5. Projeto de Software na Construção

Algumas pessoas poderiam argumentar que o projeto de software (design) não é realmente uma atividade da construção, mas em projetos pequenos muitas atividades são consideradas construção, frequentemente incluindo o projeto de software. Em determinados projetos, de maior porte, uma arquitetura formal poderia tratar apenas dos problemas em nível de sistema e grande parte do trabalho de projeto poderia ser intencionalmente deslocada para a construção. Em outros projetos grandes, poderíamos planejar o projeto de software para ser detalhado o bastante, de forma a tornar a codificação completamente mecânica, mas o projeto raramente é assim, tão completo – normalmente, o programador prepara o projeto de parte do programa, oficialmente ou não.

Em projetos pequenos e informais, grande parte do projeto de software é realizada enquanto o programador senta-se diante do teclado. Design pode significar apenas escrever uma interface de classe em pseudocódigo, antes de escrever os detalhes, pode ser também o desenho de diagramas de alguns relacionamentos entre classes, antes de sua codificação. Também pode ser perguntar a outro programador qual padrão de projeto parece ser a melhor escolha. Independentemente de como ele é feito, tanto pequenos quanto grandes projetos se beneficiam de um design cuidadoso, e reconhecê-lo como uma atividade explícita maximiza a vantagem que você obterá dele.

5.1 Desafios do design

A frase “projeto de software” significa a concepção, invenção ou criação de um esquema para transformar uma especificação em software operacional. Projeto de software é a atividade que o leva dos requisitos para a codificação e a depuração. Um bom projeto de alto nível fornece uma estrutura que pode seguramente conter vários projetos mais detalhados. Um projeto de software de boa qualidade é útil em projetos pequenos e indispensável nos de grande porte.

O projeto de software também é marcado por numerosos desafios:

Projeto de software é um problema perverso

Um ‘problema perverso’ pode ser definido como aquele que poderia ser claramente definido apenas por intermédio de sua solução ou pela solução de parte dele. Esse paradoxo sugere, basicamente, que você precisa “resolver” o problema uma vez para defini-lo claramente e depois resolvê-lo mais uma vez, para criar uma solução que funcione.

Um exemplo dramático de tal problema perverso foi o projeto original da ponte Tacoma Narrows. Na época em que foi construída, o principal aspecto no projeto de uma ponte era que ela fosse forte o bastante para suportar sua carga prevista. No caso da ponte Tacoma Narrows, o vento gerou uma ondulação harmônica transversal inesperada. Durante uma tempestade, no ano de 1940, a ondulação aumentou incontrolavelmente até que a ponte ruiu.

Esse é um exemplo claro de problema perverso, pois, até que a ponte ruísse, seus engenheiros não sabiam que a aerodinâmica precisava ser considerada até esse ponto. Somente construindo a ponte (resolvendo o problema) eles puderam aprender a respeito desse aspecto adicional no problema, o que fez com que construíssem outra ponte, que continua resistindo.

Uma das principais diferenças entre os programas que você desenvolve na universidade e aqueles que desenvolve como profissional é que os problemas de projeto resolvidos pelos programas universitários raramente são perversos, se é que chegam

a sê-lo em algum caso. As tarefas de programação na faculdade destinam-se a fazer com que você se mova em uma linha reta do início ao fim.

O projeto de software é um processo desordenado (mesmo que produza um resultado ordenado)

O projeto de Software já concluído deve estar bem-organizado e ordenado, mas o processo usado para desenvolver o projeto não é nem de longe tão ordenado quanto o resultado final.

O projeto de Software é desordenado porque você comete vários erros. Na verdade, cometer erros é o objetivo da atividade de projeto de software - é mais barato cometer erros e corrigir os projetos do que cometer os mesmos erros, reconhece-los após a codificação e ter de corrigir o código completo. O projeto de software é desordenado também porque, frequentemente, a diferença entre uma boa solução e uma ruim é sutil.

O projeto de software é desordenado ainda porque é difícil saber quando seu projeto é “bom o bastante”. Que nível de detalhe é suficiente? Quanto projeto deve ser desenvolvido com uma notação formal e quanto deve ser deixado para ser feito no teclado? Quando você sabe que terminou? Como o projeto de software é aberto, a resposta mais comum para essa pergunta costuma ser “quando você não tiver mais tempo”.

O projeto de software envolve equilíbrio e prioridades

Em um mundo ideal, todo sistema poderia funcionar instantaneamente, não consumir nada de espaço de armazenamento, não usar nenhuma largura de banda de rede, nunca apresentar erros e não custar nada para ser construído. No mundo real, uma parte importante do trabalho do design é ponderar as características concorrentes do projeto do software e equilibrá-las. Se uma alta velocidade de resposta for mais importante do que minimizar o tempo de desenvolvimento, o designer escolherá um único projeto para o software. Se minimizar o tempo de desenvolvimento for mais importante, um bom designer fará um projeto diferente.

O projeto de software envolve restrições

Em parte, o objetivo do projeto de software é criar possibilidades. Em parte é restringir as possibilidades. Se as pessoas tivessem tempo, recursos e espaço infinitos para construir estruturas físicas, você veria prédios esparramando-se de maneira incrível. É dessa maneira que o software pode se apresentar quando não há restrições deliberadamente impostas. As restrições de recursos para a construção de prédios impõem simplificações da solução que, em última análise, aprimoram essa solução. O objetivo no projeto de software é o mesmo.

O projeto de software não é determinístico

Se você mandar três pessoas projetarem o mesmo programa, elas poderão facilmente apresentar três projetos totalmente diferentes, cada um deles podendo ser perfeitamente aceitável. Pode até haver mais de uma maneira de escalpelar um gato, mas normalmente existem **dezenas de maneiras de fazer o projeto de um programa** de computador.

O projeto de software é um processo heurístico

Como o projeto de software não é determinístico, as técnicas de projeto tendem a ser heurísticas – “maneiras empíricas de proceder” ou “procedimentos a serem experimentados e que às vezes funcionam” –, em vez de serem processos repetidos, que com certeza produzem resultados previsíveis. O projeto de software envolve “tentativa e erro”. Uma ferramenta ou técnica de projeto de software que tenha funcionado bem

para uma tarefa, ou para um aspecto de uma tarefa, pode não funcionar bem no projeto seguinte. Nenhuma ferramenta se adapta a todas as circunstâncias.

O projeto de software possui comportamento emergente

Uma forma apropriada de resumir esses atributos de projeto de software é afirmar que seu comportamento, enquanto atividade, é “emergente”. Os projetos de software não surgem prontos diretamente do cérebro de alguém. Eles evoluem e melhoram por meio de revisões, discussões informais, experiência de escrita do código em si e experiência de revisão do código.

Praticamente todos os sistemas passam por algum grau de alterações no seu projeto durante seu desenvolvimento inicial; mais tarde, normalmente passam por alterações maiores, quando são ampliados em versões posteriores. O grau no qual uma alteração é benéfica ou aceitável depende da natureza do software que está sendo construído.

5.2 Principais conceitos do projeto de software

Um bom projeto de software depende da assimilação de diversos conceitos importantes. Esta seção trata sobre o papel da complexidade, das características desejáveis dos projetos de software e dos níveis de projeto.

Controle da Complexidade

Autores argumentam que o desenvolvimento de software se torna difícil por causa de duas diferentes classes de problemas - **a essencial e a acidental**. Ao se referir a esses dois termos. Brooks recorre a uma tradição filosófica que remonta à época de Aristóteles. Na filosofia, as propriedades essenciais são aquelas que alguma coisa deve ter para ser o que é. Um carro deve ter um motor, rodas e portas para ser um carro. Se ele não tiver nenhuma dessas propriedades essenciais, não será realmente um carro.

Já propriedades acidentais são aquelas que alguma coisa pode ter, propriedades que não se relacionam com o fato de ser o que é. Um carro poderia ter um motor V8, 4 cilindros turbo ou algum outro tipo de motor e ainda assim ser um carro, independentemente desse detalhe. Um carro poderia ter duas ou quatro portas; poderia ter rodas comuns ou de magnésio. Todos esses detalhes são propriedades acidentais. Você poderia conceber propriedades acidentais também como incidentais, arbitrárias, opcionais e fortuitas.

As principais dificuldades acidentais no software foram tratadas há muito tempo. Por exemplo, as dificuldades acidentais relacionadas às sintaxes de linguagem mal-definidas foram eliminadas em grande parte na evolução da linguagem assembly para as linguagens de terceira geração e, desde então, tiveram sua importância cada vez mais reduzida. As dificuldades acidentais relacionadas aos computadores não-interativos foram resolvidas quando os sistemas operacionais de compartilhamento de tempo substituíram os sistemas que operavam em lotes. Os ambientes de programação integrados eliminaram ainda mais as ineficiências no trabalho de programação provenientes de ferramentas que funcionavam mal juntas.

O avanço nas dificuldades essenciais remanescentes do software está fadado a ser mais lento. O motivo é que, em sua essência, o desenvolvimento de software consiste em elaborar todos os detalhes de um conjunto de conceitos interligados e altamente complexos. As dificuldades essenciais surgem da necessidade da ligação com o complexo e desordenado mundo real; da identificação precisa e completa das dependências e exceções; do projeto de soluções que não podem ser apenas praticamente corretas, mas sim totalmente corretas: e assim por diante. Mesmo que pudéssemos inventar uma linguagem de programação que usasse a mesma terminologia que o problema do mundo real

que estamos tentando resolver, a programação ainda seria difícil, devido ao desafio apresentado em se determinar precisamente como o mundo real funciona. À medida que o software trata de problemas do mundo real constantemente maiores, as interações entre as entidades do mundo real se tomam cada vez mais complicadas e isso, por sua vez, aumenta a dificuldade essencial das soluções de software.

A raiz de todas essas dificuldades essenciais é a complexidade - tanto acidental como essencial.

A importância do controle da complexidade

Quando os levantamentos de projetos de desenvolvimento de software apuram as causas de uma falha, eles raramente identificam motivos técnicos como causas principais. Mais frequentemente, os projetos falham devido a requisitos mal elaborados, planejamento malfeito ou gerenciamento deficiente. Mas quando os projetos falham por motivos preponderantemente técnicos, quase sempre a razão é uma complexidade não-controlada.

O software pode ficar tão complexo que ninguém saberá realmente o que ele faz. Quando um projeto chega ao ponto de ninguém entender completamente o impacto que as alterações de código em uma área terão em outras áreas, deixa de haver avanço no trabalho.

Controlar a complexidade é o assunto técnico mais importante no desenvolvimento de software.

A complexidade não é uma característica nova do desenvolvimento de software. Edsger Dijkstra, pioneiro da computação, sugeriu que a computação é a única profissão na qual uma única cabeça é obrigada a transpor a distância de um bit a algumas centenas de megabytes, uma relação de 1 para 10^9, ou nove ordens de magnitude (Dijkstra 1989). Essa relação gigantesca é desconcertante. Dijkstra expressa isso da seguinte maneira: “Comparada a esse número de níveis semânticos, a teoria da matemática normal é quase simplória. Evocando a necessidade de hierarquias conceituais profundas, o computador automático nos confronta com um desafio intelectual radicalmente novo, sem precedentes em nossa história”. É claro que o software se tornou ainda mais complexo e a relação de 1 para 10^9 poderia facilmente ser de 1 para 10^{15} atualmente.

Dijkstra afirma que ninguém tem o cérebro grande o bastante para conter um programa de computador moderno, o que significa que nós, como desenvolvedores de software, não devemos tentar empanturrar nossos cérebros com programas inteiros de uma só vez, devemos tentar organizar nossos programas de tal maneira que possamos concentrar-nos com segurança em uma parte dele por vez. O objetivo é minimizar a quantidade de um programa sobre a qual você precisa pensar em dado momento. Você poderia considerar isso como um malabarismo mental - quanto mais bolas mentais o programa exigir que você mantenha no ar simultaneamente, mais chance haverá de deixar uma das bolas cair, levando a um erro de projeto ou de codificação.

Em nível de arquitetura de software, a complexidade de um problema é reduzida dividindo se o sistema em subsistemas. Os seres humanos têm mais facilidade em compreender várias informações simples do que uma única informação complexa. O objetivo de todas as técnicas de projeto de software é dividir um problema complicado em partes simples. Quanto mais independentes são os subsistemas, mais você torna seguro enfocar um pouquinho de complexidade por vez. Objetos cuidadosamente definidos separam as responsabilidades para que você possa focalizar uma coisa por vez. Os pacotes oferecem a mesma vantagem, em um nível mais alto de agregação.

Manter as rotinas curtas ajuda a reduzir sua carga de trabalho mental. Escrever programas em termos do domínio do problema, em vez de escrevê-lo em termos dos detalhes da implementação de baixo nível, assim como trabalhar em níveis mais altos de abstração, reduz a carga sobre seu cérebro.

O resultado é que os programadores que neutralizam as limitações intrinsecamente humanas escrevem um código mais fácil para eles mesmos e para os outros entenderem, contendo menos erros.

Como atacar a complexidade

Projetos de software demasiadamente dispendiosos e ineficazes surgem de três fontes:

- Uma solução complexa para um problema simples
- Uma solução simples e incorreta para um problema complexo
- Uma solução inadequada e complexa para um problema complexo

Conforme Dijkstra sugeriu, o software moderno é inerentemente complexo e não importa o quanto você tente, finalmente acabará encontrando algum nível de complexidade inerente ao problema do mundo real em si. Isso sugere uma estratégia de duas ramificações para o controle da complexidade:

- Minimizar a quantidade de complexidade essencial com que o cérebro de alguém tem de lidar em dado momento.
- Impedir que a complexidade acidental prolifere desnecessariamente.

Tão logo você entenda que todos os outros objetivos técnicos no software são secundários relativamente ao controle da complexidade, muitas considerações de projeto se tornam simples.

Características desejáveis em um projeto de software

Um projeto de software de alta qualidade tem várias características gerais. Se você pudesse atingir todos esses objetivos, seu projeto seria realmente muito bom. Alguns objetivos contradizem outros, mas esse é o desafio do projeto de software - criar um bom equilíbrio entre objetivos antagônicos. Algumas características da qualidade do projeto também são características de um bom programa: confiabilidade, desempenho e coisas do gênero outras, são características internas do projeto de software. Eis aqui uma lista das características internas do projeto de software:

Complexidade mínima - O principal objetivo do projeto de software deve ser minimizar a complexidade por todos os motivos que acabamos de descrever. Evite fazer projetos muito “engenhosos”. Projetos de software engenhosos normalmente são difíceis de entender. Em vez disso, crie projetos “simples” e “fáceis de entender”. Se o seu projeto não permite que você ignore com segurança a maioria das outras partes do programa, quando está imerso em uma parte específica, então ele não está fazendo seu papel.

Facilidade de manutenção - Significa projetar o software com o programador de manutenção em mente. Imagine as perguntas que um programador de manutenção faria sobre o código que você está escrevendo. Considere o programador de manutenção como seu público e, então, faça o projeto do sistema ser autoexplicativo.

Baixo acoplamento - Significa projetar o software de modo a minimizar as conexões entre as diferentes partes de um programa. Use os princípios das boas abstrações nas interfaces de classe, no encapsulamento e no ocultamento de informações, para criar um projeto de classes com o mínimo de interconexões possível. Um acoplamento mínimo reduz o trabalho durante a integração, os testes e a manutenção.

Extensibilidade - Significa que você pode aprimorar um sistema sem violentar a sua estrutura básica. Você pode alterar uma parte de um sistema sem afetar outras partes. As alterações mais prováveis causam o mínimo trauma no sistema.

Reutilização - Projetar o sistema de modo que você possa reutilizar partes dele em outros sistemas.

Portabilidade - Significa projetar o sistema de modo que você possa movê-lo (portá-lo) facilmente para outro ambiente.

Objetividade - Significa criar o projeto do sistema de modo que ele não tenha partes extras. Voltaire considerava que um livro estava terminado não quando nada mais havia para ser acrescentado, mas quando nada mais havia para ser tirado. No software, isso é especialmente verdadeiro, porque código extra precisa ser desenvolvido, examinado, testado e avaliado quando outro código é modificado. As versões futuras do software devem permanecer compatíveis com o código extra. A pergunta fatal é: "É fácil; portanto, que mal faremos em colocar isso?"

Estratificação - Significa tentar manter os níveis de decomposição estratificados (organizados em camadas), para que você possa visualizar o sistema em qualquer nível de forma consistente. Projete o sistema de modo que você possa vê-lo em um nível sem precisar examinar outros níveis.

Por exemplo, se você estiver escrevendo um sistema moderno que precisa usar muito código antigo e mal-projetado, desenvolva uma camada do novo sistema que seja responsável por fazer a interface com o código antigo. Projete essa camada de modo que ela oculte a má qualidade do código antigo, apresentando um conjunto de serviços consistente para as camadas mais recentes. Depois, faça o restante do sistema usar essas classes, em vez do código antigo. Os efeitos benéficos do projeto estratificado, em tal caso. são: (1) ele separa a desordem do código malfeito em compartimentos isolados e (2) se você puder jogar fora o código antigo ou refazê-lo, não precisará modificar nenhum código novo, exceto a camada da interface.

Técnicas-padrão - Quanto mais um sistema contar com partes exóticas, mais amedrontador ele será para alguém que esteja pela primeira vez tentando entendê-lo. Tente dar ao sistema inteiro uma impressão simples, usando estratégias comuns e padronizadas.

Níveis de projeto

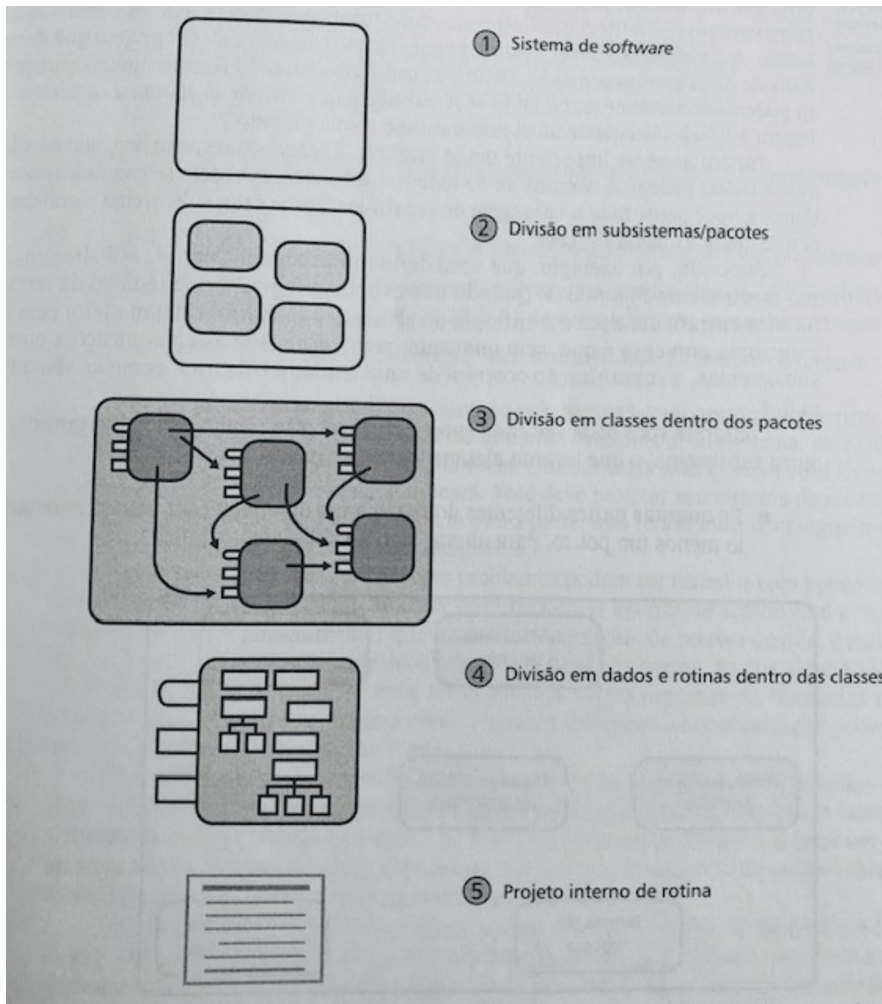
O projeto de software é necessário em vários níveis de detalhe no desenvolvimento de um sistema. Algumas técnicas de projeto se aplicam a todos os níveis e algumas se aplicam somente a um ou dois. A Figura mostra os níveis de projeto em um software. O sistema (1) é primeiro, organizado em subsistemas (2). Os subsistemas são subdivididos em classes (3) e estas são divididas em rotinas e dados (4). O projeto interno de cada rotina também é feito (5).

Nível 1: Sistema de software

Em outras palavras o primeiro nível é o sistema inteiro. Alguns programadores pulam diretamente do nível de sistema para o projeto das classes, mas normalmente é vantajoso cogitar a respeito de combinações de mais alto nível entre as classes, como os subsistemas ou pacotes

Nível 2: Divisão em subsistemas ou pacotes

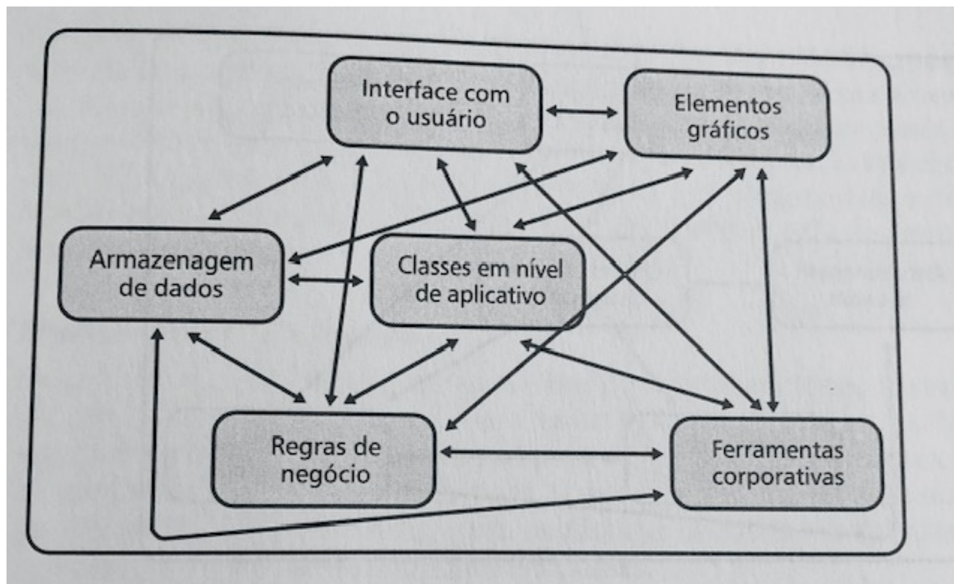
O principal produto do projeto de software neste nível é a identificação de todos os subsistemas importantes. Os subsistemas podem ser grandes: banco de dados, interface com o usuário, regras de negócio, interpretador de comandos, mecanismo de relatório e etc. A principal atividade de projeto neste nível é decidir como se vai desmembrar o software em subsistemas importantes e definir como cada subsistema poderá usar os demais. A divisão neste nível é normalmente necessária em qualquer projeto que demore mais do que algumas semanas. Dentro de cada subsistema, diferentes métodos de projeto podem ser usados - escolhendo-se a estratégia que melhor se adapta a cada caso.



Particularmente importante nesse nível são as regras a respeito dc como os vários subsistemas podem se comunicar. Se todos os subsistemas podem se comunicar com os demais, você perde toda a vantagem de separá-los. Torne cada subsistema significativo, restringindo as comunicações.

Suponha, por exemplo, que você defina um sistema com seis subsistemas. Quando não existirem regras, a entropia do sistema aumentará. Uma maneira pela qual a entropia aumenta é que, sem quaisquer restrições sobre as comunicações entre os subsistemas, a comunicação ocorrerá de uma maneira irrestrita, como se vê na Figura a seguir. Conforme pode ver, cada subsistema acaba se comunicando diretamente com outro subsistema, o que levanta algumas questões importantes:

- De quantas partes diferentes do sistema um desenvolvedor precisa entender, pelo menos um pouco, para alterar algo no subsistema gráfico?
- O que acontece quando você tenta usar as regras de negócio em outro sistema?
- O que acontece quando você quer colocar uma nova interface com o usuário no sistema, talvez uma interface de linha de comando para propósitos de teste?
- O que acontece quando você quer usar armazenamento remoto de dados?



Você poderia pensar nas linhas entre os subsistemas como sendo mangueiras com água correndo por elas. Se você quiser meter a mão e retirar um subsistema, esse subsistema terá algumas mangueiras ligadas a ele. Quanto mais mangueiras você tiver que desconectar e religar, mais molhado ficará. Você deve projetar seu sistema de modo que, se retirar um subsistema para usá-lo em outra parte, não tenha muitas mangueiras para religar e que elas sejam religadas facilmente.

Havendo antecipação, todos esses problemas podem ser tratados com pouco trabalho extra. Permita a comunicação entre os subsistemas apenas de acordo com a "necessidade de saber" - e seria melhor que fosse uma razão. Se houver dúvida, é mais fácil restringir a comunicação antecipadamente e abrandá-la depois, do que abrandá-la antes e depois tentar estancá-la, após ter codificado várias centenas de chamadas entre subsistemas.

Para conservar as conexões fáceis de entender e manter, peque pelo lado das relações simples entre os subsistemas. O relacionamento mais simples é fazer um subsistema chamar rotinas em outro. Um relacionamento mais complexo é fazer um subsistema conter classes de outro. O relacionamento mais complicado de todos é fazer as classes de um subsistema herdarem das classes de outro.

Uma regra geral é a de que um diagrama em nível de sistema, deve ser um grafo acíclico. Em outras palavras, um programa não deve conter quaisquer relacionamentos circulares nos quais a Classe **A** usa a Classe **B**, a Classe **B** usa a Classe **C** e a Classe **C** usa a Classe **A**.

Nível 3: Divisão em classes

Neste nível, o projeto de software inclui identificar todas as classes do sistema. Por exemplo, um subsistema de interface com o banco de dados poderia ser subdividido em classes de acesso a dados e classes de estrutura de persistência, assim como em metadados do banco de dados.

Os detalhes das maneiras pelas quais cada classe interage com o restante do sistema também são especificados à medida que as classes são especificadas. Em particular, a interface da classe está definida. De um modo geral, a principal atividade de projeto nesse nível é certificar-se de que todos os subsistemas tenham sido decompostos em um nível de detalhe suficiente para que você possa implementar suas partes como classes individuais.

A divisão de subsistemas em classes normalmente é necessária em qualquer projeto que demore mais do que alguns dias. Se o projeto é grande, a divisão é claramente

distinta daquela do programa que aparece no Nível 2. Se o projeto é muito pequeno, você pode mudar diretamente da visão do sistema inteiro do Nível 1 para a visão de classes do Nível 3.

Nível 4: Divisão em rotinas

Neste nível, o projeto de software inclui a divisão de cada classe em rotinas. A interface de classe definida no Nível 3 definirá algumas dessas rotinas. No Nível 4, o projeto detalhará as rotinas privadas da classe. Quando você examina os detalhes das rotinas dentro de uma classe, pode ver que muitas rotinas são simples caixas, mas algumas são compostas de rotinas organizadas hierarquicamente, as quais exigem ainda mais detalhamento do projeto.

O ato de definir totalmente as rotinas da classe em geral resulta em um entendimento melhor da interface da classe e isso provoca alterações correspondentes na interface - isto é, alterações novamente no Nível 3.

Esse nível de decomposição e projeto de software é frequentemente relegado ao programador como atividade individual, sendo necessário em qualquer projeto que demore mais do que algumas horas. Ele não precisa ser feito formalmente, mas deverá ser feito pelo menos mentalmente.

Nível 5: Projeto interno de rotina

O projeto de software em nível de rotina consiste em esboçar a funcionalidade detalhada das rotinas individualmente. O projeto interno de rotina é normalmente deixado para ser tratado individualmente pelo programador. O projeto consiste em atividades como a escrita do pseudocódigo, pesquisa de algoritmos em livros de referência, a decisão sobre como organizar os parágrafos de código em uma rotina e a escrita do código em linguagem de programação. Esse nível de projeto é sempre executado, embora às vezes mal e inconscientemente.

5.3 Blocos de construção do projeto de software: heurísticas

Os desenvolvedores de software tendem a apreciar nossas respostas prontas: “Faça A. B e C, e sempre resultarão X, Y, Z”. Orgulhamo-nos de aprender conjuntos secretos de passos que produzem os efeitos desejados e aborrecemo-nos quando as instruções não funcionam conforme o anunciado. Esse desejo por um comportamento determinístico é altamente apropriado para a programação detalhada de computador, onde esse tipo de atenção rigorosa aos detalhes causa o sucesso ou o fracasso de um programa. Mas o projeto de Software é uma história muito diferente.

Como o projeto de software não é determinístico, a aplicação hábil de um conjunto de heurísticas eficientes é a principal atividade no bom projeto de software. Você pode considerar as heurísticas como guias para as tentativas na estratégia “tentativa e erro”.

Encontre objetos do mundo real

A primeira e mais popular estratégia para identificar alternativas de projeto é a abordagem orientada a objetos, que focaliza a identificação de objetos do mundo real e objetos artificiais.

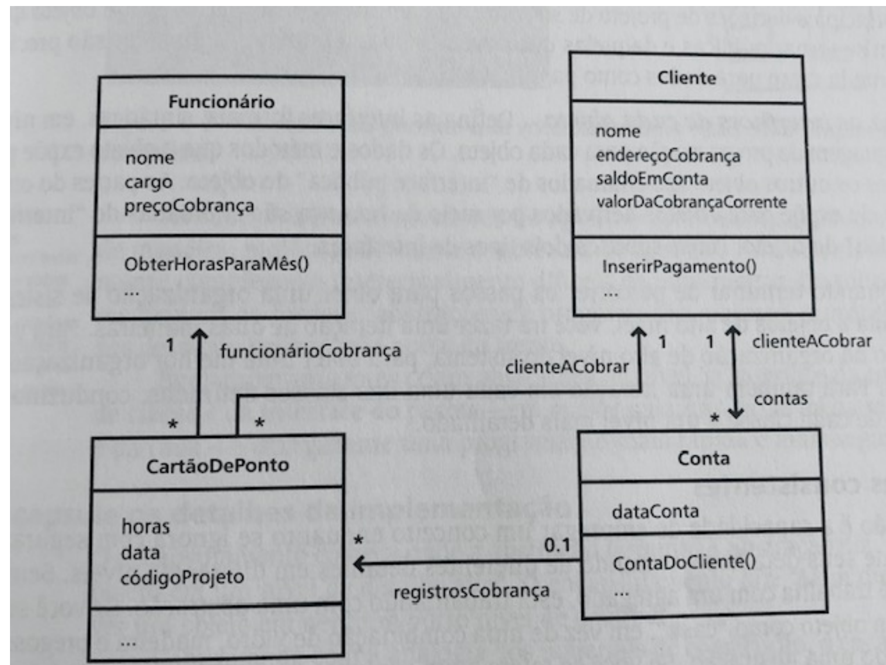
Os passos do projeto com objetos são os seguintes:

- Identifique os objetos e seus atributos (métodos e dados).
- Determine o que pode ser feito em cada objeto.
- Determine o que cada objeto pode fazer com outros objetos.
- Determine as partes (públicas e privadas) de cada objeto que serão visíveis

para outros objetos
- Defina a interface pública de cada objeto.

Esses passos não são necessariamente executados em ordem e quase sempre são repetidos. A iteração é importante.

Identifique os objetos e seus atributos - Os programas de computador normalmente são baseados em entidades do mundo real. Por exemplo, você poderia basear um sistema de cobrança por tempo em funcionários, clientes, cartões de ponto e contas do mundo real. A Figura mostra uma visão orientada a objetos de tal sistema de cobrança.



Identificar os atributos dos objetos não é mais complicado do que identificar os objetos em si. Cada objeto possui características que são relevantes para o sistema. Por exemplo, no sistema de cobrança por tempo, um objeto *funcionário* tem um nome, um cargo e um valor de cobrança. Um objeto *cliente* tem um nome, um endereço de cobrança e um saldo em conta. Um objeto *conta* tem um valor de cobrança, um nome de cliente, uma data de cobrança, etc.

Em um sistema de interface gráfica com o usuário, os objetos incluiriam janelas, caixas de diálogo, botões, fontes e ferramentas de desenho. Um melhor exame do domínio do problema poderia produzir escolhas melhores para os objetos de software do que um mapeamento direto nos objetos do mundo real, mas os objetos do mundo real são um bom ponto de partida.

Determine o que pode ser feito em cada objeto - Uma variedade de operações pode ser executada em cada objeto. No sistema de cobrança mostrado, um *funcionário* poderia ter uma alteração no cargo ou no valor de cobrança, um objeto *cliente* poderia ter seu nome ou endereço de cobrança alterado, etc.

Determine o que cada objeto pode fazer com outros objetos - Este passo é exatamente o que parece. As duas coisas genéricas que os objetos podem fazer uns com os outros são continência e herança. Quais objetos podem *conter* quais outros objetos? Quais objetos podem herdar de quais outros objetos? Na Figura, um objeto

cartãodeponto pode conter um objeto *funcionário* e um objeto *cliente*, e uma *conta* pode conter um ou mais objetos *cartãodeponto*. Além disso, uma *conta* pode indicar que um *cliente* foi cobrado e um *cliente* pode registrar pagamentos para uma conta. Um sistema mais complexo incluiria interações adicionais.

Determine as partes de cada objeto que serão visíveis para outros objetos - Uma das principais decisões de projeto de software é a identificação das partes de um objeto que devem se tornar públicas e daquelas que devem se manter privadas. Essa decisão precisa ser tomada tanto para dados como para métodos.

Defina as interfaces de cada objeto - Defina as interfaces formais, sintáticas, em nível de linguagem de programação para cada objeto. Os dados e métodos que o objeto expõe para todos os outros objeto são chamados de "interface pública" do objeto. As partes do objeto que ele expõe para objetos derivados por meio da herança são chamadas de "interface protegida" do objeto. Pense sobre os dois tipos de interfaces.

Quando terminar de percorrer os passos para obter uma organização de sistema orientada a objetos de alto nível, você irá fazer uma iteração de duas maneiras. Fará uma iteração na organização de alto nível do sistema, para obter uma melhor organização de classes. Fará também uma iteração em cada uma das classes definidas, conduzindo o projeto de cada classe a um nível mais detalhado.

Crie abstrações consistentes

Abstração é a capacidade de empregar um conceito enquanto se ignora com segurança alguns de seus detalhes - tratando de diferentes detalhes em diferentes níveis. Sempre que você trabalha com um agregado, está trabalhando com uma abstração. Se você se refere a um objeto como "casa", em vez de uma combinação de vidro, madeira e pregos, está fazendo uma abstração. Se você se refere a um conjunto de casas como "cidade", esta fazendo outra abstração.

As classes de base nas relações de herança são abstrações que permitem a você enfocar os atributos comuns de um conjunto de classes derivadas e ignorar os detalhes das classes específicas, enquanto trabalha na classe de base. Uma boa interface de classe é uma abstração que permite enfocar a interface sem a necessidade de se preocupar com o funcionamento interno da classe. A interface para uma rotina bem projetada oferece o mesmo benefício em um nível de detalhe mais baixo e a interface para um pacote ou subsistema bem projetado fornece essa vantagem em um nível de detalhe mais alto.

Do ponto de vista da complexidade, a principal vantagem da abstração é que ela permite a você ignorar detalhes irrelevantes. A maioria dos objetos do mundo real já é composta de abstrações de algum tipo. Conforme acabamos de mencionar, uma casa é uma abstração de janelas, portas, tapumes, fiação, encanamento, material isolante e uma maneira particular de organizá-los. Uma porta, por sua vez, é uma abstração de um arranjo particular de uma peça retangular, com dobradiças e uma maçaneta. E a maçaneta é uma abstração de uma máquina específica, feita de latão, níquel, ferro ou aço.

Bons programadores criam abstrações ao nível da interface da rotina, da interface de classe e da interface do pacote e isso garante uma programação mais rápida e mais segura.

Encapsule os detalhes da implementação

O encapsulamento começa onde a abstração termina. A abstração diz: “Você pode ver um objeto em um nível de detalhe alto”. O encapsulamento diz: “Além disso, você não pode ver um objeto em nenhum outro nível de detalhe”.

Continuando com a analogia dos materiais de construção, o encapsulamento é uma maneira de dizer que você pode ver o exterior da casa, mas não pode chegar perto o suficiente para perceber os detalhes da porta. Você pode saber que existe uma porta e também se ela está aberta ou fechada, mas não consegue saber se a porta é de madeira, fibra de vidro, de aço ou de algum outro material, e obviamente não consegue ver cada fibra de madeira.

Herde - quando a herança simplifica o projeto de software

Ao projetar um sistema de software, você frequentemente encontrará objetos quase iguais a outros, exceto por algumas poucas diferenças. Em um sistema de contabilidade, por exemplo, você poderia ter funcionários de tempo integral e de meio período. A maior parte dos dados associados aos dois tipos de funcionários é igual, mas alguns são diferentes. Na programação orientada a objetos, você pode definir um tipo genérico de funcionário e, então, definir funcionários de tempo integral como funcionários genéricos, exceto por algumas poucas diferenças, e funcionários de meio período também como funcionários genéricos, exceto por algumas poucas diferenças. Quando uma operação sobre um funcionário não depende do tipo de funcionário, ela é tratada como se esse funcionário fosse genérico, simplesmente. Quando a operação depende de o funcionário ser de tempo integral ou de meio período, ela é tratada de forma distinta, conforme o tipo de funcionário.

A definição de semelhanças e diferenças entre tais objetos é chamada de “herança”, pois os funcionários de meio período e de tempo integral específicos herdam características do tipo de funcionário genérico.

A vantagem da herança é que ela funciona em sinergia com a noção de abstração. A abstração trata com objetos em diferentes níveis de detalhe. Lembre-se da porta, que era um agrupamento de certos tipos de moléculas em um nível, uma formação compacta de fibras de madeira no nível seguinte e uma peça que mantém os ladrões fora de sua casa no nível seguinte. A madeira tem certas propriedades - por exemplo, você pode corta-la com um serrote ou colá-la com cola de madeira - e as ripas de madeira grossa ou cedro tem as propriedades genéricas da madeira, assim como algumas propriedades especificas próprias.

A herança simplifica a programação porque você escreve uma rotina genérica para tratar de tudo que diz respeito às propriedades genéricas de uma porta e depois escreve rotinas específicas para tratar de operações específicas nos tipos específicos de portas. Algumas operações, como Abrir() ou Entrar(), podem se aplicar independentemente de a porta ser sólida, interior, exterior, blindada, dupla ou de vidro corrediça. A capacidade de uma linguagem de suportar operações como Abrir() ou Fechar() sem saber, até o momento da execução, com que tipo de porta você está tratando, é chamada de “polimorfismo”.

A herança é uma das ferramentas mais poderosas da programação orientada a objetos.

Segredos escondidos (ocultamento de informações)

O ocultamento de informações faz parte da base do projeto estruturado e do projeto orientado a objetos. No projeto estruturado, a noção de "caixas pretas" é proveniente do ocultamento de informações. No projeto orientado a objetos, ele dá origem aos conceitos de encapsulamento e modularidade, e é associado ao conceito de abstração.

O ocultamento de informações é uma das ideias embrionárias no desenvolvimento de software. É uma heurística particularmente poderosa pois, começando com seu nome e em todos os seus detalhes, ele acentua o ocultamento da complexidade.

Segredos e o direito à privacidade

No ocultamento de informações, cada classe (ou pacote ou rotina) é caracterizada por decisões de projeto ou construção que ela oculta de todas as outras classes. O segredo poderia ser uma área de provável alteração, o formato de um arquivo, a maneira como um tipo de dados é implementado, ou uma área que precisa ser protegida do restante do programa para que os erros nessa área causem o mínimo dano possível. A tarefa da classe é manter essas informações ocultas e proteger seu próprio direito à privacidade. Pequenas alterações em um sistema podem afetar várias rotinas dentro de uma classe, mas elas não devem passar para fora da interface da classe.

Uma tarefa importante no projeto de uma classe é decidir quais recursos devem ser conhecidos fora da classe e quais devem permanecer secretos. Uma classe poderia usar 25 rotinas e expor apenas 5 delas, usando as outras 20 internamente. Uma classe poderia usar vários tipos de dados e não expor nenhuma informação sobre eles. Esse aspecto do projeto de classe também é conhecido como "visibilidade", pois está relacionado a quais recursos da classe são "visíveis" ou "expostos" fora da classe.

A interface para uma classe deve revelar o mínimo possível sobre seu funcionamento interno. Uma classe é muito parecida com um iceberg: sete-oitavos ficam sob a água e você só pode ver a oitava parte que fica acima da superfície.

O projeto da interface de classe é um processo iterativo, assim como qualquer outro aspecto do projeto de software. Se você não conseguir a interface correta na primeira vez, tente mais algumas vezes, até que ela se estabilize. Se ela não se estabilizar, você precisará tentar uma estratégia diferente.

Um exemplo de ocultamento de informações

Suponha que você tenha um programa no qual cada objeto deve ter um ID exclusivo armazenada em uma variável-membro chamada *id*. Uma estratégia de projeto seria usar valores inteiros para os IDs e armazenar o ID mais alto atribuído ate o momento em uma variável global chamada *g_maxId*. Conforme cada novo objeto fosse alocado, talvez no construtor de cada objeto, você poderia simplesmente usar a instrução id = ++*g_maxId*, o que garantiria um id exclusivo e acrescentaria o mínimo absoluto de código em cada lugar onde um objeto fosse criado. O que poderia dar erra do com isso?

Muitas coisas poderiam dar errado. E se você quiser reservar intervalos de IDs para propósitos especiais? E se você quiser usar IDs não-sequenciais para melhorar a segurança? E se você quiser reutilizar os IDs dos objetos que foram destruídos? E se você quiser adicionar um trecho de código que é ativado ao se alocar mais IDs do que o número máximo previsto? Se você alocasse IDs espalhando instruções id = ++g_maxId por todo o seu programa, teria que alterar o código associado a cada uma dessas instruções. E se seu programa tivesse múltiplas threads, essa estratégia não seria "thread-safe".

A maneira pela qual novos IDs são criados é uma decisão de projeto que você deve ocultar. Se você usar a instrução ++g_maxId ao longo de todo o seu programa, irá expor a maneira como um novo ID é criado, que se dá simplesmente por meio do incremento. Se, em vez disso, você colocar a instrução id = NewId() ao longo de todo o seu programa, ocultará a informação a respeito de como novos IDs são criados. Dentro da rotina NewId(), você ainda poderia ter apenas uma linha de código: return(++g_maxId) ou sua equivalente, mas se decidisse posteriormente reservar certos intervalos de IDs para propósitos especiais ou reutilizar IDs antigos, poderia fazer essas alterações dentro da própria rotina NewId(), sem tocar em dezenas ou centenas de instruções id

= NewId(). Independentemente do quanto as revisões dentro de NewId () possam se tornar complicadas, elas não afetariam nenhuma outra parte do programa.

Agora, suponha que você descubra que precisa alterar o tipo do ID de inteiro para string. Se você tivesse espalhado declarações de variável, como int id, ao longo de todo o seu programa, o uso da rotina NewId() não ajudaria. Você ainda teria que percorrer o seu programa e fazer dezenas ou centenas de alterações.

O ocultamento de informações é útil em todos os níveis de projeto de software, desde o uso de constantes, em vez de literais, até a criação de tipos de dados, projeto de classes, rotinas e subsistemas.

Duas categorias de segredos

No ocultamento de informações, os segredos se dividem em dois campos gerais:

- Ocultamento da complexidade, para que seu cérebro não precise lidar com ela, a menos que você esteja especificamente interessado nisso
- Ocultamento das fontes de alteração, para que, quando ocorrer uma alteração, os efeitos sejam localizados

As fontes de complexidade incluem tipos de dados complexos, estruturas de arquivo, testes booleanos, algoritmos complexos, etc.

Identifique as áreas de provável alteração

Um estudo a respeito de grandes designers revelou que um atributo que eles tinham em comum era sua capacidade de prever as alterações. Acomodar as alterações é um dos aspectos mais desafiadores do bom projeto de programas. O objetivo é isolar áreas instáveis para que o efeito de uma alteração fique limitado a uma rotina, classe ou pacote. Eis aqui os passos que você deve seguir na preparação contra tais transtornos.

1. **Identifique os itens que provavelmente irão mudar**. Se os requisitos foram bem elaborados, eles incluem uma lista de alterações em potencial e a probabilidade de cada uma delas ocorrer. Nesse caso. é fácil identificar as prováveis alterações.

2. **Separe os itens que provavelmente irão mudar**. Separe em compartimentos isolados cada componente volátil identificado no passo 1, em sua própria classe ou em uma classe com outros componentes voláteis que sejam prováveis mudar ao mesmo tempo.

3. **Isole os itens que provavelmente irão mudar**. Faça o projeto das interfaces entre as classes de modo a ser insensível às alterações em potencial, de modo que as alterações fiquem limitadas ao interior da classe e que o lado externo permaneça inalterado. Qualquer outra classe que utilize a classe alterada não deve ter conhecimento de que a alteração ocorreu. A interface da classe deve proteger seus segredos.

Mantenha o acoplamento baixo

O acoplamento descreve o quanto uma classe ou rotina está relacionada com outras classes ou rotinas. O objetivo é criar classes **e** rotinas com relações pequenas, diretas, visíveis e flexíveis com outras classes **e** rotinas, o que é conhecido como “baixo acoplamento”.

Um bom acoplamento entre módulos é baixo o suficiente para que um módulo possa ser facilmente usado por outros módulos. Os vagões ferroviários normais são acoplados por meio de engates opostos que se prendem quando um vai de encontro ao outro. Conectar dois vagões, portanto, é fácil - basta empurrar um vagão contra o outro. Imagine como seria mais difícil se você tivesse que atarraxar as coisas, conectar um conjunto de

fios ou se pudesse conectar apenas certos tipos de vagões a determinados outros tipos de vagões. O acoplamento dos vagões ferroviários comuns funciona porque é o mais simples possível. No software, faça o mesmo, torne as conexões entre os módulos a mais simples possível.

Tente criar módulos que dependam pouco de outros módulos. Uma rotina como *sin*() tem baixo acoplamento, porque tudo que ela precisa saber é passado com um único valor representando um ângulo em graus. Uma rotina como *InitVars(var1, var2, var3,..., varN)* tem um acoplamento maior, porque com todas as variáveis que devem ser passadas, o módulo que faz a chamada praticamente sabe o que está acontecendo dentro de *Initvars()*. Duas classes que dependem do uso dos mesmos dados globais têm um acoplamento ainda maior.

Procure padrões de projeto comuns

Os padrões de projeto (design patterns) fornecem as bases para soluções prontas que podem ser usadas para resolver muitos dos problemas mais comuns em software. Alguns problemas de software exigem soluções derivadas dos princípios básicos. Mas a maioria dos problemas é semelhante a problemas passados e podem ser resolvidos usando-se soluções semelhantes, ou padrões. Os padrões comuns incluem Adapter, Bridge, Decorator, Facade, Factory Method, Observer, Singleton, Strategy e Template Method. Os padrões oferecem diversas vantagens que o projeto totalmente personalizado não oferece:

Outras Heurísticas

Tenha como objetivo uma coesão forte

A coesão originou se do projeto estruturado e normalmente é discutida no mesmo contexto do acoplamento. A coesão se refere ao rigor com que todas as rotinas de uma ciasse ou de todo o código de uma rotina corroboram um propósito central - o quanto a classe é localizada em determinada funcionalidade. As classes que contém funcionalidade fortemente relacionada são descritas como tendo alta coesão e o objetivo da heurística é tornar a coesão a mais alta possível. A coesão é uma ferramenta útil para o controle da complexidade, pois quanto mais o código de uma classe corrobora um propósito central, mais facilmente seu cérebro pode focalizar tudo que o código faz.

Construa hierarquias

Hierarquia é uma estrutura de informação disposta em camadas, na qual a representação mais geral ou abstrata de conceitos está contida no topo, com as representações cada vez mais detalhadas e especializadas nos níveis mais baixos.

As hierarquias têm sido uma importante ferramenta para controle de conjuntos complexos de informações há pelo menos 2.000 anos. Aristóteles usou uma hierarquia para ordenar o reino animal. Os seres humanos frequentemente usam hierarquias de tópicos para organizar informações complexas. Os pesquisadores verificaram que as pessoas geralmente interpretam as hierarquias como uma maneira natural de organizar informações complexas. Quando desenhamos um objeto complexo, como uma casa, o fazemos hierarquicamente. Ou seja, primeiramente desenhamos um esboço da casa; em seguida, as janelas e portas; e, depois, os demais detalhes. A casa não é desenhada tijolo por tijolo, ou prego por prego, telha por telha (Simon 1996).

Formalize os contratos de classe

um nível mais detalhado, pensar na interface de cada classe como um contrato

com o restante do programa pode gerar boas ideias. Normalmente. o contrato é algo como: “Se você prometer fornecer os dados x, y e z, e prometer também que eles terão as características a, b e c, eu prometo executar as operações 1, 2 e 3 dentro das restrições 8, 9 e 10”. As promessas que os clientes da classe fazem para ela são normalmente chamadas de “pré-condições”; as promessas que o objeto faz para seus clientes são chamadas de “pós-condições”.

Os contratos são úteis para o controle da complexidade porque, pelo menos teoricamente, o objeto pode ignorar com segurança qualquer comportamento não-contratual. Na prática, esse problema é muito mais difícil.

Atribua responsabilidades

Outra heurística é pensar em como as responsabilidades devem ser atribuídas aos objetos. Perguntar pelo que cada objeto deve ser responsável é semelhante a perguntar quais informações ele deve ocultar, mas isso pode produzir respostas mais abrangentes, o que dá à heurística um valor único.

Desenvolva o projeto pensando nos testes

Um processo de abstração que pode gerar ideias interessantes para o projeto de software é perguntar como o sistema ficará se você o projetar de modo a facilitar os testes. Você precisa separar a interface com o usuário do restante do código para poder exercitá-la independentemente? Você precisa organizar cada subsistema, de modo que isso minimize as dependências de outros subsistemas? Fazer o projeto para teste tende a resultar em interfaces de classe mais formalizadas, o que geralmente é benéfico.

Evite falhas

Autores argumentam que muitas falhas espetaculares de pontes ocorreram devido ao enfoque em sucessos anteriores e à falta de uma avaliação adequada de possíveis modos de falha. Concluem que falhas como a da ponte Tacoma Narrows poderiam ter sido evitadas se os projetistas tivessem considerado criteriosamente as maneiras pelas quais a ponte poderia falhar e não apenas copiado os atributos de outros projetos de sucesso.

Escolha conscientemente o momento da vinculação

Momento da vinculação se refere ao momento em que um valor específico é vinculado a uma variável. Um código que vincula cedo tende a ser mais simples, mas também menos flexível. Às vezes, você pode conceber uma boa ideia do projeto fazendo perguntas como “E se eu vincular esses valores antes” E se eu vincular esses valores depois? E se eu iniciasse os valores dessa tabela aqui mesmo, no código? E se eu ler o valor dessa variável do usuário em tempo de execução?”

Crie pontos centrais de controle

P. J. Plauger diz que sua principal preocupação é “O Princípio de Um Lugar Certo - deve haver Um Lugar Certo para procurar qualquer código não trivial e Um Lugar Certo para fazer uma provável alteração de manutenção” (Plauger 1993). O controle pode ser centralizado em classes, rotinas, macros de pré-processador, arquivos *#include* - até uma constante é um exemplo de ponto central de controle.

A vantagem da complexidade reduzida é que quanto menos lugares você tiver para procurar algo, mais fácil e mais seguro será fazer uma alteração.

Considere o uso da força bruta

Uma poderosa ferramenta de heurística é a força bruta. Não a subestime. Uma

solução ao estilo força bruta que funciona é melhor do que uma solução elegante que não funciona. Pode demorar um longo tempo para que uma solução elegante funcione Na descrição da história dos algoritmos de pesquisa, por exemplo, Donald Knuth mostrou que, apesar de a primeira descrição de um algoritmo de busca binária tendo sido publicada em 1946, demorou outros 16 anos para que alguém publicasse um algoritmo que pesquisasse corretamente listas de todos os tamanhos (Knuth 1998). Uma busca binária é mais elegante, mas a busca sequencial no estilo força bruta frequentemente é suficiente.

Desenhe um diagrama

Os diagramas são outra poderosa ferramenta heurística. Uma imagem vale mais do que mil palavras - até certo ponto. Na verdade, você quer omitir a maior parte das mil palavras, pois um dos objetivos de usar uma figura é que ela pode representar o problema em um nível mais alto de abstração. Às vezes, você quer lidar com o problema em detalhes, mas em outras ocasiões, quer trabalhar com mais generalidade.

Mantenha seu projeto modular

O objetivo da modularidade é tornar cada rotina ou classe uma “caixa-preta”: você sabe o que entra e o que sai, mas não sabe o que acontece dentro. A caixa preta tem uma interface tão simples e uma funcionalidade tão bem-definida que, para qualquer entrada específica, você pode prever precisamente a saída correspondente.

O conceito de modularidade está relacionado com o ocultamento de informações, com o encapsulamento e com outras heurísticas de projeto. Mas, às vezes, pensar sobre como montar um sistema a partir de um conjunto de caixas pretas gera ideias que o ocultamento de informações e o encapsulamento não geram; portanto, é interessante carregar esse conceito em seu bolso traseiro.

Resumo das heurísticas de projeto

- Encontre objetos do mundo real
- Crie abstrações consistentes
- Encapsule os detalhes da implementação
- Herde quando possível
- Esconda os segredos (ocultamento de informações)
- Identifique áreas de provável alteração
- Mantenha o acoplamento baixo
- Procure padrões de projeto comuns
- Tenha como objetivo uma coesão forte
- Construa hierarquias
- Formalize os contratos de classe
- Atribua responsabilidades
- Desenvolva o projeto pensando nos testes
- Evite falhas
- Escolha conscientemente o momento da vinculação
- Crie pontos centrais de controle
- Considere o uso da força bruta
- Desenhe um diagrama
- Mantenha seu projeto modular

Pontos-chave

❖ O principal imperativo técnico do software é o controle da complexidade. Isso é bastante facilitado por um foco do projeto na simplicidade.

- A simplicidade é obtida de duas maneiras gerais: minimizando a quantidade de complexidade essencial com que o cérebro de alguém precisa lidar em dado momento e impedindo que a complexidade acidental prolifere desnecessariamente.
- Projeto de software é heurística. O apego dogmático a uma única metodologia prejudica a criatividade e seus programas.
- O bom projeto de software é iterativo; quanto mais possibilidades de projeto você tentar, melhor será seu projeto final.
- O ocultamento de informações é um conceito particularmente valioso. Perguntar “o que eu devo ocultar?” resolve muitos problemas difíceis de projeto de software.