UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

HÉLIO BRAUNER - 180182
JEFFERSON STOFFEL - 180685
JOÃO GROSS - 180171

PADRÕES DE IMPLEMENTAÇÃO

Trabalho da Disciplina Técnicas de
Construção de Programas

Prof. Marcelo Soares Pimenta

Porto Alegre, 26 de setembro de 2011.

1 MOTIVAÇÃO

Para motivar o estudo de padrões de implementação, são apresentados alguns pensamentos importantes:

- Os programadores deveriam se focar em realmente desenvolver soluções para resolver o problema, ao invés de ficar muito tempo se preocupando com o trabalho que tem para fazer.
    - Um programa, enquanto 'vivo', é mais vezes lido do que escrito.
    - Nenhum programa está pronto. Muito mais dinheiro será investido modificando o programa do que na etapa inicial de desenvolvimento.
- Padrões agilizam o desenvolvimento, pois não temos que 'inventar a roda' a cada problema que encontramos.
    - Os padrões também ajudam o desenvolvedor a ter mais tempo e a mente livre para se preocupar com outros assuntos, que ainda não tem solução, pois para os que já se tem solução utilizamos um padrão.
    - Padrões servem para ajudar humanos nas suas decisões.
    - Padrões são uma forma rápida, barata e eficiente de resolver problemas. E deixam mais tempo para a criatividade do programador.
- Padrões: descrevem o que fazer.
- Comunicação: Não devemos programar como se humanos não existissem, isso morreu há décadas. O código deve ser fácil de entender, usar e modificar. Um programa deve ser fácil de ler assim como um livro.
- Tempo é dinheiro. Um código difícil de ler demanda muito tempo de compreensão, logo se perde dinheiro. Com um código simples se ganha dinheiro e produtividade.
- Códigos simples geram menos stress, e fazem o cérebro trabalhar melhor. Assim, soluções podem ser encontradas com mais facilidade.
- Simplicidade: um código mais simples é mais fácil de entender e se for menos complexo, valor é agregado ao programa.

2 PADRÕES

2.1 Classe - Value Object

O conceito do padrão de implementação Value Object é de que “nada se transforma, tudo se cria”. Quando um objeto é criado os valores a ele associados em sua criação tornam-se imutáveis, ou seja, nenhuma operação feita sobre este objeto pode alterá-lo, podendo o objeto ser visto como um valor. Isso previne que os valores presentes no objeto sejam alterados, garantindo confiabilidade ao sistema. Já, no caso de ser necessário um novo estado do objeto, como ele não pode ser alterado, novos objetos são criados, com valores diferentes.

As operações realizadas pelo programa sempre geram novos objetos e isto pode se tornar um problema, pois um uso extensivo desse padrão pode sobrecarregar a memória do sistema, perdendo performance.

2.2 Classe – Simple Superclass Name

As classes são conceitos centrais de design. Por isso, é importante que seus nomes sejam bem escolhidos. Uma vez que as classes estiverem bem nomeadas, o nome das operações surge quase que automaticamente. A dificuldade em nomear classes está na tensão entre brevidade e expressividade. Os nomes precisam ser curtos e enérgicos. No entanto, às vezes, nomes precisos requerem várias palavras. Encontrar bons nomes pode levar tempo.

2.3 Estado: Lazy Initialization

Há aqueles que o chamam de ‘Filosofia do baiano’: só se mexer quando algo precisa ser feito, ou seja, apenas quando um bloco de código é executado, e este usa uma certa variável, é que se faz a inicialização da variável.

Mas porque fazemos isso? Pois algumas inicializações são muito caras e, caso a variável não seja usada, a inicialização não precisa ser feita.

Este padrão é importante para sistemas com poder de processamento restrito ou para aplicações que exigem processamento pesado, visto que o objetivo da Lazy Initilization é basicamente performance.

2.4 Comportamento: Guard Clause

É uma forma local e simples de expressar situações excepcionais com conseqüências locais.

Para aplicar o Guard Clause remove-se as cláusulas ‘else’ do código e ‘ifs’ encadeados. Assim o programador ao ler o código, não precisa fazer mentalmente uma pilha de cláusulas ‘else’ ou de ‘ifs’. Sua mente fica mais limpa, podendo ele se preocupar com outras coisas.

O Guard Clause é um padrão que vale a pena implementar. Ele tem uma idéia simples e apresenta resultados imediatos de clareza e manutenção do fluxo de execução do programa.

2.5 Método: Creation

Os primeiros programas eram massas indecifráveis de código e dados. O fluxo de execução podia ir de qualquer lugar para qualquer lugar e os dados podiam ser acessados de qualquer lugar. Eram rápidos e eficientes.

O problema surgiu quando se percebeu que os programas eram escritos mais para serem modificados do que executados. Todas as permissividades que tornavam os programas eficientes também os tornavam dificílimos de serem modificados.

Começou-se então a estudar modelos de computação em que mudanças não gerassem um efeito dominó em todo o código.

Uma das primeiras estratégias para fazer programas fáceis de modificar foi dividir um grande computador rodando um grande programa em um monte de computadores menores, ou seja, objetos, rodando programas pequenos. Foi então mudada a forma como os programas passaram a ser criados.

Programas pequenos são geralmente mais fáceis de modificar que programas grandes. Essa subdivisão é para propósitos humanos – para o computador não há diferença em como o código foi implementado.

3 IMPLEMENTAÇÃO

O padrão escolhido para a implementação foi o Guard Clause. O grupo considerou este o padrão mais recomendado para ser implementado, visto a sua simplicidade, aliado a resultados imediatos.

Primeiramente, é apresentado um trecho de código de um programa para login de usuários em um site, sem fazer uso do padrão de comportamento Guard Clause. O código foi implementado em php.

```php
<?php

//Sem guard clause
if(!empty($username) && !empty($password)) {
    $forumMembersInfo = $forum->getMembersInfo($username);

    if( !empty($forumMembersInfo['username']) &&
        !empty($forumMembersInfo['members_pass_hash']) &&
        !empty($forumMembersInfo['members_pass_salt']) ) {

        $hash = md5( md5( $forumMembersInfo['members_pass_salt'] ) . md5( $password ) );

        if( strcmp($hash, $forumMembersInfo['members_pass_hash']) == 0 ) {
            session_start("seguranca");
            $_SESSION['loggedUser'] = $username;
            echo "<script>document.location='index.php'</script>";
            exit;
        } else {
            echo "<script>alert('".utf8_decode("Usuário")." ou senha incorretos.
            Tente novamente.');document.location='login.php'</script>";
            exit;
        }
    } else {
        echo "<script>alert('".utf8_encode("Usuário")." ou senha incorretos.
        Tente novamente.');document.location='login.php'</script>";
        exit;
    }
} else {
    echo "<script>alert('Preencha os dois campos.');document.location='login.php'
    </script>";
    exit;
}
```

Figura 1: trecho de código sem Guard Clause

Neste exemplo podemos ver que a cada cláusula afirmativa o fluxo do programa é desviado e vai ficando cada vez mais restrito. O exemplo acima considera poucas cláusulas, mas considere um sistema mais complexo, como uma validação de login no site de um banco, a falta deste padrão poderia se tornar um problema em manutenções futuras do código.

Agora, o mesmo trecho, utilizando o padrão Guard Clause:

```php
//Com Guard Clause
if(empty($username) || empty($password)) { //testa se o usuário digitou os dados
    echo "<script>alert('Preencha os dois campos.');document.location='login.php'
    </script>";
    return;
}
$forumMembersInfo = $forum->getMembersInfo($username);

if( empty($forumMembersInfo['username']) ||
    empty($forumMembersInfo['members_pass_hash']) ||
    empty($forumMembersInfo['members_pass_salt']) ) {
    //testa se o usuário e a senha constam no banco de dados
    echo "<script>alert('".utf8_encode("Usuário")." ou senha incorretos.
    Tente novamente.');document.location='login.php'</script>";
    return;
}

$hash = md5( md5( $forumMembersInfo['members_pass_salt'] ) . md5( $password ) );

if( strcmp($hash, $forumMembersInfo['members_pass_hash']) != 0 ) {
    //testa se a senha fornecida e a senha presente no banco de dados batem
    echo "<script>alert('".utf8_decode("Usuário")." ou senha incorretos.
    Tente novamente.');document.location='login.php'</script>";
    exit;
}

session_start("seguranca");
$_SESSION['loggedUser'] = $username;
echo "<script>document.location='index.php'</script>";

?>
```

Figura 2: trecho de código com Guard Clause

Neste trecho fica evidente o benefício do padrão. O fluxo não é desviado caso as cláusulas possuam o valor que desejamos e a leitura da lógica fica facilitada.

4 CONCLUSÕES

Os padrões são ferramentas importantíssimas na elaboração de um software de qualidade. Eles nos auxiliam a deixam o código mais claro, simples, objetivo, organizado e limpo.

A experiência do grupo com padrões de implementação é muito pequena, pois até então todos nós apenas havíamos utilizado convenções criadas por nós mesmos, sem pensar muito a respeito das conseqüências de eficiência, legibilidade ou manutenção do software.

Acreditamos que para programação os conceitos vistos neste trabalho serão extensivamente utilizador no decorrer de nossas carreiras profissionais e certamente poderão ser um diferencial caso apliquemos com precisão tais padrões.

5 BIBLIOGRAFIA

- Implementation Patterns – Beck, Kent – Capítulos 5 a 9 – Conforme disponibilizado em http://moodle.inf.ufrgs.br/mod/resource/view.php?id=31665 - Acessado entre os dias 15 e 24 de setembro