



Nebentermin Jänner 2024

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag (SFKZ 8167)

Kolleg für Informatik – Tag (SFKZ 8242)

Klausurprüfung (Fachtheorie)
aus Programmieren und Software Engineering
im Nebentermin Jänner 2024

für den Aufbaulehrgang für Informatik – Tag (SFKZ 8167)
für das Kolleg für Informatik – Tag (SFKZ 8242)

Liebe Prüfungskandidatin,
lieber Prüfungskandidat!

Bitte füllen Sie zuerst die nachfolgenden Felder **in Blockschrift** aus, bevor Sie mit der Arbeit beginnen.
Trennen Sie dieses Blatt anschließend ab und geben es am Ende ab. Ohne ausgefülltes und
abgegebenes Deckblatt kann Ihre Arbeit nicht zugeordnet und gewertet werden!

Maturaaccount (im Startmenü sichtbar):

Vorname (Blockschrift)

Zuname (Blockschrift)

Klasse (Blockschrift)



Nebentermin Jänner 2024

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag (SFKZ 8167)

Kolleg für Informatik – Tag (SFKZ 8242)

Klausurprüfung aus Fachtheorie

Für das 6. Semester des Aufbaulehrganges und das Kolleg am 16. Jänner 2024.

Generelle Hinweise zur Bearbeitung

Die Arbeitszeit für die Bearbeitung der gestellten Aufgaben beträgt 5 Stunden (300 Minuten). Die 3 Teilaufgaben sind unabhängig voneinander zu bearbeiten, Sie können sich die Zeit frei einteilen. Wir empfehlen jedoch eine maximale Bearbeitungszeit von 1.5 Stunden für Aufgabe 1, 1.5 Stunden für Aufgabe 2 und 2 Stunden für Aufgabe 3. Bei den jeweiligen Aufgaben sehen Sie den Punkteschlüssel. Für eine Einrechnung der Jahresnote sind mindestens 30% der Gesamtpunkte zu erreichen.

Hilfsmittel

In der Datei *P:/SPG_Fachtheorie/SPG_Fachtheorie.sln* befindet sich das Musterprojekt, in dem Sie Ihren Programmcode hineinschreiben. Im Labor steht Visual Studio 2022 mit der .NET Core Version 6 zur Verfügung.

Mit der Software SQLite Studio können Sie sich zur generierten Datenbank verbinden und Werte für Ihre Unittests ablesen. Die Programmdatei befindet sich in *C:/Scratch/SQLiteStudio/SQLiteStudio.exe*.

Zusätzlich wird ein implementiertes Projekt aus dem Unterricht ohne Kommentare bereitgestellt, wo Sie die Parameter von benötigten Frameworkmethoden nachsehen können.

Pfade

Die Solution befindet sich im Ordner *P:/SPG_Fachtheorie*. Sie liegt direkt am Netzlaufwerk, d. h. es ist kein Sichern der Arbeit erforderlich.

Damit das Kompilieren schneller geht, ist der Ausgabepfad der Projekte auf *C:/Scratch/(Projekt)* umgestellt. Das ist zum Auffinden der generierten Datenbank wichtig.



Nullable Reference Types

Das Feature *nullable reference types* wurde in den Projekten aktiviert. Zusätzlich werden Compilerwarnungen als Fehler definiert. Sie können daher das Projekt nicht kompilieren, wenn z. B. ein nullable Warning entsteht. Achten Sie daher bei der Implementierung, dass Sie Nullprüfungen, etc. korrekt durchführen.

SQLite Studio

Zur Betrachtung der Datenbank

in *C:/Scratch/Aufgabe1_Test/Debug/net6.0* bzw. *C:/Scratch/Aufgabe2_Test/Debug/net6.0* steht die Software *SQLite Studio* zur Verfügung. Die exe Datei befindet sich in *C:/Scratch/SPG_Fachtheorie/SQLiteStudio/SQLiteStudio.exe*. Mittels *Database - Add a Database* kann die erzeugte Datenbank (*cash.db* oder *event.db*) geöffnet werden.

Auswählen und Starten der Webapplikation (Visual Studio)

In der Solution gibt es 2 Projekte: *Aufgabe3* (MVC Projekt) und *Aufgabe3RazorPages*. Sie können wählen, ob Sie die Applikation mit MVC oder RazorPages umsetzen wollen. Entfernen Sie das nicht benötigte Projekt aus der Solution und lege das verwendete Projekt als Startup Projekt fest (Rechtsklick auf das Projekt und *Set as Startup Project*).

Beim ersten Start erscheint die Frage *Would you like to trust the ASP.NET Core SSL Certificate?* Wähle *Yes* und *Don't ask me again*. Bestätigen Sie den nachfolgenden Dialog zur Zertifikatsinstallation.

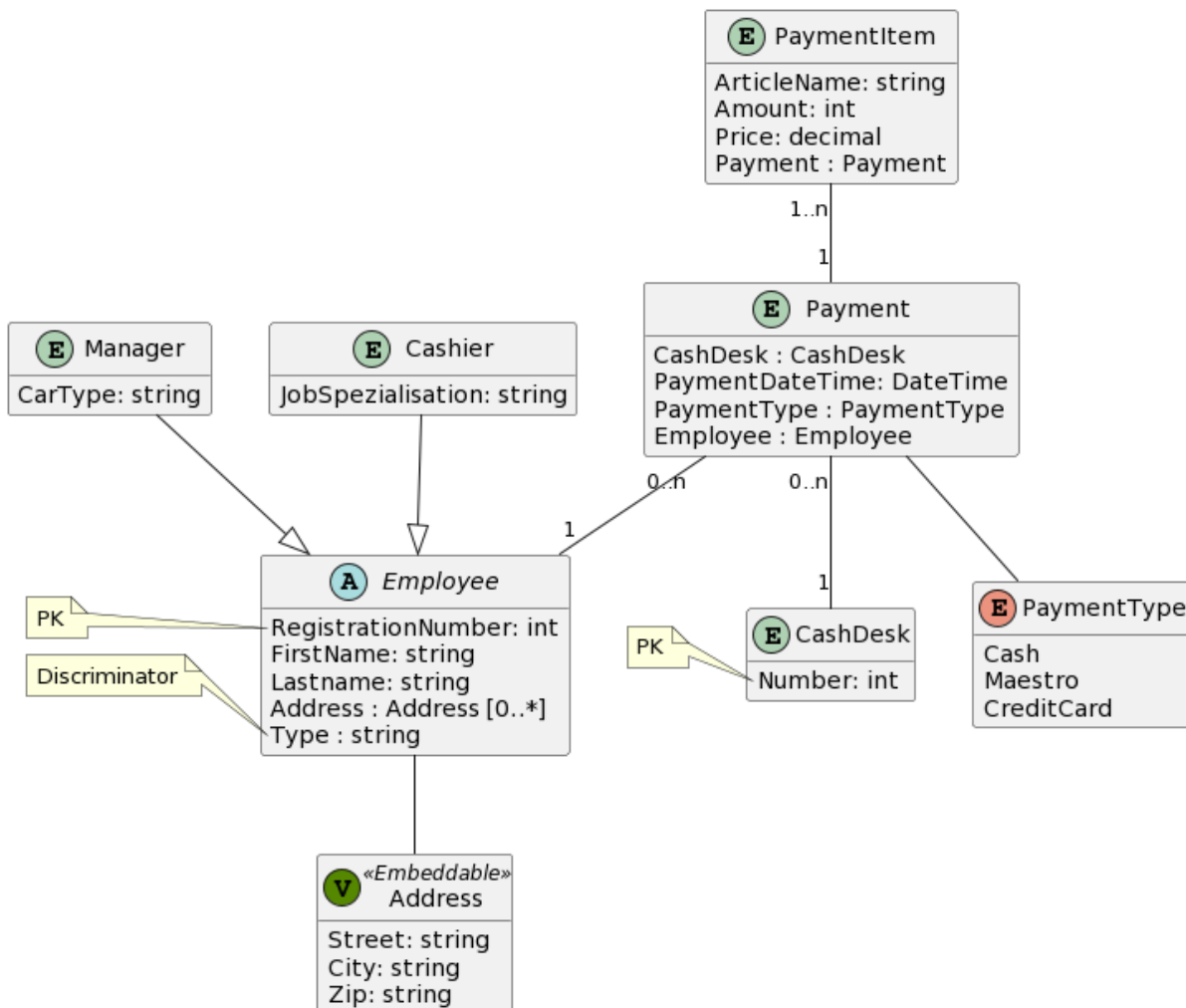


Teilaufgabe 1: Erstellen von EF Core Modelklassen

Eine Kassenverwaltung

Es ist das Domain Model für eine Verwaltungssoftware für Kassen z. B. in einem Supermarkt zu erstellen. Es werden die Mitarbeiterinnen (*Employee*) verwaltet. Diese Klasse ist *abstrakt*, denn die Mitarbeiterinnen unterteilen sich in 2 Gruppen: *Manager* und *Cashier*. Bei der Zahlung an der Kassa (*CashDesk*) entsteht ein *Payment*. Das *Payment* besteht aus mehreren Instanzen der Klasse *PaymentItem*. Das *PaymentItem* ist ein Artikel, der gekauft wurde. Hier wird die Menge (*Amount*) und der Preis (*Price*) erfasst.

Folgendes Klassendiagramm ist gegeben:





Arbeitsauftrag

Erstellung der Modelklassen

Im Projekt *SPG_Fachtheorie.Aufgabe1* befinden sich im Ordner *Model* leere Klassendefinitionen. Bilden Sie jede Klasse gemäß dem UML Diagramm ab, sodass EF Core diese persistieren kann. Beachten Sie folgendes:

- Wählen Sie selbst notwendige Primary keys. Vorgegebene Primary keys sind im Modell mit "PK" gekennzeichnet. Vorgegebene Keys werden nicht von der Datenbank generiert, sondern werden im Konstruktor übergeben.
- Die Klasse *Employee* ist abstrakt, stellen Sie dies durch eine entsprechende Klassendefinition sicher.
- Definieren Sie Stringfelder mit vernünftigen Maximallängen (z. B. 255 Zeichen für Namen, etc.).
- Durch das nullable Feature werden alle Felder als *NOT NULL* angelegt. Verwenden Sie daher nullable Typen für optionale Felder. Sie sind mit *[0..*]* im Diagramm gekennzeichnet.
- Address ist ein *value object*. Stellen Sie durch Ihre Definition sicher, dass kein Mapping diese Klasse in eine eigene Datenbanktabelle durchgeführt wird.
- Legen Sie Konstruktoren mit allen Feldern an. Erstellen Sie die für EF Core notwendigen default Konstruktoren als *protected*.
- Das Feld *Type* in *Employee* ist als Discrimiator Feld vorgesehen. Es wird von EF Core initialisiert, diese sind natürlich nicht im Konstruktor aufzunehmen. Mappen Sie in der Konfiguration das Discriminator Feld in *Employee* in das Feld *Type*
- Implementieren Sie die Vererbung korrekt, sodass eine (1) Tabelle *Employee* entsteht.
- Legen Sie die erforderlichen DB Sets im Datenbankcontext an.

Verfassen von Tests

Im Projekt *SPG_Fachtheorie.Aufgabe1.Test* ist in *Aufgabe1Test.cs* der Test *CreateDatabaseTest* vorgegeben. Er muss erfolgreich durchlaufen und die Datenbank erzeugen. Sie können die erzeugte Datenbank in *C:/Scratch/Aufgabe1_Test/Debug/net6.0/cash.db* in SQLite Studio öffnen.

Implementieren Sie folgende Tests selbst, indem Sie die minimalen Daten in die (leere) Datenbank schreiben. Leeren Sie immer vor dem *Assert* die nachverfolgten Objekte mittels *db.ChangeTracker.Clear()*.

- Der Test *AddCashierSuccessTest* beweist, dass Sie einen Kassier (Cashier) in die Datenbank einfügen können. Prüfen Sie im *Assert*, ob die eingegebene *RegistrationNumber* auch korrekt gespeichert wurde.



- Der Test *AddPaymentSuccessTest* beweist, dass Sie eine Zahlung (Payment) speichern können. Legen Sie dafür eine Instanz von *Payment* an.
- Der Test *EmployeeDiscriminatorSuccessTest* beweist, dass der Typ in Employee korrekt von EF Core geschrieben wird. Gehen Sie dabei so vor:
 - Fügen Sie einen neuen Cashier oder Manager in die Datenbank ein.
 - Prüfen Sie in der Assert Bedingung, ob das Feld *Type* den Wert "Cashier" oder "Manager" hat.

Bewertung (26P, 37.1% der Gesamtpunkte)

Jedes der folgenden Kriterien wird mit 1 Punkt bewertet.

- Die Stringfelder verwenden sinnvolle Längenbegrenzungen.
- Die Klasse *Employee* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Employee* wurde korrekt im DbContext registriert.
- Die Klasse *Employee* besitzt ein korrekt konfiguriertes value object *Address*.
- Die Klasse *Employee* besitzt einen korrekt konfigurierten Discriminator *Type*.
- Die Klasse *Employee* besitzt einen korrekt konfigurierten Schlüssel *RegistrationNumber*.
- Die Klasse *Address* beinhaltet die im UML Diagramm abgebildeten Felder und einen korrekten Konstruktor.
- Die Klasse *Address* ist ein value object, d. h. sie besitzt keine Schlüsselfelder.
- Die Klasse *Manager* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Manager* erbt korrekt von der Klasse *Employee*.
- Die Klasse *Manager* wurde korrekt im DbContext registriert.
- Die Klasse *Cashier* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Cashier* erbt korrekt von der Klasse *Employee*.
- Die Klasse *Cashier* wurde korrekt im DbContext registriert.
- Die Klasse *CashDesk* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *CashDesk* wurde korrekt im DbContext registriert.
- Die Klasse *Payment* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Payment* wurde korrekt im DbContext registriert.
- Die Klasse *PaymentItem* beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *PaymentItem* wurde korrekt im DbContext registriert.
- Der Test *AddCashierSuccessTest* ist korrekt aufgebaut.

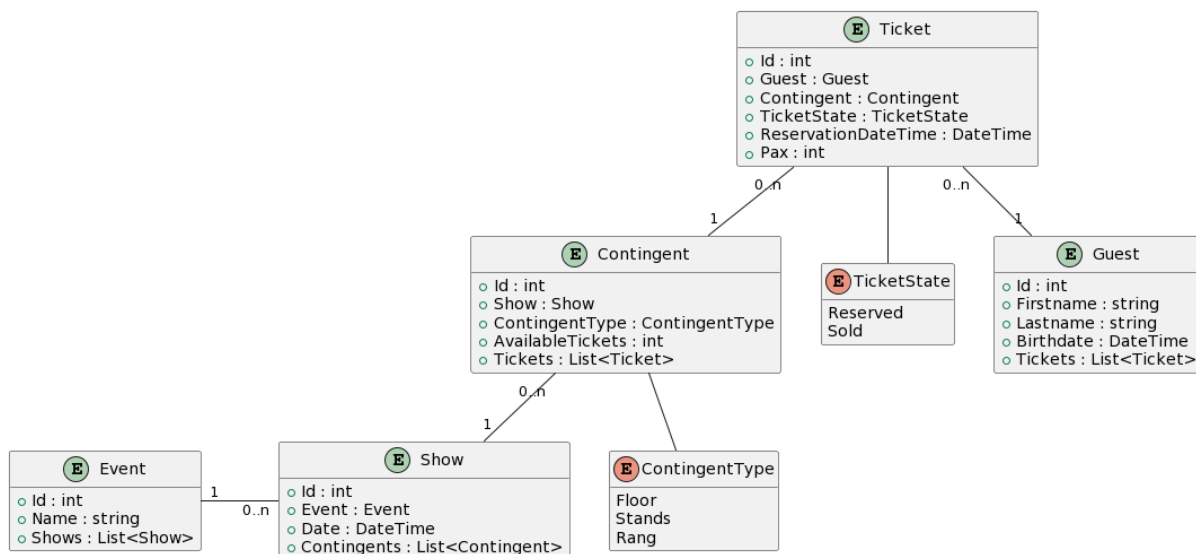


- Der Test *AddCashierSuccessTest* läuft erfolgreich durch.
- Der Test *AddPaymentSuccessTest* ist korrekt aufgebaut.
- Der Test *AddPaymentSuccessTest* läuft erfolgreich durch.
- Der Test *EmployeeDiscriminatorSuccessTest* ist korrekt aufgebaut.
- Der Test *EmployeeDiscriminatorSuccessTest* läuft erfolgreich durch.

Teilaufgabe 2: Services und Unittests

Für den Ticketverkauf bei Events soll ein Service erstellt werden. Ein Event besteht aus mehreren Shows, die zu verschiedenen Terminen statt finden. Für jede Show wird ein Kontingent an Karten (Tickets) aufgelegt. Aus diesem Kontingent werden Karten an Besucher*innen verkauft. Es ist auch möglich, Karten zu reservieren.

Folgendes Klassendiagramm ist als Domain Model bereits vorhanden und kann verwendet werden.



Arbeitsauftrag

Implementierung von Servicemethoden

Im Projekt *SPG_Fachtheorie.Aufgabe2* befindet sich die Klasse *Services/EventService.cs*. Es sind 2 Methoden zu implementieren:

ContingentStatistics CalcContingentStatistics(int contingentId)

Diese Methode soll ermitteln, wie viele Tickets für ein übergebenes Kontingent verkauft wurden. Dabei ist die Anzahl der Begleitpersonen (Pax) zu berücksichtigen. Bei einem Ticket mit 1 Begleitperson ist mit 2 Personen zu rechnen ($Pax + 1$). Der Rückgabotyp ist die im Projekt bereits definierte Klasse *ContingentStatistics*.

```

ContingentStatistics
{
    SoldTickets : int
    ReservedTickets : int
    OwnedShow : Show
}

```

Im Projekt *SPG_Fachtheorie.Aufgabe2.Test* ist der Test *CalcContingentStatisticsTest* bereits vorgegeben, der die Richtigkeit Ihrer Methode prüft.

int CreateReservation(int guestId, int contingentId, int pax, DateTime dateTime)

Diese Methode soll ein neues Ticket mit dem Status *Reserved* erstellen. Dabei sind folgende Randbedingungen zu prüfen:

- Wird das übergebene Kontingent nicht gefunden, ist eine *EventServiceException* (bereits im Projekt vorhanden) mit dem Text *Invalid contingent id.* zu werfen.
- Shows sind bis zu 14 Tage vorher buchbar. Wenn die Show weniger als 14 Tage in der Zukunft liegt, ist eine *EventServiceException* mit dem Text *The show is too close in time.* zu werfen. Verwenden Sie den übergebenen Parameter *dateTime* als Datumsbasis (**nicht** *DateTime.Now*)!
- Wird der übergebene Gast nicht gefunden, ist eine *EventServiceException* mit dem Text *Invalid guest id.* zu werfen.
- Ein Gast soll pro Kontingent und Tag nur ein Ticket reservieren können. Falls an diesem Tag ein Ticket reserviert oder gekauft wurde (Feld *ReservationDateTime*), darf auch keine Reservierung statt finden. Werfen Sie eine *EventServiceException* mit dem Text *A reservation or purchase has already been made for this contingent.* falls diese Bedingung verletzt wurde.
- Natürlich soll geprüft werden, ob das Kontingent nicht schon ausverkauft ist. Ermitteln Sie dafür die Gesamtanzahl der Tickets unter Berücksichtigung der Begleitpersonen (Pax). Ein Ticket mit 1 Begleitperson ist mit 2 zu rechnen ($Pax + 1$). Reicht das Kontingent für die Reservierung des neuen Tickets nicht mehr aus, ist eine *EventServiceException* mit dem Text *Show is sold out.* zu werfen.

Wurden alle Bedingungen erfolgreich geprüft, soll die Methode das neue Ticket mit dem Status *Reserved* in der Datenbank anlegen. Geben Sie die generierte Ticket ID zurück. Wichtig: Die generierte ID erhalten Sie im Feld *Id* erst nach dem Speichern in der Datenbank (*SaveChanges*).



Testen der Methode `CreateReservation`

Schreiben Sie im Projekt *SPG_Fachtheorie.Aufgabe2.Test* in die Klasse *EventServiceTests* Unittests, die die Korrektheit von *CreateReservation* prüfen. Mit *GetEmptyDbContext()* können Sie einen Datenbankcontext zu einer leeren Datenbank erstellen. Befüllen Sie die Datenbank selbst mit minimalen Musterdaten, sodass Sie das Methodenverhalten prüfen können. Verwenden Sie *ChangeTracker.Clear()* des Datenbankcontext, um nach dem Einfügen der Musterdaten und nach Aufrufen der Servicemethode den Changetracker zu leeren.

Mit folgendem Codesnippet können Sie prüfen, ob eine Methode eine bestimmte Exception wirft und eine korrekte Meldung liefert:

```
var ex = Assert.Throws<EventServiceException>(() => MethodToCheck());  
Assert.True(ex.Message == "This is the Message");
```

Es sind 5 Tests zu verfassen:

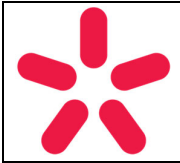
- **ShouldThrowException_WhenInvalidGuestId** prüft, ob die Methode eine *EventServiceException* mit dem Text *Invalid guest id.* wirft, wenn die übergebene Guest ID nicht in der Datenbank gefunden wurde.
- **ShouldThrowException_WhenNoTickets** prüft, ob die Methode eine *EventServiceException* mit dem Text *Show is sold out.* wirft, wenn die Kapazität für die Reservierung nicht mehr ausreicht.
- **ShouldThrowException_WhenGuestHasTicketForShowAndContingentReserved** prüft, ob die Methode eine *EventServiceException* mit dem Text *A reservation or purchase has already been made for this contingent.* wirft, wenn ein Gast an diesem Tag für dieses Kontingent bereits ein Ticket reserviert oder gekauft hat.
- **ShouldThrowException_WhenShowDateNot14DaysInFuture** prüft, ob die Methode eine *EventServiceException* mit dem Text *The show is too close in time.* wirft, wenn die Show weniger als 14 Tage in der Zukunft liegt.
- **ShouldReturnTicketId_WhenParametersAreValid** prüft, ob das Ticket korrekt in die Datenbank eingefügt wurde, wenn alle Bedingungen gültig sind. Prüfen Sie dabei, ob die zurückgegebene Id in der Datenbank gefunden wird.



Bewertung (18 P, 25.8% der Gesamtpunkte)

Jedes der folgenden Kriterien wird mit 1 Punkt bewertet.

- Die Methode *CalcContingentStatistics* berücksichtigt die Anzahl der Begleitpersonen korrekt.
 - Die Methode *CalcContingentStatistics* filtert korrekt nach dem übergebenen Kontingent.
 - Die Methode *CalcContingentStatistics* verwendet LINQ und keine imperativen Konstrukte wie Schleifen, ...
 - Die Methode *CreateReservation* prüft korrekt, ob die übergebene Kontingent und Gast ID vorhanden ist.
 - Die Methode *CreateReservation* prüft korrekt, ob die Show noch buchbar ist (14 Tage vorher).
 - Die Methode *CreateReservation* prüft korrekt, ob nur eine Reservierung pro Tag, Kontingent und Gast vorhanden ist.
 - Die Methode *CreateReservation* prüft korrekt, ob das Kontingent für das neue Ticket ausreicht.
 - Die Methode *CreateReservation* fügt das neue Ticket korrekt in die Datenbank ein.
 - Der Unittest *ShouldThrowException_WhenInvalidGuestId* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenInvalidGuestId* läuft erfolgreich durch.
 - Der Unittest *ShouldThrowException_WhenNoTickets* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenNoTickets* läuft erfolgreich durch.
 - Der Unittest *ShouldThrowException_WhenGuestHasTicketForShowAndContingentReserved* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenGuestHasTicketForShowAndContingentReserved* läuft erfolgreich durch.
 - Der Unittest *ShouldThrowException_WhenShowDateNot14DaysInFuture* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenShowDateNot14DaysInFuture* läuft erfolgreich durch.
 - Der Unittest *ShouldReturnTicketId_WhenParametersAreValid* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldReturnTicketId_WhenParametersAreValid* läuft erfolgreich durch.
-



Teilaufgabe 3: Webapplikation

Das Datenmodell aus Aufgabe 2 soll nun herangezogen werden, um eine Server Side Rendered Web Application zu erstellen. Die Ausgaben der nachfolgenden Layouts können abweichen, müssen aber alle geforderten Features anbieten.

Arbeitsauftrag

Implementieren Sie die folgenden Seiten im Projekt *SPG_Fachtheorie.Aufgabe3*, falls Sie mit MVC arbeiten möchten. Verwenden Sie *SPG_Fachtheorie.Aufgabe3.RazorPages*, falls Sie mit Razor Pages arbeiten möchten. Löschen Sie das nicht benötigte Projekt aus der Solution und legen Ihr gewünschtes Webprojekt als Startprojekt fest.

Seite /Events

Diese Seite soll alle Events darstellen. Ordnen Sie die Events nach dem Namen (Property *Name*). Pro Event sollen alle Showtermine angezeigt werden, **die in der Zukunft liegen**. Verwenden Sie als Zeitbasis den fixen Wert `new DateTime(2024, 1, 16, 0, 0, 0)`, um immer gleiche Ergebnisse zu bekommen. Ordnen Sie die Shows aufsteigend nach dem Datum (Property *Date*). Shows, die im aktuellen Monat stattfinden, sollen grün unterlegt werden. Verwenden Sie auch hier den 16.1.2024 als fixe Zeitbasis. Formatieren Sie das Datum mit dem Formatstring `dd.MM.yyyy`.

Für jede Show soll ein Link angeboten werden, der auf die Detailseite der Show verweist (siehe nächster Punkt).

Der folgende Screenshot zeigt die gewünschte Ausgabe. Es handelt sich um Echtdaten, d. h. bei Ihnen soll die Ausgabe auch so aussehen.



Nebetermin Jänner 2024

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag (SFZ 8167)

Kolleg für Informatik – Tag (SFZ 8242)

Übersicht über die Events

Funk

Showtermin	Aktionen
28.05.2024	Contingents

Rap

Showtermin	Aktionen
30.01.2024	Contingents
04.02.2024	Contingents
04.04.2024	Contingents
02.05.2024	Contingents

Stage And Screen

Showtermin	Aktionen
27.04.2024	Contingents
04.05.2024	Contingents

World

Showtermin	Aktionen
05.02.2024	Contingents
09.02.2024	Contingents
20.02.2024	Contingents
09.03.2024	Contingents

Seite /Contingents/(showId)

Diese Seite soll alle Kontingente einer Show darstellen. Die Show wird als Routingparameter übergeben. Geben Sie als Header die Informationen zur Show (Eventname und Datum der Show) aus. Geben Sie als Überschrift den Typ des Kontingents (Feld *ContingentType*) und die Anzahl der verfügbaren Tickets (Feld *AvailableTickets*) aus. Pro Kontingent sollen alle Tickets mit folgenden Informationen ausgegeben werden: Name des Gastes (Nachname, Vorname, Geburtsdatum), das Datum der Reservierung (Feld *ReservationDateTime*), der Status (Feld *TicketState*) und die Anzahl der Begleitpersonen (Feld *Pax*). Formatieren Sie das Geburtsdatum mit dem Formatstring *dd.MM.yyyy*. Formatieren Sie das Reservierungsdatum mit dem Formatstring *dd.MM.yyyy, HH:mm*. Als Status soll bei reservierten Tickets "Reserviert", bei verkauften Tickets "Verkauft" ausgegeben werden. Sortieren Sie die Tickets nach dem Namen des Gastes (zuerst Nachname, dann Vorname). Formatieren Sie das Datum der Show mit dem Formatstring *dd.MM.yyyy, HH:mm*. Bei jedem Kontingent soll ein Link *Ticket kaufen* zur Seite *BuyTicket* verweisen (siehe nächster Punkt).

Kontingente			
Event: Stage And Screen Showdate: 27.04.2024, 18:00			
Kontingent Floor (80 Tickets)			
Ticket kaufen			
Guest	Reservation Date	State	Pax
Dippl Rasmus, 26.04.2003	11.04.2024, 14:00	Verkauft	0
Franzis Estelle, 05.05.2000	10.04.2024, 14:00	Verkauft	2
Kontingent Rang (70 Tickets)			
Ticket kaufen			
Guest	Reservation Date	State	Pax
Dippl Rasmus, 26.04.2003	09.04.2024, 12:00	Reserviert	1
Franzis Estelle, 05.05.2000	12.04.2024, 07:00	Reserviert	2
Scheer Alexia, 26.08.2003	07.04.2024, 18:00	Verkauft	1
Kontingent Stands (50 Tickets)			
Ticket kaufen			
Guest	Reservation Date	State	Pax
Dippl Rasmus, 26.04.2003	11.04.2024, 10:00	Reserviert	2
Franzis Estelle, 05.05.2000	08.04.2024, 05:00	Reserviert	0
Gollnow Mina, 17.09.2004	11.04.2024, 23:00	Verkauft	0
Gunkel Linn, 05.02.2000	08.04.2024, 04:00	Verkauft	0
Kurnicki Svea, 29.08.2003	07.04.2024, 05:00	Reserviert	2
Michallek Mia, 11.01.2002	12.04.2024, 08:00	Reserviert	1
Mohrhard Susanne, 26.07.2004	11.04.2024, 13:00	Verkauft	1
Nwachukwu Mandy, 03.08.2000	12.04.2024, 12:00	Reserviert	1
Scheer Alexia, 26.08.2003	09.04.2024, 23:00	Reserviert	2
Schlawitz Steffen, 12.07.2002	10.04.2024, 09:00	Verkauft	1
Schwidde Emmely, 15.05.2000	10.04.2024, 14:00	Reserviert	2
Schönherr Marika, 27.03.2003	13.04.2024, 06:00	Reserviert	0
Stolle Alicia, 08.09.2001	08.04.2024, 22:00	Reserviert	0
Tonn Vivian, 31.12.2002	11.04.2024, 13:00	Reserviert	0
Wölpert Ivan, 18.09.2004	10.04.2024, 23:00	Reserviert	2

Seite der Show des Events *Stage And Screen* am 27.4.2024, <http://localhost:5000/Contingents/14>

Seite /BuyTicket/(contingentId)

Diese Seite soll den Kauf eines Tickets für ein bestimmtes Kontingent ermöglichen. Das Kontingent wird als Routingparameter übergeben. Geben Sie als Header die Informationen zum Event und der Show aus: Eventname, Datum der Show mit dem Formatstring *dd.MM.yyyy, HH:mm* und den Typ des Kontingentes (Feld *ContingentType*). Ein Dropdownfeld listet alle im System gespeicherten Gäste sortiert nach Zu- und Vorname auf. Ein Textfeld soll die Anzahl der Begleitpersonen erfassen. Es sollen nur Werte zwischen 0 und 8 Personen möglich sein. Stellen Sie dies durch Validierung sicher. Geben Sie Validierungsfehler auf der Seite aus.

Nach dem Klick auf *Kaufen* soll ein neues Ticket mit dem Status *Sold* erstellt und in der Datenbank gespeichert werden. Für den Parameter *ReservationDate* können

Sie *DateTime.UtcNow* verwenden. Mögliche Fehler sollen als Validierungsfehler ausgegeben werden.

Konnte der Datensatz gespeichert werden, soll auf die Seite *Contingents* verwiesen werden. Achten Sie darauf, den korrekten Routingparameter (showId) beim Redirect zu übergeben: *RedirectToPage("Contingents", new { ShowId=yourValue });*

Ticket kaufen

Event: Stage And Screen
 Show: 27.04.2024, 06:00
 Typ: Floor

Gast:

Conrad Emily

Pax:

-2

Max. 8 Begleitpersonen möglich

Kaufen

<http://localhost:5000/BuyTicket/40>

Bewertung (26P, 37.1% der Gesamtpunkte)

- Die Seite */Events* besitzt eine korrekte Dependency Injection der Datenbank.
- Die Seite */Events* fragt die benötigten Informationen aus der Datenbank korrekt ab.
- Die Seite */Events* zeigt die Events mit dem Namen an.
- Die Seite */Events* zeigt alle zukünftigen Shows des Events an.
- Die Seite */Events* sortiert die Shows korrekt.
- Die Seite */Events* unterlegt alle Shows des aktuellen Monats.
- Die Seite */Events* formatiert die Datumswerte korrekt.
- Die Seite */Events* verweist korrekt auf die Seite der Kontingente der Show.
- Die Seite */Contingents* besitzt eine korrekte Dependency Injection der Datenbank.
- Die Seite */Contingents* fragt die benötigten Informationen aus der Datenbank korrekt ab.
- Die Seite */Contingents* besitzt einen Routingparameter für die Show ID.
- Die Seite */Contingents* zeigt den Eventnamen und das Showdatum korrekt an.
- Die Seite */Contingents* zeigt jedes Kontingent mit dem Namen und den verfügbaren Tickets an.
- Die Seite */Contingents* verweist pro Kontingent korrekt auf die Seite BuyTickets.
- Die Seite */Contingents* listet alle Tickets der Kontingente wie definiert auf.
- Die Seite */Contingents* formatiert den Status des Tickets korrekt.
- Die Seite */Contingents* sortiert die Gäste nach Zuname und nach Vorname.
- Die Seite */BuyTicket* besitzt eine korrekte Dependency Injection der Datenbank.
- Die Seite */BuyTicket* fragt die benötigten Informationen aus der Datenbank korrekt ab.




Nebentermin Jänner 2024

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag (SFKZ 8167)

Kolleg für Informatik – Tag (SFKZ 8242)

- Die Seite */BuyTicket* besitzt einen Routingparameter für die Kontingent ID.
- Die Seite */BuyTicket* zeigt Eventnamen, Showdatum und Kontingenttyp korrekt an.
- Die Seite */BuyTicket* zeigt ein Dropdownfeld mit allen Gästen an.
- Die Seite */BuyTicket* zeigt das Eingabefeld für die Begleitpersonen an.
- Die Seite */BuyTicket* validiert die Eingaben.
- Die Seite */BuyTicket* speichert das Ticket korrekt in der Datenbank.
- Die Seite */BuyTicket* verweist nach dem Speichern auf die Kontingentseite.

	<p style="text-align: center;">Nebetermin Jänner 2024</p> <p style="text-align: center;">PROGRAMMIEREN UND SOFTWARE ENGINEERING</p> <p style="text-align: center;">Aufbaulehrgang für Informatik – Tag (SFZ 8167) Kolleg für Informatik – Tag (SFZ 8242)</p>
--	--

Bewertungsblatt (vom Prüfer auszufüllen)


Für jede erfüllte Teilaufgabe gibt es 1 Punkt. In Summe sind also 70 Punkte zu erreichen. Für eine Berücksichtigung der Jahresnote müssen mindestens 30 % der Gesamtpunkte erreicht werden. Für eine positive Beurteilung der Klausur müssen mindestens 50 % der Gesamtpunkte erreicht werden.

Beurteilungsstufen:

70 – 62 Punkte: Sehr gut, 61 – 53 Punkte: Gut, 52 – 44 Punkte: Befriedigend, 43 – 35 Punkte: Genügend

Aufgabe 1 (jew. 1 Punkt, 26 in Summe)	Erf.	Nicht erf.
Die Stringfelder verwenden sinnvolle Längenbegrenzungen.		
Die Klasse Employee beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse Employee wurde korrekt im DbContext registriert.		
Die Klasse Employee besitzt ein korrekt konfiguriertes value object Address.		
Die Klasse Employee besitzt einen korrekt konfigurierten Discriminator Type.		
Die Klasse Employee besitzt einen korrekt konfigurierten Schlüssel RegistrationNumber.		
Die Klasse Address beinhaltet die im UML Diagramm abgebildeten Felder und einen korrekten Konstruktor.		
Die Klasse Address ist ein value object, d. h. sie besitzt keine Schlüsselfelder.		
Die Klasse Manager beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse Manager erbt korrekt von der Klasse Employee.		
Die Klasse Manager wurde korrekt im DbContext registriert.		
Die Klasse Cashier beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse Cashier erbt korrekt von der Klasse Employee.		
Die Klasse Cashier wurde korrekt im DbContext registriert.		
Die Klasse CashDesk beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse CashDesk wurde korrekt im DbContext registriert.		
Die Klasse Payment beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse Payment wurde korrekt im DbContext registriert.		
Die Klasse PaymentItem beinhaltet die im UML Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse PaymentItem wurde korrekt im DbContext registriert.		
Der Test AddCashierSuccessTest ist korrekt aufgebaut.		
Der Test AddCashierSuccessTest läuft erfolgreich durch.		
Der Test AddPaymentSuccessTest ist korrekt aufgebaut.		
Der Test AddPaymentSuccessTest läuft erfolgreich durch.		
Der Test EmployeeDiscriminatorSuccessTest ist korrekt aufgebaut.		
Der Test EmployeeDiscriminatorSuccessTest läuft erfolgreich durch.		

Aufgabe 2 (jew. 1 Punkt, 18 in Summe)	Erf.	Nicht erf.
Die Methode CalcContingentStatistics berücksichtigt die Anzahl der Begleitpersonen korrekt.		
Die Methode CalcContingentStatistics filtert korrekt nach dem übergebenen Kontingent.		
Die Methode CalcContingentStatistics verwendet LINQ und keine imperativen Konstrukte wie Schleifen, ...		
Die Methode CreateReservation prüft korrekt, ob die übergebene Kontingent und Gast ID vorhanden ist.		

	<p style="text-align: center;">Nebetermin Jänner 2024</p> <p style="text-align: center;">PROGRAMMIEREN UND SOFTWARE ENGINEERING</p> <p style="text-align: center;">Aufbaulehrgang für Informatik – Tag (SFKZ 8167) Kolleg für Informatik – Tag (SFKZ 8242)</p>
--	--

Die Methode CreateReservation prüft korrekt, ob die Show noch buchbar ist (14 Tage vorher).		
Die Methode CreateReservation prüft korrekt, ob nur eine Reservierung pro Tag, Kontingent und Gast vorhanden ist.		
Die Methode CreateReservation prüft korrekt, ob das Kontingent für das neue Ticket ausreicht.		
Der Unittest ShouldThrowException_WhenInvalidGuestId hat den korrekten Aufbau.		
Der Unittest ShouldThrowException_WhenInvalidGuestId läuft erfolgreich durch.		
Die Methode CreateReservation fügt das neue Ticket korrekt in die Datenbank ein.		
Der Unittest ShouldThrowException_WhenNoTickets hat den korrekten Aufbau.		
Der Unittest ShouldThrowException_WhenNoTickets läuft erfolgreich durch.		
Der Unittest ShouldThrowException_WhenGuestHasTicketForShowAndContingentReserved hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest ShouldThrowException_WhenGuestHasTicketForShowAndContingentReserved läuft erfolgreich durch.		
Der Unittest ShouldThrowException_WhenShowDateNot14DaysInFuture hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest ShouldThrowException_WhenShowDateNot14DaysInFuture läuft erfolgreich durch.		
Der Unittest ShouldReturnTicketId_WhenParametersAreValid hat den korrekten Aufbau.		
Der Unittest ShouldReturnTicketId_WhenParametersAreValid läuft erfolgreich durch.		

Aufgabe 3 (jew. 1 Punkt, 26 in Summe)	Erf.	Nicht erf.
Die Seite /Events besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite /Events fragt die benötigten Informationen aus der Datenbank korrekt ab.		
Die Seite /Events zeigt die Events mit dem Namen an.		
Die Seite /Events zeigt alle zukünftigen Shows des Events an.		
Die Seite /Events sortiert die Shows korrekt.		
Die Seite /Events unterlegt alle Shows des aktuellen Monats.		
Die Seite /Events formatiert die Datumswerte korrekt.		
Die Seite /Events verweist korrekt auf die Seite der Kontingente der Show.		
Die Seite /Contingents besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite /Contingents fragt die benötigten Informationen aus der Datenbank korrekt ab.		
Die Seite /Contingents besitzt einen Routingparameter für die Show ID.		
Die Seite /Contingents zeigt den Eventnamen und das Showdatum korrekt an.		
Die Seite /Contingents zeigt jedes Kontingent mit dem Namen und den verfügbaren Tickets an.		
Die Seite /Contingents verweist pro Kontingent korrekt auf die Seite BuyTickets.		
Die Seite /Contingents listet alle Tickets der Kontingente wie definiert auf.		
Die Seite /Contingents formatiert den Status des Tickets korrekt.		
Die Seite /Contingents sortiert die Gäste nach Zuname und nach Vorname.		
Die Seite /BuyTicket besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite /BuyTicket fragt die benötigten Informationen aus der Datenbank korrekt ab.		
Die Seite /BuyTicket besitzt einen Routingparameter für die Kontingent ID.		
Die Seite /BuyTicket zeigt Eventnamen, Showdatum und Kontingenttyp korrekt an.		
Die Seite /BuyTicket zeigt ein Dropdownfeld mit allen Gästen an.		
Die Seite /BuyTicket zeigt das Eingabefeld für die Begleitpersonen an.		
Die Seite /BuyTicket validiert die Eingaben.		
Die Seite /BuyTicket speichert das Ticket korrekt in der Datenbank.		
Die Seite /BuyTicket verweist nach dem Speichern auf die Kontingentseite.		
Die Seite /Events besitzt eine korrekte Dependency Injection der Datenbank.		