração em apenas um lugar. No caso ideal, a(s) alíquota(s) de imposto deve(m) ser representada(s) em exatamente um lugar para permitir que você faça uma alteração facilmente.

Normalmente, os princípios do SRP e do DRY funcionam muito bem juntos. Aderir ao SRP frequentemente leva à adesão ao DRY também, e vice-versa. No entanto, às vezes, aderir a ambos exige algumas etapas extras. Sei que você está ansioso para aprender quais são essas etapas extras e como resolver o problema, mas, neste ponto, basta destacar a ideia geral do SRP e do DRY. Prometo revisitar este problema e mostrar como resolvê-lo (consulte “Diretriz 35: Use Decoradores para Adicionar Personalização Hierarquicamente” na página 348).

29 Você pode perceber que escolhi os nomes das três conferências das quais participo regularmente: CppCon, Conhecendo C++, e C++ no mar. No entanto, existem muitas outras conferências sobre C++. Para dar alguns exemplos: ACCU, Núcleo C++, pacífico++, CppNorte, emBO++, e CPPP. Conferências são uma ótima e divertida maneira de se manter atualizado com C++. Não deixe de conferir a página inicial do Standard C++ Foundation para quaisquer conferências futuras.

Evite a Separação Prematura de

Responsabilidades. Neste ponto, espero tê-lo convencido de que aderir ao SRP e ao DRY é uma ideia bastante razoável. Você pode até estar tão comprometido que planeja separar tudo — todas as classes e funções — nas menores unidades de funcionalidade. Afinal, esse é o objetivo, certo? Se é isso que você está pensando agora, por favor, pare! Respire fundo. E mais uma vez. E então, por favor, ouça atentamente a sabedoria de Katerina Trajchevska:30

> Não tente alcançar o SOLID, use o SOLID para alcançar a manutenibilidade.

Tanto o SRP quanto o DRY são suas ferramentas para alcançar melhor manutenibilidade e simplificar mudanças. Eles não são seus objetivos. Embora ambos sejam de extrema importância a longo prazo, pode ser muito contraproducente separar entidades sem uma ideia clara sobre que tipo de mudança afetará você. Projetar para mudanças geralmente favorece um tipo específico de mudança, mas infelizmente pode dificultar outros tipos de mudança. Essa filosofia faz parte do conhecido princípio YAGNI. (You Aren't Gonna Need It), que alerta sobre excesso de engenharia (veja também “Diretriz 5: Design para Extensão” na página 35). Se você tem um plano claro, se sabe que tipo de mudança esperar, aplique SRP e DRY para simplificar esse tipo de mudança. No entanto, se você não sabe que tipo de mudança esperar, não tente adivinhar — apenas espere. Espere até ter uma ideia clara sobre que tipo de mudança esperar e, em seguida, refatore para tornar a mudança o mais fácil possível.

Só não se esqueça de que um aspecto da facilidade de mudar as coisas é ter testes de unidade em vigor que lhe dão a confirmação de que a mudança não quebrou o comportamento esperado.

Em resumo, mudanças são esperadas em softwares e, portanto, é vital projetar para mudanças. Separe as preocupações e minimize a duplicação para que você possa alterar as coisas facilmente sem medo de quebrar outros aspectos ortogonais.

30 Katerina Trajchevska, “Tornando-se um desenvolvedor melhor usando os princípios de design SOLID”, Laracon UE, 30 e 31 de agosto de 2018.

Diretriz 2: Design para Mudança

- Espere mudanças no software.
- Projete para mudanças fáceis e torne o software mais adaptável.
- Evite combinar aspectos ortogonais não relacionados para evitar o acoplamento.
- Entenda que o acoplamento aumenta a probabilidade de mudança e torna as mudanças mais difícil.
- Siga o Princípio da Responsabilidade Única (SRP) para separar as preocupações.
- Siga o princípio de Não se Repetir (DRY) para minimizar a duplicação.
- Evite abstrações prematuras se não tiver certeza sobre a próxima mudança.

Diretriz 3: Interfaces separadas a serem evitadas Acoplamento Artificial

Vamos revisitar o exemplo de Documento da “Diretriz 2: Design para Mudanças” na página 11. Eu sei, você provavelmente já acha que já viu documentos suficientes, mas acredite, ainda não terminamos. Ainda há um aspecto importante de acoplamento a ser abordado. Desta vez, não nos concentraremos nas funções individuais da classe Documento , mas na interface como um todo:

```
classe Documento

{ público: // ...
    virtual
    ~Documento() = padrão;

    vazio virtual exportToJSON( /*...*/ ) const = 0; vazio virtual
    serialize( ByteStream& bs, /*...*/ ) const = 0; // ...

};
```

Segregue Interfaces para Separar Preocupações: O

Documento requer classes derivadas para lidar tanto com exportações JSON quanto com serialização. Embora, do ponto de vista de um documento, isso possa parecer razoável (afinal, todos os documentos devem ser exportáveis para JSON e serializáveis), infelizmente causa outro tipo de acoplamento. Imagine o seguinte código de usuário:

```
void exportDocument( Documento const& doc ) {

    // ...
    doc.exportToJSON( /* passe os argumentos necessários */ );
```