CURSO: Ciência da Computação	**DATA: ____/____/ 2013**
PERÍODO: 4º.	**PROFESSOR: Andrey**
DISCIPLINA: Técnicas Alternativas de Programação	**AULA: 12**

APRESENTAÇÃO

O objetivo desta aula é apresentar e discutir os conceitos de Refatoração e Padrões.

DESENVOLVIMENTO

PADRÕES

Os padrões de projeto de software ou padrões de desenho de software, também muito conhecido pelo termo original em inglês: *Design Patterns*, descrevem soluções para problemas recorrentes no desenvolvimento de sistemas de software orientado a objetos Um padrão de projeto estabelece um nome e define o problema, a solução, quando aplicar esta solução e suas consequências.

Os padrões de projeto visam facilitar a reutilização de soluções de desenho - isto é, soluções na fase de projeto do software, sem considerar reutilização de código. Também acarretam um vocabulário comum de desenho, facilitando comunicação, documentação e aprendizado dos sistemas de *software*.

Um padrão de projeto é uma estrutura estrutura recorrente no projeto de software orientado a objetos. Pelo fato de ser recorrente, vale a pena que seja documentada e estudada. Um padrão de projeto nomeia, abstrai e identifica os aspectos chave de uma estrutura de projeto comum para torná-la útil para a criação de um projeto orientado a objetos reutilizável.

Os principais atributos de uma boa descrição de um padrão de projeto são:

- Nome: referência que descreve de forma bastante sucinta o padrão.
- Problema (motivação, intenção e objetivos, aplicabilidade): apresenta o contexto do padrão e quando ele pode ser usado.
- Solução (estrutura, participantes, exemplo de código): descreve a solução e os elementos que a compõem.
- Consequências e padrões relacionados: analisa os resultados, vantagens e desvantagens obtidos com a aplicação do padrão.

Em geral os padrões de projeto podem ser classificados em três diferentes tipos:

- Padrões de criação: abstraem o processo de criação de objetos a partir da instanciação de classes.
- Padrões estruturais: tratam da forma como classes e objetos estão organizados para a formação de estruturas maiores.
- Padrões comportamentais: preocupam-se com algoritmos e a atribuição de responsabilidade entre objetos.

Padrões GRASP (General Responsibility Assignment Software Pattern)

- Especialista
- Criador
- Alta Coesão
- Baixo Acoplamento
- Controlador

Padrão Especialista

Problema: Qual é o princípio mais básico pelo qual as responsabilidades são atribuídas em um projeto Orientado a Objetos?

Solução: Atribuir a responsabilidade ao especialista da Informação – a classe que tem a informação necessária para resolver a responsabilidade.

Padrão Criador

Problema: De quem é a responsabilidade de criar uma nova instância de uma classe?

Solução: Atribuir à uma classe B a responsabilidade de criar uma instância de uma classe A se:

- B agrega objetos de A
- B contém objetos de A
- B registra objetos de A
- B usa muito de perto objetos de A
- B tem a informação para a inicialização dos objetos de A

Padrão Alta Coesão

Problema: Como manter a complexidade gerenciável?

Solução: Atribua a responsabilidade de tal forma que a coesão se mantenha alta.

Padrão Baixo Acoplamento

Problema: Como dar suporte à baixa dependência e aumentar o reuso?

Solução: Atribua a responsabilidade de tal forma que o acoplamento se mantenha baixo.

Padrão Controlador

Problema: Quem deve ser responsável pelo tratamento de um evento do sistema?

Solução: Atribua a responsabilidade de tratar o evento à uma classe que:

- represente o sistema como um todo
- represente a organização ou negócio
- represente algo no mundo real que seja ativo e que esteja envolvido na tarefa
- represente um tratador artificial de todos os eventos de um caso de uso.

Padrões GoF

Padrão Singleton

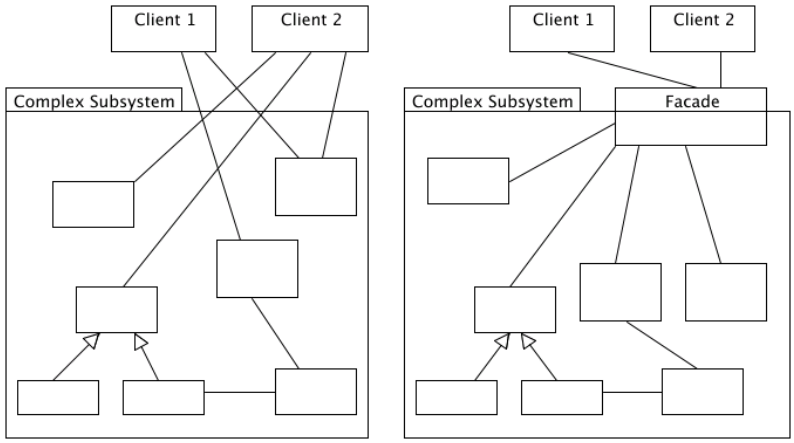
Garante que a classe tenha apenas uma instância e fornece um ponto global de acesso para a mesma. O padrão *Singleton* torna a classe responsável por manter o controle da sua única instância, garantindo que nenhuma outra instância seja criada, interceptando solicitações para criação de novos objetos, e por fornecer um meio de acesso a esta única instância.

Exemplo de Implementação:

```
public class Singleton {
    private static Singleton uniqueInstance;

    private Singleton() {
    }

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null)
            uniqueInstance = new Singleton();
        return uniqueInstance;
    }
}
```

Padrão *Facade*

Fornece uma interface unificada para um conjunto de interfaces em um subsistema. A figura demonstra a diferença entre um sistema sem fachadas, onde os clientes acessam os subsistemas da aplicação diretamente, e outro em que o padrão *Facade* define uma interface de nível mais alto que torna o subsistema mais fácil de ser usado.



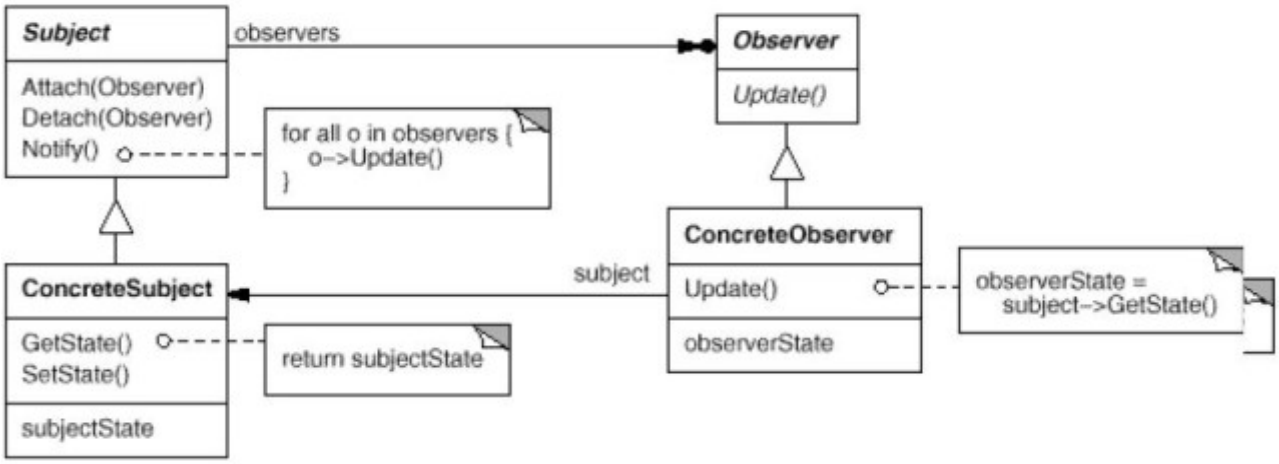
Exemplo de Implementação:

```
public class Cpu {

    public void start() {
        System.out.println("inicialização inicial");
    }
    public void execute() {
        System.out.println("executa algo no processador");
    }
```

```
    public void load() {
        System.out.println("carrega registrador");
    }
    public void free() {
        System.out.println("libera registradores");
    }
}

public class Memoria {
    public void load(int position, String info) {
        System.out.println("carrega dados na memória");
    }
    public void free(int position, String info) {
        System.out.println("libera dados da memória");
    }
}

public class HardDrive {
    public void read(int startPosition, int size) {
        System.out.println("lê dados do HD");
    }
    public void write(int position, String info) {
        System.out.println("escreve dados no HD");
    }
}

public class ComputadorFacade {
    private Cpu cpu = null;
    private Memoria memoria = null;
    private HardDrive hardDrive = null;

    public ComputadorFacade(Cpu cpu, Memoria memoria, HardDrive
hardDrive) {
        this.cpu = cpu;
        this.memoria = memoria;
        this.hardDrive = hardDrive;
    }

    public void ligarComputador() {
        cpu.start();
        String hdBootInfo = hardDrive.read(BOOT_SECTOR, SECTOR_SIZE);
        memoria.load(BOOT_ADDRESS, hdBootInfo);
        cpu.execute();
        memoria.free(BOOT_ADDRESS, hdBootInfo);
    }
}
```

Padrão Observer

Definir uma dependência um-para-muitos entre objetos, de maneira que quando um objeto muda de estado todos os seus dependentes são notificados e atualizados automaticamente. O padrão Observer descreve como estabelecer esta dependência. Os objetos chave neste padrão são subject e observer. Um subject pode ter um número qualquer de observadores dependentes. Todos os observadores são notificados quando o subject muda de estado.



Exemplo de Implementação:

```
import java.util.Observable;
import java.util.Observer;

public class RevistaInformatica extends Observable {

    private int edicao;

    public void setNovaEdicao(int novaEdicao) {
        this.edicao = novaEdicao;

        setChanged();
        notifyObservers();
    }

    public int getEdicao() {
        return this.edicao;
    }

    public static void main(String[] args) {
        //poderia receber a nova edicao atraves de um recurso externo
        int novaEdicao = 3;
        RevistaInformatica revistaInformatica = new RevistaInformatica();
        Assinante1 assinante1 = new Assinante1(revistaInformatica);

        revistaInformatica.setNovaEdicao(novaEdicao);
    }

}

class Assinante1 implements Observer {

    Observable revistaInformatica;

    int edicaoNovaRevista;

    public Assinante1(Observable revistaInformatica) {
        this.revistaInformatica = revistaInformatica;
        revistaInformatica.addObserver(this);
    }
```

```
    @Override
    public void update(Observable revistaInfSubject, Object arg1) {
        if (revistaInfSubject instanceof RevistaInformatica) {
            RevistaInformatica revistaInformatica = (RevistaInformatica)
revistaInfSubject;
            edicaoNovaRevista = revistaInformatica.getEdicao();
            System.out.println("Atenção, já chegou a mais uma edição da
Revista Informatica. " +
                    "Esta é a sua edição número: " + edicaoNovaRevista);
        }
    }
}
```

PADRÃO ITERATOR

Fornecer um meio de acessar, sequencialmente os objetos de uma coleção sem expro sua representação interna. Por exemplo, listar os elementos de uma lista encadeada sem a necessidade de manipular os ponteiros que encadeiam os elementos.

Exemplo de Implementação:

```
class MenuItem {
    String nome;

    MenuItem(String nome) {
        this.nome = nome;
    }

}

interface Iterator {
    boolean hasNext();
    Object next();
}

public class MenuIterator implements Iterator {
    MenuItem[] itens;
    int posicao = 0;

    public MenuIterator(MenuItem[] itens) {
        this.itens = itens;
    }

    public Object next() {
        MenuItem menuItem = itens[posicao];
        posicao++;
        return menuItem;
    }

    public boolean hasNext() {
        if (posicao >= itens.length || itens[posicao] == null) {
            return false;
        } else {
            return true;
        }
    }
}
```

REFATORAÇÃO

O código escrito é o documento mais preciso de seu sistema. Quanto mais legível, mais se pode saber sobre o sistema, mais pessoas podem dar manutenção e mais confiável é o sistema. Nesta aula, vamos aprender técnicas para deixar seu código legível. Vamos ver como diferenciar código ruim de código bom. E vamos aprender técnicas de melhorar código (refactoring).

O código é uma maneira de comunicação e deve deixar claro suas intenções. Isto é tão importante que você sempre deve codificar levando isto em conta.

Código duplicado ("a doença do copiar e colar")
Em informática, tenha como princípio que uma informação deve ter um único endereço. Deve estar em um único lugar e fácil de achar. Duplicação de informação, leva a falta de sincronismo das cópias e a confusão em achar a informação. Código é informação.
Códigos duplicados podem aparecer em um mesmo método, em métodos diferentes e em classes diferentes.
O que fazer: separe o código duplicado e crie uma função (refactoring Extract method). Se o código aparece em mais de uma classe, crie uma classe que represente este código.

Métodos longos
Programas vivem melhor, rodam melhor com métodos curtos. Se um método passou do tamanho de sua tela, ele já é longo. Você nunca deve ficar rolando a tela, fazendo anotações do lado para entender o método inteiro. Um método não deve fazer mais de uma atividade. Deve ter uma linha de raciocínio homogênea.
Para consertar um método longo, separe trechos que fazem uma única atividade e construa um método com ele (refactoring Extract method).

Classes muito longas
Classes devem se limitar a menor contexto possível. Este contexto pode ser uma entidade, um algoritmo. Uma classe não deve implementar mais de uma entidade ou algoritmo. É possível que classes representem mais de uma entidade (pattern FACADE), mas não implementem mais do que uma.
Para transformar uma classe grande em várias menores, isole as variáveis membros e métodos que são afins e criei uma nova classe. Se você consegue dar um nome adequado para esta nova classe, você deve estar indo para o caminho certo.

Métodos com muitos parâmetros
Se você passa 5, 6 ou mais parâmetros para um método, isto confunde. Um grande número de parâmetros indica algo que não foi modelado, uma entidade esquecida e oculta.

Passe um menor número de parâmetros possível para um método. Investigue se o método pode deduzir os parâmetros por si só.

Um saída bastante comum é criar uma classe que represente os parâmetros. Exemplo:

```
CadastrarPessoa( "Jose", "tito", 1, "555-1111" );

// melhor seria:

EnderecoPessoa endereco= new EnderecoPessoa();
endereco.rua= "tito";
endereco.numero=  1;
endereco.telefone = "555-1111";
CadastrarPessoa( "Jose", endereco );
```

Métodos em classes erradas
Fique atento se um método está mais interessado em propriedades e métodos de outra classe do que da sua própria. Em muitos casos, é melhor mudar o método de classe.

Dados que gostam de andar juntos
Você já viu aquelas variáveis que sempre andam juntas, quer em classes ou em parâmetros? Elas estão pedindo para virar uma classe. Atenda-as!
Um bom teste é: imagine que você tire uma delas, as outras perdem o sentido? Se sim, elas estão pedindo para ser um objeto.

Exemplo:
```
class Aluno
{
	String _Nome;
	String _Logradouro;
	String _Numero;
	String _Bairro;
}

//fica melhor:

class Endereco
{
	String _Logradouro;
	String _Numero;
	String _Bairro;
};

class Aluno
{
	String _Nome;
	Endereco _Endereco;
};
```

Classe de dados (data class)
Não deve ter atributos públicos, torne-os privados e crie get_ e set_ métodos. Se um atributo nunca deve ser alterado, só consultado, coloque só o método get_.

Expressões complexas
Não deixe expressões complexas soltas no código. Crie funções cujo nome digam o que a expressão faz.
Veja o seguinte código:
```
		if( ano % 4 == 0 && (ano % 100 != 0 || ano % 400 == 0) )
		{
```
não ficaria melhor assim?
```
		if( ehBisexto(ano) )
		{
		}
	/**
	 * Method ehBisexto.
	 * @param ano
	 * @return boolean
	 */
	private boolean ehBisexto(int ano)
	{
		return ano % 4 == 0 && (ano % 100 != 0 || ano % 400 == 0);
	}
```

Dar nomes a variáveis, classes e métodos
Sempre dê nomes que dispensem comentários. Você cria uma variável e dá a ela o nome de 'k' e depois comenta dizendo que ela é um contador. Por quê não a chama de 'contador'?
Não relute em mudar nomes, quando eles não estão claros o suficiente. Há editores (exemplo: eclipse) que já lhe dão ferramentas para mudar nomes e ele sai mudando tudo que referencia.

Não tente reutilizar uma variável local para mais de uma finalidade. Esta prática faz a variável ter um nome que não diz nada.

Comentários

Não é algo ruim, mas muitas vezes são colocados para melhorar um código ruim. Um comentário não deve dizer o que o código faz, o código deve dizer. Você pode usar um comentário para dizer por quê você tomou uma decisão em vez de outra.

Exemplo:

```
int CalcularArea()
{
      // l --> largura
      // c --> comprimento
      int l= calc1(x2,x1); // calc1 calcula a diferenca entre x2 e x1
      int c= calc1(y2,y1);
      return c * l;
}
```

ficaria melhor:

```
int CalcularArea()
{
      int largura= CalcularDiferenca(x2,x1);
      int comprimento= CalcularDiferenca(y2,y1);
      return comprimento * largura;
}
```

ATIVIDADE

1. Para cada um dos problemas relatados, escreva um exemplo onde ocorre o problema e mostre uma solução
2. O que é um Padrão?
3. Qual a vantagem de usar padrões de projeto?
4. Cite alguns padrões de projeto?
5. O que é o padrão facade?

BIBLIOGRAFIA BÁSICA

PRESSMAN, R. S.. *Engenharia de Software*. Makron Books. 1995

FOWLER, Martin - REFACTORING, Improving the design of existing code.

FOWLER, Martin e BECK, Kent - Bad Smells in Code - http://home.sprintmail.com/~wconrad/refactoring-live/smells.html

GAMMA, Erich et. al., Padrões de projeto: soluções reutilizáveis de software orientado a objetos. Porto Alegre: Bookman, 2000.

Booch, G.; Rumbaugh, J.; Jacobson, I.. UML guia do usuário. Editora Campus. 2000.

BEZERRA, E.. Princípios de Análise e Projeto de Sistemas com UML. Ed. Campus. 2003.