СОФИЙСКИ УНИВЕРСИТЕТ
"СВ. КЛИМЕНТ ОХРИДСКИ"

Факултет по Математика и Информатика

Катедра „Информационни Технологии”

Д И П Л О М Н А Р А Б О Т А

на тема:

EJB контейнер за сесийни бийнове в J2EE платформа

Дипломант: Иво Владимиров Симеонов, фак. № 42240

Специалност: Информатика

Специализация: Информационни и комуникационни технологии

Научен ръководител: доцент д-р Боян Бончев

София, 2006 г.

Съдържание

Увод

Широкото разпространение на езика за програмиране Java наложи разработката на платформи, върху които да може да се разработват промишлени (enterprise) приложения. Основна част от тези платформи заема *EJB Container* спецификацията, която дефинира общ компонентен модел за промишлени приложения. Високите изисквания за стабилност (robust), сигурност (secure) и скалируемост (scalable) на тези приложения са крайна цел на всяка реализация.

Идеята за промишлена платформа е разработена през 1999 година, когато излиза и първата версия на спецификацията J2EE (Java 2 Enterprise Edition)[7]. Нейната цел е да обедини съществуващите Java технологии, гарантирайки на разработчиците на приложения базов набор от функционалности, които се предоставят от платформата. Всяка следваща версия на J2EE се разширява в две посоки. Едната е чрез добавяне на нови съществуващи спецификации и тяхното интегриране в платформата, докато другата е чрез ревизиране на спецификациите вече част от J2EE стандарта, с цел подобряване на качествата на платформата и улесняване нейното ползване.

Езикът за програмиране Java предполага обектно ориентиран модел за разработване на приложения. Основна част от него са на компонентите JavaBeans. Те обхващат функционалност и данни позволявайки тяхното използване както в няколко различни части от приложението, така и в различни приложения. J2EE стандартът разширява JavaBeans модела до EJB (Enterprise JavaBeans) като гарантира стабилност, прозрачна сигурност и скалируемост.

Цел и задачи на дипломната работа

Целта на настоящата дипломна работа е да бъдат анализирани и систематизирани основните изисквания за успешна реализация на EJB контейнер *(EJB Container)* за сесийни промишлени Java бийнове (*Session Enterprise Java Beans).* Като резултат от това изследване ще бъде предложен дизайн и реализация на *EJB Container* като допълнение към работеща *Java 2 Enterprise Edition* платформа. Реализацията ще позволява поддържането на разработен промишлен сесиен бийн (enterprise session bean) от *J2EE* платформа и неговото ползване от други компоненти на системата.

Задачите, произтичащи от тази цел са:

- Да се направи обзор и анализ на EJB 3.0 Core и EJB Simplification спецификациите относно сесийни бийнове.
- Да се определят изискванията към дизайна на EJB 3.0 контейнер за сесийни бийнове
- Да се разработи дизайн на EJB 3.0 контейнер за сесийни бийнове и негова реализация
- Тестване на разработената система
- Да се анализират резултатите и предимствата на разработката

Структура на дипломната работа

Дипломната работа включва: Увод, пет глави, Заключение, Речник на използваните съкращения и термини, списък с използвана литература и Приложение.

Уводът съдържа кратко въведение в областта на дипломната работа описват се темата и начинът на структуриране на изложението.

В глава 1 се прави обзор на свързаните с областта технологии. Представя се отдалеченото програмиране в Java, използването на анотации, поименни и указателни услуги (Java Naming and Directory Interface), програмния интерфейс за транзакции и технологията EJB.

В глава 2 се прави анализ на EJB спецификациите с цел определяне изискванията и условията необходими за реализиране на J2EE съвместима реализация на EJB контейнер.

В глава 3 се разработва дизайна на контейнера с оглед резултатите от анализа.

В глава 4 се описва реализацията на контейнера спрямо разработения дизайн.

В глава 5 се описва тестването на реализацията.

Следва Заключение включващо описание на възможните насоки за развитие на дизайна и реализацията на EJB контейнер и описание на внедряването му промишления продукт SAP NetWeaver Application Server Java EE 5 Edition.

Приложени са още речник на използваните съкращения и термини, списък на използвани източници и UML Class диаграми на интерфейсите и Java код на по-важните класове от реализацията.

1. Въведение в свързаните с *EJB* технологии

1.1. Отдалечено извикване на методи в Java

Отдалеченото извикване на методи в Java – Java Remote Method Invocation RMI е механизъм, който позволява един участник да извика метод на обект, който съществува в друго адресно пространство. Това адресно пространство може да е както на същата физическа машина така и на друга такава. В известен смисъл RMI механизмът е обектно ориентирана версия на RPC (Remote Procedure Call) механизма[5].

При сценариите на отдалечено извикване има три процеса, които са замесени. Първият от тях е процесът на клиента, който извиква метода на отдалечения обект. Вторият от тях е процеса, който е създал и притежава отдалечения обект. Реално той се намира в неговото адресно пространство и за него обекта е напълно нормален. Третият процес е специален и се нарича *регистратура на обекти* (object registry). Той служи за свързване на обекти с имена. За да може обект да бъде достъпен отдалеч е необходимо да бъде регистриран в *регистратура на обекти.*

1.1.1.Програмен модел

Моделът за отдалечено извикване работи с две групи от обекти. Първата група са така наречените отдалечени обекти. Необходимо условие един обект да може да се регистрира като отдалечен е той да имплементира *отдалечен интерфейс*(remote interface). Един интерфейс се счита за отдалечен ако негов родител е интерфейса *java.rmi.Remote* и всички негови методи декларират *java.rmi.RemoteException* като изключение. Протокол RMI третира отдалечените обекти по специален начин. Когато такъв обект се предава от една Java виртуална машина в друга такава, RMI генерира специален обект наричан стъб. Стъбът действа като локален представител или посредник на отдалечения обект при клиента. Той реализира *отдалечения интерфейс,* през който клиента ползва отдалечения обект и е отговорен за транспортирането на всяка заявка от клиент към обекта като и на резултата обратно от обекта към клиента[9].

Втората група обекти, с които работи RMI са така наречените *сериализируеми* обекти. Един обект се счита за *сериализируем,* ако той реализира интерфейса *java.io.Serializable*. Когато такъв обект се предава от една Java виртуална машина в друга такава, RMI точно генерира копие на обекта там където се предава като запазва пълният граф на реферирane.

1.1.2.Разширение на протокола с IIOP

Стандартната реализация на RMI в Java е силно обвързана с езика Java. Това не позволява клиенти реализирани на друг програмен език да взаимодействат с отдалечени Java обекти. Това се променя с разширяването на стандартния RMI протокол с протокола за комуникация IIOP (Internet Inner ORB Protocol). Този стандарт е известен под името RMI-IIOP [10].

Протоколът IIOP е специфичен за стандарта CORBA (Common Object Request Broker Architecture). За разлика от RMI, CORBA специфицира начин за споделяне и достъп до обекти, който е независим от езика за програмиране. По-конкретно IIOP е протокола който описва съдържанието на комуникационните пакети които се транспортират между участниците, както и начина на тяхното предаване.

Чрез комбинирането на транспортния протокол IIOP и съществуващия RMI обектен модел се получава модел, в който е възможна комуникация между Java и не-Java системи. По този начин е възможно да се реализира една пълна разпределена система.

1.2. Анотации в Java

Стандартното издания на Java версия 5, въвежда нова особеност за описание на езика наречена анотации (annotations) [8]. Анотациите позволяват на разработчиците на Java да декорират своя код със свои специфични атрибути. Тези атрибути може да се използват за документирането, за автоматичното създаване на допълнителен код и дори по време на работа на

приложението за доставяне на специални услуги като например по сигурността или бизнес логиката.

В езика за програмиране Java винаги са съществували ограничен брой описателни данни за кода. Например такъв елемент е *javadoc,* в който със стандартни поделементи се въвежда описание от което в по-късен етап се получава документацията на кода. До момента този вид данни се използва само с тази цел и само от един стандартно помощно средство. Съществуват проекти с *отворен код –XDoclet,* които се възползват от този вид описание на по-високо ниво, разширявайки набора от поделементи с цел по-добро описание и на тяхна база автоматично създават код.

Стандартното издание на Java предлага много по-усъвършенствана система за прилагане на информация към кода на приложението, наречени анотации. В допълнение към стандартен набор от такива, разработчика е свободен на дефинира свои специфични такива, с които може да декорира класове, интерфейси, конструктори, полета или методи на своя код. В дефиницията на анотациите може да се специфицира точно към кои части от кода може да се прилагат те и в кой момент от работа може да се използват – по време на компилиране, когато кода се зарежда от Java виртуалната машина или дори по време на работа с класа.

1.3. Поименни и указателни услуги в Java

Поименните и указателните услуги имат важна роля в инфраструктурата на локални или глобални или мрежи предоставяйки на широк диапазон от клиенти споделената информация за потребители, машини, мрежи, услуги и приложения[4].

Програмния интерфейс на Java за поименни и указателни услуги- JNDI, е достъпен до всички Java приложения. Той е разработен изцяло за нуждите на Java платформата и следва стереотипите на обектното програмиране заложени в него. Използвайки JNDI, Java базираните приложения могат да съхраняват и получават достъп до именувани обекти от всеки тип. Допълнително JNDI осигурява методи са извършване на стандартни операции с указател като

асоцииране на признаци с обекти и откриване на обекти по дадени техни признаци.

1.3.1.Архитектура

Архитектурата на JNDI се състои от програмни интерфейси достъпни за ползване от приложенията и от интерфейси за доставчици на услуги разработени с цел произволен доставчик да може да се включи в процеса на предоставяне на услуги и то прозрачно за приложенията. Примерни доставчици на услуги са RMI, CORBA, LDAP и DNS протоколите.

1.3.2.Обзор на основните интерфейси

Интерфейсът *javax.naming.Context* е базовия интерфейс за всички именувани контексти. Той дефинира базови операции като регистрация на обект под дадено име, търсене на обект по име, премахване на обект по името му, разглеждане на всички обекти и техните имена, създаване и унищожаване на вложени контексти от същият тип и други.

В JNDI всички имена са относителни спрямо някой контекст. За да започнат работа приложенията използват класа *javax.naming.InitialContext* като създават инстанция от него ползвайки публичните му конструктори. Този клас предоставя възможност на клиентите да се свържат с различни споделени контексти, специфицирайки имена от стандартни пространства от имена като URL или DNS.

Различните реализации на интерфейса *Context* могат да регистрират различни обекти по специфичен начин. Особено полезен клас обект, който е препоръчително да се поддържа от всички генерални реализации на контекст е класа *javax.naming.Reference*. Инстанция от този клас представлява обект, който се намира извън указателя. Обикновено инстанции *Reference* се ползват, за да се заблудят клиентите, че произволни обекти може да се регистрират в контекста, особено такива обекти, които нямат специфична поддръжка. Когато

такъв обект се потърси, JNDI реализацията опитва да възстанови истинския обект и да го върне на клиента.
В JNDI има стандартен механизъм за създаване на обекти, използвайки съхранената информацията. Информацията може да бъде от всеки вид – *java.lang.Object*. Например това може да е *Reference*, или URL, или всеки друг вид данни необходими за създаването на обекта. Превръщането на тази съхранена информация в обект се осъществява с помощта на фабрики за обекти. Фабрика за обект е клас който имплементира интерфейса *javax.naming.spi.ObjectFactory*. Ползвайки предоставената й информация фабриката се опитва да реконструира обекта.

1.4. Транзакционен програмен интерфейс в Java

1.4.1. Транзакции

Транзакцията дефинира логическа единица от работа. Тя дефинира прост модел за успех или неуспех чрез който работата или се извършва успешно, тоест всички подзадачи приключват успешно, или се прекратява като действията на всички подзадачи се анулират.

В контекста на релационна база транзакциите има четири основни свойства [12]

- **атомарност** – транзакцията позволява група от промени върху една или няколко релационни таблици да формира атомарна операция която или приключва успешно или оставя базата непроменена
- **съгласуваност –**транзакциите винаги оперират върху съгласувана съвкупност от данни и след приключване оставят данните в съгласуван вид
- **изолираност –** всяка транзакция оперира върху данните все едно тя е единствената такава. Ефектът при конкурентна работа на няколко транзакции е невидим помежду им до тогава докато не приключат работа.
- **устойчивост –** веднъж приключила, ефектът от една транзакция е видим дори след ненормално спиране на системата. Ако транзакция не е

приключила то нейните промени не са устойчиви дори след ненормално спиране, а след последващ старт те ще бъдат анулирани

1.4.2.Локални и глобални транзакции

Най-простата форма на ползване на транзакции е този за достъп до данните към една релационна база. В този сценарий участват приложението което ползва данните, комуникационен модул за транспорт и достъп до базата наречен адаптер на ресурси и самата база от данни в роля на менажер (manager) на ресурси. Този вид транзакции се наричат локални транзакции.

Транзакция по време на която едно приложение осъществява достъп до повече от един менажер на ресурси се нарича глобална транзакция. Процесът на работа на такава транзакция се нарича процес на разпределена транзакция. Той включва менажер на транзакциите който е отговорен да координира работата на множеството от адаптерите на ресурси[11].

1.4.3.Програмен интерфейс

Java програмния интерфейс за транзакции – JTA (Java Transaction API), специфицира локални интерфейси между менажера на транзакциите и участниците в система с разпределени транзакции: приложението, менажера на ресурси и системата в която работи приложението [6].

JTA се състои от три основни части:

- Програмен интерфейс от високо ниво за ползване от приложения, позволявайки им да управляват и разграничават границите на транзакциите
- Съпоставяне между Java и индустриалния стандарт за *X/Open XA* протокола, което позволява менажер на ресурси да участва в глобална транзакция, контролирана от външен менажер на транзакции.
- Програмен интерфейс за менажер на транзакции от високо ниво, който позволява на системата, в която работят приложенията да разграничава и контролира границите на транзакциите. Обикновено приложенията

ясно заявяват, че искат това да се осъществява от системата прозрачно за тях.

1.4.3.1. Интерфейсът *UserTransaction*

Интерфейсът *javax.transaction.UserTransaction* осигурява на приложенията възможността програмно да контролират границите на транзакциите. Той може да бъде използван от програмни клиенти към J2EE платформа или от който и да е J2EE компонент като промишлен бийн или Web компонент.

Методът *UserTransaction.begin* стартира глобална транзакция и я асоциира с текущата нишка на приложението. Асоциацията нишка-транзакция се управлява прозрачно от менажера на транзакции.

Поддържането на вложени транзакции е незадължително. Ако нишката е вече асоциирана с транзакция и менажера на транзакции не поддържа вложени такива то при извикване метода *UserTransaction.begin* хвърля *NotSupportedException.*

Менажерът на транзакции е отговорен за препредаването на текущия контекст на транзакцията между приложенията. Форматът, по който се извършва това, е зависим от протоколът по който се осъществява комуникацията.

Препредаването се извършва прозрачно за приложенията.

1.4.3.2. Интерфейсът *TransactionManager*

Интерфейсът *javax.transaction.TransactionManager* позволява на системата, в която работят приложения да контролира границите на транзакциите от името на конкретно приложение. Например EJB контейнер управлява границите на транзакциите от името на промишлените бийнове. Контейнерът използва интерфейса *TransactionManager* главно за да определя границите на транзакциите, където операциите афектират върху транзакционния контекст на текущата нишка. Менажерът на транзакции управлява асоциацията на транзакционния контекст с нишките чрез свои вътрешни структури.

Транзакционния контекст асоцииран с една нишка е или *null* или реферира

конкретна глобална транзакция. Няколко нишки може да са асоциирани с една и съща глобална транзакция.

Поддържането на вложени транзакции е незадължително за менажера на транзакции.

Всеки транзакционен контекст е обвит с инстанция на обект от *Transaction*, който може да бъде използван за извършване на операции специфични за конкретната глобална транзакция, независимо от транзакционния контекст на текущата нишка.

Методите на интерфейсът *TransactionManager* позволяват да се стартира, приключва или абортира глобална транзакция. Допълнително интерфейсът има методи, с които текущата глобална транзакция може да бъде спряна, а по-късно възобновена без това да повлияе на успешното й приключване.

1.4.3.3. Интерфейсът *Transaction*

Интерфейсът *javax.transaction.Transaction* позволява да се извършват операции върху глобалната транзакция, с която той е асоцииран. Всяка глобална транзакция се асоциира с такъв обект още при създаването си. Операциите, които се предоставят от интерфейса са добавяне на ресурси към транзакцията, регистриране на обект за синхронизация, приключване или абортиране на транзакцията и получаване на информация за състоянието на транзакцията.

1.4.3.3.1. Синхронизиране с транзакцията

Синхронизирането с транзакцията позволява системата да получава съобщения от менажера на транзакции преди и след приключване на транзакцията. За всяка стартирана транзакция платформата може да регистрира *javax.transaction.Synchronization* обект за обратна връзка, който ще бъде извикан от менажера на транзакции.

Методът *Synchronization.beforeCompletion* бива извикан в началото на процедурата по завършване на транзакцията като текущата нишка е асоциирана с глобалната транзакция която приключва.

Методът *Synchronization.afterCompletion* бива извикат след приключване на глобалната транзакция като състоянието и се предава като параметър на метода.

1.5. EJB технология

Технологията за промишлени Java бийнове *EJB* е J2EE технология за разработване на бизнес компоненти за обектно ориентирани промишлени Java приложения. Бизнес компоненти, разработени върху EJB технологията са наричани често EJB компоненти или за по-просто промишлени бийнове. Обикновено промишлените бийнове предоставят логиката и представляват данните необходими за извършване на операции специфични за някоя област на бизнеса като банково дело и търгуване. Например някой бийн (може би в съвкупност с други такива) може да предостави данните и логиката необходима за изпълнение на операции върху банкови сметки като кредитиране и управление на дебита по сметка. Други бийнове биха могли да предоставят данните и логиката необходими за изпълнение на операции върху *"чанта за пазаруване"*, които позволяват на клиенти да пазаруват стоки *online* от магазин за търговия.

Общоприето е твърдението, че EJB технологията е мощна и усъвършенствана. Тя помага на разработчиците да изградят бизнес приложения, които отговарят на високопроизводителните нужди на промишлената среда. По-конкретно, приложения изградени чрез EJB и други J2EE технологии са сигурни, могат да поддържат едновременно голям брой потребители и транзакции, така че данните запазват своята цялостност независимо че се обработват конкурентно от много потребители.

Тези изисквания се постигат посредством услугите предоставени от EJB средата по време на работа – EJB контейнер. Това означава, че услуги като

конкурентен достъп до данните и информацията по сигурността са автоматично предоставени от EJB контейнер до всеки промишлен бийн.

Независимо от своята мощност и това че е усъвършенствана, някои разработчици са колебливи към прилагането на EJB технологията. Най-голямата пречка, които тези разработчици виждат, е комплексната сложност на технологията. Но все пак най-новото издание на EJB технологията – EJB 3.0 се фокусира главно върху преработване на технологията с цел та да стане по-лесна за използване от разработчиците, но не на цената на нейната мощност. Идеята е не само да се улеснят тези които разработват върху технологията, но и да се привлекат нови такива. Новото издание запазва съществуващите програмни интерфейси с цел приложения разработени върху предишни версии да са функциониращи и върху най-новата версия.

1.5.1.EJB Контейнер

Промишлените бийнове предоставят логиката и представляват данните необходими за извършване на специфични бизнес операции. Промишлените бийнове са преносими – може да се приложат към всяка J2EE съвместима система. Това осигурява на разработчиците гъвкавост в смисъл на разпределеност на компонентите на приложението. Също така промишлените бийнове могат да се използват многократно в различни приложения, но може би най-голямото предимство за промишлените бийнове, е че те са управляеми.EJB контейнерът осигурява системни услуги като управление на транзакции с цел за да се запази съвкупността от данни цяла в среда с много потребители и управление на сигурността, за да се предпази системата от неправомерен достъпът до ресурси. Поради това че EJB контейнерът осигурява тези и други системни услуги, разработчика не е длъжен да ги включва към приложението позволявайки му да се съсредоточи изцяло и само върху аспектите на бизнес логиката и управлението на данните.

Съществуват три типа промишлени бийнове: сесийни, единични и съобщителни. Често в едно приложение се използва съвкупност от тях.

1.5.1.1. Сесийни бийнове

Един сесиен бийн представлява уникална сесия между потребител и инстанция на бийна. Той не може да се споделя между различни потребители. Една инстанция на бийн е обвързана с конкретен клиент и конкретна негова сесия. Всеки сесиен бийн предоставя методи, които клиентът има възможност да извика с цел изпълнение на бизнес задачи в системата. Когато сесията на клиента приключи се приема, че сесийния бийн не е асоцииран с клиента.

Има два вида сесийни бийнове: със състояние и без състояние. Един състоятелен сесиен бийн управлява данните за единична и уникална клиент-бийн сесия посредством променливите си на ниво Java инстанция. Данните представляват състоянието, наричано още разговорно състояние, в което се намира конкретната сесия. Разговорното състояние се поддържа докато съществува асоциацията клиент-бийн. Същественото от това е че данните се запазват и поддържат между операциите. Когато един от методите на бийна приключи и се извика друг метод в същата сесия, разговорното състояние се запазва и предава от последната операция към предстоящата следваща. Това прави състоятелните сесийни бийнове полезни при реализирането на *"чанта за пазаруване"* или банкови услуги, в чиято реализация има множество свързани една с друга операции. За управлението на разговорното състояние в среда с ограничени ресурси – най-вече памет, EJB контейнерът може да използва второстепенно устройство като релационна база данни или файлова система.

В сравнение, несъстоятелните сесийни бийнове не поддържат разговорно състояние спрямо потребителите си. Поради тази причина те се използват предимно за реализиране на операции състоящи се само от една задача или стъпка като например изпращането на електронно писмо за потвърждаване на *online* поръчка. Контейнерът за EJB никога не записва несъстоятелен сесиен бийн на второстепенно устройство. Една съществена разлика за тези бийнове е, че те може да реализират *web* услуги.

1.5.1.2. Единични бийнове

Един единичен бийн представлява данните съхранени в междинен носител като например релационна база от данни. Всеки единичен бийн представлява таблица или релация в релационната база и на всяка инстанция на единичния бийн съответства елемент на релацията или ред от таблицата. Единичните бийнове не са ограничени само върху релационни бази. Те могат да представляват данни от всеки тип носител на данни. Но по-голямата част от бизнес приложения които използват EJB технологията осъществяват достъп предимно до данни съхранени в релационни бази.

Основната дума, на която трябва да се обърне внимание е "представляват". Един единичен бийн не е част от релационната база, а по скоро съдържа данни които са заредени от базата в определен момент. В последствие тези данни се записват отново в базата в определен момент.

Един единичен бийн се различава от сесиен бийн в няколко аспекта. Докато сесийния бийн не може да се споделя между клиенти то единичния може, тоест много клиенти могат да използват един единичен бийн едновременно. Това позволява няколко потребителя да използват едновременно данните, които даден единичен бийн представлява. Тук е мястото, където управлението на транзакциите осигурено от EJB контейнерът е важно. Допълнително единичните бийнове притежават свойството *персистентност*. Това означава че единичните бийнове продължават да съществуват дори след спиране на приложението което ги ползва. Това е възможно тъй като данните от релационната база, които даден единичен бийн представлява, са също *персистентни*. Поради това състоянието на конкретен единичен бийн може да бъде реконструирано дори след ненормално спиране на системата, в която се намира в сравнение със сесиен бийн, чието състояние се запазва само докато има потребителска сесия за състоятелен или само по време на изпълнение на метод за несъстоятелен. Също така единичен бийн може да е в релация с друг единичен бийн по същия начин както една релационна таблица може да е в релация с друга. Например в приложение за записване на студенти този единичен бийн, който репрезентира данните за студентите, може да е в релация с този единичен бийн, който

описва данните за класовете. В сравнение релации между сесийни бийнове няма.

Всеки единичен бийн описва дали той сам ще управлява своето *персистиране* (*bean-managed*) или ще разчита на работата на EJB контейнера (*container-managed*). В първия случай Java кода на бийна ще съдържа SQL заявки които осъществяват достъпа до базата. Във втория случай този код е автоматично генериран от EJB контейнера.

1.5.1.3. Съобщителни бийнове

Един съобщителен бийн обработва асинхронни съобщения изпратени предимно чрез стандартния за Java програмен интерфейс за изпращане на съобщения – JMS (Java Message Service). Асинхронното изпращане на съобщения освобождава изпращача от задължението да чака отговора на получателя.

Един съобщителен бийн може да обработва съобщения пратени от произволен J2EE компонент като например друг бийн или Web компонент, от приложение ползващо JMS и дори от друга система, която не ползва J2EE технология. Това прави този вид бийнове *приложими* към много бизнес-към-бизнес комуникационни сценарии.

1.5.2.Интерфейси и клас на бийн

Въпреки че най лесният начин да се опише извикване на бийн от негов клиент е посредством извикване на неговите методи, това не е така. За сесийни или единични бийнове клиентът извиква методи дефинирани в Java интерфейсите за съответния бийн. Тези интерфейси съставляват така наречения потребителски изглед на конкретния бийн. Всеки сесиен или единичен бийн трябва да има два такива интерфейса – компонентен интерфейс и базов интерфейс – както и Java клас представляващ класа на бийна. Никой не може да ползва директно съобщителен бийн. Достъпът до него се осъществява посредством програмния интерфейс за съобщения, който

предоставят изпратените съобщения чрез стандартния за протокола краен интерфейс, реализиран от съобщителния бийн.

Компонентните интерфейси дефинират бизнес методите на конкретен промишлен бийн. Например компонентния интерфейс на сесиен бийн, който управлява банкови сметки може да дефинира един метод за извличане по сметка, а друг за постъпване по сметка. Базовият интерфейс дефинира методи за управление на жизнения цикъл на инстанциите. Те дефинират методи за създаване на инстанции и за тяхното изтриване. Изтриването на инстанция в известен смисъл означава и правенето й недостъпна. Базовият интерфейс на единичен бийн може да дефинира още два вида методи. Това са методи за откриване и базови методи. Методите за откриване са за локализиране на една или няколко конкретни инстанции. Базовите методите са бизнес методи, които се извикват логически върху цялото множество от инстанции.
Класът на бийна реализира всички методи дефинирани от интерфейсите.

2. Анализ на *Core* и *Simplified* спецификациите за промишлени бийнове спрямо сесийни бийнове

Спецификацията за промишлени бийнове като част от Java промишлената платформа, се разработва и развива на базата на процеса на Java общността – *JCP* (Java Community Process). Основна негова цел е дефиниране на общи модели за разработване, доставяне и опериране на бизнес приложенията. Стандартизирайки живота на едно приложение от момента на създаването му до момента, в който той влиза в действие улеснява всички участници, били те разработчици на приложения, разработчици на система или системни администратори. Работейки с предварително определени програмни интерфейси и сценарии, всеки един от участниците е способен да съсредоточи усилията си в своята област. Не по-малко важен аспект е свободния избор на система и приложенията в нея, който се предоставя пред крайните потребители.

Като основа за договаряне и разпределение на отговорностите, *EJB Core* спецификацията дефинира няколко роли с които адресира участниците в различните сценарии. Под *доставчик на промишлени бийнове* се визира този който разработва и предоставя *промишлен бийн* като краен продукт. Той отговаря за реализирането на бизнес методите, като за дефинирането на конкретните интерфейси за тяхното ползване от клиенти и за осигуряване на допълнителната информация за бийна под формата на Java анотации или XML. Под *преводач на приложение* се визира ролята на този, който пакетира няколко *промишлени бийна* заедно, като и с евентуално други видове компоненти. *Внедрителят,* внедрява получения пакет от *преводач на приложение* като е отговорен за решаване на всички зависимости, които приложението дефинира към външни ресурси. Той гарантира, че тези ресурси съществуват на системата, в която се внедрява приложението. *Доставчик на EJB платформа* е роля, която описва специалист в областта на разпределени транзакционни процеси, разпределени и отдалечени обекти и други системи от ниско ниво. По принцип се счита че тази роля и *доставчика на EJB контейнер* са един и същ. От своя страна *доставчика на EJB контейнер* е част от платформата, в която работят

промишлени бийнове, предоставяйки им среда, достъп до други ресурси, поддръжка на транзакции, поддръжка по сигурност, достъп по мрежа за отдалечени клиенти, мащабност по време на работа както и други услуги, по принцип необходими, за изграждането на управляема платформа.

2.1. Цели на архитектурата

Спецификацията EJB поставя няколко основни цели пред себе си. Това са изграждането на стандартна архитектура за компоненти в обектно ориентираните бизнес приложения, поддръжка на жизнения цикъл за компонентите, възможност за комуникация на компонентите както помежду си така и с външни системи по стандартни протоколи, автоматично и прозрачно предоставяне на услуги от ниско ниво като транзакции и управление на нишки, гарантиране на съвместимост на компонентите независимо от платформата [3].

Наред със запазване на всички тези цели EJB 3.0 визира подобряване на архитектурата чрез намаляване на нейната сложност от гледна точка на разработчиците на приложения [1]. Основен пункт в това заема дефинирането на Java анотации които да заместят съществуващите описания за внедряване под формата на XML документи. С тяхна помощ се намалява броя на интерфейсите които разработчика на приложение е необходимо да реализира. Допълнително се премахва нуждата от някои интерфейси, а останалите се освобождават от необходимостта да наследяват *EJBObject* или *EJBLocalObject*. Част от задълженията на разработчика по инициализацията на бийновете се прехвърлят на EJB контейнера под формата на *внедряване по зависимост*.

2.2. Сесиен бийн от гледна точка на клиента

За клиента сесиен бийн е не персистентен обект, опериращ на платформата, която той достъпва и реализиращ конкретна бизнес логика. Един сесиен обект не се споделя между различни клиенти. Клиентът никога не оперира с

конкретна инстанция от клас на бийна директно. Вместо това той работи чрез някой от интерфейсите разработени и предоставени от *доставчика на бийнове*. Гледната точка на клиента към сесиен бийн се състои от това как той получава референция към негов интерфейс, кои негови методи извиква, как определя състоянието му и го различава от останалите бийнове.

2.2.1.Локални и отдалечени клиенти

По своето естество и по начина на достъп който те осъществяват клиентите се делят на две групи. Това са клиент, които осъществяват достъп от същата система в която е и бийна, и такива които осъществяват отдалечен достъп.

Отдалечените клиенти имат достъп до бийна само през тези интерфейси, които са обявени от *доставчика на бийна* за отдалечени. Важна характеристика за този вид достъп, е че резултатите от изпълнение на бизнес операциите се предават по стойност. Ползата за този вид клиенти е че те са по гъвкави спрямо конкретното място на което са внедрени компонентите. Негативите са в това, че подобен вид работа включва забавянето по мрежата както и непрекъснатото копиране на аргументите.

Клиентите намиращи се в същата виртуална машина достъпват бийна през тези интерфейси които са обявени от *доставчика на бийна* за локални. Предимствата на този вид обръщение, е че аргументите се предават по адрес, с което с ускорява процеса и се осъществява споделяне на данни. Тази характеристика е и недостатък, тъй като не бива да се забравя да се копират данните когато те не бива да се споделят.

2.2.2.Достъп на клиент до сесиен бийн по EJB 3.0 интерфейси

Клиент достъпва сесиен бийн или чрез изрично ползване на JNDI или чрез механизмът за *внедряване по зависимост.* По този начин той получава референция към някой от бизнес интерфейсите на бийна. Този интерфейс е нормален Java интерфейс. Декларираните методи в него са бизнес методите на

бийна. Клиентът има право да извика тези методи или да предаде референцията, която е получил като параметър на друг метод.

Бизнес интерфейса на бийна обикновено съдържа метод за инициализация на състоянието на обекта на сесията, както и метод за информиране, че клиентът е приключил работа с нея. Не е коректно да се запазва референция към бийна след приключване на работа. Достъп до подобен бийн завършва с генериране на изключението *javax.ejb.NoSuchEJBException*.

Доставчикът на EJB контейнер е отговорен за предоставянето на реализация на бизнес интерфейса, такава, че извикването на неин метод предизвиква извикване на съответните прихващащи класа от приложението и най-накрая на съответния метод от инстанцията на сесийния бийн.

От гледна точка на клиента, инстанцията на сесийния бийн започва да съществува в момента, в който той получава референция към бизнес интерфейса. От този момент клиентът може да вика методи на бизнес интерфейса. За клиента жизнения цикъл на състоятелен сесиен бийн завършва с извикването на бизнес метод отбелязан с анотацията *Remove*. При несъстоятелен сесиен бийн извикването на подобен метод е незадължително тъй като премахването на сесията се извършва от д*оставчика на EJB контейнера* прозрачно за клиента.

За различаване и определяне на конкретни референции на сесиен бийн, клиентите разчитат на реализацията на *Object.equals* метода от предоставения бизнес интерфейс. За несъстоятелен бийн две референции са еднакви тогава и само тогава, когато представляват един и същ сесиен бийн и един и същ негов бизнес интерфейс. За състоятелен бийн две референции са еднакви, ако представляват една и съща сесия през един и същ бизнес интерфейс.

2.2.3.Достъп на клиент до сесиен бийн по EJB 2.1 интерфейси

По-ранните спецификации изискват от клиента да получи референция към базов интерфейс на бийна след което използвайки този интерфейс да получи референция към компонентния интерфейс на бийна[2]. Този програмен модел продължава да се поддържа и от *EJB 3.0* спецификацията. Клиент достъпва

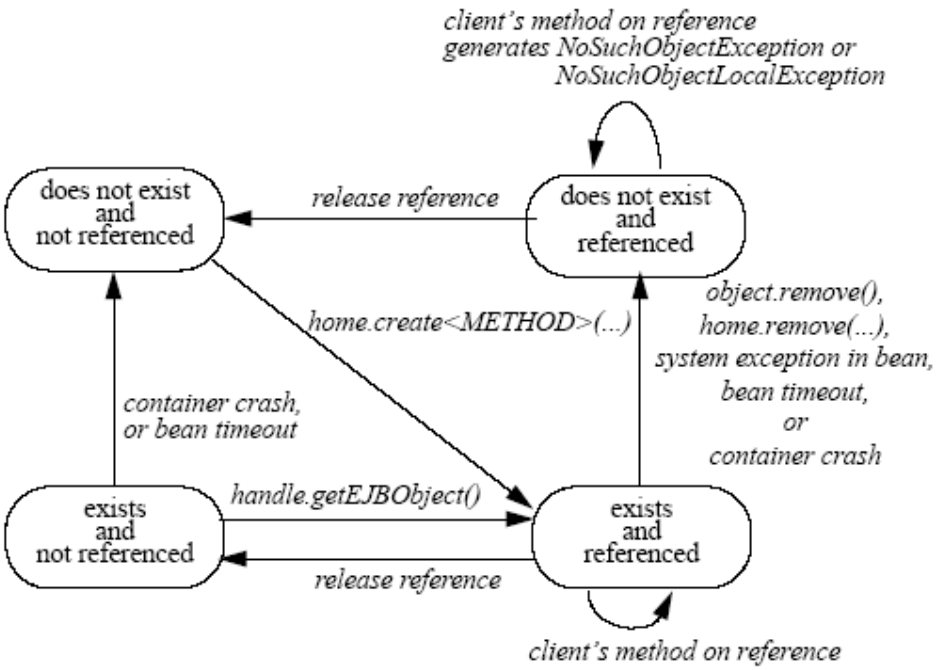
базовия интерфейс на сесиен бийн или чрез изрично ползване на *JNDI* или чрез механизмът за *внедряване по зависимост*.

В случаите на достъп до отдалечен базов интерфейс чрез ползване на *JNDI* с цел съвместимост клиентът е длъжен да приложи *PoratableRemoteObject.narrow* метода към резултата специфицирайки отдалечения интерфейс.

Доставчикът на EJB контейнер е отговорен за предоставянето на реализация на базовите и компонентните интерфейси за даден бийн в случаите, когато такива са асоциирани от *доставчика на бийна* с бийна.

Базовите интерфейси съдържат един или повече методи от вида *create<METHOD>,* чрез които се създава нов сесиен бийн. Обикновено техните аргументи служат за инициализация на състоянието на бийна, а типа на върнатата стойност е съответния компонентен интерфейс. Премахването на бийн след приключване на работа с него се извършва посредством някой от методите *EJBObject.remove*, *EJBHome.remove(Handle)* или *EJBLocalObject.remove* приложени съответно на отдалечения компонентен интерфейс, отдалечения базов интерфейс или локалния компонентен интерфейс, като това зависи от вида на клиента – отдалечен или локален.

Състоянията, в които може да се намира една сесия от гледна точка на клиента са четири както е изобразено на Фигура 1.



Фигура 1.

Състояния на сесия [3].

Това са всички комбинациите спрямо това дали сесията съществува и дали клиента я реферира. Началното състояние е, когато клиента реферира само базовия интерфейс, но не реферира компонентния. Чрез извикване на един от *create<METHOD>* методите клиентът едновременно създава сесия и получава референция към нея. В това състояние клиентът може да извиква неограничено бизнес методите от интерфейса на бийна. Ако възникне системна грешка или клиентът сам приключи работа с бийна или след дълго отсъствие на активност, сесията се премахва но клиента все още реферира интерфейса. Той може да се освободи от него, с което се връща до началното състояние. Ако клиентът се освободи от референцията преди приключване на работа, то сесията остава активна в системата до тогава докато не се премахне поради изтичане на определено време или поради спирането на системата.

Клиентът различава сесиите, които ползва чрез методите *EJBLocalObject.isIdentical* или *EJBObject.isIdentical*. Както при *EJB 3.0* референции към компонентен интерфейс са еднакви тогава и само тогава

когато представляват една и съща сесия. Всички референции към компонентен интерфейс за несъстоятелна сесия на един и същи бийн са еднакви.

2.3. Компонентен договор за сесиен бийн

Инстанция на сесиен бийн е инстанцията на класа на бийна. Тя съдържа състоянието на сесията. Инстанцията на сесиен бийн представлява абстрактно разширение на състоянието на клиента, който я създава. Тя съдържа разговорно състояние от името на клиента, изпълнява операции от името на клиента и в случай на състоятелна сесия, клиента управлява жизнения й цикъл. Контейнерът управлява жизнения цикъл на инстанциите на бийна, съобщавайки на инстанциите за събития, когато е нужно действие от тяхна страна. Той осигурява пълен набор от услуги за да гарантира, че реализацията на бийна е мащабна и може да поддържа голям брой потребители.

2.3.1.Разговорно състояние на състоятелни сесийни бийнове

Разговорното състояние (conversational state) на конкретен състоятелен сесиен бийн се дефинира като множеството състоящо се от всички полета на инстанцията на бийна, всички асоциирани инстанции на прихващащи класове и техните полета също и всички транзитивни *Java* референции произтичащи от тези обекти.

За ефикасно ползване на системната памет, EJB контейнерът може временно да прехвърли състоянието на някоя неизползвана в момента сесия към второстепенно устройство за съхранение. Тази операция се нарича *пасивиране* (passivate). Обратната на тази операция, тоест когато сесията се възстановява от второстепенното устройство, се нарича *активиране*.

В по-сложен сценарий разговорното състояние на сесията може да съдържа отворени системни ресурси. Тогава разработчика на сесийния бийн е длъжен да затвори и в последствие отвори отново тези ресурси в обозначените съответно с *PrePassivate* и *PostActivate* анотации методи на някой от прихващащите

класове или бийна. Контейнерът може да *пасивира* бийн, само ако той не е асоцииран с транзакция.
Разговорното състояние на сесиен бийн не е транзакционно в смисъл, то не се възстановява до на началното си, ако текущата транзакция се абортира. В такъв случай разработчика на бийна е длъжен сам да синхронизира състоянието на бийна използвайки *afterCompletion* метода.

2.3.2.Протокол между инстанцията на бийна и EJB контейнера

Един клас се дефинира като сесиен бийн или ако са му е приложени анотациите *Stateless* или *Stateful или ако* е описан като такъв в *XML* описанието за внедряване. Тези анотациите определят дали бийна е съответно несъстоятелен или състоятелен.

Контейнерът за бийнове не изисква никакви услуги от инстанцията на бийна. Негово задължение е да управлява инстанциите, осигурявайки им услуги и доставяйки им съобщения за възникналите операции.

Механизмът за *внедряване по зависимост* е част от задълженията на контейнера. Той е разработен с цел да замести необходимостта разработчика на бийн да реализира локализирането и инициализирането на ресурсите които ползва инстанцията. Използвайки анотации, например *Resource*, *EJB*, *PersistenceContext* и други върху определени полета или методи на бийна разработчика дефинира ресурсите, които изисква да бъдат *внедрени*. От своя страна контейнерът е длъжен да инициализира тези полета или да извика методите непосредствено след създаването на инстанцията на бийна и преди да се извика, който и да е негов бизнес метод. Ако класа на бийна реализира EJB 2.1 интерфейса *SessionBean,* то неговия метод *setSessionContext(SessionConext)* се възприема за част от *внедряването по зависимост*.

Освен пасивния протокол за *внедряване по зависимост* и свободния достъп до *JNDI*, контейнерът предоставя на бийна реализация на интерфейса *SessionContext*. Той представлява програмния интерфейс за комуникация между бийна и контейнера. Чрез него бийна получава достъп до информация по сигурността, контрол върху транзакциите, улеснено ползване на *JNDI* и

функции за получаване на референции към конкретен свой базов, компонентен или бизнес интерфейс.

Контейнерът информира инстанцията на бийна за прехвърлянето и от едно състояние в друго по време на управление на жизнения и цикъл, чрез извикване на конкретни методи на инстанцията на бийна и на прихващащите класове. Кои методи трябва да се извикат, контейнерът определя по поставените им анотации:

- *PostConstruct* – тази анотация определя метода, който контейнерът извиква преди да извика първият бизнес метод и след изпълнението на *внедряването по зависимост*. Ако бийна е несъстоятелен и е реализиран по *EJB 2.1* или по-ранна версия то такъв метод е и *ejbCreate* метода на класа на бийна (*доставчика на бийна* е длъжен да реализира точно един такъв). Разликата между състоятелен и несъстоятелен бийн е основна, тъй като при състоятелния бийн, клиентът е този, който инициира извикването на *SesionBean.ejbCreate<METOD>* методите за да инициализира разговорното състояние на бийна.
- *PreDestroy* **–** тази анотация определя метод, който контейнерът извиква за да информира инстанцията на бийна, че тя се премахва от системата. Ако бийна е реализиран по *EJB 2.1* или по-ранна версия за такъв метод се приема и *SessionBean.ejbRemove*.
- *PrePassivate* **–** тази анотация определя метод, който контейнерът извиква за да информира инстанцията на бийна, че тя се *пасивира*. Ако бийна е реализиран по *EJB 2.1* или по-ранна версия за такъв метод се приема и *SessionBean.ejbPassivate*
- *PostActivate* – инстанцията на бийна, че тя се *активира*. Ако бийна е реализиран по *EJB 2.1* или по-ранна версия за такъв метод се приема и *SessionBean.ejbActivate*

За да реализира синхронизиране на разговорното си състояние с транзакцията, един сесиен бийн реализира интерфейса *javax.ejb.SessionSynchronization*.В този случай контейнерът е задължен да извика методите на бийна от този интерфейс в определени моменти спрямо началото и края на транзакцията.

- *SessionSynchronization.afterBegin* – контейнерът е длъжен да извика този метод преди извикването на първия бизнес метод на конкретния бийн в конкретна транзакция. Той служи да информира бийна, че е асоцииран с нова транзакция и му дава възможността да запази копие от текущото си състояние с цел то да бъде възстановено при неуспех.
- *SessionSynchronization.beforeCompletion* – контейнерът е длъжен да извика този метод, когато транзакцията приключва, но преди всеки от включените в нея менажери на ресурси да се завършил изпълнението си. В голяма степен този метод е като метода *Synchronization.beforeCompletion.*
- *SessionSynchronization.afterCompletion* – контейнерът е длъжен да извика този метод, когато транзакцията е приключила като специфицира като аргумент дали това е било успешно – *true* или неуспешно – *false*. При неуспешно приключване, инстанцията има възможността да възстанови състоянието си преди началото на транзакцията. В голяма степен този метод е като метода *Synchronization.afterCompletion*

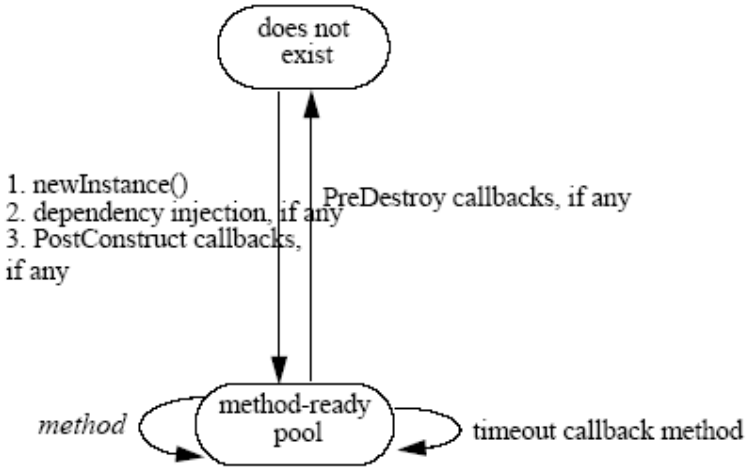
Бизнес методите на бийна са тези методи, които клиентите извикват през предоставените от контейнера реализации на интерфейсите на бийна. Контейнерът е длъжен на всяко извикване на бизнес метод от клиента да съпостави извикване на съответния метод от класа на бийна. В допълнение на това контейнерът е длъжен да извика всички методи от бийна или прихващащите класове, които имат анотацията *AroundInvoke* преди да извика съответния бизнес метода на бийна.

Процеса на инициализиране на състоятелен сесиен бийн се предизвиква от извикването на някой от *create<METHOD>* методите, дефинирани в базовите интерфейси. За всеки такъв метод *доставчикът на бийна* е длъжен да предостави съответна реализация на метод със същата сигнатура. Ако се следва EJB 2.1 или по-ранните версии този съответен метод трябва да се казва *ejbCreate<METHOD>*. В случаите, когато името на метода не съвпада, *доставчика на бийна* е длъжен да съпостави метод използвайки анотацията *Init*. Следвайки тези правила за именуване, контейнерът е длъжен на всяко

извикване на *create<METHOD>* да съпостави извикване на съответния *ejbCreate<METHOD>* метод от бийна.
Тъй като сесиите на един бийн са несподелени, контейнерът е длъжен да гарантира, че в един момент само един клиент работи с инстанция на бийн.

2.3.3.Жизнени състояния на сесийни бийнове

Контейнерът на бийнове управлява живота на инстанцията на бийна прозрачно за клиентите. Фигура 2 описва състоянията в които може да се намира една инстанцията на несъстоятелен бийн.

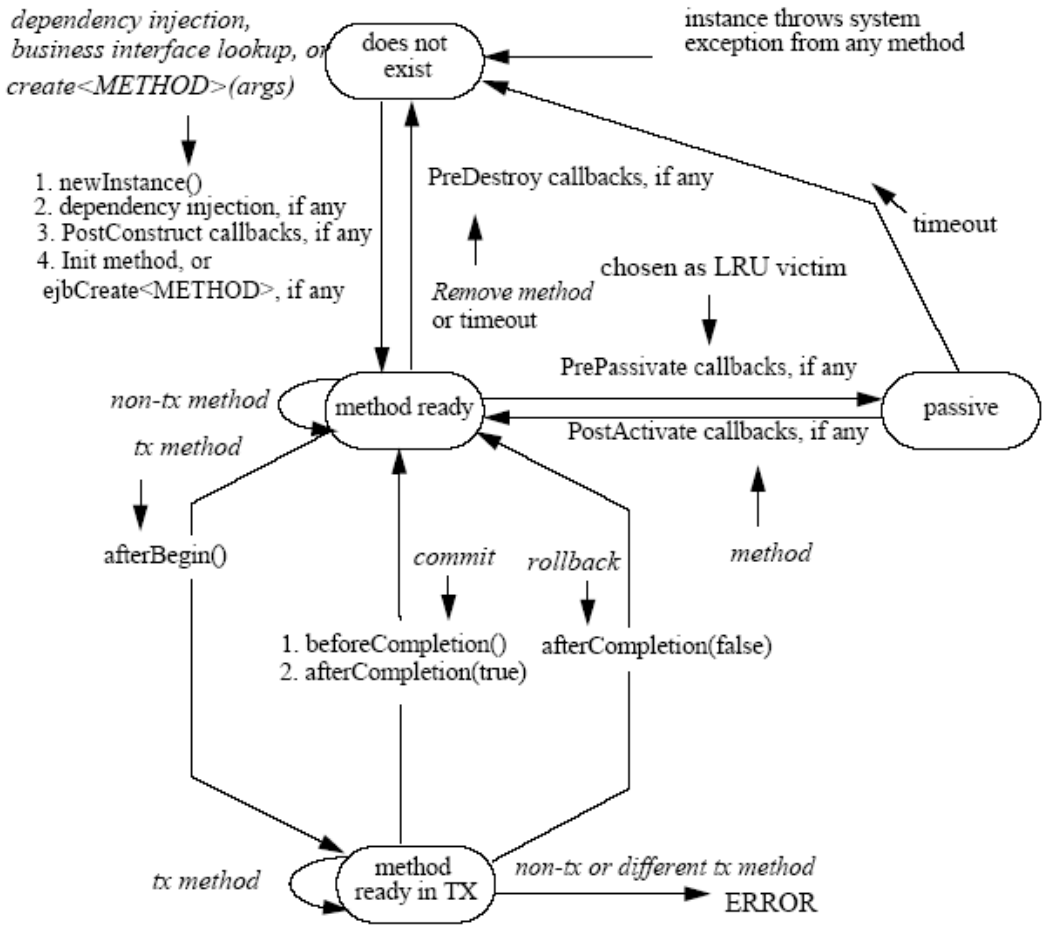


Фигура 2.
Жизнени състояния на несъстоятелен сесиен бийн[3].

Веднага след създаването си, несъстоятелната сесия преминава в единственото състояние в което може да се намира тя. В процеса на обслужване на клиенти, контейнера използва която и да е инстанция намираща се в това състояние, или създава нова ако в момента няма свободна такава. Премахването на инстанция се контролира изцяло от контейнера, който преценява кога то да се извърши. Например някои реализации на контейнер може да създава и премахва инстанциите всеки път. Реално този подход е твърде неефективен, тъй като се налага изпълняване на *PostConstruct* и

PreDestroy методите както и *внедряването по зависимост* при всеки бизнес метод. По-ефикасен подход е реализирането на *пул* от инстанции в *готово* състояние. Атрибутите на този *пул* ефективно може да следят за използваната памет свеждайки използваните ресурси до предварително известни граници. По-този начин малък брой инстанции може да обслужат многократно по-голям брой клиенти.

За разлика от несъстоятелните бийнове, жизнения цикъл на състоятелните е много по-сложен. Фигура 3 описва кои са състоянията на състоятелен сесиен бийн и начините, по които контейнера реализира преминаването им в тях.



Фигура 3.

Жизнени състояния на състоятелен сесиен бийн[3].

Инстанция на състоятелен сесиен бийн се създава и унищожава само при формулирана заявка от клиент като през времето през което тя съществува е

асоциирана само с този клиент. Това изключва възможността за редуциране броя на инстанциите спрямо броя на клиентите, тоест във всеки един момент контейнерът съхранява всички активни инстанции на бийнове. Жизнения цикъл дефинира стандартен начин за контролиране на използваната памет от контейнера чрез *активиране* и *пасивиране* на инстанциите. Ако клиент използва една сесия в транзакционен контекст, то контейнерът е отговорен да асоциира съответната инстанция на бийн с този контекст. Инстанцията остава асоциирана с транзакционния контекст до тогава докато не приключи транзакцията.

2.4. Прихващащи класове

Прихващащите класове е нова концепция, която *EJB 3.0* спецификацията въвежда. Тя е заимствана от някой съществуващи спецификации и подкрепя в известна степен *аспектно* ориентираното програмиране. Идеята за прихващане визира това, конкретен модел за изпълнение на заявки от клиент да бъде динамично разширяем. Това става с помощта на нова компонентна част, наречена *прихващащ клас.* При дефинирана такава част, системата обработваща заявките я прилага при обработката и изпълнението на всяка една заявка. По този начин всяко поведение, което не зависи от конкретното приложение и компонент може да се реализира веднъж и да се приложи динамично за всички компоненти без да е необходимо самите компоненти да се променят.

Прихващащ клас в *EJB 3.0* спецификацията е нормален *Java* клас, различен от класа на бийна, чийто методи се изпълняват при изпълнение на бизнес методи или събитие от жизнения цикъл не бийна. Всеки бийн се асоциира с нула или повече прихващащи класове като това става чрез прилагане на анотацията *Interceptors* или чрез *XML* описанието за внедряване.
Инстанциите на прихващащите класове споделят информация помежду си и с бийна, използвайки интерфейса *InvocationContext*

2.4.1.Жизнен цикъл на инстанциите на прихващащ клас

Жизнения цикъл на инстанция на прихващащ клас е същият както този на инстанцията на бийна, с която той е асоцииран. Когато контейнерът изпълнява дадена операция от жизнения цикъл на бийна той е длъжен да я приложи и върху прихващащите класове асоциирани с бийна. Създаването на инстанция на бийн води до създаване на инстанции на прихващащите класове. Премахването на тези инстанции става само ако се премахне и бийна.
Както класът на бийна така и всеки прихващащ клас може да съдържа състояние и подлежи на *внедряването по зависимост.* Реализацията им има същия достъп до ресурси както и реализацията на бийна. Прихващащите класове споделят същата *JNDI* среда както и бийна.

2.4.2.Прихващане на бизнес методи

Метод на прихващащ клас или на бийна с анотация *AroundInvoke* се изпълнява по време на извикване на бизнес метод от клиент. Такъв метод трябва винаги да има *Java* сигнатура

Object <METHOD>(InvocationContext) throws Exception.

Тъй като може да има няколко такива метода в различни класове то изпълнението им се извършва в определен ред. Този ред зависи от реда на дефиниране на прихващащите класове, от това дали метода е от класа на бийна или не, дали е специфичен само за този бизнес метод или се прилага за всички бизнес методи както и от позицията на метода спрямо родителското дървото на прихващащия клас. Определянето на този ред изисква съпоставяне на много и различна по рода си информация.
При извикване на метод от прихващащ клас, контейнерът предава изцяло контрола на метода. Реализацията на метода има право да върне контрола на контейнера чрез извикване на *InvocationContext.proceed*. При възникване на изключение, методът може да опита да изпълни повторно *InvocationContext.proceed*, да обработи изключението и да върне нормален резултат или да предизвика изключение. Клиентът ще получи точно този резултат, който първият метод на прихващащ клас върне. Подобен вид дейност наподобява поточна линия в завод – параметрите и резултата от бизнес метода

се дообработва от всеки прихващащ клас преди да се предаде на следващия във веригата. Реално контейнерът е отговорен да свърже тази верига, реализирайки интерфейса *InvocationContext*

2.4.3.Прихващане на събития от жизнения цикъл на бийна

Методите на прихващащ клас или на бийна с анотации *PostConstruct, PreDestroy, PostActivate* и *PrePassivate* се изпълняват по време на събитията от жизнения цикъл на бийна. Един метод може да има повече от една от тези анотации. Ако е дефиниран в прихващащ клас сигнатурата на метода трябва да е от вида

void <METHOD>(InvocationContext) throws Exception

Ако е дефиниран в класа на бийна сигнатурата на метода трябва да е от вида

void <METHOD>() throws Exception

Казаното за методите с *AroundInvoke* анотация в голяма степен важи и за тези методи. Изключение е факта, че веригата от методи може да не завършва с метод от класа на бийна, тъй като той може да няма метод с конкретната анотация. В такъв случай последното извикване на *InvocationContext.proceed* представлява празна операция.

2.4.4.Интерфейс *InvocationContext*

Обектът от реализацията на *InvocationContext* осигурява на веригата от прихващащи обекти данни и функционалност за контролиране на поведението на извикване. По време на извикване на бизнес метод, на неговите прихващащи методи се подава една и съща инстанция на *InvocationContext.* Тази инстанция не се споделя между няколко бизнес метода.

Методите на *InvocationContext* са:

- *Object getTarget()* – връща инстанцията на бийна
- *Method getMethod()* – връща метода на класа на бийна или *null*

- *Object[] getParameters()* – връща аргументите, с които е извикан бизнес метода или стойностите последно предадени на *setParameters(Object[]),* ако е бил извикван
- *void setParameters(Object[])* – подменя параметрите, с които ще се извика бизнес метода
- *Map<String,Object>getContextData()* – връща обект съдържащ текущите данни, които всички участници по веригата са асоциирали с нея
- *Object proceed()* – предава задачата по изпълнението на веригата на контейнера. Връща резултата от изпълнените на бизнес метода или *null*

2.5. Поддръжка на транзакции

Основна характеристика на *EJB* архитектурата е поддръжката на разпределени транзакции, които може да включват достъп до различни ресурсни системи дори при съвместна работа на различни *EJB* контейнер реализации. Спецификацията не изисква от реализациите поддръжка на вложени транзакции.
Доставчика на бийн преценява дали да бийна програмно ще определя границите на транзакциите или ще остави това задължение на контейнера.

2.5.1.Контрол на границите на транзакции от бийн

Бийнът контролира границите на транзакциите през интерфейса *UserTransaction*. Поради това, че има пълен контрол върху транзакцията, бийна не може да разчита на методите на *SessionSynchronization* интерфейса. Сесиен бийн няма право да започва втора транзакция преди да е приключил първата.

На състоятелен сесиен бийн е позволено да приключи изпълнението на бизнес метод без да приключи транзакцията. В този случай контейнерът е отговорен да запази транзакцията и следващия бизнес метод да се изпълни отново в нея. Това важи до момента в който бийна реши да приключи транзакцията сам.

На несъстоятелен сесиен бийн не е позволено да приключи изпълнението на бизнес метод без да приключи транзакцията. Контейнерът е длъжен да определя случаите в които това не е изпълнено, да абортира транзакцията и да премахне инстанцията на бийна.

2.5.2.Контрол на границите на транзакции от контейнера

Предоставяйки контрола на транзакциите на контейнера на бийна се забранява ползването на *UserTransaction* интерфейса както и програмно да приключва какъвто и да е ресурс, които се добавен към транзакцията. Единствения контрол и информация върху транзакцията бийна има от методите на *EJBContext* интерфейса *setRollbackOnly* и *getRollbackOnly*. С тяхна помощ той може да маркира една транзакция за абортиране или да провери дали тя вече е маркирана.

Всеки внедрен сесиен бийн дефинира информация за това в по какъв начин иска от контейнера да се обработи транзакционния контекст по време на изпълнение на бизнес методите. Спецификацията дефинира следните възможности, наречени *транзакционни атрибути*

- *NOT_SUPPORTED* – контейнера е длъжен да извика бизнес метода без транзакционен контекст. Ако по време на извикване, клиента има транзакционен контекст, контейнерът е длъжен временно да го прекрати, да изпълни бизнес метода и след това да го възстанови
- *REQUIRED* – контейнерът е длъжен да извика бизнес метода във валиден транзакционен контекст. Ако по време на извикване клиент има транзакционен контекст, контейнерът изпълнява бизнес метода в него. Ако такъв контекст няма, то контейнерът автоматично стартира нов, изпълнява бизнес метода, след което се опитва да приключи транзакцията
- *SUPPORTS* – контейнерът извиква бизнес метода в транзакционния контекст на клиента или без такъв ако клиента не е асоцииран с транзакция

- *REQUIRES_NEW* – контейнерът е длъжен да извика бизнес метода в нов транзакционен контекст. Ако по време на извикване клиента има транзакционен контекст, контейнерът е длъжен временно да го прекрати и след приключване на работа да го възстанови. Контейнерът е длъжен да стартира нов транзакционен контекст да изпълнява бизнес метода е него след което да приключи транзакцията
- *MANDATORY* – контейнерът извиква бизнес метода в транзакционния контекст на клиента като верифицира че такъв има
- *NEVER* – контейнерът извиква бизнес метода без транзакционния контекст като верифицира че и клиента няма такъв

Ако бийна използва в бизнес метода си *setRollbackOnly,* то контейнерът е длъжен текущата транзакция да не приключи успешно. Ако транзакцията е стартирана непосредствено преди бизнес метода, контейнерът я абортира. Контейнерът е длъжен да извиква методите с анотации *PostConstruct, PreDestroy, PrePassivate* и *PostActivate* без транзакционния контекст.

2.6. Обработка на изключения

Спецификацията *EJB 3.0* въвежда понятието *приложни изключения* за да различи изключенията, които се предизвикват от приложението по време на работа, но са част от определен техен сценарий, от тези които се предизвикват поради системна грешка.

Приложните изключения се дефинират от *доставчика на бийна* чрез поставянето им в списъка от изключения на конкретен бизнес метод. Ако изключението има за базов клас *RuntimeException*, то *доставчика на бийна* трябва допълнително да постави анотацията *ApplicationException* на класа или да го опише в XML за внедряване.

Изключенията имащи за базов клас *java.rmi.RemoteException* или *RuntimeException* (но не са *приложни изключения*) се приемат са системни изключения.

Контейнеът е длъжен да установява и да се справя със системни изключения различавайки ги от *приложните изключения*. При възникване на системно

изключение контейнера трябва да реагира относно текущия транзакционния контекст, жизнения цикъл на инстанцията на бийна и изключението което ще се хвърли към клиента. При възникване на *приложно изключения,* контейнерът е длъжен да го хвърли непроменен към клиента, запазвайки инстанцията на бийна.

Извикване на метод на прихващащ клас с анотация *AroundInvoke* може да предизвика *приложно изключения*. Контейнерът трябва да различи този сценарий и да реагира все едно то се е хвърлило от бизнес метода на бийна. Клиента получава системните грешки винаги или като *javax.ejb.EJBException* или като *java.rmi.RemoteException*.

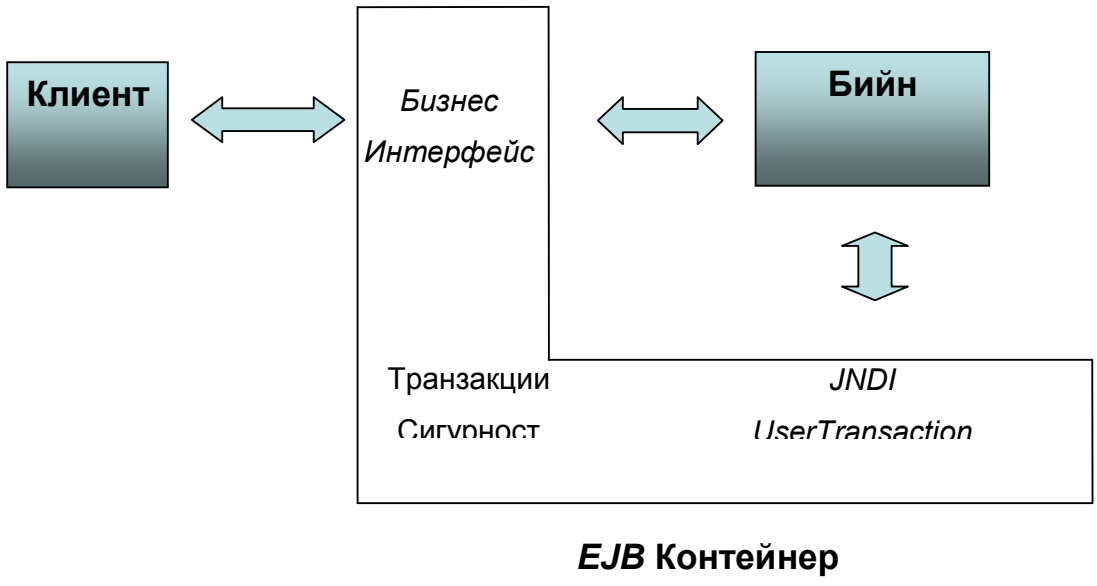
2.7. Среда на работа на сесиен бийн

Средата на работа на сесиен бийн е механизъм, който позволява конфигуриране на бизнес логиката на бийна, без да се налага промяна на неговия код. Реализираните в кода анотации и XML документите за внедряване са основните средства, които позволяват на *преводача на приложение* да реализира нуждите на бийна по конфигурирането му.

Контейнерът реализира средата на работа на сесиен бийн като *JNDI* контекст. *Доставчика на бийна* описва всички елементи които участват в средата на работа на бийна чрез анотации или в XML документа за внедряване. Контейнерът публикува тези елементи в средата на работи и те са за достъп до бийна чрез *EJBContext.lookup* метода или стандартните интерфейси на *JNDI*. Допълнително контейнерът *внедрява* (*по зависимост*) част от тези елементи в полета или чрез методи на бийна.
Средата на работа на сесиен бийн се споделя с всички прихващащи класове, асоциирани с него. Средата на работа на сесиен бийн не се споделя с друг бийн или *J2EE* компонент.

2.8. Изводи

Основни роли в контекста на описание и реализация на EJB контейнер заемат клиента, бийна като съвкупност от всички обвързани класове, анотации и XML документи и самият EJB контейнер.Фигура 4 схематично описва процеса на работа на участниците.

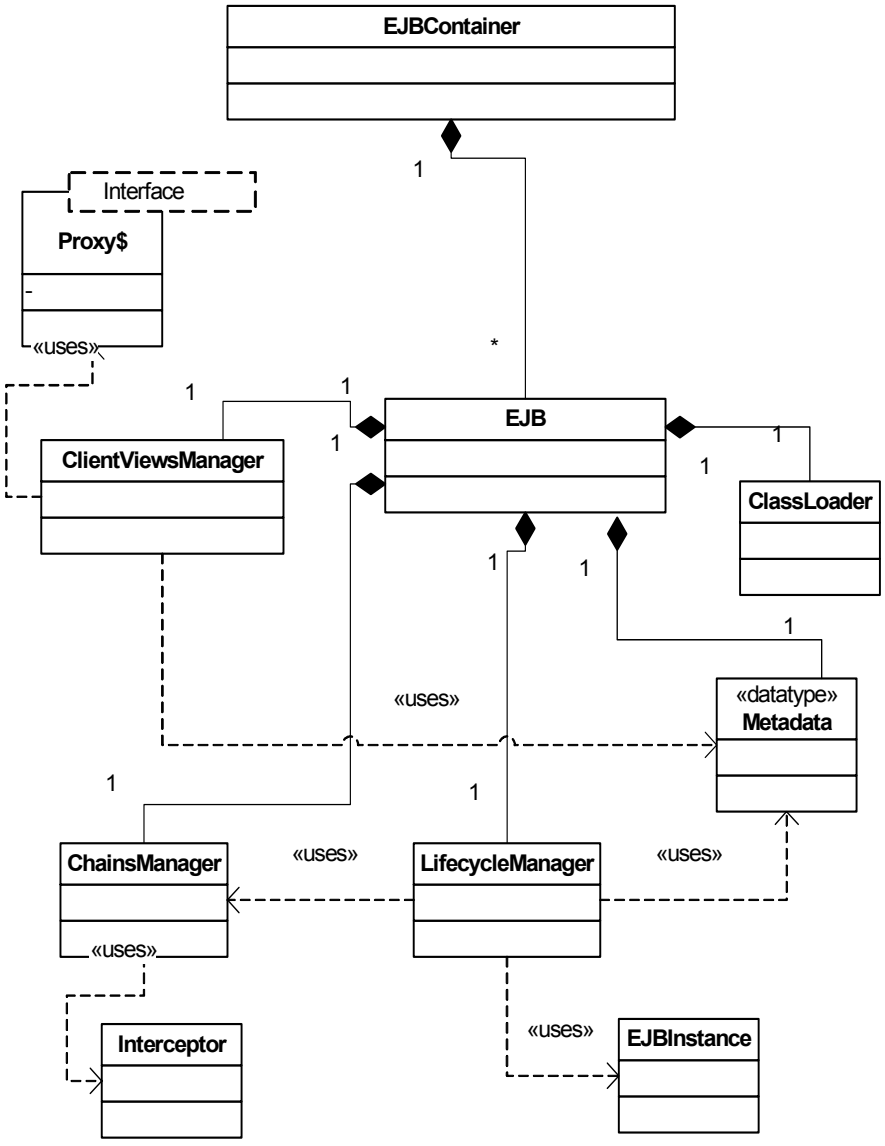


Фигура 4.
Схема на EJB контейнер, клиент и бийн.

Контейнерът предоставя на клиента достъп до бийна през неговите бизнес интерфейси. Реализацията на тези интерфейси трябва да е такава че да предизвика протичането на необходимите процеси по сигурността, транзакции, жизнения цикъл на бийна и извикването на конкретния бизнес метод.
От своя страна по време на работа, бийна има право да ползва предоставените му програмни интерфейси и среда за работа. Контейнера е длъжен да ги осигури спазвайки изискванията на спецификацията.

3. Дизайн на EJB контейнер за сесийни бийнове

Контейнера за сесийни бийнове е системата, в която съществуват сесийните бийнове, от момента на тяхното доставяне до момента на премахването им от системата. Разработеният от мен дизайн на *EJB* контейнер съдържа дефиниции на *Java* компоненти, които обединени реализират целите и нуждите на спецификацията. На Фигура 5 са представени основните компоненти дефинирани в него.



Фигура 5.

Основни компоненти в *EJB* контейнера

3.1. Интерфейса *EJBContainer*

Интерфейсът *EJBContainer* е базовият компонент контролиращ работата на всички останали. Той се инициализира по време на старт на системата и се премахва при спиране на системата. Основна негова функция е да пази информация за всички бийнове доставени и работещи в системата и асоциираните им компоненти. Той предоставя функционалност за откриване на компонентите на конкретен бийн по името на бийна или по описание запазено в *javax.naming.Reference*.

Основна цел на реализацията на интерфейса е по време на своята инициализация да стартира процеса на доставяне на нови бийнове. Този процес е специфичен за конкретната реализация и за това не е част от дизайна. Например стандарта *J2EE* специфицира изискване за съществуване на външно приложение, чрез което се осъществява доставянето на пакетирани бийнове или приложения до системата.

В една пълна реализация на *J2EE* платформа, интерфейсът *EJBContainer* би служил за доставяне до другите компоненти на *EJB* контейнера, специфичните за конкретната *J2EE* платформа интерфейси и компоненти. Това например може да са *TranasactionManager*, *JNDI*, RMI, или интерфейсите по сигурността.

3.2. Интерфейс *EJB*

Интерфейсът *EJB* служи да идентифицира конкретен бийн. Той обединява информацията от *доставчика на бийн* и от компонентите създадени за този бийн от контейнера.По време на доставянето на сесиен бийн, реализацията на *EJBContainer* създава *EJB* инстанция и я регистрира при себе си, правейки я достъпна за други компоненти. От своя страна, при създаването си *EJB* инстанцията се грижи да направи бийна достъпен за клиенти, както и да създаде необходимите ресурси и компоненти за коректната работа на инстанциите на бийна.

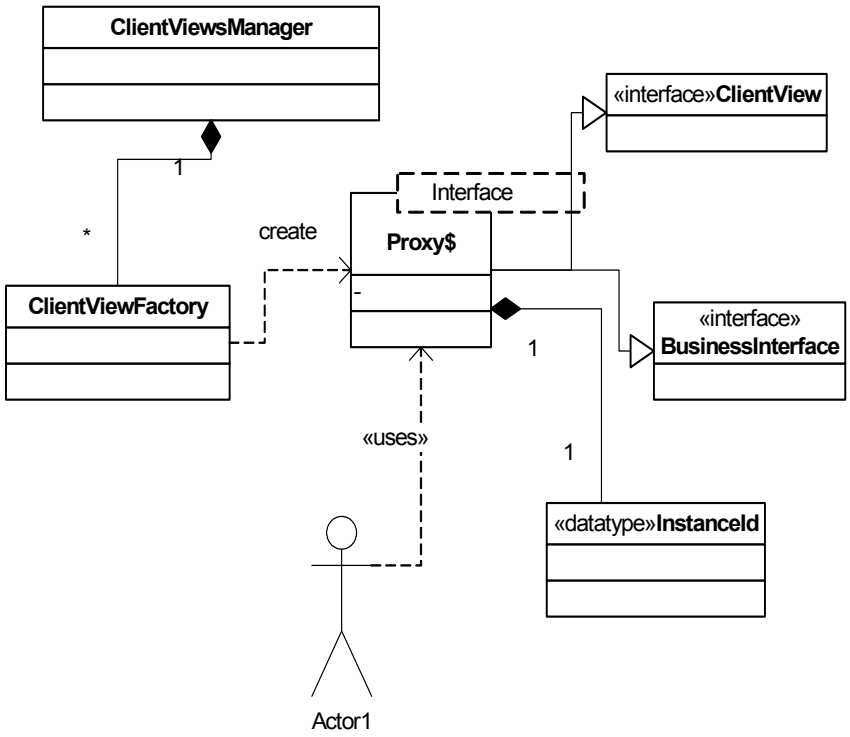
Основните компоненти, които *EJB* интерфейса реферира са *ClientViewsManager* – управляващ компонентите за достъп от клиенти, ChainsManager – управляващ методите осъществяващи бизнес извикванията и операциите по жизнения цикъл на бийна*, LifecycleManager* – управляващ методите от жизнения цикъл на бийна и *ClassLoader* – зареждащия *Java* класовете на бийна обект.

3.3. Интерфейс *EJBInstance*

Интерфейсът *EJBInstance* обгражда конкретна инстанция на бийн заедно с инстанциите на прихващащите класове, асоциирани с бийна. Конкретната негова реализация допълнително би могла да обхваща всички системни ресурси асоциирани с инстанцията и имащи същия цикъл на живот като нея. Всяка инстанция на *EJBInstance* при създаването си се асоциира с *InstanceID,* което е неин идентификатор. Този идентификатор може да не е уникален спрямо стоп и старт на системата тъй като сесийните бийнове не преживяват тези операции. Допълнително *EJBInstance* е асоцииран със конкретен обект реализиращ стандартния програмен интерфейс *javax.ejb.SessionContext* предоставен на бийна.

3.4. Управление на достъпа от клиенти

Основният компонент отговарящ за обектите предоставяни на клиенти на бийна е интерфейса *ClientViewsManager.* За всеки интерфейс на бийна, дефиниран от *доставчика на бийн*, *ClientViewsManager* управлява съответен обект от интерфейса *ClientViewFactory* – както е представено на Фигура 6. всички *ClientViewFactory* обекти се инициализират при доставянето на бийна . Те са достъпни до отделните компоненти на контейнера през *ClientViewsManager* интерфейса.



Фигура 6.

Модел на компонентите за управление на

достъпа до клиенти

Всяка инстанция на интерфейс на бийн, генерирана от контейнера, реализира допълнително и интерфейса *ClientView*.Той служи за общ модел, по който контейнера и по-конкретно обектите *ClientViewFactory* управляват всички инстанции при клиента. Всяка *ClientView* инстанция е асоциирана с *InstanceID.* С негова помощ се осъществява връзката между конкретни *ClientView* и *EJBInstance*.

Обектите *ClientViewFactory* се асоциират с конкретен интерфейс на бийн. По този начин, знаейки само името на интерфейса и *InstanceID* на бийн инстанцията, *EJB* контейнера генерира *ClientView* обект.

Скривайки конкретната реализация на *ClientViewFactory* от компонентите на контейнера, позволява да се оптимизира във времето начина на доставяне и работа на *ClientView* обектите, предоставяни на клиента. В тази насока например е възможно интегриране на RMI реализацията с тези *ClientViewFactory* отговарящи за отдалечените интерфейси на бийна.

3.4.1.Механизмът *java.lang.reflect.Proxy*

По своето естество, реализацията на *ClientView* се състои от генериране на код реализиращ методите на конкретни *Java* интерфейси. В стандартната *Java* има вграден подобен модел който предоставя на програмата възможност динамично да създаде реализация на списък от интерфейси. Реализацията има за базов интерфейс *java.lang.reflect.Proxy*. Тя е генерирана по такъв начин,че извикването на метод от нея рефлектира в извикване на метода

Object invoke(Object proxy, Method m , Object[] args) throws Throwable

от реализацията на интерфейса *InvocationHandler*, предоставена от клиента при създаването на *Proxy* инстанцията. Конкретната *InvocationHandler* инстанция има възможността да прецени как да проведе изпълнението на метода, имайки референция към конкретния *Proxy* обект, неговия метод и стойността на аргументите на метода.

Този *Proxy* модел напълно задоволява нуждите на реализацията на *EJB* контейнер. Дизайна на контейнера асоциира всяко *ClientViewFactory* с определен вид *Proxy* обекти, и *InvocationHandler* инстанции. Интерфейсът *EJBProxyInvocationHandler* е базов интерфейс за всички *InvocationHandler* реализирани от контейнера.

3.4.2.Работа на *EJBProxyInvocationHandler* инстанциите

Конкретните *EJBProxyInvocationHandler* инстанции служат да преведат извикването на метод от клиент към контейнера. Тяхната реализация трябва да е такава, че да различи метода в зависимост от това дали идва от вътрешния интерфейс *ClientView,* от *java.lang.Object* или от интерфейса на бийна. Методите от *ClientView* и от *java.lang.Object* са реализирани директно в конкретните *EJBProxyInvocationHandler.* За обработка на методите от бизнес, базов или компонентен интерфейс се използва обекта *ChainsManager* асоцииран с бийна, който осъществява конкретното управление по извикването. Инстанцията на *EJBProxyInvocationHandler* предава върнатия резултат на клиента.

Допълнително, някои конкретни *EJBProxyInvocationHandler* може да различават някои други специфични интерфейси, които *Proxy* инстанцията реализира.

3.4.3.Видове *ClientViewFactory* и *EJBProxyInvocationHandler*

Използвайки *Proxy* механизма, обектите от *ClientViewFactory* лесно създават реализации на набор от интерфейси, които се предоставят на клиент. За целта те използват последователността от създаване на EJBProxyInvocationHandler инстанция и асоцииране и с конкретното *InstanceID* и *Proxy*. Така двойката *ClientViewFactory*, *EJBProxyInvocationHandler* определя поведението на обектите предоставени на клиента.

Не винаги модела на работа на контейнера се реализира с една и съща *ClientViewFactory*, *EJBProxyInvocationHandler* двойка за конкретен бийн и негов интерфейс. Контейнерът дефинира различни реализации както с цел оптимизация на ресурси, така и породени от изискванията на спецификацията. Комбинирайки различни *ClientViewFactory* и *EJBProxyInvocationHandler* контейнерът успява да покрие изискванията към него и да подобри работата си.

3.4.3.1. *ClientViewFactory* за отдалечени реализации

За да реализира отдалечен достъп към *Proxy* инстанциите, контейнерът разширява базовата функционалност на *ClientViewFactory, като използва* RMI-IIOP механизма. Стандартно генерираната инстанция се излага за достъп при създаването си чрез метода *ClientViewFactory.newInstance(InstanceID)* и се премахва за достъп при унищожаването си чрез метода *ClientViewFactory.destroyInstance(ClientView)*. Този вид *ClientViewFactory* може да се приложи само за тези интерфейси, които спазват изискванията за RMI. Това са отдалечените базов и компонентен интерфейс на бийна.

3.4.3.2. Запазващ *ClientViewFactory*

За някои интерфейси на бийна, контейнерът предоставя и управлява само една инстанция за клиент. За да реализира това контейнерът използва метода на съдържане. Конкретното *ClientViewFactory* съдържа второ *ClientViewFactory*. Всички методи на първото *ClientViewFactory* се предават към методите на второто като резултата от изпълнението на методите *ClientViewFactory.newInstance* и *ClientViewFactory.newLocalInstance* се запазва и връща при повторно им извикване. По този начин това *ClientViewFactory* създава само една *Proxy* инстанция от съответния вид, независимо от конкретната реализация на второто *ClientViewFactory*. Този вид *ClientViewFactory* може да се приложи за реализации на базовите интерфейси на бийн, както и за отдалечения компонентен интерфейс на несъстоятелен бийн.

3.4.3.3. Реализация на състоятелен бийн управляващ сам транзакции

Състоятелни бийнове, които управляват сами транзакции може да приключат бизнес извикването без да са приключили транзакцията. Контейнерът, и по-конкретно някой системен клас за прихващане, управлява тази ситуация през интерфейса *ClientTransactionAssociation*. Конкретно този интерфейс се реализира от специфичен *EJBProxyInvocationHandler* клас, който запазва в свое поле транзакцията и асоциацията й с инстанцията на бийна. Комбинирайки специфичната *EJBProxyInvocationHandler* инстанция и нормалните (според вида интерфейс) *ClientViewFactory* контейнерът реализира исканата функционалност.

3.4.3.4. Реализация на отдалечени бизнес интерфейси

Бийнът декларира бизнес интерфейси за отдалечен достъп чрез анотацията *Remote*. По този начин отдалечените бизнес интерфейси не спазват изискванията на RMI и правят директното прилагане на RMI невъзможно. За да разреши този проблем, контейнерът дефинира разширен модел на *ClientViewFactory* и *EJBProxyInvocationHandler*, който позволява отдалеченото изпълнение.

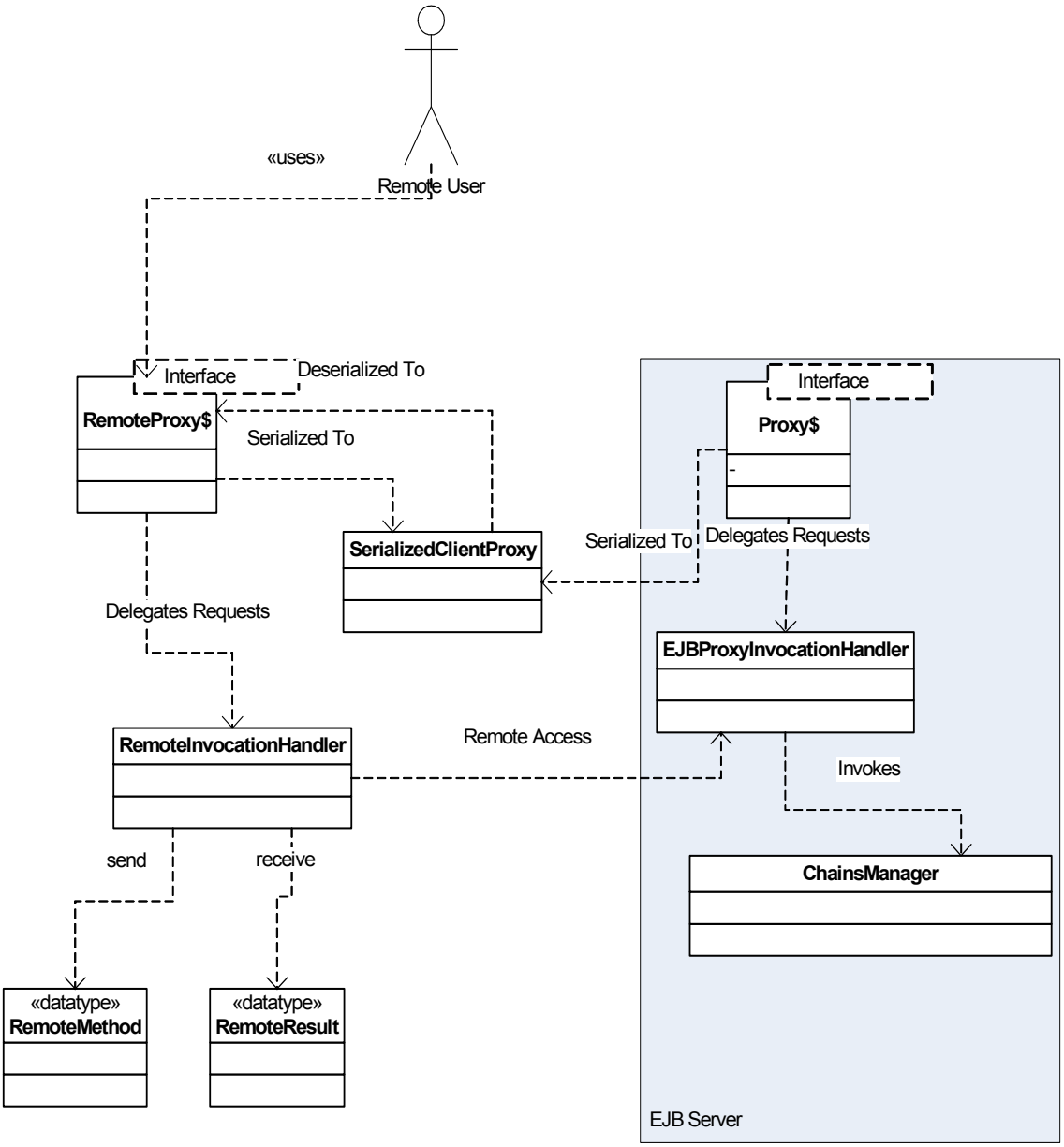
Тъй като RMI работи само със *сериализируеми* и отдалечени обекти, за да е RMI съвместим, контейнера реализира *Proxy* обектите като *сериализируеми*. За целта той използва функционалността на протокола за *сериализиране* предоставена от стандартната *Java.*

Всички *Proxy* обекти за отдалечен бизнес интерфейс реализират интерфейса *ReplaceableProxy,* който наследява *java.io.Serializable* и декларира единствено метода

Object ReplaceableProxy.writeReplace()

Този метод, както се определя от протокола за *сериализация*, служи за заместването на *Proxy* обекта от друг по време на отдалечено му транспортиране. Обектът, с който се замества е инстанция на *SerializedClientProxy*, която точно го описва и при своето де-сериализиране на отдалечената система се реконструира.

Реализацията на *writeReplace* метода на *Proxy* обектът е в специфичен *EJBProxyInvocationHandler.* Този *EJBProxyInvocationHandler* е специален още в отношението, че се използва за отдалечен достъп през интерфейса *RemoteInvocationHandler*, както е показано на Фигура 7.



Фигура 7.

Схема на отдалечено викане през бизнес интерфейс.

Единствения метод, който *RemoteInvocationHandler* декларира е

RemoteResult invoke(RemoteMethod m) throws RemoteException

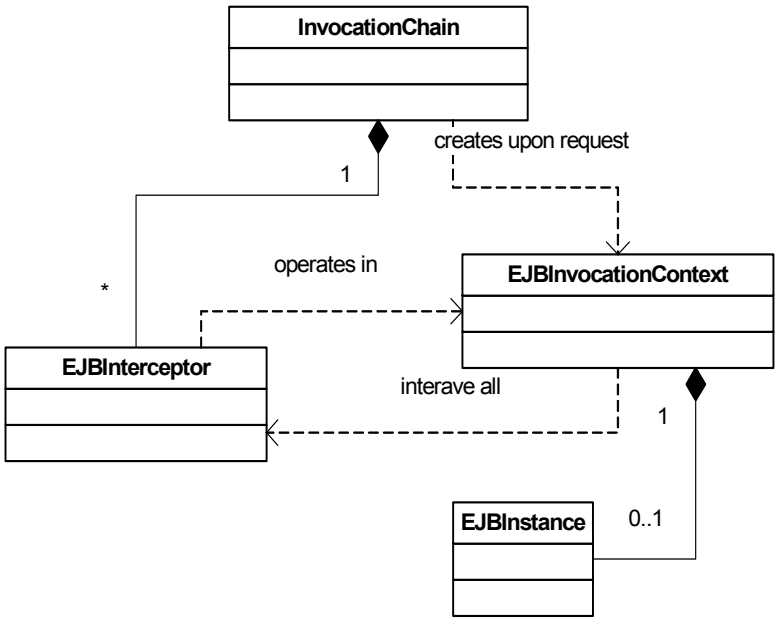
При реконструкцията на *Proxy* обекта при клиента, *SerializedClientProxy* реализацията асоциира с него инстанция на *RemoteInvocationHandlerAdapter* като негов *InvocationHandler*. Тази инстанция съдържа референция към отдалечения обект *RemoteInvocationHandler* и делегира всяко извикване

InvocationHandler.invoke към него, превеждайки параметрите до инстанция на *RemoteMethod*. Инстанцията на *RemoteMethod* съдържа стойността на аргументите на извикания метод и самия метод като инстанция на *java.lang.reflect.Method* като по този начин тя е *сериализируема*. След изпълнение на отдалеченото извикване, *EJBProxyInvocationHandler* обекта връща резултата към клиента, обвит в инстанция на *RemoteResult*. Резултатът е изключение или конкретна стойност и за това *RemoteResult* съдържа референция към *java.rmi.RemoteException* и *java.lang.Object* като в конкретен случай едната референция винаги е *null*. При получаването на резултата *RemoteInvocationHandlerAdapter* извлича обекта или изключението и го предава на клиента, като в случай на изключение то се хвърля към него, за да пресъздаде прозрачност на отдалеченото извикване.

3.5. Изпълнения в контейнера

Изискванията за управление в контейнера се свеждат до набор от правила и код за изпълнение. За всеки бийн и негов конкретен бизнес метод този код и правила са известни предварително на контейнера. Това позволява описанието на действията които извършва контейнера да се реализира чрез модела за прихващане.

Следвайки принципите за прихващане, на всеки метод който клиента извиква и на всяка операция по жизнения цикъл на бийн, контейнера съпоставя верига от прихващащи класове. За да не се смесват дефинираните от доставчика на бийн и дефинираните от контейнера прихващащи класове, последните са наречени системни прихващащи класове . Фигура 8 изобразява компонентите, които са асоциирани със системните прихващащи класове и ги представляват.



Фигура 8.

Модел на системни прихващащи класове.

3.5.1.Интерфейс *EJBInterceptor*

Интерфейсът *EJBInterceptor* представлява инстанция на един системен прихващащ клас. Негов основен метод е

Object proceed(EJBInvocationContex) throws ApplicationException, EJBException

Извикването на този метод предизвиква изпълнението на кода реализиращ специфична дейност от задълженията на контейнера. Например това може да е определянето на транзакционния контекст, проверка по сигурността, активиране на сесия на бийн или друга, специфична за конкретната *J2EE* платформа дейност. Реализациите на този метод трябва да извикат метода *EJBInvocationContex.proceed()* за да предадат управлението отново на контейнера. Стойността която връща методът е стойността върната от *EJBInvocationContex.proceed()* или друга специфична за конкретния системен прихващащ клас.

Изключенията, дефинирани от метода са само две и описват двете групи, с които работи *EJB* спецификацията. Всички изключения определени като *приложни* се обграждат от инстанция на *ApplicationException* и се прехвърлят по

този начин до клиента. Този, който пръв инициира извикване на верига от прихващащи класове е отговорен за превеждането на *ApplicationException* до съответното *приложно изключение*. Второто декларирано изключение е *EJBException* и то определя системната грешка възникнала по време на работа.

В повечето реализации на системни прихващащи класове се очаква те да обработват грешките възникнали само в техния код, но не и тези от *EJBInvocationContex.proceed()* метода.

3.5.2.Вериги от прихващащи класове

Основна сила на използването на прихващащи класове е, че те може да се нареждат и групират в различни вериги. Обикновено една верига отговаря за пълната група от операции по изпълнението на конкретна заявка на клиента.

Всяка инстанция на интерфейса *EJB* се асоциира при създаването си с инстанция на *ClientViewsManage,* който управлява инстанциите на системните прихващащи класове и образуваните от тях вериги. Конкретните вериги и инстанции от прихващащи класове, се определят и създават спрямо това кои и какви са методите които контейнера предоставя на клиента и какъв е жизнения цикъл на бийна. Създаването на веригите й се осъществява в момента на доставяне на бийна в системата. Инстанциите от прихващащите класове се създават по време на конструиране на конкретните вериги. Всяка такава се регистрира в *ClientViewsManage* при което получава уникален номер с който се цитира във веригата. По този начин се контролира броя на тези обекти и се позволява те да се споделят от различните вериги.

3.5.2.1. Интерфейс *EJBInvocationContext*

Реализацията на *EJBInvocationContex* управлява извикването на всеки прихващащ клас от веригата, в реда в който те са описани в нея, както и за извикването на бизнес метода от класа на бийна в края. Тя поддържа информация за текущия прихващащ клас а конкретната негова инстанция извлича от *ClientViewsManage* чрез уникалния му номер описан във веригата.

В конкретен момент, когато контейнерът изпълнява заявка от клиент или операция от жизнения цикъл на бийна, инстанция на *EJBInvocationContex* се асоциира с конкретна верига. Ако два клиента на бийн изпълняват една и съща операция едновременно то контейнера асоциира различни инстанции на *EJBInvocationContex* с една и съща верига. По този начин различните клиенти споделят не само обектите от прихващащите класове, но и веригите, които те образуват.

Поради това, че веригите се споделят, информацията за конкретния клиент и текущата информация по изпълнението от контейнера се асоциира с обекта *EJBInvocationContex*. Например това са *ClientView* обекта, неговия метод, извикан от клиента, транзакционния контекст на клиента, EJBInstance обекта обвиващ ресурсите на инстанцията на бийна. Всяка инстанция на прихващащ клас има достъп до *EJBInvocationContex* и по време на своята работа определя специфична част от информацията съхранена там. Например прихващащия клас който управлява транзакциите запазва транзакционния контекст на клиента и той е достъпен до всички други следващи обекти от веригата.

3.5.2.2. Интерфейси *InvocationChain, InvocationChains* и *CallbackInvocationChains*

Интерфейса *InvocationChain* представлява конкретна верига от прихващащи класове. Той съхранява информация за това кои са класовете във веригата под формата на списък от идентификатори, както и за дължината на веригата. По време на работа *EJBInvocationContex* използва тази информация, за да обходи веригата в правилния ред.

Всички вериги за изпълнение дефинирани относно методи от един интерфейс на бийн са асоциирани с обект от интерфейса *InvocationChains*. По този начин се разграничават начините, по които контейнера управлява достъпа до бизнес методите на бийна. Реализацията може да асоциира различни прихващащи класове за един и същи бизнес метод, в зависимост от изискванията на *доставчика на бийн*. Конкретен *InvocationChains* предоставя информация за кой интерфейс се отнася и достъп до веригите създадени от

контейнера за този интерфейс по инстанция на *java.lang.reflect.Method*. Този начин на работа се съвместява добре с *Proxy* и *InvocationHandler* механизмите тъй като се ползва общ програмен интерфейс.

Всички вериги относно жизнения цикъл на бийна са асоциирани и достъпни през интерфейса *CallbackInvocationChains*. За разлика от стандартните методи на бийна в този случай няма *Method, с която* да се асоциира конкретна верига. За целта, дизайна на контейнера дефинира изброимия тип *Callback*, който описва конкретни операции по състояния на бийна. Всяка верига в *CallbackInvocationChains* е достъпна по конкретна *Callback* стойност.

Създаването на *InvocationChains* и *CallbackInvocationChains* се осъществява по време на създаването на конкретните вериги за изпълнение. Те се асоциират с бийна през неговия *ClientViewsManage*.

3.5.2.3. Управление по извикване на вериги

Управлението по извикване на вериги се контролира от *ClientViewsManage*. Всеки вътрешен за контейнера компонент инициира извикване по управление на бийна чрез метод от този интерфейс, предоставяйки конкретната верига, клиента на извикването като *ClientView*, метода от интерфейса на бийна и стойностите на аргументите. За извикване по жизнения цикъл на бийна *ClientViewsManage* предоставя други подобни методи, които изискват или конкретната *EJBInstance* или съответното и *InstanceID.* На всяко извикване на верига, *ClientViewsManage* съпоставя инстанция на *EJBInvocationContex*, като създава нова конкретна такава. След приключване на извикването инстанцията на *EJBInvocationContex* се освобождава. По този начин всички текущи данни асоциирани с извикването на клиента се освобождават.

3.5.3.Съвместимост със *приложни* прихващащи класове

Дефинирайки системни прихващащи класове, контейнера гарантира че те са съвместими с *приложните* такива*.* Спецификацията дефинира конкретното време спрямо жизнения цикъл на бийна или бизнес извикването, в което трябва

да се изпълнят те. Контейнера дефинира своята работа само чрез системни прихващащи класове и към тях добавя специални такива предназначени да осъществят извикването на *приложните*. Реално тези специални класове са винаги в края на веригата от извикване. Те работят с *EJBInstance* от където получават конкретната инстанция на приложен прихващащ клас чийто метод извикват. Преминаването от програмния интерфейс на контейнера към програмния интерфейс на *EJB* спецификацията се осъществява от обекта *EJBInvocationContex,* който предоставя инстанция на *InvocationContext* асоциирана с него при създаването си. За конкретна операция, всички системни прихващащи класове споделят една инстанция на *EJBInvocationContex*. Така всички приложни прихващащи класове споделят една инстанция на *InvocationContext*, както се изисква от спецификацията.

3.6. Жизнен цикъл на сесийни бийнове

Контейнерът управлява жизнения цикъл на сесийните бийнове в съвкупност от тяхната инстанция, инстанциите на прихващащите класове и ресурсите асоциирани с тях. Тази съвкупност се представя от компонента *EJBInstance*. Компонента, който извършва това управление е представен от интерфейса *LifecycleManager.* Всеки вид сесиен бийн се асоциира с инстанция на този интерфейс. Той предоставя на останалите компоненти на бийна възможност да създадат нова инстанция на бийн, да преведат съществуваща инстанция от едно състояние в друго или да унищожат инстанцията.

От гледна точка на *LifecycleManager* компонента, *EJBInstance* е в активно състояние, тоест в момента клиент извършва бизнес управление чрез нея, или е в неактивно състояние като заема едно от състоянията дефинирани от спецификацията. Докато е в активно състояние *EJBInstance* е асоциирана с *EJBInvocationContex* инстанцията. В неактивно състояние *LifecycleManager* се грижи за асоциирането на инстанцията с определена структура, така че тя да не бъде унищожена и изгубена преди клиента да е поискал това.

Реализациите на методите на *LifecycleManager* се възползват от създадените за целта вериги от системни прихващащи класове. По този начин конкретен

метод може да се реализира чрез извикването на една или повече от тези вериги, използвайки интерфейса *ChainsManager* и асоциираните с него *CallbackInvocationChains*.

Общите методи, които интерфейса *LifecycleManager* дефинира са

- *void destroy(EJBInstance instance)* – специфицираната инстанция се премахва по нормален начин, като се извикват всички необходими приложни прихващащи класове
- *void discard(EJBInstance instance)* – специфицираната инстанция се премахва незабавно като се освобождават всички ресурси. Не се извикват никакви методи от приложението

Тъй като жизнения цикъл на състоятелните и несъстоятелните бийнове не споделя обща схема то за всеки от тях, в контейнера е разработена специфична реализация на *LifecycleManager, която* специфицира и реализира за допълнителни състояния.

3.6.1.Управление при несъстоятелни бийнове

Несъстоятелните бийнове в неактивно състояние споделят общ статус и са неразличими от контейнера. За целта контейнерът дефинира *InstanceID* за несъстоятелни бийнове като идентифициращо съвкупността от инстанции а не конкретна една от тях. Реализацията е *StatelessInstanceID*, която съдържа информация само за името на бийна.

Конкретната реализация на *LifecycleManager* за несъстоятелни бийнове е *StatelessLifecycleManager* дефиниращ допълнително методите

- *EJBInstance getReady()* – връща съществуваща инстанция от *пул* с такива които се намират в готово състояние за изпълнение на бизнес методи. Ако такава инстанция в момента няма, то *StatelessLifecycleManager* създава нова инстанция на *EJBInstance*, асоциирана с инстанцията на бийна, прихващащите класове и всички необходими системни ресурси и привежда бийн инстанцията в готово състояние за изпълнение на бизнес методи

- *void releaseReady(EJBInstance instance)* **–** контейнера съобщава на *StatelessLifecycleManager,* че е приключил работа със специфицираната инстанция и му позволява да я върне в *пула* от такива

3.6.2.Управление при състоятелни бийнове

Инстанциите на състоятелните бийнове стриктно се различават една от друга. За да може да управлява процеса на откриването им, контейнера дефинира идентификатор на сесия за всяка инстанция. Реализацията на *InstanceID* за състоятелни бийнове е *StatefulInstanceID.* Тя разширява *StatelessInstanceID* със стойността на идентификатора на сесия за конкретната инстанция на бийн.

Конкретната реализация на *LifecycleManager* за състоятелно бийнове е *StatefulLifecycleManager* дефиниращ допълнително методите

- *EJBInstance create()* **–** създава нова инстанция на *EJBInstance*, асоциирана с инстанцията на бийна, прихващащите класове и всички необходими системни ресурси и превежда бийн инстанцията в готово състояние за изпълнение на бизнес методи
- *void activate(InstanceID id)* **–** *активира* бийна със специфицирания *InstanceID*
- *void passivate(EJBInstance instance)* – *пасивира* бийна със специфицирания *InstanceID*
- *EJBInstance getActiveInTransaction(InstanceID id)* – връща *EJBInstance*, имаща специфицираното *InstanceID* и асоциирана с текущата транзакция
- *void putActiveInTransaction(Transaction t, EJBInstance instance)* – асоциира специфицираната *EJBInstance* към специфицираната транзакция

3.7. Осъществяване на достъп от клиенти

Освен генерирането на реализация на интерфейса, по който клиента работи с бийна, контейнерът е длъжен да предостави механизъм, по който клиента да получи референция към тази реализация. За целта контейнера разчита на

съществуваща JNDI реализация, която предоставя достъп както на клиенти в същата *Java* виртуална машина, така и на отдалечени.

Механизмът на достъп, който контейнера предоставя се състои от дефиниране на протокол по който се конструират *javax.naming.Reference* обекти. Контейнера или който и да е друг *J2EE* компонент, регистрира тези обекти в JNDI под специфично име, което предварително се знае от клиентите на бийна. Когато клиент потърси в JNDI обект под това име, реализацията на JNDI открива *Reference* инстанцията и опитва да създаде обект от нея използвайки специфицираното в нея *javax.naming.spi.ObjectFactory*.

Реализацията на този *ObjectFactory* обект е част от *EJB* контейнера и се състои от класа *EJBObjectFactory*. При извикване от JNDI, той използва информацията съхранена в предоставения му *javax.naming.Reference* обект за да открие чрез реализацията на *EJBContainer* интерфейса кой е съответния *EJB* обект. Когато има достъп до интерфейса *EJB*, *ObjectFactory* обекта използва съответните му *ChainsManager* и *ClientViewsManager, за да* създаде необходимите *EJBInstance* и *ClientView* обекти.

3.7.1.Протокол за информацията съхранена в *javax.naming.Reference*

Информацията се съхранява в *Reference* обектите под формата на *StringRefAddr* обекти. Един *StringRefAddr* представлява двойка, съставена от ключ от тип *String* и съответна стойност от тип *Object*. Контейнера специфицира набора от ключове, които разпознава и съответните им стойности като:

- **local –** името на локален бизнес или компонентен интерфейс
- **remote –** името на отдалечен бизнес или компонентен интерфейс
- **jndi-name –** уникалното име в JNDI под което е публикуван бийна
- **ejb-link –** името на бийна
- **ejb-jar –** името на архива в който е бийна
- **local-home –** името на базовия локален интерфейс на бийна
- **home –** името на базовия отдалечен интерфейс на бийна

- **interfaceType –** или **local** или **remote** в зависимост дали клиента иска нормален или отдалечен достъп към бийна или **undetermined** ако не е ясно
- **undetermined –** името на интерфейс на бийна към който клиента иска да получи достъп

Тъй като някой от ключовете специфицират взаимно изключващи се интерфейси на бийна, една инстанция на *Reference* специфицира само набор от тях с оглед на това каква информация е достъпна по време на създаването на *Reference* обекта. Възможните ситуации са следните:

- Контейнера описва клиент, който ще ползва бийна през локалните интерфейси съвместими с *EJB 2.1* спецификацията добавяйки *StringRefAddr* с ключове **interfaceType, local-home** и **local**
- Контейнера описва клиент, който ще ползва бийна през отдалечените интерфейси съвместими с *EJB 2.1* спецификацията, добавяйки *StringRefAddr* с ключове **interfaceType, home** и **remote**
- Контейнера описва клиент, който ще ползва бийна през конкретен локален бизнес интерфейс, добавяйки *StringRefAddr* с ключове **interfaceType** и **local**
- Контейнера описва клиент, който ще ползва бийна през конкретен отдалечен бизнес интерфейс, добавяйки *StringRefAddr* с ключове **interfaceType** и **remote**
- Контейнера описва клиент, който специфицира само *Java* анотация към свое поле или метод, добавяйки *StringRefAddr* с ключове **interfaceType** и **undetermined**

Във всички случаи специфицирането на **ejb-link** е незадължително, но препоръчително тъй като ускорява процеса на откриване и създаване на бийна, ако в системата има доставени много такива. Специфицирането на **jndi-name** е както незадължително така и зависимо от реализацията на конкретната *J2EE* платформа тъй като *EJB* спецификацията не определя правилата за глобално публикуване в *JNDI.*

3.7.2.Работа на *EJBObjectFactory*

Инстанциите на *EJBObjectFactory* осъществяват връзката между изпълнените на клиента и изпълнението на системата, в която работи *EJB* контейнера. Тъй като клиентите може да са отдалечени то *EJBObjectFactory* трябва да разпознава дали работи при клиента или не. За целта се ползва системна променлива дефинирана при стартирането на системата.

Ако *EJBObjectFactory* определи, че работи при реализацията на *EJB* контейнер то той осъществява достъп директно до контейнера през неговите класове. За случая, когато EJBObjectFactory определи, че е при клиента, той използва специален отдалечен обект имащ интерфейса *RemoteObjectFactory,* който извършва същата JNDI операция както клиента, но при *EJB* контейнера и връща по RMI резултата. Ако *EJBObjectFactory* реализацията определи, че клиента изисква референция към базов интерфейс то за създаването й се използва само съответното *ClientViewFactory, тъй* като базовите интерфейси не са свързани с конкретна инстанция на бийн.

3.7.3.Създаване на *ClientView* за несъстоятелни бийнове

При създаване на обект реализиращ бизнес интерфейс на несъстоятелен бийн, контейнера не създава съответна *EJBInstance*. Поради тази причина реализацията на *EJBObjectFactory* ползва само съответното *ClientViewFactory* и специфицира общото за всички инстанции на бийна *StatelessInstanceID*.

3.7.4.Създаване на *ClientView* за състоятелни бийнове

Създаването на обект реализиращ бизнес интерфейс на състоятелен бийн изисква преди използването на съответното *ClientViewFactory,* да се създаде конкретна *EJBInstance*. Контейнера реализира това с помощта на специална верига от прихващащи класове, асоциирана с метода *null* в съответния на интерфейса обект *InvocationChains*. Тази верига се създава по подобие на веригите за *create<METHOD>* методите от базовите интерфейси на бийна с това

изключение, че тя не съдържа извикване на съответен *ejbCreate<METHOD>* метод на бийна.

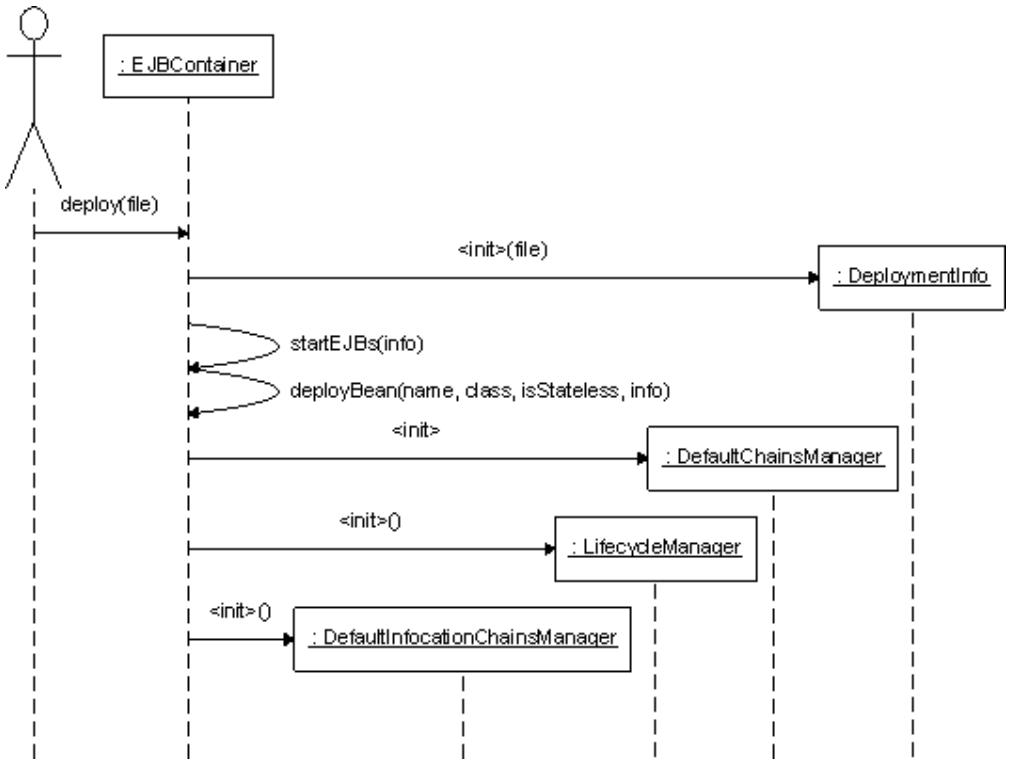
4. Реализация на EJB контейнер за сесийни бийнове

Реализацията на контейнер се състои основно от методи осъществяващи доставянето на бийн, от прихващащи класове реализиращи задачите, поставени към контейнера и от реализации на интерфейсите на компонентите описани в дизайна на контейнера. В приложението към дипломната работа можете да намерите UML class диаграми на тези интерфейси.

4.1. Процес на доставяне на бийн

Процеса на доставяне на бийн се състои от доставяне на пакетирани *jar* архиви до системата, която ги разпознава и активира съдържащите се в пакета бийнове по начин, такъв че те са да достъпни до клиенти на системата.

Входна точка за доставяне на бийна за контейнера е методът *EJBContainer.deploy(File archive)* , както е изобразено на Фигура 9.



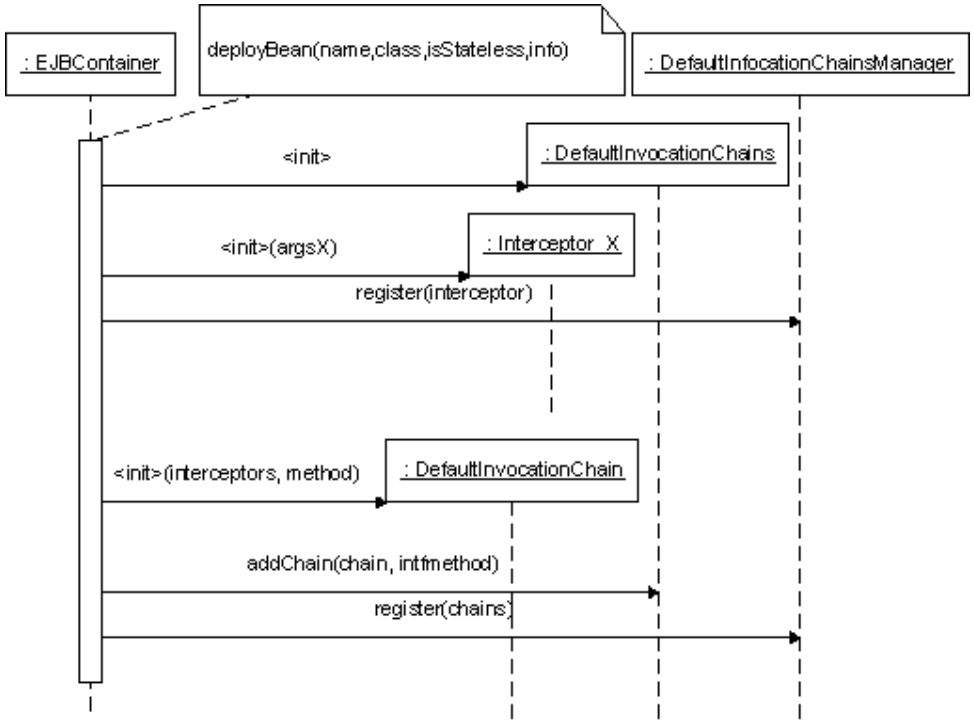
Фигура 9.

Доставяне на бийнове до контейнера.

След неговото извикване от вътрешната система за доставяне на бийнове, контейнера обработва архива като търси анотации за сесийни бийнове в класовете в архива, както и XML документи за внедряване. За всеки открит бийн, контейнера създава инстанция на *DefaultEJB* – реализацията на *EJB* интерфейса и асоциираните с нея обекти от класове *DefaultChainsManager*, *StatelessLifecycleManager* или *StatefulLifecycleManager*, *DefaultClientViewsManager* и *EJBClassLoader*. Всеки един от тези обекти е реализация на интерфейс асоцииран с интерфейса *EJB*.

Ако бийна е несъстоятелен то доставянето му включва инициализация на обект *StatelessLifecycleManager.* Тя се състои от определяне на параметрите и инициализацията на *пул* от инстанции от интерфейсът *EJBInstance*. Ако бийна е състоятелен то доставянето му включва инициализация на обект от вида *StatefulLifecycleManager*. Този обект създава обектите свързващи го със системата за вторично съхранение на бийнове както и списък за съхранение на *непасивираните* такива.

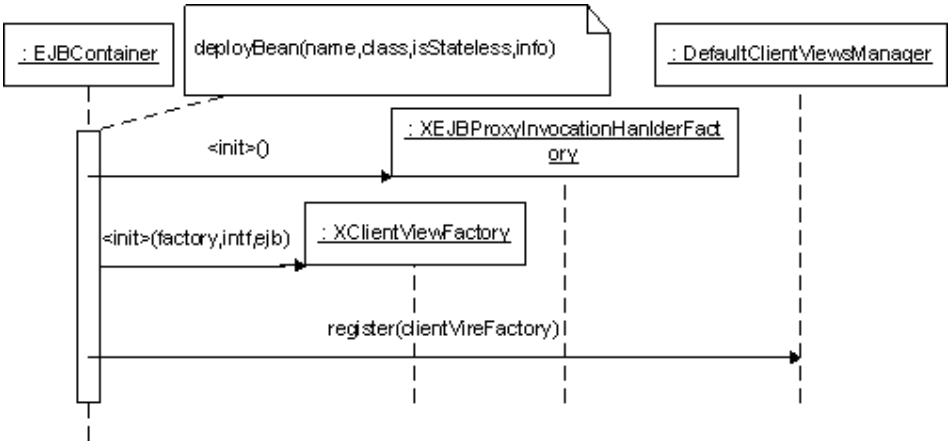
Инициализацията на *DefaultChainsManager* включва създаването на вътрешни структури за съхранение на системните прихващащи класове и техните вериги. Тези структури се запълват в по-късен етап от работата по доставяне в съответсвие с данните открити за класовете на съответния бийн, както е изобразено на Фигура 10.



Фигура 10.

Създавана на веригите от прихващащи класове.

Инициализацията на *DefaultClientViewsManager* се състои от създаването на вътрешни структури съдържащи всички конкретни *ClientViewsFactory*. За да създаде тези обекти контейнера обработва информацията за това кои и какви са всички интерфейси, които *доставчика на бийн* е разработил и асоциирал с бийна. За всеки такъв се създава една от конкретните инстанции на *ClientViewsFactory* реализирани в контейнера и на нея се съпоставя инстанция на *EJBProxyInvocationHandlerFactory*, както е изобразено на фигура 11.



Фигуре 11.

Създаване на *ClientViewsFactory* инстанции за бийн.

Всяко *EJBProxyInvocationHandlerFactory* служи за създаване на инстанция на *EJBProxyInvocationHandler*. По този начин се осъществява групирането по двойки на реализации на *ClientViewsFactory* и *EJBProxyInvocationHandler* заложен от дизайна на контейнера.

Обекта *EJBClassLoader* разширява стандартния клас *URLClassLoader* като специфицира името на бийна за който се отнася. Той се инициализира със списък от *URL* обекти съдържащи само един обект сочещ към архива който се доставя.

След създаването на *DefaultEJB* и свързаните с него обекти, контейнера провежда създаването на всички *DefaultInvocationChains* обекти реализиращи интерфейса *InvocationChains*. За всеки интерфейс на бийна се създава инстанция от този клас и се асоциира с *DefaultChainsManager* обекта. За всеки метод от интерфейса на бийна, както и за *null* метода съответстващ на създаване породено от внедряване по зависимост, се създава обект *DefaultInvocationChain* реализиращ интерфейса *InvocationChain* и се асоциира със списък от системни прихващащи класове. Допълнително се създават *DefaultInvocationChain* обекти асоциирани със всеки *Callback* обект имащ отношение към жизнения цикъл на бийна. Всички те се обвиват в инстанция на *DefaultCallbackInvocationChains* реализираща интерфейса *CallbackInvocationChains*.

След като всички вериги се инициализират контейнера създава JNDI средата на бийна попълвайки я с *javax.naming.Reference* обекти за съответния вид ресурси като например друг бийн. Ако системата поддържа глобален достъп до бийна, то контейнера публикува бийна в JNDI под специални имена които описват достъп до базовите или конкретен бизнес интерфейс на бийна. Системи, които подържат премахване на вече доставен архив оперират като премахват вече описаните обекти от системата и изриват достъпа в JNDI за всеки бийн. С това бийна престава да съществува в системата.

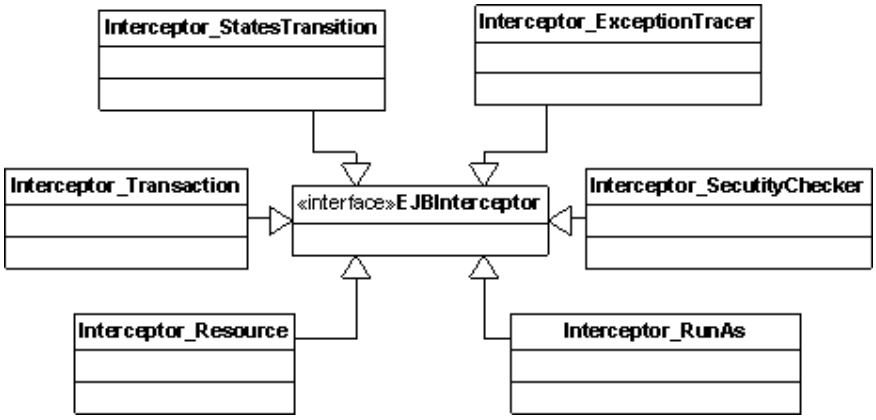
4.2. Системни прихващащи класове

Както бе многократно описано, основна роля за работата на контейнера заемат веригите от системни прихващащи класове. Те образуват списъци, подмножества на множеството от всички реализирани от контейнер такива. Всеки прихващащ клас допринася за осъществяване конкретна операция изисквана от спецификацията на контейнера и неговия дизайн.

4.2.1.Реализации на прихващащи класове

По своите функции системните прихващащи класове се делят на такива обслужващи контейнера и такива обслужващи приложните прихващащи класове.

Следните класове изобразени на Фигура 12 са общи са всички видове вериги.



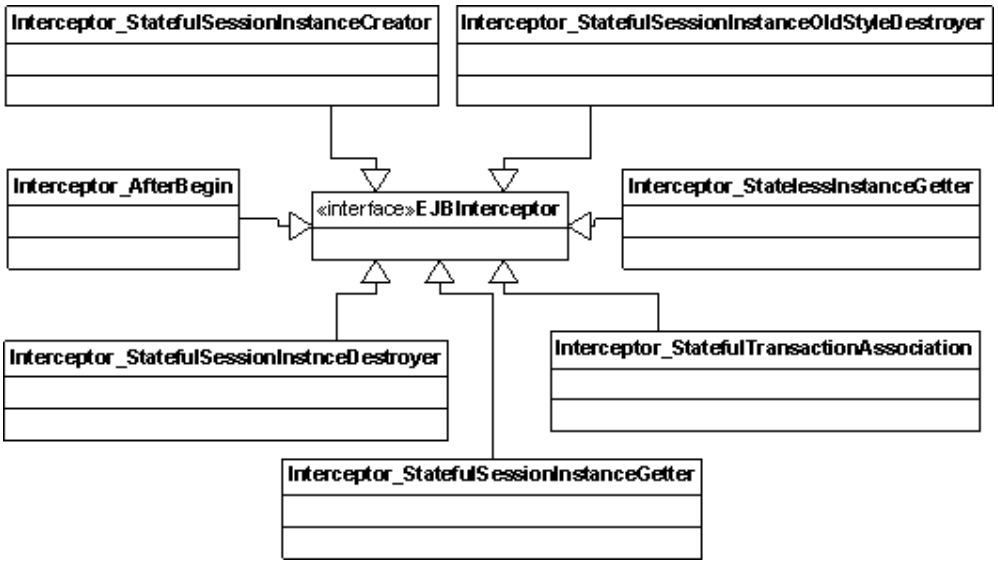
Фигура 12.

Системни прихващащи класове общи за всички вериги.

- **Interceptor_ExceptionTracer –** обработва изключенията предизвикани от изпълнението на бизнес извикването като го запазва в системния списък от съобщения
- **Interceptor_SecurityChecker –** проверява достъпа на текущия клиент към операцията която той извиква.
- **Interceptor_RunAs –** променя текущата сесия по сигурността да е асоциирана с името на друг потребител съобразно информацията предоставена от *доставчика на бийн*
- **Interceptor_Resource –** асоциира ресурсите и достъпа до тях на бийна с текущото извикване. След приключване на извикването, ресурсите се де –асоциират отново
- **Interceptor_StatesTransition –** поставя определено състояние на бийна с оглед достъпът му до определени ресурси и реализации на стандартни програмни интерфейси в *J2EE* платформата
- **Interceptor_Transaction –** определя транзакционния контекст, в който ще се изпълнят следващите операции от веригата. Транзакцията на клиента се запазва и възстановява след изпълнението

Следните класове изобразени на Фигура 13, участват във веригите с ролята да определят коя е текущата инстанция на бийн, да я асоциират с извикването от клиента в подходящо състояние и след приключване на извикването да

оставят инстанцията в съответно подходящо състояние. За тези си цели те използват методите предоставени от съответната реализация на *LifecycleManager*:



Фигура 13.

Системни прихващащи класове управляващи преходи от жизнения цикъл на бийна.

- **Interceptor_StatefulSessionInstanceCreator –** използвайки метода *create* създава нова сесия на състоятелен бийн и я асоциира с текущия *EJBInvocationContext*. След приключване на работа сесията се *пасивира* използвайки метода *passivate* или се маркира за *пасивиране* в края на транзакцията ако има такава
- **Interceptor_StatefulSessionInstanceGetter –** използвайки методите *getActiveInTransaction* и *activate* открива активна или активира съществуваща сесия на състоятелен бийн и я асоциира с текущия *EJBInvocationContext*. След приключване на работа сесията се пасивира използвайки метода *passivate* или се маркира за *пасивиране* в края на транзакцията ако има такава
- **Interceptor_StatefulSessionInstanceDestroyer –** използвайки методите *getActiveInTransaction* и *activate* открива активна или активира съществуваща сесия на състоятелен бийн и я асоциира с текущия

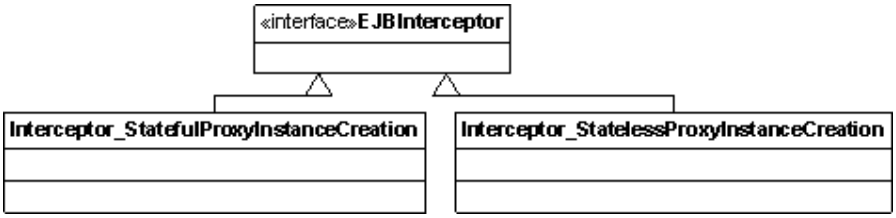
EJBInvocationContext. След приключване на работа сесията се премахва използвайки метода *destroy* или се маркира за премахване в края на транзакцията ако има такава. Използва се само във вериги за метод за премахване на бийн от бизнес интерфейс

- **Interceptor_StatefulSessionInstanceOldStyleDestroyer –** този клас е различен от *Interceptor_StatefulSessionInstanceDestroyer* само в отношение на сценариите, в които възниква грешка по време на извикването. Използва се само във вериги за метод за премахване на бийн по *EJB 2.1* спецификацията
- **Interceptor_StatelessInstanceGetter –** използвайки метода *getReady* получава активна сесия на несъстоятелен бийн и я асоциира с текущия *EJBInvocationContext*. След приключване на работа сесията се освобождава използвайки метода *releaseReady*

Следните класове са специфични само за веригите на състоятелен бийн спрямо ролата си:

- **Interceptor_StatefulTransactionAssociation –** асоциира текущата инстанция на сесия с транзакционния контекст ако има такъв. След приключване на работа оставя асоциацията до момента на приключване на транзакцията. Обекта управляващ асоциацията премахва или пасивира сесията в края на тразакцията в зависимост от това как е маркирана тя. Този обект изпълнява методите *beforeCompletion* и *afterCompletion* на бийна от интерфейса *SessionSynchronization*
- **Interceptor_AfterBegin –** изпълнява метода *afterBegin* на бийна от интерфейса *SessionSynchronization*

Прихващащите класове, които участват при създаването на *ClientView* обекти са изобразени на Фигура 14.

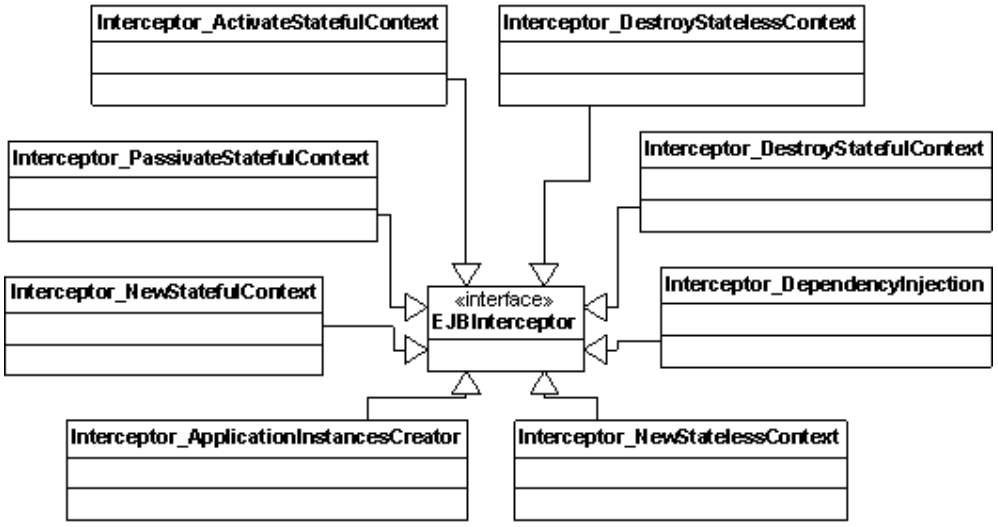


Фигура 14.

Системни прихващащи класове създаващи *ClientView* обекти.

- **Interceptor_StatefulProxyInstanceCreation –** създава нова инстанция на реализация на бизнес или компонентен интерфейс на състоятелен бийн използвайки *DefaultClientViewsManager* и съответното *ClientViewFactory*
- **Interceptor_StatelessProxyInstanceCreation –** създава нова инстанция на реализация на бизнес или компонентен интерфейс на несъстоятелен бийн използвайки *DefaultClientViewsManager* и съответното *ClientViewFactory*

Прихващащите класове, които участват във веригите за управление на жизнения цикъл на бийновете са изобразени на Фигура 15.

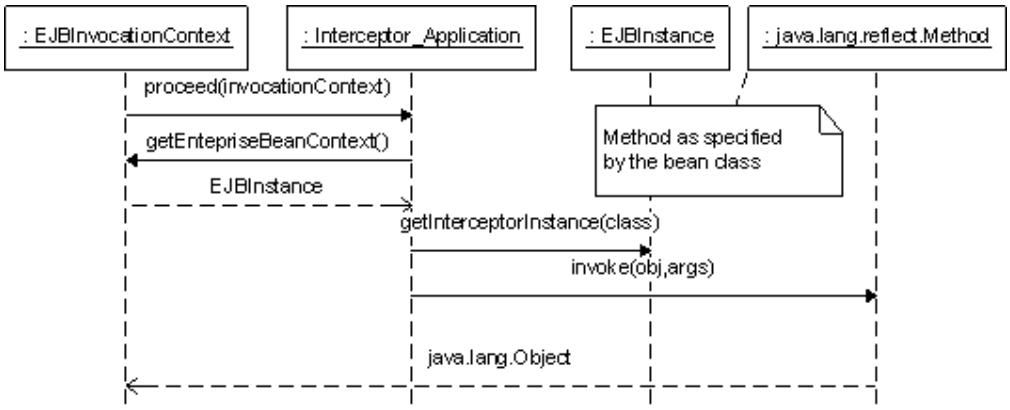


Фигура 15.

Системни прихващащи класове реализиращи жизнения цикъл на бийна.

- **Interceptor_ActivateStatefulContext –** реконструира инстанцията на състоятелен бийн от системата за вторично съхранение
- **Interceptor_PassivateStatefulContext –** прехвърля инстанцията на състоятелен бийн в системата за вторично съхранение
- **Interceptor_NewStatefulContext –** създава нова инстанция на *StatefulEJBInstance* реализираща интерфейса *EJBInstance* и я асоциира с текущия *EJBInvocationContext*
- **Interceptor_ApplicationInstancesCreator –** създава инстанцията от класа на бийна и от приложните прихващащи класове асоциирани с него и ги асоциира с текущата *EJBInstance* асоциирана с *EJBInvocationContext*
- **Interceptor_DependencyInjection –** реализира операциите на внедряване по зависимост върху инстанцията на бийна и инстанциите на приложните прихващащи класове
- **Interceptor_DestroyStatefulContext –** премахва обекта *StatefulEJBInstance* като освобождава заетите от него системни ресурси
- **Interceptor_NewStatelessContext –** създава нова инстанция на *StatelessEJBInstance* реализираща интерфейса *EJBInstance* и я асоциира с текущия *EJBInvocationContext*
- **Interceptor_DestroyStatelessContext –** премахва обекта *StatelessEJBInstance* като освобождава заетите от него системни ресурси

Приложните прихващащи класове се съвместяват във веригата от системни такива, както е показано на Фигура 16.



Фигура 16.

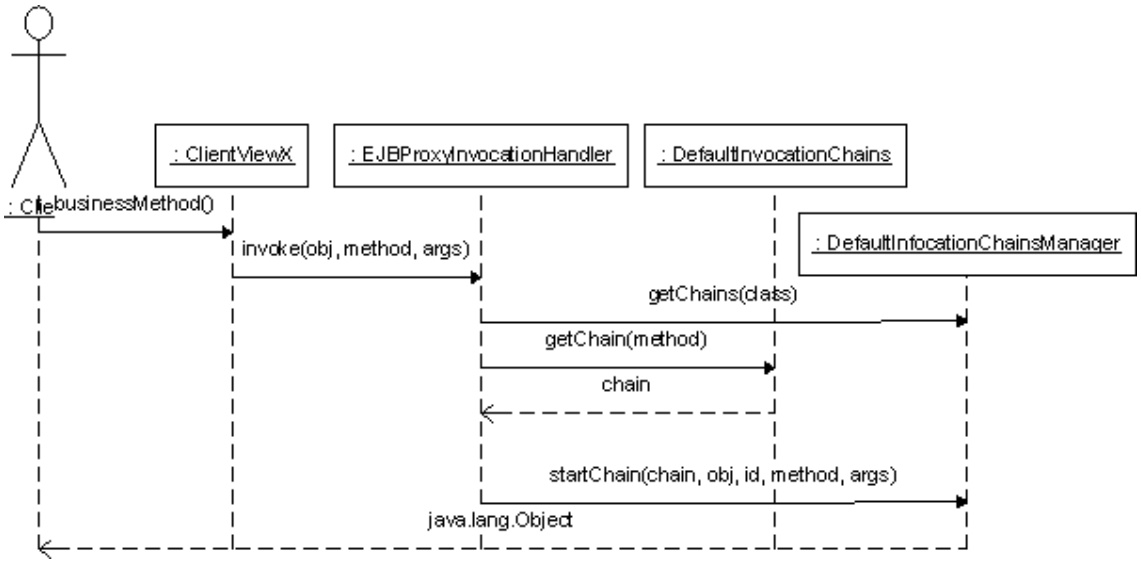
Съвместяване на приложните прихващащи класове със ситемните.

За целта се използват следните системни прихващащи класове:

- **Interceptor_Application –** извиква метод на приложен прихващащ клас с анотация *ArroundInvoke*
- **Interceptor_CallbackApplication –** извиква метод на приложен прихващащ клас с анотация *PostConstruct, PreDestroy, PrePassivate* или *PostActivate*

4.2.2.Конкретни вериги на прихващащи класове в контейнера

Прихващащите класове се обединяват в конкретни вериги. Те се изпълняват в съответствие на конкретна заявка от клиента както е изобразено на Фигура 17.



Фигура 17.

Извикване на вериги от прихващащи класове.

При доставянето на състоятелен бийн контейнера създава следните вериги от прихващащи класове разпределени по вид на интерфейси и операции, с които се асоциират:

- За метод на базов интерфейс – *Interceptor_ExceptionTracer, Interceptor_SecurityChecker, , Interceptor_StatefulProxyInstanceCreation, Interceptor_StatefulSessionInstanceCreator, Interceptor_StatesTransition, Interceptor_Resource, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове)
- За бизнес метод на компонентен или бизнес интерфейс – *Interceptor_ExceptionTracer, Interceptor_SecurityChecker, Interceptor_RunAs* (ако е специфицирано), *Interceptor_StatefulSessionInstanceGetter* (за метод за премахване се специфицира *Interceptor_StatefulSessionInstanceDestroyer*), *Interceptor_Transaction, Interceptor_Resource, Interceptor_AfterBegin, Interceptor_StatesTransition, Interceptor_Application* (по един за всички асоциирани прихващащи класове)
- За създаване породено от внедряване по зависимост – *Interceptor_ExceptionTracer, Interceptor_StatefulProxyInstanceCreation, Interceptor_StatefulSessionInstanceCreator*
- За активиране – *Interceptor_PassivateStatefulContext Interceptor_StatesTransition, Interceptor_Resource, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове)
- За пасивиране – *Interceptor_ActivateStatefulContext, Interceptor_StatesTransition, Interceptor_Resource, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове)
- За създаване – *Interceptor_NewStatefulContext, Interceptor_Resource, Interceptor_ApplicationInstancesCreator, Interceptor_StatesTransition, Interceptor_DependencyInjection, Interceptor_StatesTransition, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове)
- За премахване – *Interceptor_StatesTransition, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове), *Interceptor_DestroyStatefulContext*

- За унищожаване поради грешка - *Interceptor_DestroyStatefulContext*

При доставянето на несъстоятелен бийн контейнера създава следните вериги от прихващащи класове, разпределени по вид на интерфейси и операции, с които се асоциират:

- За метод на базов интерфейс **–** *Interceptor_SecurityChecker, Interceptor_StatelessProxyInstanceCreation*
- За бизнес метод на компонентен или бизнес интерфейс – *Interceptor_ExceptionTracer, Interceptor_SecurityChecker, Interceptor_RunAs* (ако е специфицирано), *Interceptor_StatelessInstanceGetter, Interceptor_Transaction , Interceptor_Resource, Interceptor_StatesTransition, Interceptor_Application* (по един за всички асоциирани прихващащи класове)
- За създаване породено от внедряване по зависимост – *Interceptor_ExceptionTracer, Interceptor_StatelessProxyInstanceCreation*
- За създаване – *Interceptor_NewStatelessContext, Interceptor_Resource, Interceptor_ApplicationInstancesCreator, Interceptor_StatesTransition, Interceptor_DependencyInjection, Interceptor_StatesTransition, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове)
- За премахване – *Interceptor_StatesTransition, Interceptor_CallbackApplication* (по един за всички асоциирани прихващащи класове), *Interceptor_DestroyStatelessContext*
- За унищожаване поради грешка - Interceptor_DestroyStatelessContext

5. Тестване

Изискванията, които поставят разработчиците на приложения към промишлената среда включват както продуктивност така и коректност на изпълнение. За да се гарантират тези качества на разработената реализация на EJB Container са приложени различни начини на тестване съвместими с продукта.

5.1. Вътрешно тестване

Вътрешното тестване (white box testing) се базира на конкретните Java класове реализиращи контейнера. Класовете от реализацията се разделят на единици (units) и за всяка от тях са разработени набор от JUnit тестове. Това са всички прихващащи класове, менажерите за достъп от клиент, за жизнения цикъл на бийн и за веригите от прихващащи класове и *EJBContainer* реализацията като компонента отговорен за тяхното конструиране.

Всеки тест за прихващащите класове се състои от три стъпки. Първо се създават необходимите обекти, които да пресъздават реалната среда на работа. Например ако се тества прихващащия клас за транзакционния контекст то е необходимо да се инициализира JTA. Следващата стъпка е изпълнение на метода, който се тества. За прихващащите класове има две основни тестови точки. Това са, когато прихващащия клас предава изпълнението на следващия такъв и когато изпълнението на метода завърши. Последната стъпка от тестването е проверка дали във всяка от тестовите точки изпълнението на метода е превело системата в правилно състояние. В примера с транзакционния прихващащ клас това означава дали той е гарантирал правилния транзакционен контекст на бийна и дали след завършване е превел състоянието на транзакционния контекст до очакваното състояние от клиента.

Всеки тест за *EJBContainer* реализацията се състо от две стъпки. В първата се предава изпълненито на *EJBContainer* реализацията с информация за конкретен бийн постъпил в системата. Втората се състои от проверка дали са създадени правилните менажери и дали данните, които те съдържат са коректни.

5.2. Външно тестване

Външното тестване (black box testing) се основава на сценариите описани в реализираните спецификациите. То се базира на конкретни приложения съдържащи промишлени бийнове и верифицира реализацията на договорите клиент-контейнер и клиент-бийн. Всеки тест съответства на конкретно изискване и сценарий. На няколко сценария, които засягат еднаква група бийнове съответства едно приложение.

Спецификацията за платформата J2EE съдържа набор от тестове за съвместимост – CTS (Compatibility Test Suite). Този набор се състои от приблизително 3000 сценария, пакетирани в повече от 1000 тестови приложения. Те са разработени като външни тестове и служат като гарант, че дадена промишлена платформа е реализирана в съответствие с изискванията. При промишленото внедряване на изложената реализация този набор от тестове бе използван за да се верифицира както качеството на контейнера така и начина му на внедряване в системата.

5.3. Тестване за производителност

Тестовете за производителност (benchmark testing) се състоят от няколко компонента. Първият е приложение, което съдържа няколко промишлени функции максимално приближаващи се до реалните. Вторият е система, с която се симулират променлив брой виртуални потребители. Третият е система за отчитане, която натрупва данните за скоростта и количеството на изпълнените заявки от всеки виртуален потребител.

Приложението се доставя до реализацията, след което се стартират другите две системи за определено тестово време. След изминаването на това време се съставят таблици на всички отчетени показатели спрямо времето на изпълнение и количеството виртуални потребители. Средното време за изпълнение на промишлена функция е показател за сравнение с други промишлени системи. Ако с нарастването на броя виртуални потребители това

време се запазва то това показа, че системата е скалируема (scalable). От друга страна в различни етапи на подобряване на продукта този вид тестване показва количествена оценка на подобрението.

При промишленото внедряване на разработената системата, за тест за производителност бе използвано приложението SPECjAppServer2004 [14]. Чрез него бе потвърдена скалируемостта на реализацията и нейната добра производителност.

Заключение

Процеса на развитие на EJB и J2EE архитектурата е постоянен. Обществото от Java разработчици се стреми да въведе доминираш модел за работа с Java, който едновременно да улесни работата им и подобри качеството и съвместимостта на продуктите. В контекста на това развитие реализацията на EJB контейнер е заемала и заема основна роля. За да покрие както съществуващите така и предстоящите изисквания, дизайна на контейнера трябва да е максимално гъвкав и разчупен спрямо функционалността на обединяващите го компоненти.

Основна цел при разработката на предложения дизайн е с минимални добавки той да покрива произволна система от правила върху компоненти. Изолирайки отделните подсистеми една от друга и поставяйки общ интерфейс между тях позволява реализиране на произволен изглед към клиенти, жизнен цикъл и управление на бизнес извикванията.

В не по-малка степен е важна възможността за интеграция на предложения дизайн и реализация с произволна Java платформа предлагаща, JNDI, RMI и JTA услуги.

Предложената реализация е внедрена успешно в продукта SAP NetWeaver Application Server Java EE 5 Edition [13]. Той успешно премина всички необходими тестове и се сертифицира като Java EE 5.0 съвместим (compliant). За да се постигне това реализацията на прихващащите класове бе адаптирана към предоставените от платформата реализации на Java EE технологиите.

Насока за бъдещо развитие е адаптирането на реализацията към система за тестване с цел улесняване тестването на разработвани бийнове. По този начин ще се съкрати процеса на тестване, тъй като няма да е необходимо бийновете да се доставят до промишлената система преди да се изпълни даден набор от тестове.

Списък на използваните съкращения и термини

1. CORBA (Common Object Request Broker Architecture) – общ стандарт за система приемаща и обработваща заявки за изпълнение към обекти

2. DNS (Domain Name System) – протокол за достъп до система предлагаща поименен указател за устройства

3. EJB (Enterprise Java Bean) – промишлени бийнове на Java

4. IIOP (Internet Inter-ORB Protocol) – вътрешен протокол за комуникация на ORB

5. J2EE (Java 2 Enterprise Edition) – промишлена платформа на Java версия 2

6. JMS (Java Messaging Service) – стандарт в Java предоставящ възможност за изпращане на съобщения и регистриране за получаването им към централизирана система

7. JNDI (Java Naming and Directory Interface) – стандарт за поименен указател в Java

8. LDAP (Lightweight Directory Access Protocol) – протокол за достъп до поименен указател

9. Online – чрез Интернет

10. ORB (Object Request Broker) – компонент от CORBA приемащ и обработващ заявки за изпълнение към обекти

11. RMI (Remote Method Invocation) – отдалечено извикване на методи

12. URL (Unified Resource Location) – система за обозначаване на ресурси със уникален идентификатор чрез който те може да се достъпят по определен протокол

13. Web –предоставящ достъп през интернет

14. XML (Extensible Markup Language) – саморазширяващ се език за описание и обозначение на данни

15. бийн – наименование на промишлени компоненти в Java

16. Прихващащ – който осъществява прозрачен достъп до данните на процес в определен момент от изпълнението на процеса.

17. Промишлен – който участва в система за производство, контрол или съхранение на данни.

18. сесиен – последователен
19. състоятелен - който запазва състоянието си в сесия

Използвана литература

1. Enterprise JavaBeans, v 3.0 EJB 3.0 Simplified API, http://java.sun.com/products/ejb

2. Enterprise JavaBeans, v 2.1, http://java.sun.com/products/ejb

3. Enterprise JavaBeans, v 3.0 EJB Core Contract and Requirements, http://java.sun.com/products/ejb

4. Java Naming and Directory Interface (JNDI), http://java.sun.com/products/jndi

5. Java Remote Method Invocation (RMI), http://java.sun.com/products/rmi

6. Java Transaction API (JTA), http://java.sun.com/products/jta

7. Java Platform, Enterprise Edition (Java EE), v 5, http://java.sun.com/javaee

8. JSR-250: Common Annotations for the Java Platform, http://jcp.org/en/jsr/detail?id=250

9. RMI Tutorial, http://www.ccs.neu.edu/home/kenb/com3337/rmi_tut.html

10. RMI-IIOP Tutorial, http://www.science.uva.nl/ict/ossdocs/java/jdk1.3/docs/guilde/rmi-iiop/rmi_iiop_pg.html

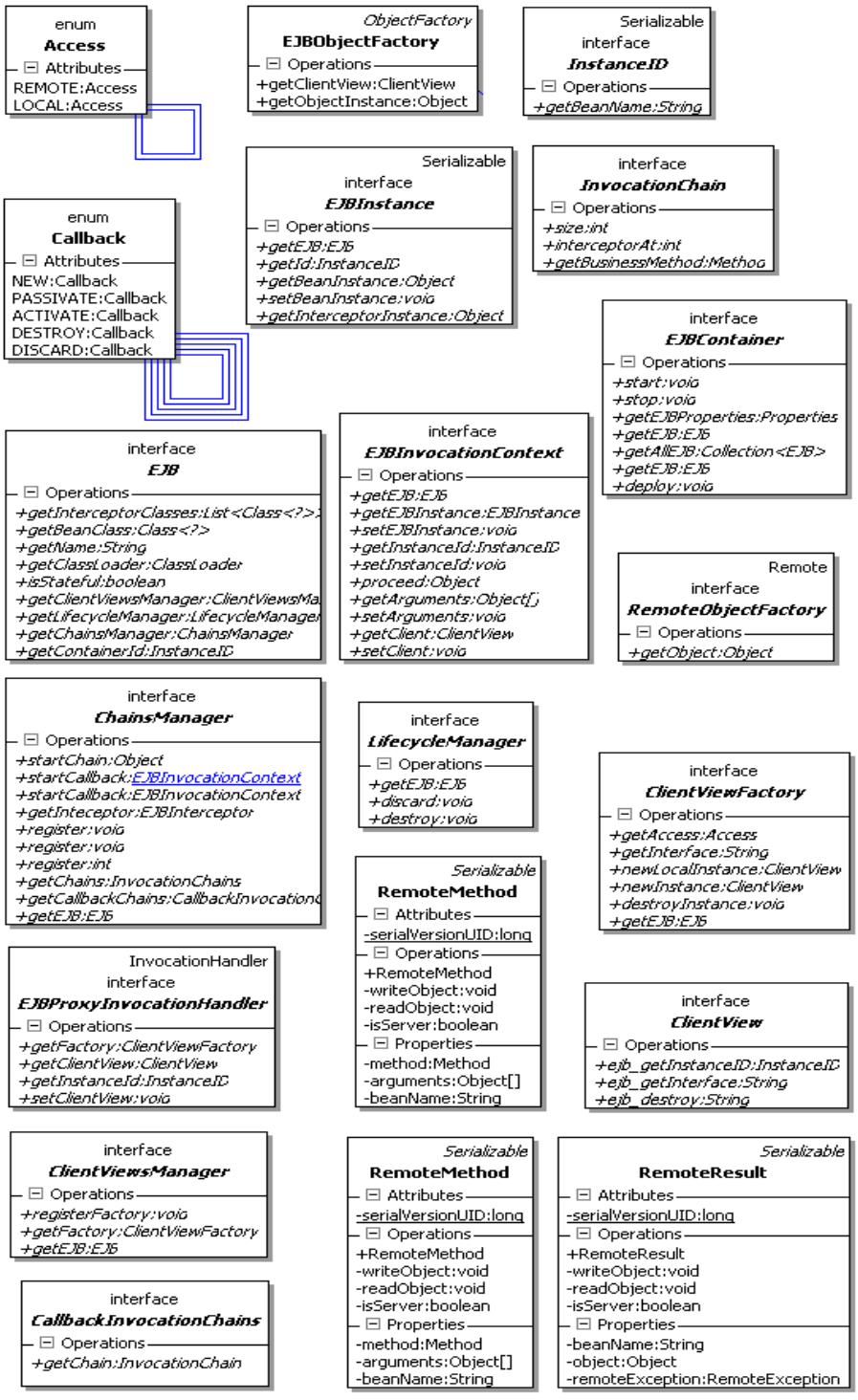
11. JTA tutorial, http://www.datadirect.com/developer/jdbc/topics/jta/index.ssp

12. ACID Transactions, http://www.nusphere.com/products/library/acid_transactions.htm

13. https://www.sdn.sap.com/irj/sdn?rid=/library/uuid/da699d27-0b01-0010-99b0-f11458f31ef2 - SAP NetWeaver Application Server Java EE 5 Edition

14. http://www.spec.org/jAppServer2004/index.html - SPECjAppServer2004 application server benchmark

Приложение – UML class диаграми и Java код на класове от реализацията



Диаграма 1.

UML Class диаграма на интерфейсите в пакета org.fmi.ejb

Диаграма 2.

UML Class диаграма на пакета org.fmi.ejb.impl

```
package org.fmi.ejb.deployer;
public class Deployer implements Runnable {
        private File appDir;
        private EJBContainer container;
        private List<String> names;
        private final PrintStream ERR = System.err;
        private long lastModified;
        private boolean stop = false;
        public Deployer(File dir,EJBContainer container) {
                this.appDir = dir;
                this.container = container;
                this.names = new ArrayList<String>();
        }
```

```
	public void run() {
		lastModified = appDir.lastModified();
		checkAppDirContent();

		while (!stop) {
			if ( lastModified < appDir.lastModified() ) {
				lastModified = appDir.lastModified();
				checkAppDirContent();
			}
			try {
				Thread.sleep(2000);
			} catch(InterruptedException ex) {
				ex.printStackTrace(ERR);
			}
		}
	}

	private void checkAppDirContent() {
		File[] jars = appDir.listFiles(JarFilter.FILTER);
		for (File jar:jars) {
			if (names.contains(jar.getName())) {
				continue;
			}
			try {
				container.deploy(jar);
				names.add(jar.getName());
			} catch (EJBException ex) {
				ex.printStackTrace(ERR);
			}
		}
	}

	private static class JarFilter implements FileFilter {
		private JarFilter() {}
		private static final JarFilter FILTER = new JarFilter();
		public boolean accept(File pathname) {
			return pathname.isFile() &&
				pathname.getName().endsWith(".jar");
		}
	}
	public void stop() {
		stop = true;
	}
}
```

```
package org.fmi.ejb.deployer;

public class EJBContainerImpl implements EJBContainer {

	private static final PrintStream PS = System.out;
	private Deployer deployer;
	private IIOPRemoteObjectFactory iiopAccessFactory;

	private List<DeploymentInfo> deploymentInfo;
	private Map<String,EJB> ejbs;
//	private ORB orb;


	EJBContainerImpl(File deployDir) {
		deployer = new Deployer(deployDir,this);
		deploymentInfo = new ArrayList<DeploymentInfo>();
		ejbs = new HashMap<String,EJB>();
		instance = this;
		initIIOPAccess();
	}


	private void initIIOPAccess() {
		try {
			InitialContext ictx = new InitialContext();
			String host = System.getProperty("org.fmi.ejb.orbhost","localhost");
			String port = System.getProperty("org.fmi.ejb.orbport","900");
			Context ctx = (Context)ictx.lookup("corbaname:iiop:1.2@" + host + ":" + port +
"#/");
			iiopAccessFactory = new EJBObjectFactory();
			PortableRemoteObject.exportObject(iiopAccessFactory);
			ctx.rebind("IIOPRemoteObjectFactory",iiopAccessFactory);
		} catch(Exception ex) {
			throw new EJBException(ex);
		}
	}

	public void start() {
		PS.print("Starting EJB Container...");
		Thread t = new Thread(deployer,"EJB Deployer");
		t.setDaemon(true);
		t.start();
		PS.println(" done.");
	}
```

```
public void stop() {
	PS.print("Stopping EJB Container...");
	deployer.stop();
	PS.println(" done.");
}

public Properties getEJBProperties() {
	// TODO Auto-generated method stub
	return null;
}

public EJB getEJB(String name) {
	return ejbs.get(name);
}

public Collection<EJB> getAllEJB() {
	// TODO Auto-generated method stub
	return null;
}

public EJB getEJB(Reference ref) {
	// TODO Auto-generated method stub
	return null;
}

public void deploy(File jar) {
	PS.println("Start deploying '" + jar + "' ... ");
	DeploymentInfo info = new DeploymentInfo(jar);
	deploymentInfo.add(info);
	startEJBs(info);
	PS.println("End deploying '" + jar + "' ... ");
}

private void startEJBs(DeploymentInfo info) {
	Map<String,Class> stateless = info.getStatelessBeans();
	for (String beanName:stateless.keySet()) {
		Class clazz = stateless.get(beanName);
		assert clazz != null;
		deployBean(beanName,clazz,true,info);
	}
	Map<String,Class> stateful = info.getStatefulBeans();
	for (String beanName:stateful.keySet()) {
		Class clazz = stateful.get(beanName);
		assert clazz != null;
		deployBean(beanName,clazz,false,info);
```

```
		}
	}


	private void deployBean(String name,Class clazz,boolean isStateless,DeploymentInfo info) {
		DefaultEJB ejb = new DefaultEJB(name,clazz,info.getClassLoader(),!isStateless);
		ejb.setChainsManager(new DefaultChainsManager(ejb));
		ejb.setClientViewsManager(new DefaultClientViewsManager(ejb));
		if (isStateless) {
			ejb.setLifecycleManager(new StatelessLifecycleManager(ejb));
		} else {
			ejb.setLifecycleManager(new StatefulLifecycleManager(ejb));
		}
		ejbs.put(name,ejb);

		// init remote client view factories and invocation chains
		Remote remoteAnnotation = (Remote)ejb.getBeanClass().getAnnotation(Remote.class);
		if ( remoteAnnotation != null ) {
			for (Class remoteInterface:remoteAnnotation.value()) {
				// create factory
				RemoteClientViewFactory factory = new
RemoteClientViewFactory(Access.REMOTE,
							new
RemoteEJBProxyInvocationHandlerFactory(),remoteInterface,ejb);
				ejb.getClientViewsManager().registerFactory(factory);

				DefaultInvocationChains chains = new
DefaultInvocationChains(remoteInterface);

				//add all chains
				for (Method method:remoteInterface.getMethods()) {
					List<Integer> interceptors = new ArrayList<Integer>();
					// TODO
					interceptors.add(ejb.getChainsManager().register(
							new Interceptor_StatelessInstanceGetter()));
					DefaultInvocationChain chain = new
DefaultInvocationChain(interceptors,

	getBusinessMethod(method,ejb.getBeanClass()));
					chains.addChain(method,chain);
				}
				//add chains for creation due to dependency injection
				List<Integer> interceptors = new ArrayList<Integer>();
				if (isStateless) {
					interceptors.add(ejb.getChainsManager().register(
```

```
                                                                        new
Interceptor_StatelessProxyInstanceCreation(remoteInterface.getName())));
                                        } else {
                                                // TODO
                                        }
                                        DefaultInvocationChain chain = new
DefaultInvocationChain(interceptors,null);
                                        chains.addChain(null,chain);

                                        ejb.getChainsManager().register(chains);
                                }
                        }
                        // init local client view factories and invocation chains

                        // init lifecycle invocation chains
                        DefaultCallbackInvocationChains callbackchains = new DefaultCallbackInvocationChains();
                        if (isStateless) {
                                // NEW
                                List<Integer> interceptors = new ArrayList<Integer>();
                                interceptors.add(ejb.getChainsManager().register(new
Interceptor_NewStatelessContext()));
                                interceptors.add(ejb.getChainsManager().register(new
Interceptor_ApplicationInstancesCreator()));
                                DefaultInvocationChain chain = new DefaultInvocationChain(interceptors,null);
                                callbackchains.addChain(Callback.NEW,chain);
                                // DESTROY
                                // DISCARD
                        } else {

                        }
                        ejb.getChainsManager().register(callbackchains);
                }

                private Method getBusinessMethod(Method intf, Class<?> beanClass) {
                        try {
                                return
beanClass.getDeclaredMethod(intf.getName(),(Class[])intf.getParameterTypes());
                        } catch(Exception ex) {
                                throw new EJBException();
                        }
                }

                /*
                 * MAIN method
                 */
```

```
	public static void main(String[] args) throws Exception {
		EJBContainerImpl container = new EJBContainerImpl(
				new File(".","deploy"));
		final Object LOCK = new Object();
		Runtime.getRuntime().addShutdownHook(new Thread(new Runnable(){
			public void run() {
				synchronized(LOCK) {
					LOCK.notifyAll();
				}
			}
		}));
		container.start();
		synchronized (LOCK) {
			LOCK.wait();
		}
		container.stop();
	}


	private static class DeploymentInfo {

		private EJBClassLoader loader;
		private File file;
		private Map<String,Class> stateless;
		private Map<String,Class> stateful;

		DeploymentInfo(File file) {
			this.file = file;
			try {
				this.loader = new EJBClassLoader(
						new
URL[]{file.toURL()},DeploymentInfo.class.getClassLoader());
			} catch(MalformedURLException ex) {
				throw new EJBException(ex);
			}
			stateless = new HashMap<String,Class>();
			stateful = new HashMap<String,Class>();
			initBeans();
			List<String> names = new ArrayList<String>();
			names.addAll(stateless.keySet());
			names.addAll(stateful.keySet());
			loader.setBeanNames(names);
		}

		public Map<String,Class> getStatelessBeans() {
```

```
            return Collections.unmodifiableMap(stateless);
    }

    public Map<String,Class> getStatefulBeans() {
            return Collections.unmodifiableMap(stateful);
    }

    public EJBClassLoader getClassLoader() {
            return loader;
    }

    private void initBeans() {
            try {
                    JarFile jar = new JarFile(file);
                    Enumeration<JarEntry> entries = jar.entries();
                    while (entries.hasMoreElements()) {
                            JarEntry entry = entries.nextElement();
                            if ( entry.isDirectory() ) {
                                    continue;
                            }
                            String name = entry.getName();
                            if ( !name.endsWith(".class") ) {
                                    continue;
                            }

                            String className = name.substring(0,name.length() -
                                            ".class".length()).replace("/",".");
                            Class<?> clazz = loader.loadClass(className);

                            Stateless sl = null;
                            if ( (sl = (Stateless)clazz.getAnnotation(Stateless.class)) != null ) {
                                    String beanName = null;
                                    if ( sl.name() != null && !sl.name().equals("")) {
                                            beanName = sl.name();
                                    } else {
                                            beanName = className.substring(
                                                            className.lastIndexOf(".")
+ 1,
                                                            className.length());
                                    }
                                    stateless.put(beanName,clazz);
                                    PS.println(" found stateless bean '" + beanName + "' in
class " +
                                                    className);
                                    continue;
```

```
                                                }
                                                Stateful sf = null;
                                                if ( (sf = (Stateful)clazz.getAnnotation(Stateful.class)) != null ) {
                                                        String beanName = null;
                                                        if ( sf.name() != null && !sf.name().equals("")) {
                                                                beanName = sf.name();
                                                        } else {
                                                                beanName = className.substring(
                                                                                className.lastIndexOf(".")
+ 1,
                                                                                className.length());
                                                        }
                                                        stateful.put(beanName,clazz);
                                                        PS.println(" found stateless bean '" + beanName + "' in
class " +
                                                                        className);
                                                        continue;
                                                }
                                        }
                                } catch(NoClassDefFoundError er) {
                                        throw new EJBException(new RuntimeException(er));
                                } catch(ClassNotFoundException ex) {
                                        throw new EJBException(ex);
                                } catch(IOException ex) {
                                        throw new EJBException(ex);
                                }
                        }

                }

                static EJBContainer instance;

                public static EJBContainer getInstance() {
                        return instance;
                }
        }

        public class EJBClassLoader extends URLClassLoader {

                private List<String> beanNames;

                public EJBClassLoader(URL[] urls, ClassLoader parent) {
                        super(urls, parent);
                }
```

```
	protected void setBeanNames(List<String> beanNames) {
		if (this.beanNames != null) {
			throw new IllegalStateException("beanNames already initialized");
		}
		this.beanNames = beanNames;
	}

	protected List<String> getBeanNames() {
		if (this.beanNames == null) {
			throw new IllegalStateException("beanNames not initialized");
		}
		return beanNames.subList(0,beanNames.size());
	}
}

public class DefaultEJBProxyInvocationHandler implements EJBProxyInvocationHandler {

	private ClientViewFactory factory;
	private ClientView clientView;
	private InstanceID id;
	protected Class clazz;

	DefaultEJBProxyInvocationHandler(InstanceID id,ClientViewFactory factory) {
		this.id = id;
		this.factory = factory;
		try {
			clazz = factory.getEJB().getClassLoader().loadClass(factory.getInterface());
		} catch(ClassNotFoundException cnfex) {
			throw new EJBException(cnfex);
		}
	}

	public ClientView getClientView() {
		return clientView;
	}

	public InstanceID getInstanceId() {
		return id;
	}

	public ClientViewFactory getFactory() {
		return factory;
	}

	public void setClientView(ClientView clientView) {
```

```
            this.clientView = clientView;
        }

        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            if (method.getDeclaringClass() == Object.class) {
                return method.invoke(this,args);
            }

            if (method.getDeclaringClass() == ClientView.class) {
                if (method.getName().equals("ejb_getInstanceID")) {
                    return id;
                }
                if (method.getName().equals("ejb_getInterface")) {
                    return factory.getInterface();
                }
                if (method.getName().equals("ejb_destroy")) {
                    factory.destroyInstance(clientView);
                    return null;
                }
                throw new IllegalArgumentException("Unknown method " + method );
            }
            ChainsManager manager = factory.getEJB().getChainsManager();
            InvocationChain chain = manager.getChains(clazz).getChain(method);
            try {
                return manager.startChain(chain,clientView,id,method,args);
            } catch(ApplicationException ex) {
                throw ex.getCause();
            }
        }
    }
```