



Depuração & Testes

Prof. Guilherme N. Ramos

A civilização roda em software, e a maioria das atividades de engenharia envolve programas de computador [1]. Eles permeiam tantas aplicações que defeitos podem causar sérios prejuízos (como aplicações financeiras) e danos físicos (controle de aviões ou usinas), tornando a qualidade do programa extremamente relevante [2].

“Errar é humano, mas para estragar tudo é preciso um computador.”

William E. Vaughan

Pequenos defeitos podem, potencialmente, causar enormes problemas: um erro em um módulo obscuro de um sistema grande é geralmente a causa mais frequente de problemas [2]. A medida que os programas tornam-se mais rápidos, complexos, e ubíquos, os danos podem ser cada vez maiores. Geralmente, estes pequenos erros são simples descuidos dos desenvolvedores, mas esta simplicidade não quer dizer que não possam ser sérios ou que é fácil encontrá-los e resolvê-los. A qualidade de sistemas complexos depende da qualidade dos pequenos programas que o compõem [2].

Erros de programação *sempre* existirão [pelo menos enquanto o processo de geração de código continuar como agora], dizem que qualquer código significativo terá defeitos inicialmente, geralmente 2 para cada 100 linhas de código [3]. Isso ocorre por diversos motivos, mas principalmente por que o código é gerado por seres humanos (e, portanto, falhos), e mesmos sistemas pouco complexos possuem tantas possibilidades que, na prática, é impossível testar todas as situações (e assim garantir a correção).

“Há duas maneiras de se escrever programas sem erros; apenas a terceira funciona.”

Alan Perlis

Os erros geralmente se originam nas 3 primeiras etapas e, como são “inevitáveis”, as duas últimas se fazem necessárias. para para as seguintes origens:

Especificação: se um problema não for especificado corretamente, mesmo um programa perfeito não terá o desempenho desejado [3]. Estes erros podem ser minimizados (ou mesmo eliminados) pela revisão atenta do entendimento e análise do problema apresentado, entenda *claramente* o que o seu programa deverá fazer.

Algoritmo: o programa precisa ser planejado antes de ser implementado, partir da especificação diretamente para a programação e torcer para que funcione não é uma estratégia de muito sucesso. Estes erros podem ser minimizados com um planejamento adequado do algoritmo, considerando que estratégia utilizar, como gerar o código, quais as estruturas de dados e como elas devem ser usadas. Um tempo maior gasto em um planejamento cuidadoso quase sempre é recompensado com menos refatorações.

Codificação: a tradução do algoritmo em código não é uma ciência exata (quem nunca confundiu = com ==?), e alguns destes erros podem ser tão simples que nem são considerados quando se busca a origem do erro (prolongando o processo). Podem ser minimizados pela simples releitura do código, tanto pelo autor quanto por outro desenvolvedor. Por incrível que pareça, o pato de borracha sabe das coisas.

Ainda assim, os erros persistem, então é preciso considerar outras estratégias para mitigá-los...

“Se depurar é o processo de remover bugs, então programar deve ser o processo de inserí-los.”

Edsger W. Dijkstra

Em termos genéricos, a *depuração* serve para lidar com um problema conhecido no programa, e os *testes* servem para tentar, sistematicamente, verificar situações que “quebrem” um programa [que você acha que funciona] para analisar se ele tem o comportamento esperado.

1 Depuração

“bug: defeito ou falha em uma máquina, plano, ou similar.”

Dicionário Oxford

“A partir daí, sempre que alguma coisa dava errado com um computador, nós dizíamos que tinha um bug nele.”

Grace Hopper

A *depuração* é feita quando se sabe que o programa não funciona (erros de execução, de segmentação de memória), não tem o desempenho desejado, ou simplesmente não tem o comportamento esperado.

Consertar um programa com *bugs* é um processo de confirmar, uma a uma, que as suas crenças sobre o seu código são, de fato, verdadeiras [4]. Quando algo não é confirmado, você pelo menos tem uma dica da natureza ou de onde está o problema. A situação começa a complicar com a propagação de *bugs* no código (um *bug* acaba gerando outro(s) e assim sucessivamente). Eventualmente se descobre um *bug* no meio da cadeia, e é preciso ir retrocedendo até encontrar o *bug* original. A maioria das vezes isto não é um processo óbvio, nem simples, certamente demorado¹, e potencialmente caro. A execução de testes sobre código tende a diminuir consideravelmente a quantidade de *bugs* presentes, e quanto antes são descobertos, menor será o esforço para removê-los.

Uma das melhores práticas de programação é realizar pequenas alterações no código e testá-las adequadamente a medida que são feitas.

Se o código funcionava antes da última mudança no código, e os testes mostraram um erro, fica claro que esta mudança provocou o problema e, portanto, que o código novo tem um *bug*, ou expôs um *bug* existente. Os passos de depuração tendem a ser simples:

1. *Teste* o código para descobrir quais problemas existem.
2. *Defina* as condições que o erro pode ser reproduzido.
3. *Encontre* onde no código está a instrução que causa o erro.
4. *Corrija* a instrução;
5. *Verifique* que a correção funciona (com testes).

¹Uma vez certo profissional de programação passou 8 dias inteiros rastreando um *bug*, passando por diversas GUIs, alguns *forks* e muitas definições inconsistentes, apenas para descobrir um ponteiro não inicializado...

A *depuração* é inevitável, mas há diferentes formas de avaliar a execução do programa ao longo deste processo. A primeira, e mais eficiente, é *pensar* a respeito do que está sendo feito.

“Todos sabem que depurar é duas vezes mais difícil que programar. Se você gera um programa usando toda a sua esperteza, como vai depurá-lo?”

Brian W. Kernighan

Caso não tenha conseguido achar o problema, é preciso investigar um pouco mais a fundo, e as ferramentas mais simples são o bom e velho `printf` em uma busca binária, tentando localizar em que ponto do código as coisas começam a dar errado.

“A ferramenta de depuração mais eficaz ainda é pensamento cuidadoso, juntamente com instruções de impressão bem colocadas.”

Brian Kernighan

Se mesmo assim não foi possível encontrar o problema, ferramentas mais poderosas - os depuradores - são necessárias. Um depurador é um programa de computador usado para testar outros programas, e geralmente têm funcionalidades mais sofisticadas como: execução passo a passo do programa; a suspensão do programa para examinar seu estado atual, em pontos predefinidos, chamados pontos de parada; o acompanhamento do valor de variáveis que podem ser usadas inclusive para gerar uma suspensão, ou ativar um ponto de parada.

1.1 The GNU Project Debugger

O `gdb` é o depurador padrão para ambientes GNU/Linux, e permite que você veja o que está acontecendo “dentro” de outro programa enquanto este é executado - ou o que outro programa estava fazendo no momento que falhou [5]. As principais funções do `gdb` são:

1. iniciar o programa, especificando qualquer coisa que possa influenciar seu comportamento;
2. interromper a execução do programa sob condições específicas;
3. examinar o que aconteceu quando o programa foi interrompido; e
4. alterar coisas no programa, permitindo que você experimente com as correções do *bug* (e aprenda sobre outro).

O `gdb` é usado iterativamente pela linha de comando, mas há diversas interfaces com ele. É preciso indicar ao compilador que gere um executável depurável, utilizando a opção `-g`:

```
$ gcc [flags] -g <arquivo> -o <saída>
```

A interação com o depurador é simples, basta ter familiaridade com os comandos (ou seja, pratique). Há diversos exemplos disponíveis para estudar.

“Se você quer programadores mais eficientes, descobrirá que eles não deveriam gastar tempo depurando, eles não deveriam gerar bugs para começar.”

Edsger W. Dijkstra

O `gdb` é uma ferramenta fundamental para análise de código com gerenciamento de memória (outra muito interessante é `Valgrind`) e, portanto, para a depuração. Mas este processo é custoso, e uma das formas de tentar minimizar os gastos é evitando a propagação de *bugs*.

2 Testes

O processo de testar um programa é planejado para garantir que o código de computador faça o que foi projeto para fazer e, contrariamente, que não faz o que não deveria [6]. A ideia, mais que ter um comportamento definido, é “garantir” que o a execução do programa não tenha resultados/consequências imprevisíveis.

Os testes buscam investigar a qualidade do programa no contexto em que ele deve operar. Mais especificamente, podem detectar erros no código, de modo que as falhas do programa possam ser encontradas e corrigidas. Esta investigação afeta diretamente a *qualidade de software* (veja também a ISO/IEC 25010). É fácil ver como isto pode ser traduzido em termos financeiros, no valor cobrado pelo produto de maior qualidade gerado, mas o verdadeiro impacto está nos custos de desenvolvimento.

“Em um típico projeto de programação, 50% do tempo e mais de 50% do custo total são gastos em testes do programa ou sistema em desenvolvimento.”

Myers, Badgett & Sandler

A única forma de mostrar que um programa é correto é testar *todas* as possibilidades de execução [7] (é possível provar a corretude de um algoritmo, mas isso não tem implicações em sua implementação). Como isto é inviável, na prática o que se tenta é maximizar as possibilidades testadas (minimizando a quantidade de testes), portanto a qualidade do testes depende da qualidade dos profissionais que definem *o que testar*. Testar também é um trabalho criativo e difícil, o responsável pelos testes deve ser experiente e conhecedor de técnicas mais úteis. Por fim, testar é uma tarefa complexa, testes efetivos dependem de conhecimento detalhado do sistema, que geralmente não é simples e nem de fácil entendimento.

“Testes de programas podem ser usados para revelar a existência de erros, mas nunca para mostrar sua ausência!”

Edsger W. Dijkstra

2.1 Testes de Caixas

Os testes podem ser divididos em duas categorias básicas: *caixa-preta* e *caixa-branca*. Um teste de caixa-preta visa testar a funcionalidade do programa sem analisar sua implementação: a ideia é verificar pares de entrada/saída de dados. Neste caso, quanto mais abrangentes as entradas (em função das especificações), melhor será a qualidade do teste. As entradas testadas são definidas a partir das especificações/requisitos, e focam nas funcionalidades projetadas.

Um teste de caixa-branca considera a implementação para definir os casos de teste, buscando analisar a execução de todos os fluxos de possíveis. Este tipo de teste é mais custoso que o de caixa-preta, pois exige conhecimento interno do sistema, e precisa ser ajustado em caso de alterações na implementação; além da “limitação” de testar o programa como ele está implementado (ou seja, não detecta ausência de funcionalidades). Entretanto, como a estrutura interna é usada como referência, é mais fácil definir os valores de entrada mais úteis para os testes.

```

1 /* Testes de Caixa-Preta */
2 assert(min(1, 2, 3) == 1);
3 assert(min(1, 3, 2) == 1);
4 assert(min(2, 1, 3) == 1);
5 assert(min(2, 3, 1) == 1);
6 assert(min(3, 1, 2) == 1);
7 assert(min(3, 2, 1) == 1);
8 assert(min(1, 1, 1) == 1);
9 assert(min(1, 1, 2) == 1);
10 assert(min(1, 2, 2) == 1);
11 assert(min(-1, 2, 3) == -1);
12 assert(min(1, -2, 3) == -2);
13 assert(min(1, 2, -3) == -3);

```

```

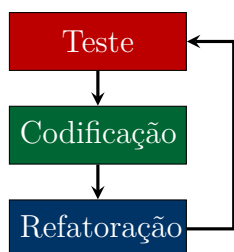
1 /* Implementação */
2 if(A > 1 && B == 0)
3     x /= A;
4 if(A == 2 || x > 1)
5     ++x;

1 /* Testes de Caixa-Branca */
2 A=2, B=0; /* todos os processos */
3
4 A=3, B=0, X=0; /* todas decisões e */
5 A=2, B=1, X=1; /* todas as condições */

```

2.2 Test Driven Development

TDD é uma técnica para desenvolvimento de software baseada em um ciclo curto de repetições em que a evolução do programa é guiado pelos testes gerados.



1. Escreva um caso de teste que define uma melhoria desejada ou uma nova funcionalidade *ainda não implementada*.
2. Escreva o mínimo código seja validado pelo teste.
3. Refatore o código para padrões adequados ao projeto.

Os procedimentos são simples, e baseiam-se em 3 regras:

1. Você não pode criar código antes de ter criado um teste que falhe.
2. Você não pode criar mais em um teste além do necessário para falhar, e a não compilação é uma falha.
3. Você não pode criar mais código que o necessário para passar no teste que está falhando..

Esta abordagem implica em uma série de coisas (a mais óbvia é uma grande quantidade de código gerada para os testes), mas também tem efeitos na qualidade do programa gerado. Para *adicionar um teste*, o desenvolvedor precisa entender bem as especificações e requisitos da funcionalidade. Ao *executar o [novo] teste*, o desenvolvedor verifica um aspecto de funcionalidade que ainda não havia sido avaliado. Ao *gerar código [para passar o teste]*, ele está efetivamente incluindo a funcionalidade sendo avaliada, mesmo que sem se preocupar com a qualidade deste código. Ao *repetir todos os testes*, ele verifica que o código atende a nova funcionalidade e não quebra as funcionalidades existentes. Caso haja algum problema, cada apenas o código suficiente para resolver este novo erro é gerado até que não haja mais erros. Por fim, ao *refatorar o código* o resultado será código limpo (repita os passos até que todas as funcionalidades sejam implementadas).

As principais vantagens são que os diversos testes fornecem uma forma simples de validar o código, potencialmente gera apenas o código realmente necessário (de forma mais modular) e, principalmente, diminuem a quantidade de defeitos e assim o tempo total de desenvolvimento (mais codificação, menos depuração).

Entretanto, TDD não se aplica a todos os projetos e apresenta uma série de dificuldades (necessidade de suporte gerencial, dificuldade de ser incorporado a um projeto em andamento, cobertura parcial de funcionalidades, etc.).

Referências

- [1] Bjarne Stroustrup. *Programming: principles and practice using C++*. Pearson Education, 2014.
- [2] Watts S Humphrey. The software quality profile. *Fundamental Concepts for the Software Quality Engineer*, page 1, 2002.
- [3] Neil Matthew and Richard Stones. *Beginning Linux programming*. Wrox programmer to programmer. Wrox ; John Wiley [distributor], Indianapolis, Ind. : Chichester, 4th ed edition, 2008.
- [4] Norman Matloff and Peter Jay Salzman. *The Art of Debugging with GDB, DDD, and Eclipse*. No Starch Press, San Francisco, CA, USA, 2008.
- [5] Richard Stallman, Roland Pesch, Stan Shebs, et al. Debugging with gdb. *Free Software Foundation*, 2015.
- [6] Glenford J. Myers, Tom Badgett, Todd M. Thomas, and Corey Sandler. *The art of software testing*. John Wiley & Sons, Hoboken, N.J, 2nd ed edition, 2004.
- [7] Luis Joyanes Aguilar. *Fundamentos de Programação: Algoritmos, estruturas de dados e objetos*. McGraw-Hill, 3a edition, 2008.