PROGRAMAÇÃO POR OBJETOS

Tópico 2
Os problemas da programação estruturada que vieram a motivar a orientação a objetos

Leonel Morgado, Universidade Aberta, 2023

Dos primórdios da otimização à programação estruturada

Os primeiros computadores tinham como propósito fazer cálculos rapidamente. O objetivo era massificar a quantidade de cálculos para permitir resolver em tempo útil problemas de cálculo matemático que estavam além da capacidade de seres humanos o fazerem em tempo útil ou mesmo além da capacidade de batalhões de seres humanos. Desde o mecanismo de Anticítera para cálculo astronómico aos famosos esforços de Alan Turing para quebrar os códigos militares do Eixo na Segunda Grande Guerra, passando por pioneiros dos séculos anteriores, como Pascal ou Charles Babbage, o foco não era permitir programar facilmente as máquinas: era obter delas o máximo de cálculo possível.

Quando surgiu o EDSAC, entre 1946 e 1949, o primeiro computador capaz de executar programas diferentes sem ter de se reorganizar a cablagem que o fazia funcionar, passou a ser mais importante olhar para a qualidade do código para evitar erros. Não se gastava tempo a reorganizar cabos ou ligar e desligar manualmente interruptores, como no famoso ENIAC, era importante poder mudar rapidamente entre programas que estivessem corretos. Programas que pudessem mais facilmente ser escritos e verificados por seres humanos. Este anseio levou à criação e lançamento da linguagem Fortran em 1957… e fomos evoluindo até às linguagens atuais em toda a sua diversidade.

Contudo, tal também permitiu que se abraçassem problemas e situações cada vez mais ambiciosos e complexos, não necessariamente na sua formulação matemática, mas na diversidade da experiência humana: problemas que mudam com o tempo, que se têm de reescrever para se adaptar ao mundo. Em breve se constatava que não bastava uma linguagem ser legível e interpretável por seres humanos: as pessoas, na sua diversidade e na complexidade das vidas pessoais e das organizações, acabam por produzir soluções que funcionam mas… contêm sementes perversas, que lhes provocam mais tarde problemas. Foi ao constatá-lo que surgiram várias propostas de estruturação da forma de escrever código-fonte.

Programação por Objetos - Tópico 2 - Os problemas da programação estruturada que vieram a motivar a orientação a objetos © 2023 por Leonel Morgado está licenciado segundo a licença CC BY 4.0

Uma das propostas mais famosas foi o relatório "Notes on Structured Programming" de Edsger Dijkstra, em 1970. Nele afirmava (negritos meus) "*torna-se da máxima urgência **parar** de encarar a programação fundamentalmente como a minimização da razão custo/desempenho. Devemos reconhecer que já hoje a programação é acima de tudo um desafio intelectual: **a arte de programar é a arte de organizar a complexidade**, de dominar a multiplicidade e evitar o seu caos maldito.*"

Alguns exemplos de programação estruturada

Nesse relatório, Dijkstra exemplifica vários conceitos que foram levando à estruturação da programação, como este caso de dois programas equivalentes, mas difíceis de aceitar como tal:

```
if B2 then
  begin while B1 do S1 end
else
  begin while B1 do S2 end
```

é equivalente a:

```
while B1 do
  begin if B2 then S1 else
S2 end
```

Afirma assim:
«*Consigo demonstrar a equivalência do resultado destas computações, mas não consigo encará-las como equivalente de qualquer outra forma útil. Tive de obrigar-me a chegar à conclusão que (1) and (2) são "difíceis de comparar". Inicialmente, esta conclusão incomodou-me muito. Entretanto, consegui encarar esta incomparabilidade como um facto da vida. Consequentemente, é uma das principais razões pelas quais encaro que a escolha entre (1) e (2) é uma decisão de projeto relevante, que não deve ser tomada sem ponderação cuidada. É precisamente esta aparente trivialidade que me sensibilizou para as considerações que devem influenciar tais escolhas.*»

Nesse relatório, cuja consulta encorajo, surgem muitas outras recomendações de estruturação de programas. Boas práticas que se mantêm e que originaram muitos outros trabalhos e técnicas de código. Exemplos dessas técnicas são a criação de blocos de agrupamento de variáveis, ou seja "tipos compostos" (as `structs` do C, por exemplo) ou a possibilidade de ter variáveis locais a uma função, ou seja, permitir programar blocos de código sem a preocupação de que outras partes do código pudessem influenciar ou alterar as suas variáveis, além de poder haver, por exemplo, mil variáveis "i", cada uma em sua função, em vez de i, i0, i1, i2, inova, inovissima, etc. (estas duas novidades surgiram na linguagem ALGOL, um momento transformador na evolução da programação estruturada).

O percurso para programação de qualidade não terminou aí, está bem de ver: neste documento mostram-se alguns exemplos de problemas que podem surgir, ainda que se usem princípios de programação estruturada, devido à complexidade inerente aos nossos objetivos: à medida que os programas crescem, que ficam mais dependentes de ser "bem" ou "corretamente" empregue a linguagem de programação, é inevitável que os problemas desse equilíbrio instável se manifestem.

Assim, surgiu a proposta de programação orientada a objetos como uma forma de ir mais além na estruturação e qualidade, mas isso ficará para outro tópico. Neste exemplificam-se problemas.

Estado global: como rebentar um foguetão com programação estruturada

Nos primórdios da programação, nem sempre era viável (até por motivos de desempenho) estar a passar toda a informação entre funções. O recurso a variáveis globais era uma inevitabilidade. Ainda hoje assim ocorre, por exemplo para lidar com grandes volumes de dados que partilhamos entre funções (por exemplo, no tratamento de imagens de alta resolução).

Esta "partilha de estado global" pode originar comportamentos complicados, como ciclos sem fim explícito, mas com uma saída por efeito secundário de uma das instruções internas:

```
int sair = 0; // Inicialmente, não é para sair do ciclo

while (sair == 0) {
    // ...
    vamos_fazer_isto();
    vamos_fazer_tambem_isto();
    vamos_fazer_ainda_isto();
    // ...
}

void vamos_fazer_isto() {
    // Várias coisas
}

void vamos_fazer_tambem_isto() {
    // Várias coisas, podem levar a fazer, nalgum if...
    sair=1;
    // Mais coisas
}

void vamos_fazer_ainda_isto() {
    // Várias coisas, que podem levar a fazer, nalgum if...
    sair=0;
    // Mais coisas
}
```

Assuma-se que hoje em dia este tipo de código já é (felizmente) pouco habitual, mas serve como exemplo com pouca extensão, compaginável com o espaço de formatação visual, para exemplificar o conceito.

Neste código, as funções `vamos_fazer_tambem_isto` e `vamos_fazer_ainda_isto` alteram a variável sair de forma não explícita, tornando o controle de fluxo confuso e difícil de acompanhar. Nesta situação, quem lê o código principal não sabe se se sai do `while` (nem como).

Além disso, pode suceder uma das funções decidir sair, mexendo na variável global `sair`, mas outra função depois anular essa situação.

Pior: nada obriga a que uma função veja qual é o estado atual da variável `sair`, antes de decidir alterá-la. Imagine-se esta situação:

```c
#include <stdio.h>
#include <stdbool.h>

int termoestato = 0; //Variável global com o nível do termoestato
bool aquecimento_ligado = false; // Variável global para saber o estado do
aquecimento

void aquecer() {
    aquecimento_ligado = true;
    //Só mudou a variável do aquecimento... não está realmente
    //a chamar nenhuma função para o ligar...
    termoestato = termoestato + 1;
    MexerNoTermoestato(termoestato);
    //se estiver o aquecimento desligado, isto falha
}

int main() {
    // Vê-se se está frio, com alguma função.
    // Estando, pede-se para aquecer:
    aquecer();

    return 0;
}
```

Neste código, a função `main` está dependente, para funcionar, do comportamento adequado sobre as variáveis globais de todas as funções do programa, que podem ser imensas, criadas por programadores diferentes com meses ou anos de distância.

Há vários episódios famosos de problemas decorrentes da alteração indevida de variáveis globais por alguma função, normalmente devido precisamente à mudança das equipas e da distância temporal entre a programação inicial e a sua utilização, mesmo quando os programadores são experientes. Refira-se, por exemplo, o desastre com o foguetão Ariane 5 em 1996 por alteração da infame variável global BH. Como se relata nessa página Web, que se traduz aqui parcialmente:

> *Vários fatores contribuíram* (…) *o valor BH nem sequer era necessário depois do lançamento, tinha sido deixado no código-fonte do antecessor, o Ariane 4, que precisava deste valor para se alinhar depois do lançamento.* (…) *o código que teria detetado e tratado esses erros* (…) *tinha sido desativado para o valor BH, devido a limitações de desempenho no hardware do Ariane 4, que não se aplicavam ao Ariane 5. Um último fator foi uma alteração nos requisitos* (…)*. O Ariane 5 foi lançado com uma trajetória muito mais inclinada* (…) *levando a maior velocidade vertical.* (…) *o foguetão acelerava mais rapidamente, maior certeza havia de que o valor BH encontraria o erro de conversão.*

Reutilização de código: os clones intermináveis

Além do desafio da manutenção do estado global, a programação sem objetos também se depara com a questão da reutilização de código. Em programação, são frequentes as situações em que temos de fazer coisas segundo um padrão, apenas com diferenças de pormenor. Isso leva à necessidade frequente de termos blocos de código repetido, para sustentar essas diferenças.
Por exemplo, listas longas de funções como estas eram comuns:

```
void ordenarClientesPorNome(Cliente clientes[], int num_clientes);
void ordenarClientesPorID(Cliente clientes[], int num_clientes);
void ordenarClientesPorSaldo(Cliente clientes[], int num_clientes);
void ordenarVeiculosPorModelo(Veiculo veiculos[], int num_veiculos);
void ordenarVeiculosPorMatricula(Veiculo veiculos[], int num_veiculos);
void ordenarVeiculosPorAno(Veiculo veiculos[], int num_veiculos);
void ordenarProdutosPorNome(Produto produtos[], int num_produtos);
void ordenarProdutosPorCodigo(Produto produtos[], int num_produtos);
void ordenarProdutosPorPreco(Produto produtos[], int num_produtos);
```

Embora fosse possível, com técnicas elaboradas, partilhar internamente código entre estas funções, na verdade elas são funções independentes. Ou seja: tanto `ordenarClientesPorNome` como `ordenarClientesPorSaldo` estão a aceder diretamente ao vetor de clientes. Ainda que façam tudo com um código estruturadíssimo, recorrendo a algumas funções mais gerais para evitar duplicar código, não têm nenhuma garantia, absolutamente nenhuma, que noutro teclado outro programador não pense "preciso de ordenar os clientes por data de nascimento" e não faça uma `ordenarClientesPorDataNasc`… com código totalmente diferente do dessas funções.

As consequências perniciosas são imensas. Por exemplo, imagine-se que as duas primeiras funções usam algoritmos que estão a contar, por algum motivo de otimização, que os ID estejam sempre por ordem crescente entre clientes empatados na ordenação: a terceira função pode não saber isso e ao ordenar por data de nascimento partir essa organização, afetando as duas anteriores sem que se saiba porque é que deixaram de funcionar. Note-se que este tipo de erros acontece muitas vezes já em produção, quando se combinam sequências não testadas de valores no acesso às funções…

Complexidade na manipulação de dados: o malabarismo permanente

O exemplo anterior permite recordar que frequentemente temos estruturas de dados complexas, de dados interligados. Por exemplo, admite-se desse exemplo que:

- Um cliente tem nome, ID e saldo
- Um veículo tem modelo, matrícula e ano
- Um produto tem nome, código e preço.

Logo o primeiro caso, o registo de clientes, é uma fonte corrente de problemas. Quando analisamos dados de sistemas de informação com algum historial, é frequente encontrar coisas com este aspeto:

Cliente	Idade	Telefone
Manuel Silva	20	(054) 123456
Maria Afonsina	??	(036) 789012
João Fortunato	50	(043) 345678V

Neste caso, temos uma idade com valor inválido e um telefone com letras acrescentadas, como aquele “V”, que era usado para indicar “este telefone é do vizinho”.

Repare-se que quando várias funções diferentes têm acesso aos dados, é perfeitamente possível (e até comum) que uma delas, perante uma falta de elementos para agir tome uma opção de registar na idade um valor inválido, por exemplo “-1”, como forma de indicar esse problema. Se noutra parte do programa, contudo, há a expetativa da idade ser sempre positiva, esse valor não será interpretado como indefinido. Já o telefone com a letra apensa funcionará durante anos… até um dia em que é preciso transferir os dados para um sistema novo, ou se pretende instalar um sistema de contacto automático com os clientes, ou simplesmente a operadora telefónica implementa uma renumeração que obriga a atualizar toda a base de dados – em qualquer destas situações, as primeiras implementações dos algoritmos irão falhar porque não concebem a presença de letras apensas aos números.

Acrescem a estas situações regras mais complexas. Por exemplo, imagine-se que aquele ano associao ao veículo é o ano de matrícula. Não há matrículas sem ano, pelo que se deve querer garantir que sempre que se atribui uma matrícula se atribui também o ano respetivo. Contudo, se todas as funções acedem diretamente aos dados, a responsabilidade de manter essa coerências reside simultaneamente em todas elas, o que representa um risco enorme de incoerências.