Sistemas de Informação 1

2010.2

Hyggo Almeida

Grupo:

Caio Paes
Carlos Artur
Catharine Quintans
Demontiê Júnior
Matheus Araújo

MILESTONE 2 (PADRÕES DE PROJETO)

Neste relatório explicaremos onde usamos os padrões aprendidos em sala de aula, definindo a necessidade de uso do padrão e explicando o ganho de flexibilidade que a sua utilização causa no projeto.

Camadas

O sistema é organizado em 3 camadas. No mais alto nível se encontra a camada de **serviço**, responsável pela comunicação com APIs de clientes. A segunda camada é a camada responsável pela **lógica** do sistema. A última camada contempla o **modelo** de representação do sistema.

Camada de Serviço:

- Classes que controlam o *Web Service* e a sessão de usuário, verificando se o mesmo está logado para utilizar o serviço.

Camada de Lógica:

- Responsável por criação de usuários, blogs, post, comentários.
- Responsável pela modificação e atualização dos elementos junto à persistência.
- Verifica se a requisição do cliente tem atributos válidos.

Camada de Modelo:

- Representa o modelo do sistema. (Blog, Post, UserIF, InteractiveContent…)

Model-View-Controller

O sistema utiliza o padrão MVC, pois dado que é voltado para a web, este padrão facilita a organização do projeto. Assim, permite que os clientes desenvolvedores implementem sua própria GUI *(View)*, utilizando a interface do *Web Service*.

Factory Method

Utilizamos o *Factory Method* no pacote *persistence*. Visando evitar a repetição de código para fazer a persistência dos objetos, criamos uma classe abstrata *AbstractDAO*, que implementa o CRUD de objetos utilizando o XStream (framework para persistência XML). A única diferença na persistência de um objeto para o outro é o seu *path* no banco de dados e a exceção lançada quando erros ocorrem.

Assim, cada objeto herda a clase *AbstractDAO* e apenas precisa indicar o caminho onde este deverá ser salvo.

```
public class BlogDAO extends AbstractDAO<Blog>{

    private final String BLOGS_PARENT_PATH = "resources" + SEP + "db" + SEP + "blogs" + SEP;
    private final String BLOGS_FILE_EXTENSION = ".blog";
    private final String EXISTENT_BLOG_MESSAGE = "Blog existente";
    private final String UNEXISTENT_BLOG_MESSAGE = "Blog inexistente";

    @Override
    protected File createFileReference(String fileName) {
        return new File(BLOGS_PARENT_PATH + fileName + BLOGS_FILE_EXTENSION);
    }
    @Override
    protected File createDirectoryReference() {
        return new File(BLOGS_PARENT_PATH + SEP);
    }
```

Figura 1. Exemplo de *Factory Method.*

Com a implementação desse padrão, reduzimos a repetição de código nas classes DAO e facilitamos o trabalho para alguma mudança na forma de persistência.

Singleton

O singleton é utilizado em 4 classes controladoras. Nas classes *UsersHandler, WebElementManager, DatabaseFacade, SessionManager.*

A intençao de permitir uma única instancia de cada uma dessas classes, é que as operação sobre os objetos como Blog, User, Post, Sound, sejam controladas em casos de uso sincronizado. Pois, muitas funcionalidades dependem do determinismo das ações com os elementos.

```
private WebElementManager() {
}

/**
 * Return the instance of the WebElementManager.
 * @return The instance of the WebElementManager.
 */
public static synchronized WebElementManager getInstance() {
    if (selfInstance == null) {
        selfInstance = new WebElementManager();
    }
    return selfInstance;
}
```

Figura 2. Exemplo de Singleton em *WebElementManager.*

Facade

Utilizamos o padrão *Facade* em dois pontos principais do sistema, na fachada para o banco de dados e para o *Web Service*. Foi decidido o uso de fachada no banco de dados, para evitar chamadas às classes DAO e assim proteger o uso indevido das mesmas. (Ex.: Usar a classe PostDAO para serializar um Blog.) O uso da fachada também é justificado para promover organização do código.

Observer

O padrão *observer* foi utilizado na funcionalidade de notificação de usuário. A classe Blog é a observadora e a classe UserImpl é a escutadora. Quando ocorre alguma mudança em um Post pertencente à um Blog, ele envia a notificação para todos os UserImpl que estão registrados.

```
public void addPostAnnouncement(String blogId, String userId) throws Exception {
    UserIF usr = DatabaseFacade.getInstance().retrieveUser(userId);
    Blog blog = getBlog(blogId);
    blog.addNotifiable(usr);
    DatabaseFacade.getInstance().updateBlog(blog);
}
```

Figura 3. Adicionando um escutador (UserImpl) a um observador (Blog).

```
/**
 * Notifies all the notifiers.
 */
public void doNotifyAll(String announcementId) {
    for (Notifiable usr : notifiables) {
        usr.receiveAnnouncement(announcementId);
    }
}
```

Figura 4. O Observador (Blog) notifica todos os escutadores (UserImpl).

Strategy

O padrão *Strategy* é utilizado, por exemplo, para que a persistência seja realizada para um objeto que implementa a interface *InteractiveContent*, como um *Sound*, ou *Picture*, ou *Movie*.

```
public String attachPictureOnPost(String postId, String description, String data) throws Exception {
    Post post = getPost(postId);
    if (isInvalidString(data)) {
        throw new IllegalArgumentException(INVALID_DATA_MESSAGE);
    }
    // Can be an empty data.
    description = description == null? "" : description;
    StaticContent pictureDescription = new Text(description);
    String pictureId = IdGenerator.getInstance().getNextId();
    InteractiveContent picture = new Picture(pictureId, postId, pictureDescription, data);
    DatabaseFacade.getInstance().insertInteractiveContent(picture);
    post.attachPicture(picture.getId());
    DatabaseFacade.getInstance().updatePost(post);
    return picture.getId();
}
```

Figura 5. A DatabaseFacade recebe um *InteractiveContent* instanciado como *Picture*.