Padrão Depurador

André Gustavo Andrade
Rodrigo Moreira Barbosa

October 2, 2002
Versão 0.1

Abstract

Em todas as aplicações que desenvolvemos, sempre necessitamos, em um dado momento, depurar nosso código. Este padrão visa a tornar essa tarefa um pouco mais simples e organizada, sem contudo perder flexibilidade e configurabilidade.

1 Exemplo

Imagine uma certa classe de uma aplicação recentemente desenvolvida por você. A partir de um certo momento, depois de depurar esta classe razoavelmente bem, você se convence de que não há erros, e portanto, remove aquele monte de *System.outo.println*'s da classe.

Depois de seis meses de uso (e conseqüentemente seis meses de desgaste da sua memória), você infelizmente descobre que existe uma entrada para a qual sua classe não funciona.

Pior: você não sabe **exatamente onde** no meio de todo aquele código de seis meses atrás está o defeito.

Se esse fenômeno já pode ser um desastre restrito a uma classe isolada, imagine o que pode acontecer se você não puder determinar com certeza em qual classe o erro se deu?

Este caso é por si só desesperador, porque não apenas exige uma nova etapa de depuração, como a reinserção dos códigos apagados, e portanto, duplicação de trabalho.

Por outro lado, deixar as mensagens de depuração para sempre dentro do código implica uma péssima estética para o usuário, que ocasionalmente vai ver passar à sua frente um monte de mensagens que não fazem o menor sentido para ele (e ocasionalmente, fazendo-o ignorar mensagens realmente importantes as quais ele deveria ver).

Este é um exemplo típico de quando devemos usar o padrão *depurador*. Quando estamos desenvolvendo aplicações grandes e complexas que necessitem de manutenção ou expansão constantes.

2 Contexto

Desenvolvimento de aplicações que necessitem de constante depuração ou aplicações que necessitem de um sistema de registro (*log*).

3 Problema

Aplicações quando são inicialmente programadas necessitam de depuração. Muitas pessoas tem um pouco de aversão a usar depuradores, e isso por alguns motivos:

- Muitas vezes é difícil analisar as mensagens mostradas pelo depurador, bem como seguir a pilha inteira para saber onde está a origem do erro.
- Estabelecer *break points* pode ser cansativo, e executar o programa até onde um *break point* esteja inserido, por exemplo dentro de um laço, pode ser extenuante.
- Apesar dos avanços feito na área de depuração, com a criação de interface gráfica para os depuradores mais conhecidos, ainda assim, lidar com esses softwares exige o aprendizado de uma nova ferramenta

Por esses e outros motivos, o que se faz em geral é imprimir algum tipo de mensagem de depuração no meio do código, muitas vezes exibindo valores de variáveis, e outras informações importantes.

Esse método tem a vantagem de ser **persistente**, enquanto a depuração com um depurador é sempre nova a cada seção.

Por outro lado, algumas vertentes veêm alguns problemas com essa técnica, a saber:

- Ha poluição do código com comandos de impressão de mensagens. Nós não consideramos isso uma poluição do código, uma vez que se você ao ler o código possa saber o que exatamente o programa está fazendo, sem ter de recorrer a análise de linhas e linhas de programa, então essas mensagens também funcionaram como documentação
- Quando o software estiver pronto, as mensagens de depuração devem ser removidas, o que acarreta em um novo trabalho. Quando não removidas, devem ser ao menos comentadas.

4 Solução

A proposta apresentada pelo padrão Depurador é simples, e se baseia no seguinte paradigma:

- Manter as mensagens de depuração, podendo facilmente habilitar e desabilitar as mesmas.
- Permitir depuração tanto por classe quanto por característica funcional.
- Permitir vários tipos de saída, bem como a especificação e a implementação de novos tipos de saída, além de configurações.

Baseado nesses princípios surgiu o padrão Depurador.

5 Estrutura

A estrutura básica do padrão pode ser observada na Figura 1.

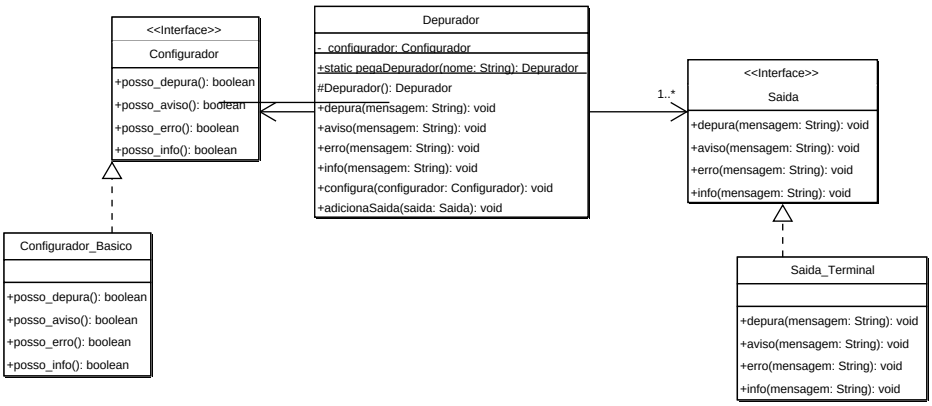

Figure 1: Diagrama de classes
A figura pode não ficar muito legível, exigindo zoom. Para maior clareza, existe o arquivo fig1.ps com a mesma em tamanho maior.

Assim, o padrão é constituído de duas interfaces e três classes:

5.1 Configurador

Esta é a interface responsável por definir os Configuradores. Configuradores são objetos que são capazes de determinar quais dos diferentes tipos de mensagens existentes podem ser impressas.

Deixá-lo apenas como uma interface possibilita ao usuário implementar diferentes tipos de configuradores, podendo inclusive serem alterados dinâmicamente ou até mesmo remotamente, através de chamadas CORBA, por exemplo.

5.2 Saída

Esta interface descreve pra onde as mensagens de cada tipo devem ir. Em geral, todas vão para o mesmo dispositivo, sendo que estes podem ser livremente programados pelo usuário, como arquivos, terminais, redes, etc. Mas além disso, permite que se programe a saída para mandar as mensagens de cada tipo para dispositivos diferentes.

5.3 Configurador_Basico

Responsabilidades: É um configurador padrão, para o caso do usuário não puder (souber) desenvolver um novo configurador. Tem como responsabilidade configurar o sistema para imprimir todas as mensagens.

5.4 Saida_Terminal

Responsabilidades: Saida padrão. Tem como responsabilidade imprimir no terminal todas as mensagens.

5.5 Depurador

Responsabilidades: Devolver a instância certa do depurador, controlar o destino de cada mensagem enviada.

Colaboradores: Configurador, Saida.

6 Dinâmica

No método a ser depurado, uma chamada `Depurador pegaDepurador (id)` é realizada. *id* é um identificador único para cada depurador. Se duas chamadas com o mesmo id forem feitas a `pegaDepurador`, a mesma instância de `Depurador` é devolvida (algo semelhante ao padrão **singleton**). De posse do objeto concreto da clase `Depurador`, faz-se uma instanciação de um objeto que implementa a interface `Configurador`, esse objeto pode ser da classe `Configurador_Basico`, por exemplo. Esse objeto é associado ao objeto `Depurador` através do método configura. O objeto do tipo `Configurador` determina se uma mensagem de um dado tipo (depuração, erro, informação, aviso) deverá ser considerada ou não. Em seguida, instancia-se um objeto que implementa Saida, que pode ser o `Saida_Terminal`. Esse objeto implementa os métodos de exibição de mensagens de depuração, aviso, erro, e informações gerais. Esse objeto `Saida`, é associado a `Depurador` através do método `adicionaSaida`. Vale notar, que vários objetos Saida podem ser associados, o que implica várias formas diferentes

de se tratar um dado tipo de mensagem. Feitas essas configurações, toda vez que o programador quiser exibir uma mensagem qualquer para um dado método, basta chamar o método `pegaDepurador` com o id correto e de posse do objeto `Depurador` usar os métodos `depura, aviso, erro, info` para exibição de mensagens de depuração.

7 Implementação

```
/* Exemplo de codigo */
public class Depurador {

    static Hashtable d = new Hashtable ();
    Set saidas;
    Configurador conf;

    private Depurador ();
    public static void pegaDepurador (string s) {                    pegaDepurador
        if d.containsKey (s) return d.get (s);                        10
        else {
            Depurador aux = new Depurador ();
            d.put (s, aux);
            return aux;
        }
    }

    public void adicionaSaida (Saida s) {                             adicionaSaida
        saidas.add (s);
    }                                                                 20
    public void configura (Configurador c) {                          configura
        conf = c;
    }
    public void depura () {                                           depura
        if (conf.possoDepurar () == true) {
            Iterator i;
            i = saidas.iterator ();
            for(; i.hasNext (); i = i.next) {
                saida.depura ();
            }                                                         30
        }
    }
    public void aviso () {                                            aviso
        if (conf.possoAviso () == true) {
            Iterator i;
            i = saidas.iterator ();
            for(; i.hasNext (); i = i.next) {
                saida.aviso ();
            }
```

```
        }                                                                    40
    }
    public void erro () {                                                    erro
        if (conf.possoErro () == true) {
            Iterator i;
            i = saidas.iterator ();
            for(; i.hasNext (); i = i.next) {
                saida.erro ();
            }
        }
    }                                                                        50
    public void info () {                                                    info
        if (conf.possoInfo () == true) {
            Iterator i;
            i = saidas.iterator ();
            for(; i.hasNext (); i = i.next) {
                saida.info ();
            }
        }
    }
}                                                                            60

public class ConfiguradorBasico extends Configurador {
    boolean depura, aviso, erro, info;
    depura = aviso = erro = info = true;
    void possoDepura () {                                                    possoDepura
        return depura;
    }
    void possoAviso () {                                                     possoAviso
        return aviso;
    }                                                                        70
    void possoErro () {                                                      possoErro
        return erro;
    }
    void possoInfo () {                                                      possoInfo
        return info;
    }
}

public class SaidaTerminal extends Saida {
    void depura (String s) {                                                 80 depura
        system.out.println (s);
    }
    void aviso (String s) {                                                  aviso
        system.out.println (s);
    }
    void erro (String s) {                                                   erro
```

```
        system.out.println (s);
    }
    void info (String s) {                                          info
        system.out.println (s);                                     90
    }
}

class Principal {
    public static void main () {                                    main
        Depurador d;
        Configurador c;
        Saida s;
        Depurador pegaDepurador ("main");
        c = new ConfiguradorBasico ();                              100
        s = new SaidaTerminal ();
        d.adicionaSaida (s);
        d.configura (c);
        pegaDepurador ("main").depura ("mensagem de depuracao");
    }
}
```

8 Usos conhecidos

Depois da formulação deste padrão, em pesquisas na Internet, constatamos que uma variação dele foi usada no framework *log4j* do projeto jakarta do Apache.

9 Conseqüências

9.1 Vantagens

As vantagens do padrão são claras: há uma maior padronização da depuração, além de uma maior versatilidade na exibição de mensagens, com a eliminação fácil de mensagens idesejáveis. Além disso, com a utilização desse padrão, a inserção de um log na aplicação torna-se trivial.

9.2 Desvantagens

Entre algumas desvantagens, podemos citar, obviamente, a necessidade de seu aprendizado. Além disso, sua complexidade é maior do que simplesmente inserir-se `system.out.println`'s . Em aplicações muito simples, tal padrão acaba sendo inadequado.

10 Veja também

Seria interessante a revisão de padrões como Singleton(130) e Strategy(292), que aparecem (meio disfarçadamente) neste padrão.