<Online Ticket System>

Object Design

<1.0>

<23.12.2019>

<Özay Ezerceli, Mert Mısırlıoğlu, Gizem Gümüşçekiççi, Dilara Ünbay>

Prepared for

SE301 Software Engineering



Table of Contents

[1. Introduction 1](#_Toc436772639)

[1.1. Object Design Trade-offs 1](#_Toc436772640)

[1.2. Interface Documentation Guidelines 1](#_Toc436772641)

[1.3. Definitions, Acronyms, and Abbreviations 1](#_Toc436772642)

[1.4. References 1](#_Toc436772643)

[2. Packages 1](#_Toc436772644)

[3. Class Interfaces 1](#_Toc436772645)

OBJECT DESIGN DOCUMENT

Object Design Document (ODD) describes object design trade-offs made by developers, guidelines they followed for subsystem interfaces, the decomposition of subsystems into packages and classes, and the class interfaces. The ODD is **used** to exchange interface information among teams and **as a reference during testing**. The audience for the ODD includes system architects (i.e., the developers who participate in the system design), developers who implement each subsystem, and testers.

Among three approaches to generate ODD, we follow “**ODD embedded into source code**” approach in SE301, since the other methods create many redundancies, inconsistencies.

The initial version of the ODD can be written soon after the subsystem decomposition is stable. Both packages and class interfaces can be generated from source code (comments!) by using a tool, which is named Javadoc. Keeping material for the ODD with the source code enables the developers to maintain consistency more easily and rapidly.

# Introduction (Gizem)

Online Ticket System is designed to present a better and simpler solution for selling tickets. The complete system includes functions for the user, the operator and for the admin. In the system the user will be able to preview events and buy tickets with multiple payment methods. The operator can request to add events to the system and operator also authorized to change the event details as for admin, the admin will accept the requests coming from the operator and has wide authority in the system.

The objective of the system is to be simple but also satisfy user needs and every function needed for a ticket selling system. As we said previously our purpose is to create and effective and simple system, for that purpose the complexity of the website has been reduced. The system will include a simple interface consists of the event views and the functions. In the system the user can preview events without logging in, but the user must register and log-in to activate the buy ticket function. When a user registering to the website the information of the users will remain private. Our website has privacy protocols to ensure the privacy constraints are satisfied according to universal privacy policy.

The system will provide a Graphical User Interface (GUI) and will run on a single computer. The tickets, users, events etc. will be stored in the database.

The subsystem decomposition from which the systems and objects in this document were developed is summarized in the figure below:

A close up of a map

Description automatically generated

## Object Design Trade-offs

As with any software system, several explicit trade-off decisions had to be made in the object design of the system. All the code in the system is our own work as a project team.

Our first trade-off agreement was on memory vs response time(performance) since the database of our interface is crucial for the website, we have decided to choose the memory over the response time because it is crucial to keep the data safely over the performance. Another trade-off is about security vs performance since security is important for buying tickets because we need to keep the payment information of the user safely, so we have chosen security over performance.

As a last trade-off we had to decide between delivery time vs. functionality, in this situation we have chosen delivery time because for the testing we wanted to show every function with most of the functionality of the entire system in a limited time.

## Interface Documentation Guidelines

We have adopted specific interface documentation guidelines and conventions in order to make communication more uniform among our team. First, we have named our classes with singular nouns or noun phrases, with each word beginning with a capital letter. All objects derived from the use cases we have made earlier in the RAD document. Error status is returned to the user. Furthermore, every class will have full interfacing information commented before the implementation of that class, including all attributes and methods.

## Definitions, Acronyms, and Abbreviations

*RAD* : Requirements Analysis Document

*GUI* : Graphical User Interface

*Event* A visual and a written information in which enables users to view.

*Ticket*  The prerequisite to attend to the an event

*Buy* Function to get ticket.

*Publish* Admins making events publicly available for the user on the website.

*Consumer* An individual using the system.

*Operator* An individual which hosts events through the system.

*Admin* User to arrange events and communicates with the operators.

*Search* Different criteria functions to find an event

*MVC* Model-View-Controller Structure

*HTTP* Hyper-Text Transfer Protocol.

## References

Biletix.com

# Packages (Mert)

Describes the decomposition of subsystems into packages and the **file organization of the code.** This includes an overview of each package, its dependencies with other packages, and its expected usage.

# Class Interfaces (Dilara-Özay)

* 1. **Event Class** (Dilara)

The Event class allows the operator to add a new event, cancel an existing event, view an event, edit (modify) an existing event and search for events. Also, it allows the admin to view a list of events and the user to view an event. The Event class is as follows:

A screenshot of a cell phone

Description automatically generated

**Class Name: Event**

**Attributes:**

1. *-name: String*

name is a value of type String. It is a private attribute. It holds the name of the event.

1. *-id: Integer*

id is an event identification number value of type positive Integer. It is a private attribute. It is used in order to distinguish events, since name is not a unique value.

1. *-stage: Stage*

stage is a value of type Stage. It is a private attribute. It holds the stage that event is being held at.

1. *-date: Date*

date is a value of type Date. It is a private attribute. It is used so that the users can view events held on the given date.

1. *-rules: String*

rules is a value of type String. It is a private attribute. It holds the rules description of the event.

1. *-isAvailable: Boolean*

isAvailable is a value of type Boolean. It is a private attribute. It is used in order to check whether the event is available or not. Any event that has been passed will be defined as unavailable in this attribute.

1. *-isAccepted: Boolean*

isAccepted is a value of type Boolean. It is a private attribute. It is used in order to check whether the event is accepted by the admin or not. Any event that has been confirmed by the admin will be defined as accepted in this attribute.

1. *-quota: Integer*

quota is a value of type Integer. It is a private attribute. It holds the maximum amount of people that can attend the event.

1. *-isConcert: Boolean*

isConcert is a value of type Boolean. It is a private attribute. It is used in order to check whether the event is a concert or not.

1. *-isTheatre: Boolean*

isTheatre is a value of type Boolean. It is a private attribute. It is used in order to check whether the event is a theatre or not.

1. *-isSport: Boolean*

isSport is a value of type Boolean. It is a private attribute. It is used in order to check whether the event is a sport or not.

**Operations:**

1. *+addEvent()*

addEvent() is a public function which returns a value of type Event. When this function is called, a new event will be created and sent to admin for confirmation.

1. *+cancelEvent()*

cancelEvent() is a public function which does not return any type of value. When this function is called, the event will be removed from public display based on the given event id. However, it will remain in the database.

1. *viewEvents()*

viewEvents() is a public function which returns a value of type List. When this function is called, a list consisting of all the events will be displayed.

1. *viewEvent()*

viewEvent() is a public function which returns a value of type Event. When this function is called, the event will be displayed in detail based on the given event id.

1. *editEvent()*

editEvent() is a public function which does not return any type of value. When this function is called, the event will be updated based on the changed attributes and sent to admin for confirmation based on the given event id.

1. *searchEvent()*

searchEvent() is a public function which return a value of type List. When this function is called, a list of events will be created and displayed based on the given search criteria.

**Dependencies with Other Classes and Packages:**

A close up of a sign

Description automatically generated

The Event class interacts with Website class, Stage class, Ticket class and Event Storage package. Users/Visitors can handle various event related tasks by accessing the Event class through Website class. For an operator to create an event, there needs to be a stage where the event will be held. Therefore, Event class interacts with Stage class to be created. For a user to attend the event, he or she needs a ticket. Therefore, Event class interacts with the Ticket class. For an admin to see the lists of events, Event class needs to interact with Event Storage package and store the events to display later.

**Exceptions Raised and Exception Handling:**

There are several exceptions that can be raised by the Event class. For example, if an operator tries to create an event leaving some of the information unprovided an exception is raised. To handle this exception, an error message is displayed on the screen and operator is asked to provide the necessary information.

Also, if an operator deletes an information without providing a new one during an event editing process an exception is raised. To handle this exception, an error message is displayed on the screen and operator is asked to provide the necessary information.

Another exception is raised if an operator tries to add an event quota that exceeds the quota of the chosen stage. To handle this exception, an error message is displayed on the screen and operator is asked to change the event quota or the stage.

Lastly, exception is raised if an operator tries to add an event on the same stage and the same date as another existing event. To handle this exception, an error message is displayed on the screen and operator is asked to change the event date or the stage.

* 1. **Ticket Class** (Dilara)

The Ticket class allows the user to buy a ticket to an available event and preview an existing ticket. The Ticket class is as follows:

A close up of a sign

Description automatically generated

**Class Name: Ticket**

**Attributes:**

1. *-price: Integer*

price is a value of type positive Integer. It is a private attribute. It holds the price of a ticket.

1. *-id: Integer*

id is a ticket identification number value of type positive Integer. It is a private attribute. It is used in order to distinguish tickets.

1. *-seat\_number: String*

seat\_number is a value of type String. It is a private attribute. It holds the seat that is assigned to the ticket’s owner.

**Operations:**

1. *+buyTicket()*

buyTicket() is a public function which returns a value of type Ticket. When this function is called, a new ticket will be created and displayed.

1. *+previewTicket()*

previewTicket() is a public function which returns a value of type Ticket. When this function is called, the ticket will be displayed based on the given ticket id.

**Dependencies with Other Classes and Packages:**

A screenshot of a cell phone

Description automatically generated

The Ticket class interacts with Event class and Ticket Storage package. Users can buy or preview tickets by accessing the Ticket class through the Event class, which is accessed through the Website class. Ticket class interacts with Event class because it is a product of an Event. Also, Ticket class interacts with the Ticket Storage package to store the tickets.

**Exceptions Raised and Exception Handling:**

There are several exceptions that can be raised by the Ticket class. For example, if a user tries to buy an amount of tickets that exceeds the quota of the event an exception is raised.

To handle this exception, an error message is displayed on the screen and the user is asked to decrease the quantity of the tickets.

Also, if a user tries to buy a ticket with an amount of credits less than the price of the tickets an exception is raised. To handle this exception, an error message is displayed on the screen and user is redirected to the home page.

* 1. **Stage Class** (Dilara)

The Stage class defines the type of the stage that is chosen. The Stage class is as follows:



**Class Name: Stage**

**Attributes:**

1. *-address: String*

address is a value of type String. It is a private attribute. It holds the address of the stage.

1. *-quota: Integer*

quota is a value of type positive Integer. It is a private attribute. It holds the maximum amount of people that the stage can have.

1. *-place: String*

place is a value of type String. It is a private attribute. It holds the city in which the stage is located.

**Dependencies with Other Classes and Packages:**

A screenshot of a cell phone

Description automatically generated

The Stage class interacts with Event class and Event Storage package. Events can only be created if it includes the stage where it will be held. So, every time an event is created Stage class interacts with Event class. Also, Stage class interacts with the Event Storage package through Event class to store stages.

**Exceptions Raised and Exception Handling:**

The Stage class won’t raise exceptions within self because it doesn’t have any functions that provide services to other classes or packages.

* 1. **Operator Class** (Dilara)

The Operator defines a user that can manage events. Also, it allows an admin to add a new operator and view the existing operators. The Operator class is as follows:

A screenshot of a cell phone

Description automatically generated

**Class Name: Operator**

**Operations:**

1. *+addOperator()*

addOperator() is a public function which returns a value of type Operator. When this function is called, a new user with operator permissions will be created.

1. *+viewOperators()*

viewOperators() is a public function which returns a value of type List. When this function is called, a list consisting of all the operators will be displayed.

**Dependencies with Other Classes and Packages:**

A screenshot of text

Description automatically generated

The Operator class interacts with User class and Website class. Operator is a type of user that can be created by an admin; therefore, it inherits the qualities of the User class by accessing it. The Operators class manage events, so it interacts with Website class to access events.

**Exceptions Raised and Exception Handling:**

There are several exceptions that can be raised by the Operator class. For example, if an admin tries to add an operator leaving some of the information unprovided an exception is raised. To handle this exception, an error message is displayed on the screen and admin is asked to provide the necessary information.

Also, if an admin tries to add an operator with a password that contains the email or the username information an exception is raised. To handle this exception, an error message is displayed on the screen and admin is asked to change the password.

Another exception is raised if an admin tries to add an operator with a common password. To handle this exception, an error message is displayed on the screen and admin is asked to construct a stronger password.