Paulo Marcel Caldeira Yokosawa
Renato Kosaka Araújo

Anti-Padrão: Printf Debugging

Raízes do Problema

No desenvolvimento de qualquer sistema, uma das partes mais importantes, se não a mais importante, é a dos testes. Mesmo o mais perfeito dos sistemas do ponto de vista conceitual podem ter falhas em sua implementação que passem desapercebidas por seus desenvolvedores, falhas estas que só serão detectadas através do seu uso efetivo, ou através de testes simulando esse uso. E existem também erros encontrados durante o desenvolvimento do programa que só são solucionados depois de uma extensa bateria de testes sobre o fluxo de execução do programa. São erros muito difíceis de se encontrar apenas com a lógica, o programador precisa ver o funcionamento do software até o momento da ocorrência do erro, para então tentar entendê-lo.
Mas é justamente nessa fase tão importante que reside um grande problema: a fase de testes é uma das mais difíceis de ser organizada, uma vez que elaborar testes bem-feitos pode ser tão difícil quanto construir o próprio programa. Muitos testes são feitos de maneira desorganizada e aleatória, sem planejamento algum.

Conseguimos identificar um procedimento muito utilizado por diversos colegas de BCC (e, é claro, por nós mesmos) para testar seus programas. Quase que a totalidade dos que fazem uso dessa prática reconhece que a mesma é ineficiente, trabalhosa, totalmente deselegante, e que não deveria, de forma nenhuma, se tornar um padrão de testes. Porém, nem por isso a abandonam!

A "técnica" consiste no seguinte: espalhar uma quantidade enorme de mensagens de erro na tela (através de *printf*´s, ou System.out.println´s, ou qualquer equivalente em outra linguagem. Vou adotar os *printf´s* no resto do texto como uma homenagem, já que foi em C que primeiro tomamos contato com essa "prática" ☺), cada uma com um texto diferente, antes e depois de variáveis, dentro de laços, antes e depois de atribuições, dentro e fora de funções e métodos, e em qualquer outro lugar que evidencie, através de mensagens impressas na tela durante o decorrer da execução do programa, o fluxo de execução que o mesmo está seguindo.
Dessa forma, pode-se verificar em que ponto o valor de uma variável se tornou nulo, em que laço o programa entrou em *loop infinito*, qual a função ou método é responsável por aquela falha de segmentação, ou mesmo garantir que o programa, depois de pronto, está efetivamente funcionando em todas as variações possíveis.

Os *printf´s* servem como alarmes, plantados pelo programador ao longo do programa. Conforme o mesmo vai sendo executado, esses "alarmes" vão soando, indicando que parte do código do programa está sendo executada naquele momento. Quando um erro ocorre, ou a execução do programa é abortada, sabe-se o que o programa estava fazendo pelas mensagens de erro exibidas na

tela. Isso ajuda o programador a localizar o erro, exatamente como fazem os alarmes.

A idéia não parece tão ruim à primeira vista: localizar o erro para então atacá-lo. Porém, pode-se facilmente perceber que é péssima: além de se ter o penoso trabalho de remover todos os *printf´s* espalhados pelo programa, executando-o sucessivas vezes só para se ter certeza de que não há nenhuma "mensagem de teste" esquecida, os erros não são facilmente encontrados, fazendo com que o número de *printf´s* cresça de maneira exponencial com o tamanho do programa. Além disso, ela pode levar a conclusões equivocadas: pode-se achar que a causa de um *loop infinito* é um erro de lógica, quando na verdade é apenas uma atribuição falha, ou que uma falha de segmentação é causada por uma função, quando na verdade o erro está no parâmetro que está sendo fornecido à ela, entre outros.

Anecdotal Evidence

" - AAAHH!!! E agora? Já pensei em todas as possibilidades e não consigo achar a razão dessa falha de segmentação!!! Acho que ela está sendo causada por algum parâmetro nulo em alguma função, mas não imagino qual! A lógica do programa parece tão perfeita!
- Mas isso é fácil de resolver! Façamos o seguinte: vamos colocar mensagens de erro antes e depois de cada uma das 3387438 variáveis que estamos usando, antes e depois da chamada de cada um dos 5453 métodos, e dentro de todos os 343487 laços. Como colocaremos um texto diferente em cada uma das mensagens, saberemos exatamente em que ponto da execução do programa está ocorrendo o problema! Aí, só precisaremos pensar um pouco sobre as causas dele, e então resolve-lo! Muito fácil... "

Sintomas e Consequencias

1. A execução do programa está recheada de mensagens do tipo: "Entra no laço1", "Até aqui blz", "Passou"...
2. A versão final de um programador distraído contém uma das mensagens citadas acima, e a mesma ocasionamente aparecerá, sem estar prevista, na execução do programa, ...
3. O programador, após dizer que seu programa já está pronto e testado, dizer que ainda faltam "umas coisinhas" (retirar todos os *printf´s*...
4. O mesmo programador, na mesma situação acima, voltar dizendo que apareceram novos erros no seu programa (provavelmente causados por uma linha acidentalmente apagada no processo de remoção dos *printf´s*).
5. O código do programa fica extremamente poluído e a identação é destruída, até a "limpeza" do mesmo.

Causas Típicas

O anti-padrão ocorre principalmente em programadores iniciantes, mas também é largamente difundido entre os mais experientes. As razões para que ele aconteça são, principalmente, preguiça em aprender a usar um *debugger* para uma determinada linguagem, ansiedade em ver até que ponto o programa está funcionando, e a falsa impressão de que é um método pouco trabalhoso: basta colocar algumas mensagens de erro para se ter um controle do que está acontecendo com o

programa. As pessoas esquecem que vão ter de remover todas essas mensagens (procedimento que, não raro, ocasiona ainda mais erros).
Outro fator é a alta "portabilidade" do método: ele pode ser usado em qualquer linguagem, seja ela procedural ou orientada a objetos. Basta que a mesma implemente a impressão de mensagens na tela, o que é uma funcionalidade básica de qualquer linguagem de programação.

Exceções Conhecidas

O uso do anti-padrão pode até ser aceitável no caso de já ser ter uma idéia da causa do erro, para confirmar uma hipótese. O uso porém precisa ser bem restrito, e não se propagar pelo programa inteiro.
Outra situação é aquela em que se está aprendendo uma linguagem nova, ainda não se conhece nenhum programa de *debug*, e não se tem tempo para aprender nenhum. Ainda assim, deve-se fazer ressalvas: pode-se perder muito mais tempo testando o programa dessa forma do que para aprender a se usar um *debugger* e testar o programa de forma decente.

Solução Refatorada

As únicas soluções seriam usar um bom programa de debugação e criar programas de teste que "exercite" todas as partes do programa principal, ou que sejam específicos para ajudar na resolução de um determinado erro.
Nas linguagens C/C++, existe uma outra solução que ameniza uma das desvantagens do método é introduzir um "interruptor", ou um modo de testes, ao programa. Para isso, usa-se o comando *ifdef*, que, aliado a uma variável (teste, por exemplo), indicará se o programa está ou não em teste, ativando e desativando os *printf´s*. Isso eliminaria o trabalho de ter que remover todos os *printf´s* após o término dos testes, (bastaria desativá-los, comentando-se ou removendo-se o *define* teste) mas o código do programa ficaria ainda mais poluído, pois, além dos *printf´s,* teríamos um igual número de *ifdef´s* espalhados pelo mesmo. Exemplo:

```
 // define teste
.
.
.
i = 0;
ifdef teste
printf ("%d\n",i);
```

Exemplos

Exemplo básico de utilização:

```
.
.
.
printf ("%d\n",i);
```

```
i = funcao_qualquer()
printf ("%d\n",i);
.
.
.
```

Exemplos de casos em que o método é ruim:

1 - Mensagens de teste "escondidas", de rara execução, podem passar desapercebidas pelo programador, e só aparecerem depois do programa estar em uso:

```
.
.
.
i = gera_numero_aleatorio(semente);
if (i == 903894038043)
{
      printf("Xiii... O valor da função é muito complicado...\n");
      i = 1;
}
```

Na maioria das vezes, esse trecho de código vai passar desapercebido, mas se ele estiver num programa muito grande, há grandes chances de o programador se esquecer de retirá-lo. Nesse caso, a mensagem só vai ser exibida se, em uma das execuções do programa, a função retornar o valor indicado. É difícil, mas não impossível...

2 - Printf´s "enganosos" - Pela falta de precisão oferecida pelo método, o programador pode ser "enganado", perdendo muito tempo para achar o erro verdadeiro:

```
.
.
.
int calculo_complexo(double m)
{
      double i = 0;
      i =  m / 3 + 34343.43 * sqrt(54433) - m * m;
      if (i == NULL) printf("Erro!!! \n");
      else
            return i;
}
```

Há uma grande probabilidade de um programador mais distraído se esquecer de testar o valor do parâmetro m, e achar que um possível erro está no cálculo de i. Se m for nulo, por exemplo, a mensagem de erro será impressa, não importando se a expressão para o cálculo de i esteja correta.

3 - Devido à (falta de) estrutura de alguns testes, alguns erros podem passar despercebidos com o *printf debuggin*:

```
if (1 > 2)
{
```

```
.
.
.
      if (i == NULL)
            printf("Erro!!! i é nulo!!!\n");
.
.
.
}
```

No exemplo acima a mensagem de erro nunca vai ser mostrada, não importando se i é ou não nulo, porque o fluxo de execução do programa nunca vai atingi-la, pois a mesma está dentro de um comando condicional que nunca será satisfeito (if (1 > 2)). Assim, o programador pode achar que sua função está correta (i != NULL), quando na verdade esse teste não é nem realizado.

4 - Programas poluídos: o uso do *printf debuggin* é vicioso: há cada vez mais mensagens de erro espalhadas pelo programa, de forma que em níveis mais avançados de "depuração", nem mesmo o dono do programa o entende direito, tamanha a poluição:

```
certo = 0;
a = 2;
for (i = 0; i < a + 45; i++) {
      b = 3 + a + i;
      printf("b => %d\n", b);
      c = i - 2 + a;
      printf("c => %d\n", c);
}
if (c == 45) {
      certo = 1;
      d = i + b;
      printf("d => %d\n", d);
}
printf("certo => %d\n", certo);
```