	Höhere technische Bundeslehr- und Versuchsanstalt für Textilindustrie und Datenverarbeitung
	Abteilungen: Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168) Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Reife- und Diplomprüfung 2023/24

Nebentermin Jänner 2025

Aufgabenstellung für die Klausurprüfung

Jahrgang:	6AAIF, 6BAIF, 6CAIF, 6AKIF, 6BKIF, 8ABIF, 8ACIF
Prüfungsgebiet:	Programmieren und Software Engineering
Prüfungstag:	15. Jänner 2025
Arbeitszeit:	5 Std. (300 Minuten)
Kandidaten/Kandidatinnen:	
Prüfer/Prüferin:	Mag. Manfred BALLUCH; Michael SCHLETZ, BEd; DI (FH) Martin SCHRUTEK; Klaus UNGER, MSc
Aufgabenblätter:	15 Seiten inkl. Umschlagbogen

*Inhaltsübersicht der Einzelaufgaben im Umschlagbogen
(Unterschrift des Prüfers/der Prüferin auf den jeweiligen Aufgabenblättern)*

Das versiegelte Kuvert mit der der Aufgabenstellung wurde geöffnet von:

Name: _____ Unterschrift: _____

Datum: _____ Uhrzeit: _____

Zwei Zeugen (Kandidaten/Kandidatinnen)

Name: _____ Unterschrift: _____

Name: _____ Unterschrift: _____

Geprüft: Wien, am _____

Mag. Heidi Steinwender
Abteilungsvorständin

RS.

Dr. Gerhard Hager
Direktor

Genehmigt:

Wien, am _____

RS.

MinR Mag. Gabriele Winkler Rigler



Nebentermin Jänner 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Klausurprüfung (Fachtheorie)
aus Programmieren und Software Engineering
im Nebentermin Jänner 2025

für den Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

für das Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Liebe Prüfungskandidatin,
liebe 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 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Klausurprüfung aus Fachtheorie

Für das 6. Semester des Aufbaulehrganges Tag und das Kolleg Tag sowie für das 8. Semester des Aufbaulehrganges Abend und das Kolleg Abend am 15. Jänner 2025.

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 8 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.



Nebentermin Jänner 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

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 legen Sie 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.

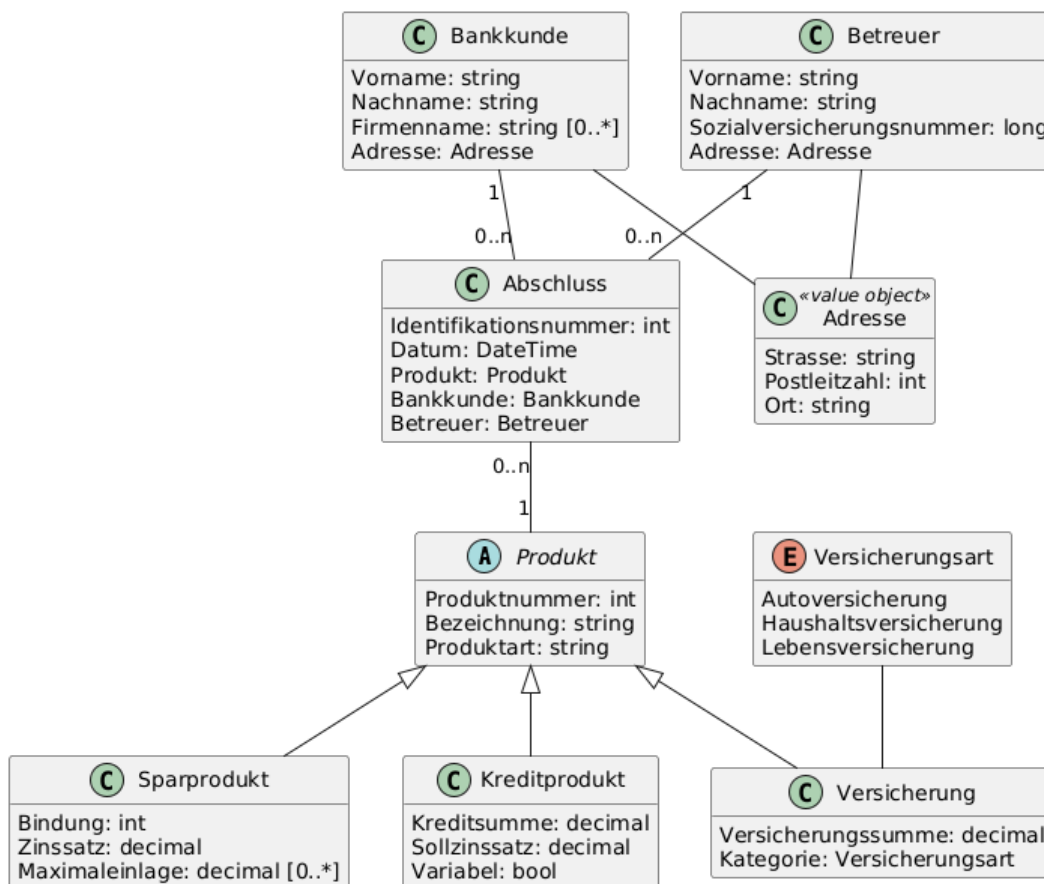


Teilaufgabe 1: Erstellen von EF Core Modelklassen

Eine Bankapplikation

In einer Bank können die Kunden unterschiedliche Finanz- oder Versicherungsprodukte abschließen. Es ist das Domain Model für diese Produktabschlüsse zu erstellen. Die abstrakte Klasse *Produkt* unterteilt sich in *Sparprodukt*, *Kreditprodukt* und *Versicherung*. Es werden *Bankkunden* und *Betreuer* verwaltet. Bei einem Produktabschluss entsteht eine Instanz der Klasse *Abschluss*. Im Zuge eines Produktabschlusses muss gespeichert werden, welches Produkt von welchem Bankkunden bei welchem Betreuer an welchem Datum abgeschlossen wurde.

Folgendes Klassendiagramm ist gegeben:



Die Klasse *Bankkunde* ist anders aufgebaut als die Klasse *Kunde* von der Teilaufgabe 2. Die Klasse *Betreuer* ist anders aufgebaut als die Klasse *Mitarbeiter* von der Teilaufgabe 2.



Arbeitsauftrag

Erstellung der Modelklassen

Im Projekt *SPG_Fachtheorie.Aufgabe1* sind im Ordner *Model* die Klassen gemäß den Anforderungen zu implementieren. Bilden Sie jede Klasse gemäß dem Klassendiagramm ab, sodass EF Core diese persistieren kann. Beachten Sie folgendes:

- Definieren Sie in der Klasse *Produkt* das Feld *Produktnummer* und in der Klasse *Abschluss* das Feld *Identifikationsnummer* als Primärschlüssel. Diese beiden Primärschlüssel werden nicht von der Datenbank generiert, sondern werden im Konstruktor übergeben. Für alle anderen Klassen definieren Sie selbst notwendige Primärschlüssel.
- Die Klasse *Produkt* ist abstrakt, stellen Sie dies durch eine entsprechende Klassendefinition sicher.
- 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.
- *Adresse* 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 *Produktart* in *Produkt* ist als Discriminator Feld vorgesehen. Es wird von EF Core initialisiert, diese sind natürlich nicht im Konstruktor aufzunehmen. Mappen Sie in der Konfiguration das Discriminator Feld in *Produkt* in das Feld *Produktart*.
- Implementieren Sie die Vererbung korrekt, sodass eine einzige Tabelle *Produkt* entsteht.
- Legen Sie die erforderlichen DB-Sets im Datenbankkontext 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 *AddInsuranceSuccessTest* beweist, dass Sie eine Versicherung in die Datenbank einfügen können. Prüfen Sie im Assert, ob die eingegebene *Produktnummer* auch korrekt gespeichert wurde.
- Der Test *ProductDiscriminatorSuccessTest* beweist, dass der Typ im Produkt korrekt von EF Core geschrieben wird. Gehen Sie dabei so vor:
 - Fügen Sie ein neues Sparprodukt, ein neues Kreditprodukt oder eine neue Versicherung die Datenbank ein.
 - Prüfen Sie in der Assert Bedingung, ob das Feld *Produktart* den korrekten Wert beinhaltet.
- Der Test *AddTransactionSuccessTest* beweist, dass Sie einen Produktabschluss speichern können. Legen Sie dafür eine Instanz von *Abschluss* an.

Bewertung (23 Punkte)

Jedes der folgenden Kriterien wird mit 1 Punkt bewertet.

- Die Klasse *Produkt* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Produkt* wurde korrekt im DbContext registriert und besitzt einen korrekt konfigurierten Schlüssel *Produktnummer*.
- Die Klasse *Produkt* besitzt einen korrekt konfigurierten Discriminator *Produktart*.
- Die Klasse *Sparprodukt* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Sparprodukt* erbt korrekt von der Klasse *Produkt* und wurde korrekt im DbContext registriert.
- Die Klasse *Kreditprodukt* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Kreditprodukt* erbt korrekt von der Klasse *Produkt* und wurde korrekt im DbContext registriert.
- Die Klasse *Versicherung* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Versicherung* erbt korrekt von der Klasse *Produkt* und wurde korrekt im DbContext registriert.
- Die Klasse *Bankkunde* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Bankkunde* wurde korrekt im DbContext registriert.
- Die Klasse *Betreuer* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Betreuer* wurde korrekt im DbContext registriert.



Nebentermin Jänner 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

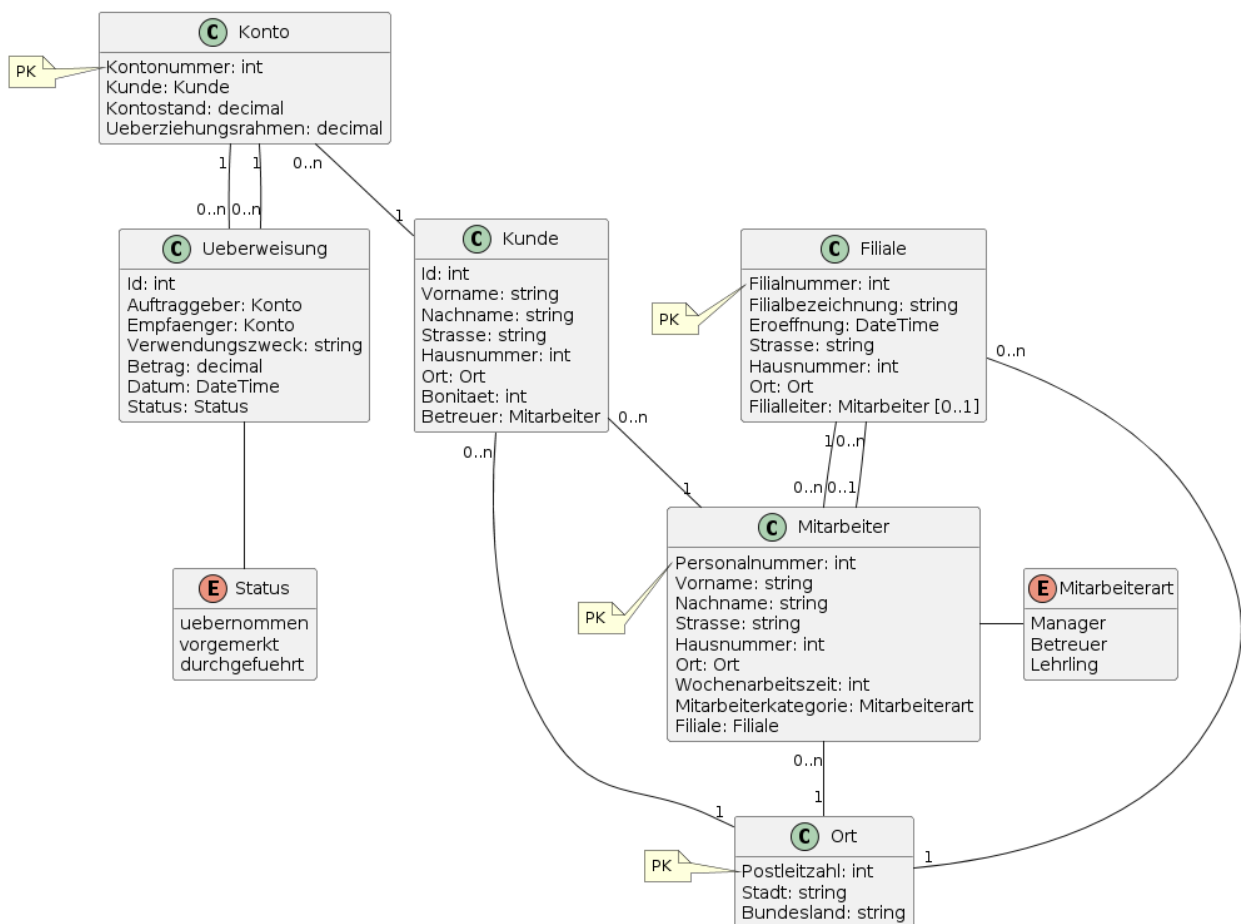
- Die Klasse *Bankkunde* und die Klasse *Betreuer* besitzen jeweils ein korrekt konfiguriertes value object *Adresse*.
- Die Klasse *Adresse* beinhaltet die im UML-Diagramm abgebildeten Felder und ist ein value object, d. h. sie besitzt keine Schlüsselfelder.
- Die Klasse *Abschluss* beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.
- Die Klasse *Abschluss* wurde korrekt im DbContext registriert.
- Der Test *AddInsuranceSuccessTest* ist korrekt aufgebaut.
- Der Test *AddInsuranceSuccessTest* läuft erfolgreich durch.
- Der Test *ProductDiscriminatorSuccessTest* ist korrekt aufgebaut.
- Der Test *ProductDiscriminatorSuccessTest* läuft erfolgreich durch.
- Der Test *AddTransactionSuccessTest* ist korrekt aufgebaut.
- Der Test *AddTransactionSuccessTest* läuft erfolgreich durch.

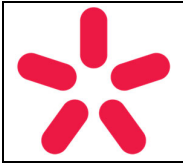


Teilaufgabe 2: Services und Unittests

Für eine Bank werden die Daten der Bankfilialen, Mitarbeiter, Kunden, Konten und Überweisungen verwaltet und es soll ein Service erstellt werden. Überweisungen können zwischen einem Konto des Auftraggebers und einem Konto des Empfängers durchgeführt werden. Vor einer Überweisung ist bereits sichergestellt, dass sowohl Auftraggeber als auch Empfänger bereits als Kunde erfasst ist. Eine Überweisung kann den Status *übernommen*, *vorgemerkt* oder *durchgeführt* haben. Jeder Kunde verfügt über eine bestimmte Bonität. Je höher die für die Bonität gespeicherte Zahl (Bonitätswert), desto schlechter die Bonität.

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/BankService.cs. Es sind 2 Methoden zu implementieren:

List<Kunde> SchlechteKunden (int bonitaet, int prozentsatz)

Diese Methode soll ermitteln, welche Kunden den angegebenen Prozentsatz des Überziehungsrahmens bei einem beliebigen Konto des jeweiligen Kunden bereits ausgeschöpft haben, wenn Sie dem angegebenen Bonitätswert oder einem höheren Bonitätswert zugewiesen sind.


Im Projekt SPG_Fachtheorie.Aufgabe2.Test ist der Test ListBadCustomersTest in der Klasse BankTests bereits vorgegeben, der die Richtigkeit Ihrer Methode prüft.

*int UeberweisungAnlegen(int auftraggeberKonto, int empfaengerKonto,
string verwendungszweck, decimal betrag, DateTime datum)*

Diese Methode soll eine neue Überweisung anlegen, wobei davon ausgegangen werden kann, dass jeder Auftraggeber bzw. Empfänger als Kunde bereits angelegt wurde, wenn eine gültige Kontonummer des Auftraggebers bzw. Empfängers übergeben wird. Dabei sind folgende Randbedingungen zu prüfen:

- Beinhaltet *AuftraggeberKonto* keine Kontonummer eines bestehenden Kontos, ist eine *ApplicationException* mit dem Text *Ungültiges Auftraggeber Konto* zu werfen.
- Beinhaltet *EmpfaengerKonto* keine Kontonummer eines bestehenden Kontos, ist eine *ApplicationException* mit dem Text *Ungültiges Empfänger Konto* zu werfen.
- Besteht der *Verwendungszweck* aus mehr als 50 Zeichen, ist eine *ApplicationException* mit dem Text *Ungültiger Verwendungszweck* zu werfen.
- Ist nach dem Abzug des *Betrags* der Überweisung vom *Kontostand* des Auftraggebers ein negativer Kontostand vorhanden und übersteigt der negative Kontostand den *Ueberziehungsrahmen* des Auftraggebers, ist eine *ApplicationException* mit dem Text *Ungültiger Betrag* zu werfen.

Wurden alle Bedingungen erfolgreich geprüft, soll die Methode eine neue Überweisung in der Datenbank anlegen. Der Status der Überweisung soll auf *vorgemerkt* gesetzt werden, wenn das *Datum* für die Überweisung in der Zukunft liegt. Der Status der Überweisung soll auf *übernommen* gesetzt werden, wenn das *Datum* für die Überweisung dem aktuellen Datum entspricht oder in der Vergangenheit liegt. Geben Sie die generierte

	<p style="text-align: center;">Nebentermin Jänner 2025</p> <p style="text-align: center;">PROGRAMMIEREN UND SOFTWARE ENGINEERING</p> <p style="text-align: center;">Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168) Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)</p>
--	---

ÜberweisungsId zurück. Wichtig: Die generierte ID erhalten Sie im Feld *Id* erst nach dem Speichern in der Datenbank (*SaveChanges*).

Testen der Methode UeberweisungAnlegen

Schreiben Sie im Projekt *SPG_Fachtheorie.Aufgabe2.Test* in die Klasse *BankTests* Unittests, welche die Korrektheit von *UeberweisungAnlegen* prüfen. Mit *GetDbContext()* können Sie einen Datenbankkontext mit einer Datenbank erstellen, welche bereits Testdaten beinhaltet. Sie können von diesen Testdaten ausgehen, um das Methodenverhalten zu überprüfen.

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_WhenInvalidTransferorAccount** prüft, ob die Methode eine *ApplicationException* mit dem Text *Ungültige Auftraggeber Konto* wirft, wenn das übergebene Auftraggeber Konto nicht in der Datenbank gefunden wurde.
- **ShouldThrowException_WhenInvalidRecipientAccount** prüft, ob die Methode eine *ApplicationException* mit dem Text *Ungültiges Empfänger Konto* wirft, wenn das übergebene Konto nicht dem Empfänger zugewiesen ist.
- **ShouldThrowException_WhenPurposeTooLong** prüft, ob die Methode eine *ApplicationException* mit dem Text *Ungültiger Verwendungszweck* wirft, wenn der Verwendungszweck zu viele Zeichen beinhaltet.
- **ShouldThrowException_WhenLimitsExceeded** prüft, ob die Methode eine *ApplicationException* mit dem Text *Ungültiger Betrag* wirft, wenn der Überziehungsrahmen durch die Überweisung überschritten werden würde.
- **ShouldReturnTransferId_WhenParametersAreValid** prüft, ob die Überweisung 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 Punkte)

Jedes der folgenden Kriterien wird mit 1 Punkt bewertet.

- Die Methode *SchlechteKunden* berücksichtigt die übergebene Bonität korrekt.
 - Die Methode *SchlechteKunden* berücksichtigt den übergebenen Prozentsatz korrekt.
 - Die Methode *SchlechteKunden* verwendet LINQ und keine imperativen Konstrukte wie Schleifen oder ähnliches.
 - Die Methode *UeberweisungAnlegen* prüft korrekt, ob das Auftraggeberkonto dem Auftraggeber zugeordnet ist.
 - Die Methode *UeberweisungAnlegen* prüft korrekt, ob das Empfängerkonto dem Empfänger zugeordnet ist.
 - Die Methode *UeberweisungAnlegen* prüft korrekt, ob der Verwendungszweck maximal 50 Zeichen beinhaltet.
 - Die Methode *UeberweisungAnlegen* prüft korrekt, ob der neue Kontostand des Auftraggeberkontos den Überziehungsrahmen des Auftraggebers nicht überschreiten würde.
 - Die Methode *UeberweisungAnlegen* fügt die neue Überweisung korrekt in die Datenbank ein.
 - Der Unittest *ShouldThrowException_WhenInvalidTransferorAccount* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenInvalidTransferorAccount* läuft erfolgreich durch.
 - Der Unittest *ShouldThrowException_WhenInvalidRecipientAccount* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenInvalidRecipientAccount* läuft erfolgreich durch.
 - Der Unittest *ShouldThrowException_WhenPurposeTooLong* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenPurposeTooLong* läuft erfolgreich durch.
 - Der Unittest *ShouldThrowException_WhenLimitsExceeded* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldThrowException_WhenLimitsExceeded* läuft erfolgreich durch.
 - Der Unittest *ShouldReturnTransferId_WhenParametersAreValid* hat den korrekten Aufbau (arrange, act, assert).
 - Der Unittest *ShouldReturnTransferId_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 /Bank/Index

Diese Seite soll alle Bankfilialen auflisten, wobei die Filialen nach der Postleitzahl sortiert werden sollen. Für jede Filiale soll die Filialbezeichnung, die Postleitzahl, die Stadt, die Straße, die Hausnummer und die Anzahl der dieser Filiale zugewiesenen Mitarbeiter ausgegeben werden. Wiener Filialen mit einer Postleitzahl kleiner als 2000 sollen mit einer anderen Hintergrundfarbe dargestellt werden. Für jede Filiale soll ein Link angeboten werden, der auf die Detailseite der Filiale verweist (siehe nächster Punkt).

Filialen

Filialbezeichnung	Postleitzahl	Stadt	Strasse	Hausnummer	Mitarbeiteranzahl	Aktionen
Niederlassung Stephansdom	1010	Wien	Stephansplatz	1	3	Details
Flagship Office Zentral	1060	Wien	Mariahilferstr.	15	3	Details
Niederlassung Westbahnhof	1060	Wien	Mariahilferstr.	20	1	Details
Niederlassung Kärnten	2345	Villach	Am See	1	0	Details
Niederlassung Linz Hptbahnhof	4020	Linz	Schützenstr	41	1	Details
Niederlassung Salzburg Stad	5020	Salzburg	Stadtplatz	32	1	Details
Niederlassung Graz Hptbahnhof	6020	Innsbruck	Bahnhofplatz	3	2	Details

<https://localhost:5000/Bank/Index>

Seite /Bank/Details

Diese Seite soll alle Mitarbeiter einer Bankfiliale anzeigen, wobei die Filialnummer als Routingparameter übergeben wird. Geben Sie zunächst die Filialbezeichnung, das Datum der Eröffnung sowie den Vor- und Nachnamen des Filialleiters aus. Für das Eröffnungsdatum ist nur das Datum im bei uns gebräuchlichen Format (beispielsweise

13.02.2024) und nicht die Uhrzeit auszugeben. Anschließend sind die Mitarbeiter der Filiale mit Vornamen, Nachnamen, Straße, Hausnummer, Postleitzahl, Stadt, Bundesland, die Mitarbeiterkategorie und der Wochenarbeitszeit aufzulisten. Die Liste der Mitarbeiter ist nach dem Nachnamen als erstes Sortierkriterium und nach dem Vornamen als zweites Sortierkriterium zu sortieren. Auf der Seite ist ein Link zum Hinzufügen eines Mitarbeiters zu der Filiale anzuführen.

Details zur Filiale 1

Filialbezeichnung: Flagship Office Zentral

Eröffnungsdatum: 01.01.1990

Filialleiter: Teo Noeh

[Mitarbeiter hinzufügen](#)

Mitarbeiter

Vorname	Nachname	Strasse	Hausnummer	Postleitzahl	Stadt	Bundesland	Mitarbeiterkategorie	Wochenarbeitszeit
Roger	Andrzejewski	Feldstr.	3	1010	Wien	Wien	BETREUER	20
Antonio	Mudersbach	Sonnenweg	40	8010	Graz	Steiermark	BETREUER	30
Teo	Noeh	Marburger Str.	9	1030	Wien	Wien	MANAGER	30

<https://localhost:5000/Bank/Details/1>

Seite /Bank/Add

Diese Seite soll das Hinzufügen eines Mitarbeiters zu einer Filiale ermöglichen, wobei die Filialnummer als Routingparameter übergeben wird. Unter der Filialbezeichnung sollen die Eingabefelder für den neuen Mitarbeiter angeordnet werden. Eingabefelder für den Namen, der Straße, der Hausnummer und der Wochenarbeitszeit sind vorzusehen. Ein Dropdownfeld listet alle im System gespeicherten Postleitzahlen mit der jeweiligen Stadt auf. Ein weiteres Dropdownfeld listet alle im System gespeicherten Mitarbeiterkategorien auf, wobei eine Liste der Mitarbeiterkategorien mit

`Enum.GetValues<Mitarbeiterart>().ToList()` generiert werden kann. Für die Wochenarbeitszeit sollen nur Werte zwischen 5 und 40 Stunden möglich sein. Stellen Sie dies durch eine Validierung sicher. Geben Sie gegebenenfalls Validierungsfehler auf der Seite aus.

Nach einem Klick auf den Button *Speichern* soll ein neuer Mitarbeiter für die entsprechende Filiale mit den eingegebenen Daten in der Datenbank gespeichert werden. Die Personalnummer des neuen Mitarbeiters können Sie mit `_db.Mitarbeiterliste.Select(m => m.Personalnummer).Max() + 1` berechnen. Mögliche Fehler sollen als Validierungsfehler ausgegeben werden.

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



Nebentermin Jänner 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Neuer Mitarbeiter für die Filiale Flagship Office Zentral

Vorname

Max

Nachname

Mustermann

Straße

Spengergasse

Hausnummer

20

Ort

1050 Wien

Mitarbeiterkategorie

BETREUER

Wochenarbeitszeit

50

Die Wochenarbeitszeit muss zwischen 5 und 40 liegen.

Speichern

<https://localhost:5000/Bank/Add/1>

Bewertung (25 Punkte)

- Die Seite `/Bank/Index` besitzt eine korrekte Dependency Injection der Datenbank.
- Die Seite `/Bank/Index` fragt die benötigten Informationen aus der Datenbank korrekt ab.
- Die Seite `/Bank/Index` zeigt die Bankfilialen mit der Filialbezeichnung und der Adresse inklusive der Stadt an.
- Die Seite `/Bank/Index` zeigt die Anzahl der jeweiligen Bankfiliale zugewiesenen Mitarbeiter an.
- Die Seite `/Bank/Index` sortiert die Bankfilialen korrekt.
- Die Seite `/Bank/Index` hebt alle Bankfilialen von Wien farblich hervor.
- Die Seite `/Bank/Index` verweist korrekt auf die Seite mit den Details zu einer Bankfiliale.
- Die Seite `/Bank/Details` besitzt eine korrekte Dependency Injection der Datenbank.
- Die Seite `/Bank/Details` fragt die benötigten Informationen aus der Datenbank korrekt ab.
- Die Seite `/Bank/Details` besitzt einen Routingparameter für die Filialnummer.
- Die Seite `/Bank/Details` zeigt die Filialbezeichnung sowie den Vor- und Nachnamen des Filialleiters an.
- Die Seite `/Bank/Details` zeigt das korrekt formatierte Eröffnungsdatum der Filiale an.
- Die Seite `/Bank/Details` listet alle Mitarbeiter der Filiale wie definiert auf.



Nebentermin Jänner 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

- Die Seite */Bank/Details* listet zu jedem Mitarbeiter auch die Stadt und das Bundesland auf.
- Die Seite */Bank/Details* sortiert die Mitarbeiter der Filiale nach Nachnamen und Vornamen.
- Die Seite */Bank/Details* verweist korrekt auf die Seite zum Hinzufügen eines Mitarbeiters zu der Filiale.
- Die Seite */Bank/Add* besitzt eine korrekte Dependency Injection der Datenbank.
- Die Seite */Bank/Add* fragt die benötigten Informationen aus der Datenbank korrekt ab.
- Die Seite */Bank/Add* besitzt einen Routingparameter für die Filialnummer und zeigt die Filialbezeichnung korrekt an.
- Die Seite */Bank/Add* zeigt ein Dropdownfeld mit allen in der Datenbank gespeicherten Postleitzahlen an.
- Die Seite */Bank/Add* zeigt ein Dropdownfeld mit allen in der Datenbank gespeicherten Mitarbeiterkategorien an.
- Die Seite */Bank/Add* zeigt die Eingabefelder für den Namen, der Straße, der Hausnummer und der Wochenarbeitszeit an.
- Die Seite */Bank/Add* validiert die eingegebene Wochenarbeitszeit gemäß der Definition.
- Die Seite */Bank/Add* speichert die Mitarbeiterdaten korrekt in der Datenbank.
- Die Seite */Bank/Add* verweist nach dem Speichern auf die Detailseite der entsprechenden Bankfiliale.



Nebetermin Jänner 2025

PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Bewertungsblatt (vom Prüfer auszufüllen)

Für jede erfüllte Teilaufgabe gibt es 1 Punkt. In Summe sind also 66 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:

66 – 58 Punkte: Sehr gut, 57 – 50 Punkte: Gut, 49 – 42 Punkte: Befriedigend, 41 – 34 Punkte: Genügend

Aufgabe 1 (jew. 1 Punkt, 23 in Summe)	Erf.	Nicht erf.
Die Klasse <i>Produkt</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Produkt</i> wurde korrekt im DbContext registriert und besitzt einen korrekt konfigurierten Schlüssel <i>Produktnummer</i> .		
Die Klasse <i>Produkt</i> besitzt einen korrekt konfigurierten Discriminator <i>Produktart</i> .		
Die Klasse <i>Sparprodukt</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Sparprodukt</i> erbt korrekt von der Klasse <i>Produkt</i> und wurde korrekt im DbContext registriert.		
Die Klasse <i>Kreditprodukt</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Kreditprodukt</i> erbt korrekt von der Klasse <i>Produkt</i> und wurde korrekt im DbContext registriert.		
Die Klasse <i>Versicherung</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Versicherung</i> erbt korrekt von der Klasse <i>Produkt</i> und wurde korrekt im DbContext registriert.		
Die Klasse <i>Bankkunde</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Bankkunde</i> wurde korrekt im DbContext registriert.		
Die Klasse <i>Betreuer</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Betreuer</i> wurde korrekt im DbContext registriert.		
Die Klasse <i>Bankkunde</i> und die Klasse <i>Betreuer</i> besitzen jeweils ein korrekt konfiguriertes value object <i>Adresse</i> .		
Die Klasse <i>Adresse</i> beinhaltet die im UML-Diagramm abgebildeten Felder und ist ein value object, d. h. sie besitzt keine Schlüsselfelder.		
Die Klasse <i>Abschluss</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Abschluss</i> wurde korrekt im DbContext registriert.		
Der Test <i>AddInsuranceSuccessTest</i> ist korrekt aufgebaut.		
Der Test <i>AddInsuranceSuccessTest</i> läuft erfolgreich durch.		
Der Test <i>ProductDiscriminatorSuccessTest</i> ist korrekt aufgebaut.		
Der Test <i>ProductDiscriminatorSuccessTest</i> läuft erfolgreich durch.		
Der Test <i>AddTransactionSuccessTest</i> ist korrekt aufgebaut.		
Der Test <i>AddTransactionSuccessTest</i> läuft erfolgreich durch.		
Die Klasse <i>Produkt</i> beinhaltet die im UML-Diagramm abgebildeten Felder und korrekte public bzw. protected Konstruktoren.		
Die Klasse <i>Produkt</i> wurde korrekt im DbContext registriert und besitzt einen korrekt konfigurierten Schlüssel <i>Produktnummer</i> .		
Die Klasse <i>Produkt</i> besitzt einen korrekt konfigurierten Discriminator <i>Produktart</i> .		



Nebentermin Jänner 2025


PROGRAMMIEREN UND SOFTWARE ENGINEERING

Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)

Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)

Aufgabe 2 (jew. 1 Punkt, 18 in Summe)	Erf.	Nicht erf.
Die Methode <i>SchlechteKunden</i> berücksichtigt die übergebene Bonität korrekt.		
Die Methode <i>SchlechteKunden</i> berücksichtigt den übergebenen Prozentsatz korrekt.		
Die Methode <i>SchlechteKunden</i> verwendet LINQ und keine imperativen Konstrukte wie Schleifen oder ähnliches.		
Die Methode <i>UeberweisungAnlegen</i> prüft korrekt, ob das Auftraggeberkonto dem Auftraggeber zugeordnet ist.		
Die Methode <i>UeberweisungAnlegen</i> prüft korrekt, ob das Empfängerkonto dem Empfänger zugeordnet ist.		
Die Methode <i>UeberweisungAnlegen</i> prüft korrekt, ob der Verwendungszweck maximal 50 Zeichen beinhaltet.		
Die Methode <i>UeberweisungAnlegen</i> prüft korrekt, ob der neue Kontostand des Auftraggeberkontos den Überziehungsrahmen des Auftraggebers nicht überschreiten würde.		
Die Methode <i>UeberweisungAnlegen</i> fügt die neue Überweisung korrekt in die Datenbank ein.		
Der Unittest <i>ShouldThrowException_WhenInvalidTransferorAccount</i> hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest <i>ShouldThrowException_WhenInvalidTransferorAccount</i> läuft erfolgreich durch.		
Der Unittest <i>ShouldThrowException_WhenInvalidRecipientAccount</i> hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest <i>ShouldThrowException_WhenInvalidRecipientAccount</i> läuft erfolgreich durch.		
Der Unittest <i>ShouldThrowException_WhenPurposeTooLong</i> hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest <i>ShouldThrowException_WhenPurposeTooLong</i> läuft erfolgreich durch.		
Der Unittest <i>ShouldThrowException_WhenLimitsExceeded</i> hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest <i>ShouldThrowException_WhenLimitsExceeded</i> läuft erfolgreich durch.		
Der Unittest <i>ShouldReturnTransferId_WhenParametersAreValid</i> hat den korrekten Aufbau (arrange, act, assert).		
Der Unittest <i>ShouldReturnTransferId_WhenParametersAreValid</i> läuft erfolgreich durch.		

Aufgabe 3 (jew. 1 Punkt, 25 in Summe)	Erf.	Nicht erf.
Die Seite <i>/Bank/Index</i> besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite <i>/Bank/Index</i> fragt die benötigten Informationen aus der Datenbank korrekt ab.		
Die Seite <i>/Bank/Index</i> zeigt die Bankfilialen mit der Filialbezeichnung und der Adresse inklusive der Stadt an.		
Die Seite <i>/Bank/Index</i> zeigt die Anzahl der jeweiligen Bankfiliale zugewiesenen Mitarbeiter an.		
Die Seite <i>/Bank/Index</i> sortiert die Bankfilialen korrekt.		
Die Seite <i>/Bank/Index</i> hebt alle Bankfilialen von Wien farblich hervor.		
Die Seite <i>/Bank/Index</i> verweist korrekt auf die Seite mit den Details zu einer Bankfiliale.		
Die Seite <i>/Bank/Details</i> besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite <i>/Bank/Details</i> fragt die benötigten Informationen aus der Datenbank korrekt ab.		
Die Seite <i>/Bank/Details</i> besitzt einen Routingparameter für die Filialnummer.		
Die Seite <i>/Bank/Details</i> zeigt die Filialbezeichnung sowie den Vor- und Nachnamen des Filialleiters an.		
Die Seite <i>/Bank/Details</i> zeigt das korrekt formatierte Eröffnungsdatum der Filiale an.		
Die Seite <i>/Bank/Details</i> listet alle Mitarbeiter der Filiale wie definiert auf.		
Die Seite <i>/Bank/Details</i> listet zu jedem Mitarbeiter auch die Stadt und das Bundesland auf.		

	<p style="text-align: center;">Nebentermin Jänner 2025</p> <p style="text-align: center;">PROGRAMMIEREN UND SOFTWARE ENGINEERING</p> <p style="text-align: center;">Aufbaulehrgang für Informatik – Tag/Abend (SFKZ 8167/8168)</p> <p style="text-align: center;">Kolleg für Informatik – Tag/Abend (SFKZ 8242/8244)</p>
--	--

Die Seite <i>/Bank/Details</i> sortiert die Mitarbeiter der Filiale nach Nachnamen und Vornamen.		
Die Seite <i>/Bank/Details</i> verweist korrekt auf die Seite zum Hinzufügen eines Mitarbeiters zu der Filiale.		
Die Seite <i>/Bank/Add</i> besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite <i>/Bank/Add</i> fragt die benötigten Informationen aus der Datenbank korrekt ab.		
Die Seite <i>/Bank/Add</i> besitzt einen Routingparameter für die Filialnummer und zeigt die Filialbezeichnung korrekt an.		
Die Seite <i>/Bank/Add</i> zeigt ein Dropdownfeld mit allen in der Datenbank gespeicherten Postleitzahlen an.		
Die Seite <i>/Bank/Add</i> zeigt ein Dropdownfeld mit allen in der Datenbank gespeicherten Mitarbeiterkategorien an.		
Die Seite <i>/Bank/Add</i> zeigt die Eingabefelder für den Namen, der Straße, der Hausnummer und der Wochenarbeitszeit an.		
Die Seite <i>/Bank/Add</i> validiert die eingegebene Wochenarbeitszeit gemäß der Definition.		
Die Seite <i>/Bank/Add</i> speichert die Mitarbeiterdaten korrekt in der Datenbank.		
Die Seite <i>/Bank/Add</i> verweist nach dem Speichern auf die Detailseite der entsprechenden Bankfiliale.		
Die Seite <i>/Bank/Index</i> besitzt eine korrekte Dependency Injection der Datenbank.		
Die Seite <i>/Bank/Index</i> fragt die benötigten Informationen aus der Datenbank korrekt ab.		