UNIVERSIDADE FEDERAL DO RIO GRANDE - FURG
CENTRO DE CIÊNCIAS COMPUTACIONAIS - C3
ENGENHARIA DE COMPUTAÇÃO

PROJETO E DESENVOLVIMENTO DE SOFTWARE II

Relatório sobre a utilização de padrões de projeto em um sistema de compra de ingressos online

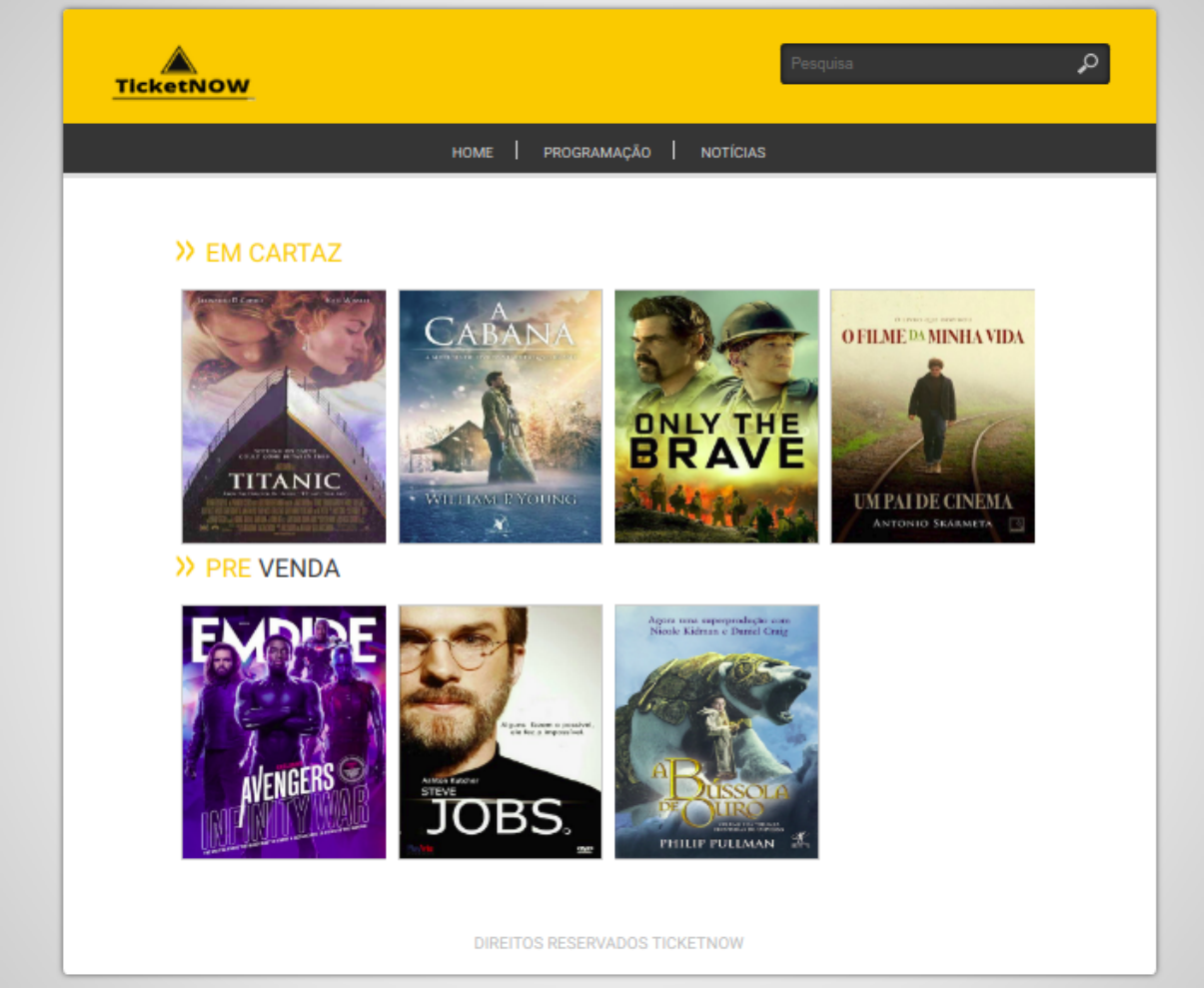
Marlon Franco, 59782
Vinicius Lucena, 85522

1. Introdução

Este trabalho descreve a implementação de um Sistema de Compras de Ingressos (simplificado), intitulado "*Ticketnow*", cujo o foco principal é a aplicação de alguns padrões de projeto. O sistema faz parte do trabalho avaliado da disciplina de Projeto e Desenvolvimento de Software II (Turma U) da Universidade Federal do Rio Grande - FURG, sob orientação do Prof. Dr. Odorico Mendizabal.

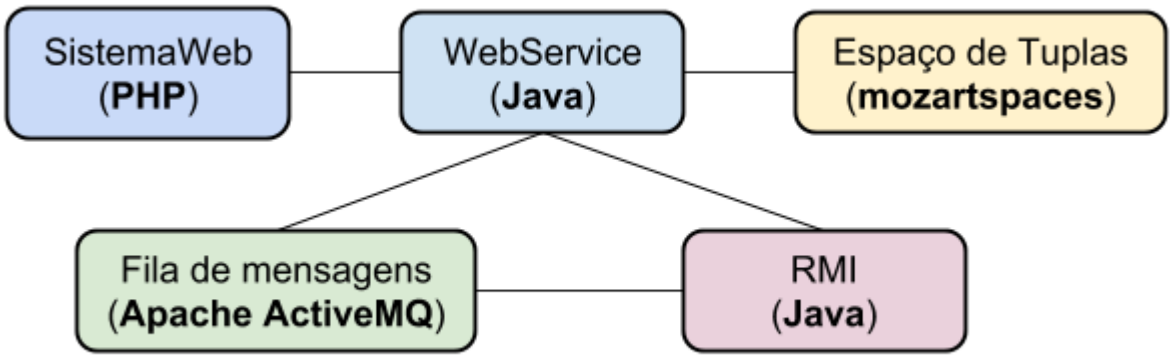
O objetivo do sistema é proporcionar um serviço de compra de ingressos de um cinema, onde os clientes possam comprar os assentos disponíveis, consultar os assentos, ou se inscrever nos filmes que ainda não estão em cartaz (padrão observer que será comentado em breve). A Figura 1 mostra a página inicial do sistema.

Figura 1. Página principal do sistema *TicketNow*.



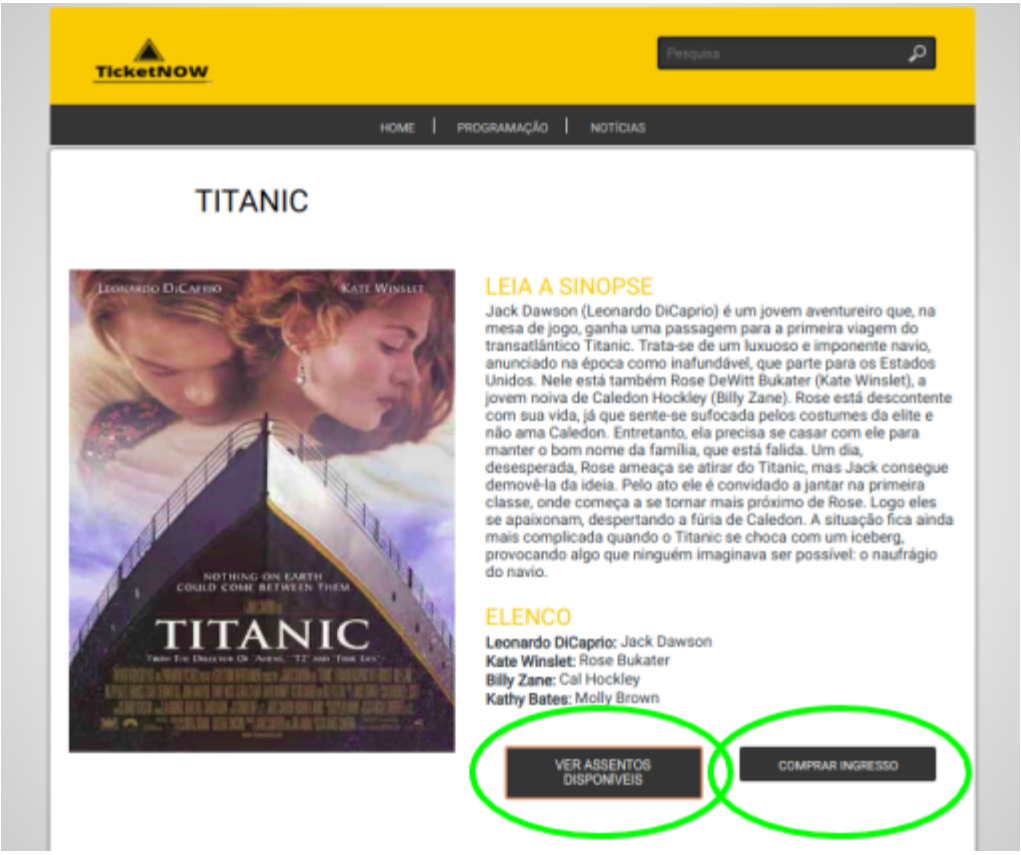
O sistema implementado é composto de cinco principais componentes, como ilustra a Figura 2.

Figura 2. Componentes do Sistema (visão geral).



O ***SistemaWeb*** é responsável por gerenciar as páginas Html que por sua vez realizam a interface com os usuários do sistema. Estas páginas invocam métodos do WebService para solicitar a compra de um determinado assento, ou a consulta dos assentos disponíveis, como ilustra a Figura 3.

Figura 3. Página de detalhes sobre o filme Titanic, com enfoque para os botões "Ver Assentos Disponíveis" e "Comprar Ingresso", que invocam métodos do WebService.



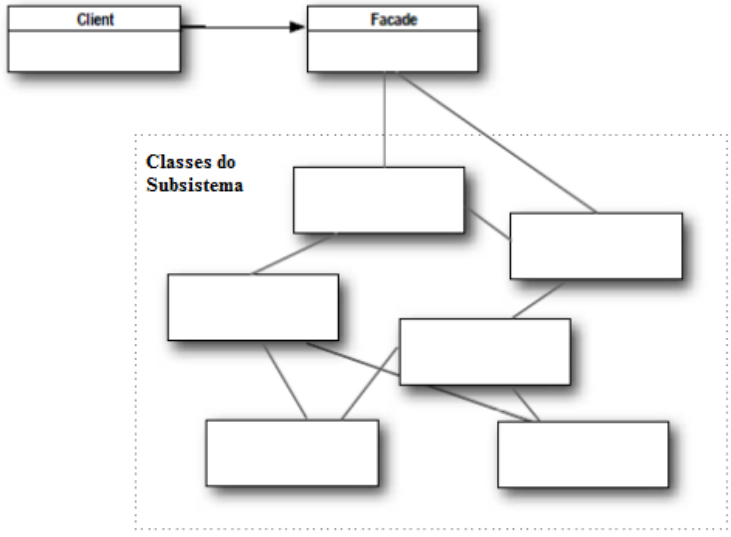
Quando qualquer um dos dois botões da página ilustrada na figura anterior é pressionado, uma requisição é enviada ao WebService, que compõe o segunda e talvez o principal componente do sistema. O WebService, implementado em Java, é responsável por acessar o terceiro componente (Espaço de Tuplas) retirando e inserindo tuplas (que neste caso são os próprios assentos) a fim de manter um registro dos assentos disponíveis a cada instante. Também é função do WebService, ao ser invocado o método de compra, verificar se o cartão de crédito informado pelo cliente é válido, verificação esta que é realizada pelo quarto componente (RMI). Após ser invocado pelo WebService, o RMI necessita dos dados do cartão a ser validado, para tanto ele busca estes dados numa Fila de Mensagens, o quinto e último componente. Lembrando que esta Fila de Mensagens possui duas filas apenas, a de "pedidos" (no caso os cartões a serem validados), e a de "concluídos" (que são os cartões já validados). A fila de "pedidos" é preenchida somente pelo próprio WebService, e quem consome os dados desta fila é o método remoto de validação de cartão de crédito (RMI). A fila e "concluídos" é preenchida apenas pelo RMI, após realizar a verificação.

Os padrões de projeto utilizados foram: **Factory, Singleton, Adapter, Observer e Facade**, sendo em sua maioria presentes nos componentes *SistemaWeb*, *WebService* e *Espaço de Tuplas*. Os detalhes sobre estes padrões, assim como a motivação da escolha de cada um são descritos a seguir.

2. Facade

"O Padrão Facade fornece uma interface unificada para um conjunto de interfaces em um subsistema. O Facade define uma interface de nível mais alto que facilita a utilização do subsistema" [1]. O Facade é um padrão **Estrutural**.

Figura 4. Diagrama de classe do padrão Facade.

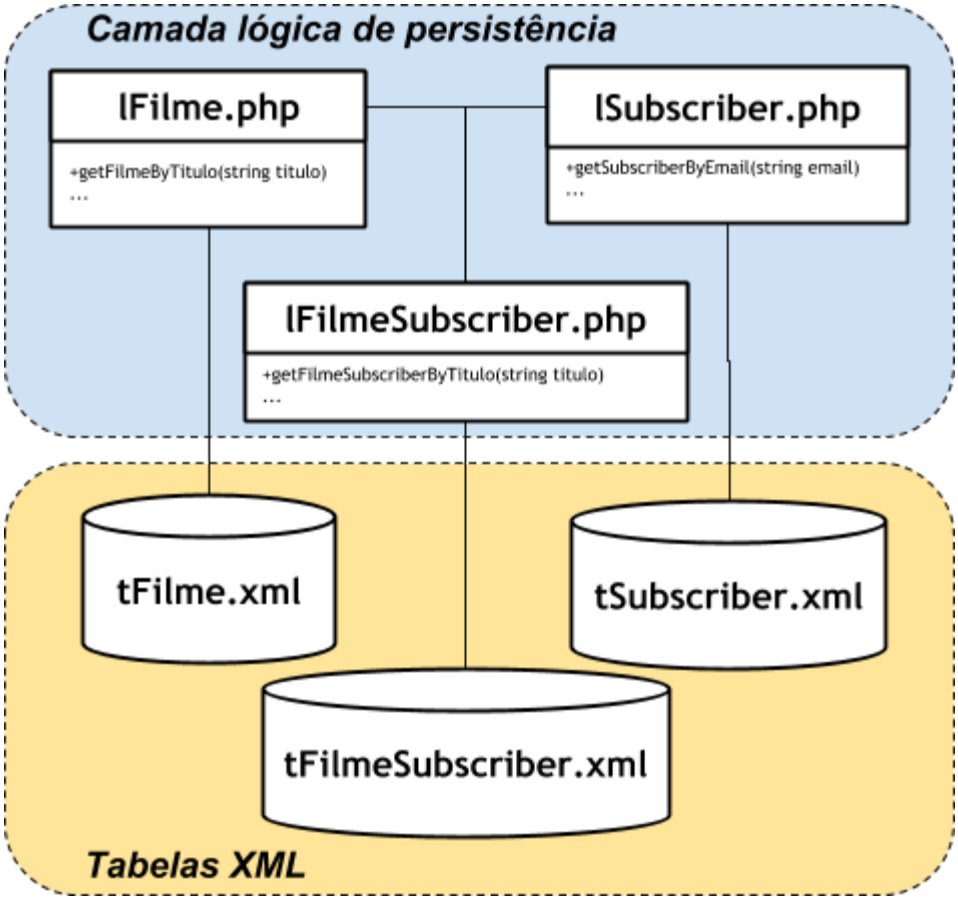


A Figura 4 mostra o diagrama do funcionamento do padrão Facade, onde a classe “Client” acessa as demais classes do Subsistema por meio da interface oferecida pela classe “Facade”, tornando transparente as invocações aos métodos das demais classes.

No contexto do trabalho, a interface oferecida pelo Facade é a interface oferecida pelo WebService ao cliente (PHP), onde a invocação dos métodos de “consulta cadeira” e “compra cadeira” ocultam ao usuário as interfaces internas do Espaço de Tuplas, RMI e Fila de Mensagens.

Outro local onde o Facade foi implementado foi na criação de uma camada de classes lógicas para acessar as tabelas .XML onde as informações de Filmes e Usuários Inscritos ficam salvas, como mostra a Figura 5.

Figura 5. Diagrama da camada lógica para acesso à persistência de dados.



Esta camada é composta de três principais classes que oferecem uma API para acessar as respectivas três tabelas do sistema:

- ***lFilme.php*** oferece uma interface para manipular a tabela ***tFilme.xml***;
- ***lSubscriber.php*** oferece uma interface para manipular a tabela ***tSubscriber.xml***;
- ***lFilmeSubscriber.php*** oferece uma interface para manipular a tabela ***tFilmeSubscriber.xml***;

Figura 6. Alguns métodos de consulta às tabelas XML, escondendo as funções de manipulação de arquivos XML.

```php
/**
 * getFilmeByTitulo
 *
 * Busca os Filmes que possuem o titulo informado e retorna um array com objetos da classe lFilme.
 *
 * @param string $titulo         Nome do Filme a ser buscado.
 * @return lFilme[] $ListaFilme  Retorna um array de objetos da classe lFilme.
 *
 */
public function getFilmeByTitulo(string $titulo) {
    $ListSimpleXMLObject = $this->selectFilme(null, $titulo);
    $ListaFilmes = $this->traduzSimpleXMLObjectToFilme($ListSimpleXMLObject);
    return $ListaFilmes;
}


/**
 * getFilmeByDescricao
 *
 * Busca os Filmes que possuem a descricao informado e retorna um array com objetos da classe lFilme.
 *
 * @param string $descricao     Descrição do Filme a ser buscado.
 * @return lFilme[] $ListaFilme Retorna um array de objetos da classe lFilme.
 *
 */
public function getFilmeByDescricao(string $descricao) {
    $ListSimpleXMLObject = $this->selectFilme(null, null, $descricao);
    $ListaFilmes = $this->traduzSimpleXMLObjectToFilme($ListSimpleXMLObject);
```

A motivação da criação desta camada lógica é ocultar as funções de manipulação de arquivos XML para as classes que precisam acessar a persistência de dados. Deste modo, quando alguma página PHP necessita acessar os dados de algum filme cadastrado, ela não precisa ter conhecimento de que estas informações estão salvas em arquivos XML, mas somente precisa invocar o método *getFilmeByTitulo(string Titulo)* e receber as informações desejadas. Esta separação permite que no futuro as tabelas XML sejam substituídas por tabelas em um Banco de Dados (ex MySQL) facilmente, apenas alterando as classes desta camada lógica de persistência.

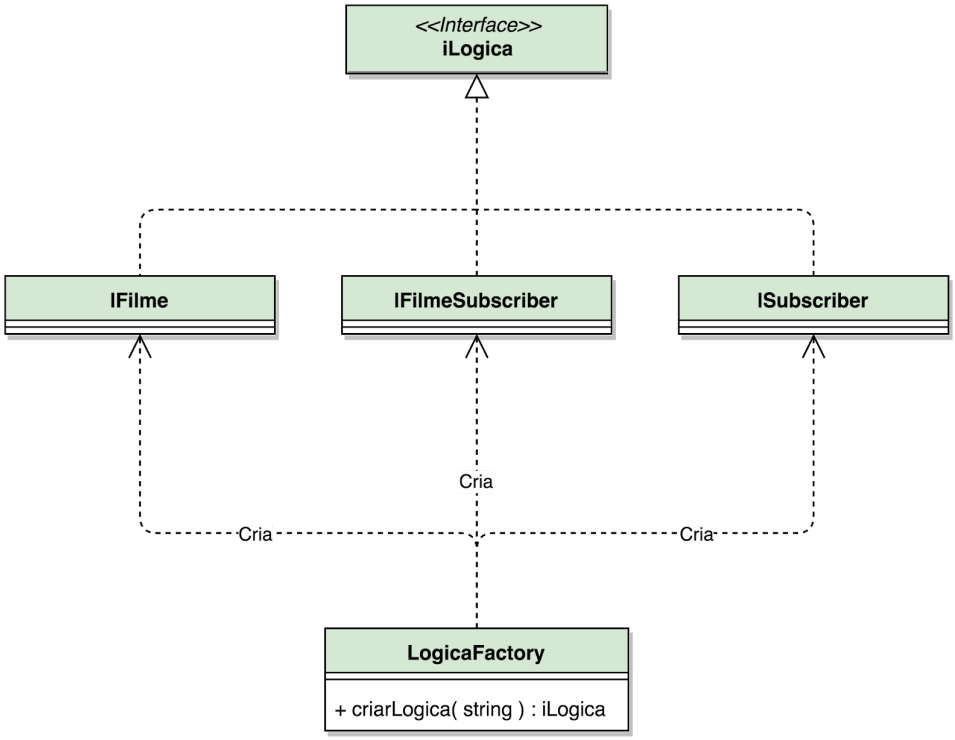
3. Factory

O padrão *Factory* é um padrão **Criacional** que permite que criamos uma fábrica para instanciar nossos objetos em tempo de execução, sem que o cliente instancie os objetos "na mão".

Na camada de persistência de dados, há diversas tabelas, sendo que cada uma possui uma lógica que a manipula, como é ilustrado na Figura 7 e melhor explicado na seção anterior.

Como há diversas lógicas, as quais cada uma manipula uma tabela diferente, surgiu a necessidade de estruturar a maneira como é instanciado um objeto de cada lógica. Nesse ponto, decidimos criar uma fábrica (LogicaFactory) para instanciar tais objetos, como podemos visualizar no Figura 7.

Figura 7. Diagrama UML utilizando o padrão Factory.



Dessa forma, se quisermos criar um objeto da classe *lFilme*, basta executarmos o seguinte comando:

```
$oFilme = $oLogicaFactory->criarLogica("filme");
```

```
public function criarLogica($nomeLogica) {
    if(strtolower($nomeLogica) == "filme")
        return new lFilme();

    if(strtolower($nomeLogica) == "filmesubscriber")
        return new lFilmeSubscriber();

    if(strtolower($nomeLogica) == "subscriber")
        return new lSubscriber();
}
```

4. Singleton

É importante que apenas uma instância da classe *LogicaFactory* esteja ativa no sistema, logo, para garantir tal requisito, utilizamos o padrão de projeto *Singleton,* o qual é **Criacional.**

Figura 8. Classe *LogicaFactory*, além de Factory também é Singleton.

```php
class LogicaFactory {
    private static $instance = NULL;

    private function __construct() {}

    static function getInstance() {
        if(self::$instance == NULL)
            self::$instance = new LogicaFactory();

        return self::$instance;
    }
```

5. Adapter

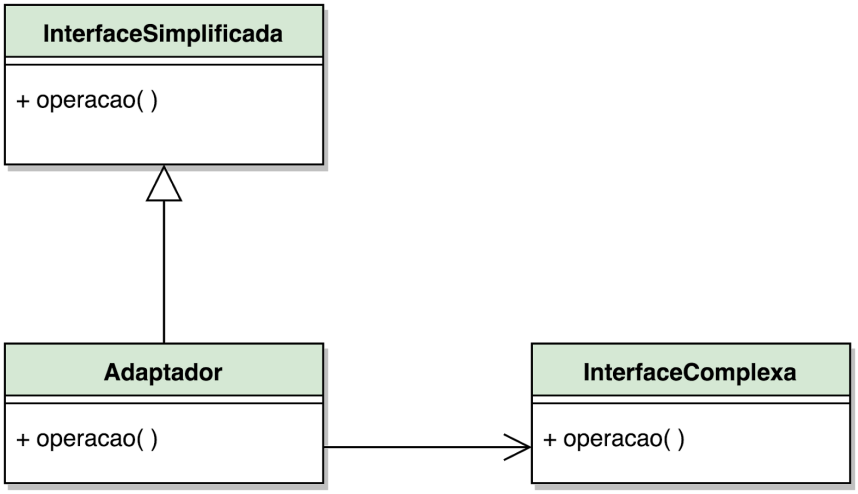
O padrão Adapter permite que classes com interfaces incompatíveis trabalhem juntas. Se trata de um padrão de projeto **Estrutural.**

Antes de explicar onde o padrão foi utilizado no presente trabalho, precisamos esclarecer algumas coisas.

- Para guardar os assentos disponíveis no cinema para cada filme, utilizamos o *middleware* MozartSpaces, o qual implementa o conceito de **Espaço de Tuplas**.
- O MozartSpaces fornece uma interface para que possamos manipular as tuplas no Espaço de Tuplas. Tal interface possui três métodos essenciais: *write, read* e *take*, os quais são complexos de serem utilizados.

Para facilitar a manipulação das tuplas, criamos uma classe chamada ClienteTS, a qual simplifica a utilização da interface do MozartSpaces.

Figura 9. Diagrama de classes do padrão Adapter.



Quem for utilizar o Espaço de Tuplas, basta instanciar um objeto da classe ClienteTS e manipular o espaço, sem se preocupar com a interface do MozartSpaces. Em

termos mais práticos, para escrever uma tupla no espaço de tuplas, basta utilizar a seguinte sintaxe:

```
oClienteTS = new ClienteTS();
oClienteTS.write(10,"A");
```

Sendo que internamente, a classe ClienteTS, utiliza o mesmo comando *write*, mas da interface do MozartSpaces, como é mostrado na Figura 10.

Figura 10. Comando write da classe ClienteTS.

```
public void write(Integer numeroAssento, String letraFileira) throws MzsCoreException {
    Assento oAssento = new Assento(numeroAssento, letraFileira);
    capi.write(cref, new Entry((Serializable) oAssento));
}
```

6. Observer

O último padrão utilizado é o padrão Observer, que por sua vez é um padrão **Comportamental**.

Este padrão veio à mente quando foi idealizada a função de notificação dos usuários interessados em saber quando um filme específico chegou ao cinema. Em linhas gerais, o usuário se inscreve no filme que deseja, e quando este filme se tornar disponível, ele é avisado.

Para implementar o padrão Observer, foram criadas duas classes: InscritoObserver e FilmeSubject, como mostram as Figuras 11 e 12 a seguir.

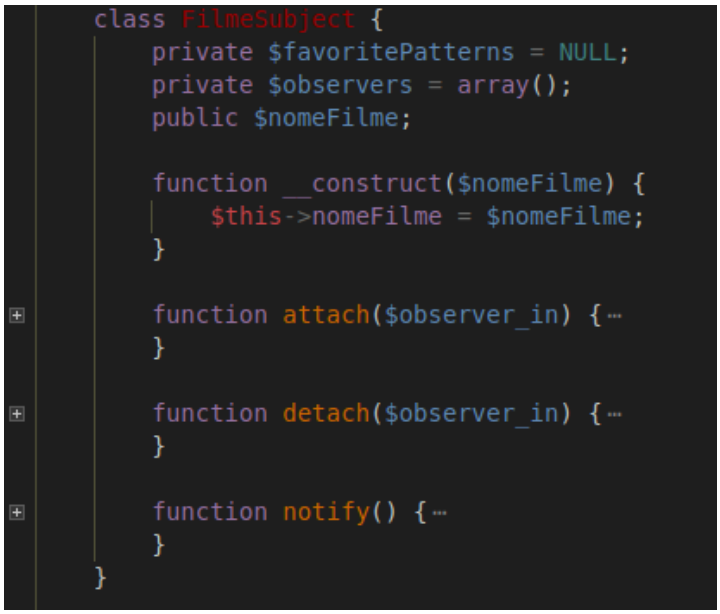
Figura 11. Classe Observer..

```
class InscritoObserver {
    public $email;
    public $nome;

    public function __construct($nome, $email) {
        $this->nome = $nome;
        $this->email = $email;
    }

    public function update($titulo) {
        print_r("Enviando email: O filme ". $titulo." chegou!");
    }
}
```

Figura 12. Classe Subject.



Quando um usuário se inscreve no filme, é instanciado um objeto da classe InscritoObserver passando-o como parâmetro do método *attach* da classe FilmeSubject. De modo análogo, quando o usuário deseja não receber mais as notificações, é utilizado o método *detach*. Quando o filme entra em cartaz, então é invocado o método *notify* da classe FilmeSubject, que por sua vez verifica na tabela tFilmeSubscriber.xml todos os usuários cadastrados, instancia um objeto da classe InscritoObserver para cada um deles, e invoca em cada um o seu respectivo método *update*, para enfim avisar os interessados.

7. Conclusão

De modo geral, pode-se concluir com a realização deste trabalho que, por ser um sistema complexo, a utilização dos padrões de projeto se mostrou ideal (se não essencial) para a realização da tarefa de integração dos componentes. Pode-se observar também que em certos momentos foram utilizados padrões sem que fosse dado conta deles, como ocorreu com o padrão Facade e Adapter.

Referências

[1] DEVMEDIA. ***Padrão de Projeto Facade em Java***. 2012. Disponível em: <https://www.devmedia.com.br/padrao-de-projeto-facade-em-java/26476>. Acesso: 05 ago. 2018.