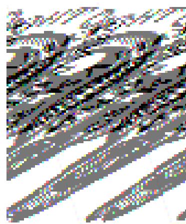
СУ "Св. Климент Охридски"
Факултет по Математика и Информатика
Катедра "Информационни Технологии"



ДИПЛОМНА РАБОТА

Система за регистрация и управление на доставчици на JNDI услуги

Дипломант: Елица Николова Панчева
Степен: Магистър
Специалност: Разпределени Системи и Мобилни Технологии
Факултетен номер: М-21360
Научен ръководител: доц. Боян Бончев
гр. София, Октомври 2007

Съдържание:

1. Увод 4
- 1.1. Въведение в проблемната област 4
- 1.2. Цел на разработката 4
- 1.3. Структура на разработката 5

2. Технологичен преглед 7
- 2.1. Java. 7
- 2.2. JNDI 8
  - 2.2.1. JNDI API 9
  - 2.2.2. JNDI SPI 10
- 2.3. Именуваща услуга (naming service). 10
- 2.4. Доставчик на именуващи услуги 12
  - 2.4.1. Интеграция с JNDI SPI 12
  - 2.4.2. Factory типове предоставящи контекст имплементации 12
- 2.5. Семантичен обект 15
- 2.6. Доставчик на семантични обекти 16
  - 2.6.1. Интеграция с JNDI SPI 16
  - 2.6.2. Factory типове за писане и четене на обекти 16
- 2.7. Механизъм за намиране и създаване на factory инстанции в JNDI 18
- 2.8. Сървър за Java EE (Enterprise Edition) приложения 19
  - 2.8.1. Проблеми при интеграцията с JNDI 20
  - 2.8.2. Частични решения 20

3. Функционален анализ на проблемната област 22
- 3.1. Стандартна интеграция на factory имплементации в JNDI средата 22
  - 3.1.1. NamingManager 22
  - 3.1.2. Създаване на нов начален контекст 23
  - 3.1.3. Именуване на обект 25
  - 3.1.4. Търсене на обект по име 26
- 3.2. Налични ограничения в JNDI SPI 27

4. Дефиниране на функционалните изисквания към разработката 29

5. Проектиране на решението 30
- 5.1. Изисквания за качество и съвместимост 30
- 5.2. Test-driven development process 30
  - 5.2.1. Автоматизирани тестове 31
  - 5.2.2. JUnit 31
- 5.3. Дефиниране на тестовите сценарии 32
  - 5.3.1. Функционална цялост 32
  - 5.3.2. Съвместимост със стандартната функционалност на JNDI средата 35
  - 5.3.3. Оптималност и бързодействие 36
- 5.4. Архитектурни концепции 38

6. Реализация 40
- 6.1. Контейнер за доставчици на jndi услуги 40
- 6.2. Интеграция с JNDI средата 42

7. Проверка за качество и коректност на услугата 51

8. Подобрения и разширения 55
- 8.1. Функционални 55
- 8.2. Оптимизационни 56

9. Ръководство за употреба 58

9.1. Инициализация 58
9.2. Клиенти на jndi услуги 58
9.3. Доставчици на jndi услуги 59
9.4. Миграция на вече съществуващи приложения 59
10. Заключение 60
10.1. Възможни подобрения и бъдещи насоки 60
11. Допълнения 62
11.1. Речник на използваните термини 62
12. Използвана литература 65

1. Увод

Todd Sundsted започва статията си „Представяне на JNDI” с разходка в градската библиотека – хиляди книги подредени на прашни рафтове. Единственият начин да намериш тази, която ти трябва е да потърсиш името и в каталога на библиотеката. [1] В света на компютърните технологии подобни каталог структури се наричат услуги за именуване. Тези услуги служат за асоцииране на име и местоположение на данни и услуги. Чрез тях компютърните програми еднозначно намират местоположението на ресурсите и услугите, от които се нуждаят.

1.1. Въведение в проблемната област

Намирането на необходимите ресурси е от особено значение в разпределените сървър (Enterprice Server) системи, където едни приложения предоставят услуги за други приложения, като различните групи приложения се разработват от различни доставчици и не могат да комуникират директно помежду си. [2] Именуващите услуги дават възможност на различните приложения да се откриват едно друго и да обменят данни и услуги.

В една голяма сървър система може да има различни реализации на именуващи услуги в зависимост от данните, които приемат, начина им на съхраняване, предназначението и насочеността им. В езика за програмиране Java има вграден механизъм за откриване и използване на различни именуващи услуги наречен JNDI – Java Naming and Directory Interfaces. На пръв поглед JNDI предоставя универсална интеграция за различни доставчици на услуги за именуване (naming service provider), но на втори план откриваме редица ограничения, които затрудняват предоставянето и използването на именуващи услуги в по-сложни приложения и сървър системи.

1.2. Цел на разработката

Настоящата дипломна работа ще опише ограниченията наложени от JNDI спецификацията, ще представи проблемите възникващи в резултат на тези ограничения и ще предложи начин за разрешаването им посредством система за регистрация и управление на доставчици на jndi услуги.

1.3. Структура на разработката

Дипломната работа съдържа следните части:

Част 1. Увод

Въвежда читателя в предметната област и същността на разглеждания проблем. Представя целта на разработката и потребността, която тя адресира. Описва структурата на дипломната работа.

Част 2. Технологичен преглед

Съдържа общ преглед на технологиите засегнати в процеса на разработката. Описва взаимодействието и интеграцията между тях като постепенно изгражда модел на проблемната област.

Част 3. Функционален анализ на проблемната област

Представя стандартната функционалност заложена в JNDI средата за интеграция на доставчици на jndi услуги и ролята на NamingManager класа в нея. Разглежда семантиката на основните операции в JNDI - създаване на нов начален контекст, именуване на обект, търсене на обект по име. Разкрива ограниченията наложени от JNDI средата.

Част 4. Дефиниране на функционалните изисквания към разработката

На база на идентифицираните ограничения наложени от JNDI средата и произтичащите от тях проблеми дефинира функционалните изисквания към разработката.

Част 5. Проектиране на решението

Описва и мотивира методологията избрана за проектиране на разработката – test-driven development, дефинира основните тестови сценарии и тяхната реализация. Представя взетите архитектурни решения по време на работния процес.

Част 6. Реализация

Описва реализацията на системата за регистрация и управление на доставчици на jndi услуги. Дефинира отделните компоненти и взаимодействието между тях.

Част 7. Проверка за качество и коректност на услугата

Описва изпълнените тестови сценарии гарантиращи функционална цялост, коректност и качество на услугата. Коментира получените резултати.

Част 8. Подобрения и разширения

Предлага функционални и оптимизационни подобрения въз основа на резултатите от направените тестове.

Част 9. Ръководство за употреба

Указва начините за използване на разработената система от доставчиците на jndi услуги и техните клиенти, и предпоставките за използване на системата - инициализация. Коментира миграцията на вече съществуващи приложения към новата система, както и съвместимостта и със стандартната функционалност на JNDI средата.

Част 10. Заключение

Оценява постигнатите резултати и актуалността на разработената система. Предлага възможни подобрения и бъдещи насоки.

Част 11. Допълнения

Съдържа кратък речник на използваните термини.

Част 12. Използвана литература

Съдържа списък на използваната литература.

2. Технологичен преглед

Следва обзор на технологиите засегнати в настоящата разработка.

2.1. Java.

Java е език за програмиране, разработен от Sun Microsystems през 1995 година, като основна част от Java-базираната платформа на фирмата. Езикът заимства голяма част от синтаксиса си от C и C++, но има опростен обектен модел и ограничен набор от възможности за работа на ниско ниво. Философията на езика е продиктувана от 5 основни принципа [3]:

- Налагане на обектно-ориентиран стил за програмиране
- Една и съща програма се изпълнява еднакво на различните поддържани платформи
- Вградена поддръжка за работа в мрежа
- Сигурно изпълнение на приложения на отдалечени системи
- Лесна употреба, взимайки основните положителни черти на другите обектно-ориентирани платформи

В едно с езика за програмиране – Sun Microsystems пакетират също и набор от инструменти, изграждащи една цялостна платформа за разработка на приложения. Те включват:

- Основни библиотеки – структури от данни, алгоритми, механизми за сигурност, механизми за интернационализация и локализация на текстове, обработка на XML и т.н.
- Интеграционни библиотеки – слой за връзка с бази данни, JNDI слой (който ще разгледаме в повече детайли в настоящата разработка), механизми за отдалечено изпълнение на приложение, както и възможност за интеграция с други платформи (CORBA).
- Библиотеки за изграждане на потребителски интерфейс.

През 1999 Sun Microsystems разработиха Java 2 Enterprise Edition (J2EE) – платформа за разработка на бизнес приложения, базирана върху езика Java и

платформата около него. В тази среда JNDI служи като свързващо звено – даващо възможност на компоненти от различни модули да си взаимодействат

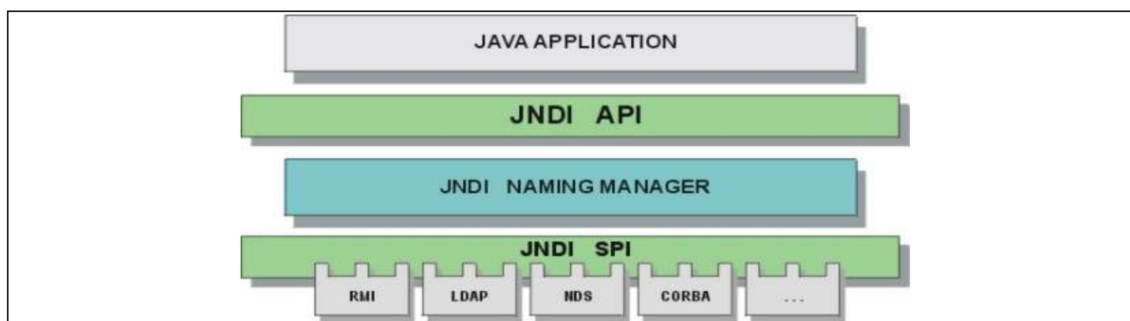
2.2. JNDI

JNDI спецификацията дава възможност на java програми да намират и използват услуги за именуване (naming service) и работа с директории (directory services). Най-общо услугите за именуване се използват за асоцииране на име и местоположение на обект, както и намиране на обект по име. Директорийните услуги са разширение на именуващите услуги, предлагащо в допълнение асоцииране на атрибути и свойства на обекта към името, както и търсене на обект по зададени атрибути [4].

От Java2 SDK, версия 1.3 JNDI библиотеката е включена в стандартната дистрибуция на JDK. Последната версия на спецификацията е 1.2.1.

Освен интерфейси JNDI предоставя и набор от имплементации реализиращи основната функционалност на JNDI средата, а именно намиране и предоставяне на именуващи и директорийни услуги. По дефиниция JNDI библиотеката е независима от имплементациите на услугите за именуване и работа с директории. Тя просто предлага универсален начин за достъп до тези услуги.

Архитектурно JNDI библиотеката е разделена на две части API (Application Provider Interfaces) и SPI (Service Provider Interfaces). Java приложенията използват JNDI API, за да получат достъп до желаната именуваща или директорийна услуга, а доставчиците на тези услуги използват JNDI SPI, за да направят услугите си достъпни за клиенти.



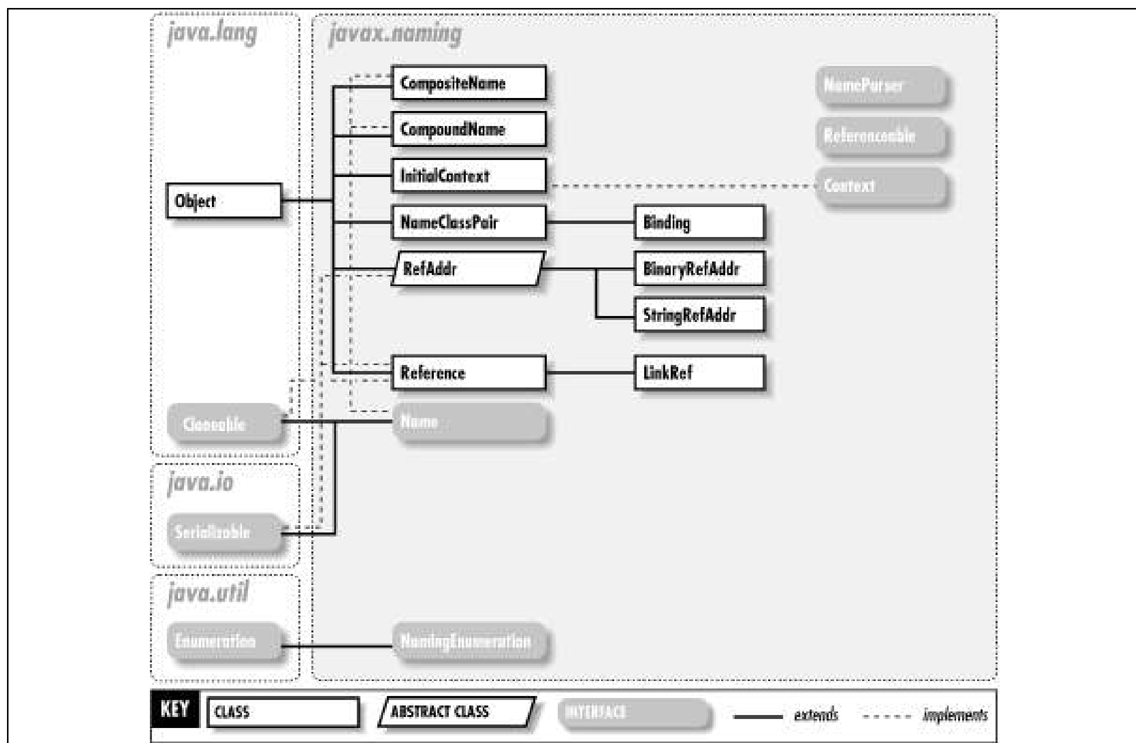
Фиг. 1: *Взаимодействие на java приложения с именуващи услуги посредством JNDI.*

JNDI се отнася както за именуващи, така и за директорийни услуги, но разлика между двата вида услуги по отношение на разглеждания проблем на практика няма. Поради това занапред ще разглеждаме само именуващите услуги с уговорката, че същите налични средства и направени разсъждения са еквивалентни и за директориините услуги.

2.2.1. JNDI API

JNDI API предлага набор от публични интерфейси, намиращи се в пакетите *javax.naming* и *javax.naming.directory*, чрез които java приложенията могат да получат достъп до различни именуващи и директорийни услуги.

Интерфейсите използвани за комуникация между клиентското приложение и именуващата или директорийна услуга са съответно *javax.naming.Context* and *javax.naming.directory.DirContext*.

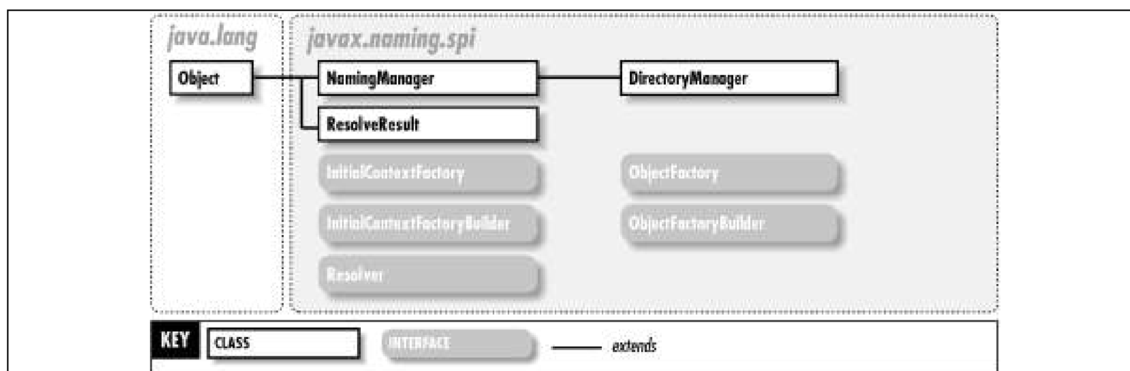


Фиг. 2: *класове и интерфейси в JNDI API*.

2.2.2. JNDI SPI

За да може едно java приложение да използва услуга посредством JNDI API трябва да има доставчик (provider) предоставящ тази услуга съгласно изискванията на JNDI SPI спецификацията.

За да се интегрират с JNDI средата доставчиците на именуващи и директорийни услуги трябва да имплементират някои от интерфейсите намиращи се в *javax.naming.spi* пакета.



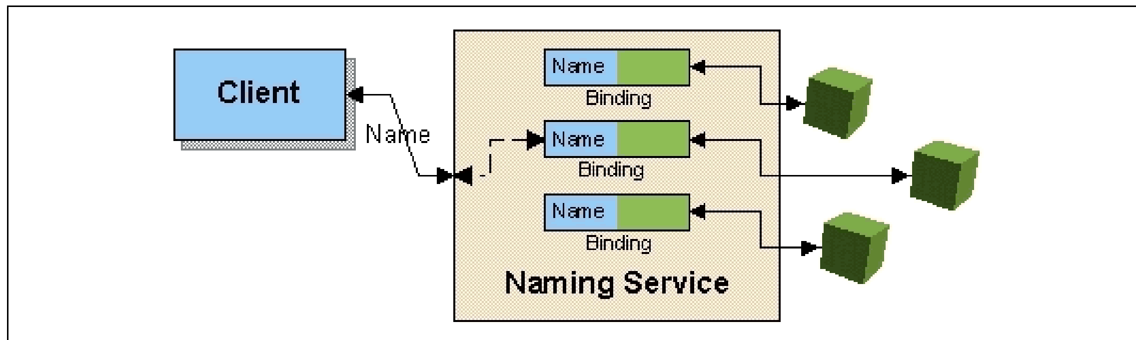
Фиг. 3: *класове и интерфейси в JNDI SPI.*

JNDI библиотеката е включена в стандартната дистрибуция на JDK, така че клиентите и не трябва да правят нищо допълнително, за да получат достъп до нея. В случай, че клиентът иска да използва именуваща или директорийна услуга, той/тя трябва да си осигури достъп до имплементацията на тази услуга.

2.3. Именуваща услуга (naming service).

Именуващата услуга (naming service) най-общо представлява механизъм, чрез който един обект може да се асоциира с дадено име и в последствие да се търси по това име. Всяко име се подчинява на набор синтактични правила наречени конвенция за именуване (naming convention). Всяка именуваща система (naming system) има точно една конвенция, така че имената, които се използват за именуване на обекти в тази система се подчиняват на тази конвенция. Най-малката градивна частица на името според дадена конвенция се нарича атомарно име (atomic name). Асоциацията на атомарно име и обект се нарича “binding”. Когато един обект не може директно да

се постави в именуващата система се използва референция (reference). Референцията е обект, който съдържа необходимата информация да се намери истинския обект, към който тя сочи; този обект най-често не се намира в текущата именуваща система, но може и да е там асоцииран с друго име – в този случай посредством референцията един и същи обект става достъпен под различни имена.



Фиг. 4: *Структура на именуваща услуга.*

Всяко име в именуващата система се интерпретира съгласно даден контекст (jndi context); съответно всяка именуваща операция се изпълнява в даден контекст. „Контекст" наричаме съвкупност от асоциации име-обект, където всички имена се подчиняват на определена именуваща конвенция.

Входна точка за изпълняване на именуващи операции е специален контекст наречен „initial context", т.е. началният контекст на именуващата система, спрямо който се интерпретират имената зададени в операциите. Задача на именуващата услуга е да групира контекстите в дървовидна структура съгласно правилата на именуващата конвенция и по зададено име еднозначно да намери асоциирания към него обект, претърсвайки структурата от контексти.

В езика за програмиране JAVA механизмът за разработка и подържане на именуващи услуги се предоставя посредством JNDI спецификацията. SPI (Service Provider Interface) разделът на JNDI спецификацията предоставя унифициран начин за разработка на приложения предлагащи именуващи услуги (naming services) и начин за публикуване на тези услуги, така че да бъдат достъпни за приложения използващи JNDI API (Application Programming Interface).

2.4. Доставчик на именуващи услуги

Доставчик на именуващи услуги (naming service provider) наричаме модул или съвкупност от модули, които предоставят функционалност удовлетворяваща JNDI API заявки.

Всеки доставчик на именуващи услуги трябва да предостави като минимум имплементация на контекст. Контекст имплементацията трябва да реализира *javax.naming.Context* интерфейса или някой от неговите под-интерфейси като *DirContext, EventContext*, или *LdapContext*.

Контекст имплементацията може да бъде получена от клиентските приложения по различни начини, най-често използваният е при създаване на нов начален контекст (initial context).

2.4.1. Интеграция с JNDI SPI

JNDI дефинира четири типа factory обекти и предоставя SPI методи за използването им. Според функционалността, която адресират можем да разделим factory обектите на две групи – такива, които предоставят контекст имплементации и други, които трансформират обектите участващи в именуващи операции.

Към първия тип спадат initial context factory и url context factory, а към втория - object factory и state factory.

За един доставчик на именуващи услуги най-важно е да предостави имплементация на factory от първия тип, което да създава контекст обекти реализиращи функционалността на услугата. Поддръжката на останалите типове factory обекти е въпрос на избор на самия доставчик на услугата; JNDI спецификацията не регламентира, колко и какъв тип factory обекти трябва да се предоставят от доставчик на именуващи услуги.

2.4.2. Factory типове предоставящи контекст имплементации

- *InitialContextFactory* – factory, което създава инстанция на начален контекст.

```
public interface InitialContextFactory {

public Context getInitialContext(Hashtable env) throws NamingException;

}
```

Използва се от конструктора на *InitialContext* класа, като по зададени от клиента конфигурации (properties) на контекст средата (environment) трябва да създаде инстанция от тип *Context*.

Конфигурацията на JNDI средата "*java.naming.factory.initial*" съдържа името на класа на InitialContextFactory имплементацията, която клиентът желае да използва. Различните доставчици на именуващи услуги предоставят различни *InitialContextFactory* имплементации и документират предлаганата функционалност и начините за използване на конкретната именуваща услуга.

Нещата, които клиентът на именуващата услуга трябва да знае като минимум, за да използва услугата са: името на класа на initial context factory имплементацията, адрес и порт за връзка с доставчика на услугата и евентуално клиентско име и парола за удостоверяване самоличността на клиента и правата, които притежава за работа със системата за именуване. Всички тези характеристики се задават като конфигурации на контекстната среда (context environment properties) и се използват при създаване на начален контекст.

```
Hashtable env = new Hashtable();
env.put(Context.INITIAL_CONTEXT_FACTORY,
"thesis.jndi.provider.initial.InitialContextFactoryImpl");
env.put(Context.PROVIDER_URL, "localhost:50004");
env.put(Context.SECURITY_PRINCIPAL, "Administrator");
env.put(Context.SECURITY_CREDENTIALS, "password");
 try {
    InitialContext ctx = new InitialContext(env);
 } catch (NamingException e) {
     e.printStackTrace();
 }
```

- *URLContextFactory* – factory, което създава инстанция на контекст, но не в зависимост от подадената клиентска конфигурация, а в зависимост от името използвано в именуващата операция.
  Ако операцията изпълнена върху *InitialContext* класа получи като параметър име с URL нотация, например *jdbc:local/connections*, ще се задейства механизъм за намиране и създаване на URL context factory по следния алгоритъм:

Конфигурацията на контекстната среда (context environment) *java.naming.factory.url.pkgs* съдържа списък от префикси на пакети разделени със символ „**:**”. Името на класа на *URLContextFactory* имплементацията, която трябва да създаде подходящ контекст за изпълнение на JNDI операцията се конструира по следния начин:

<**префикс**> + “**.**” + <**url_схема**> + “**.**” + <**url_схема**>**URLContextFactory**

Където:

- <**префикс**> е всеки от зададените пакетни префикси с конфигурацията *java.naming.factory.url.pkgs*. В края на списъка се добавя и префикса по подразбиране - *com.sun.jndi.url*.
- <**url_схема**> е знаковият низ от началото на името до знака „**:**”, ако съществува такъв. В примера *jdbc:local/connections* url схемата в името е „*jdbc*”.

Така че, ако имаме изпълнение на следния java фрагмент:

```
Hashtable env = new Hashtable();

env.put("java.naming.factory.url.pkgs", "thesis.jndi.provider.url");

(new InitialContext(env)).lookup("jdbc:local/connections");
```

URL context factory имплементацията, която ще се опита да инициализира JNDI средата ще бъде: *thesis.jndi.provider.url.jdbc.jdbcURLContextFactory* или *com.sun.jndi.url.jdbc.jdbcURLContextFactory* (в този ред).

Като тип URL context factory имплементацията реализира *ObjectFactory* интерфейса съобразявайки се с особеностите на предназначението си, т.е.

типът на върнатия обект в резултат от извикването на метода на *ObjectFactory* имплементацията задължително трябва да е *Context*.

```
public interface ObjectFactory {

    public Object getObjectInstance(Object obj, Name name, Context ctx,
Hashtable env)

}
```

2.5. Семантичен обект

След като клиентът се е сдобил с контекст инстанция, той/тя може да я използва за извършване на jndi операции. Най-популярните jndi операции са асоцииране на обект с име (bind) и търсене на обект по име (lookup). По спецификация при търсене на обект по име доставчикът на услугата е длъжен да върне на клиента копие на обекта, който е асоцииран с името, а не самия обект. Така че типът на обектите участващи в тези операции не е маловажен. Различните доставчици на именуващи услуги имат различна обработка за различните типове обекти. Някои доставчици например преобразуват всеки обект до масив от байтове и така го съхраняват в контекст имплементацията, други обвиват обекта в *CORBAObejct* или друг java тип за отдалечен достъп, трети записват референция към обекта, вместо да пазят самия обект.

За някои типове обекти е удобно да се запише само някаква информация за тях, а не самия обект. Например информация за отдалечен достъп до обекта, или локалната му конфигурация. Тази информация се използва при lookup операция за създаване на нова инстанция на търсения обект. Такива обекти, които притежават определена семантика и именно тя се използва при записване и прочитане на обекта в именуващата система ще наричаме семантични обекти.

2.6. Доставчик на семантични обекти

Доставчик на семантични обекти наричаме модул или съвкупност от модули предоставящи една или повече имплементации на семантичен обект, заедно с механизъм за обработката му при записване и прочитане от система за именуване. Разбира се каквито и трансформации да прави доставчикът на семантични обекти с обектите участващи в jndi операции те трябва да останат скрити за клиента.

2.6.1. Интеграция с JNDI SPI

JNDI спецификацията подпомага доставчиците на семантични обекти по два начина:

- Предоставя механизъм, чрез който един доставчик на семантични обекти има възможност да направи модификация или изцяло да подмени обекта преди той да бъде съхранен в системата за именуване, както и да направи съответните промени по обекта съхранен в jndi системата преди той да бъде върнат на клиента.

- Предоставя удобна имплементация на абстракцията референция към обект - *javax.naming.Reference*, която е интегрирана с гореспоменатия механизъм.

За да се възползва от механизма за трансформация на обекти при jndi операции доставчикът на семантични обекти трябва да предостави имплементации на *StateFactory* и *ObjectFactory* интерфейсите.

2.6.2. Factory типове за писане и четене на обекти

- *StateFactory* – factory, което има възможност да промени обекта подаден като параметър при JNDI операция преди записването му в системата за именуване.

```
public interface StateFactory {

    public Object getStateToBind(Object obj, Name name, Context ctx,
Hashtable env)

}
```

По спецификация всички контекст имплементации преди за запишат обекта в системата за именуване са длъжни да извикат.

```
NamingManager.getStateToBind(Object obj, Name name, Context ctx,
Hashtable env);
```

Конфигурацията на JNDI средата "*java.naming.factory.state*" съдържа списък от имена на класове на *StateFactory* имплементации разделени със символ „**:**". *NamingManager* имплементацията, ще създаде инстанция на всяко factory и ще извика съответния му *getStateToBind(Object obj, Name name, Context ctx, Hashtable env)* метод. Всяка factory имплементация е длъжна да върне *null*, в случай че не може да обработи обекта. Така се обхождат всички factory имплементации записани в списъка докато не се стигне до резултат различен от *null* или прихващане на изключение (exception).

- *ObjectFactory* – factory, което има възможност да промени обекта върнат като резултат при JNDI операция преди да е стигнал до клиента.

```
public interface ObjectFactory {

      public Object getObjectInstance(Object obj, Name name, Context ctx,
Hashtable env)

}
```

По спецификация всички контекст имплементации преди да върнат обекта към клиента изпълняващ операция търсене по име са длъжни да извикат:

```
NamingManager.getObjectInstance(Object obj, Name name, Context ctx,
Hashtable env);
```

Конфигурацията на JNDI средата "*java.naming.factory.object*" съдържа списък от имена на класове на *ObjectFactory* имплементации разделени с „**:**". *NamingManager* имплементацията, ще създаде инстанция на всяко factory и ще извика съответния му *getObjectInstance(Object obj, Name name, Context ctx, Hashtable env)* метод. Всяка factory имплементация е длъжна да върне *null*, в случай че не може да обработи обекта. Така се обхождат всички factory

имплементации записани в списъка докато не се стигне до резултат различен от *null* или прихващане на изключение (exception).

Object и state factory концепцията е малко встрани от услугите за именуване или по-точно е до голяма степен независима от тях. Услугите за именуване са специфицирани от JNDI, така че различните имплементации от гледна точка на клиента трябва да притежават една и съща функционалност.

Това обаче не е валидно за обектите, които се съхраняват в jndi системата. Даден тип обект може да работи без проблеми с един доставчик на именуващи услуги, но да води до изключение при работа с друг.

За да се избегне това поведение е удобно за конкретен тип обект да се създадат state и object factory имплементации, които да трансформират обекта до вид разбираем за jndi системата - например до *Properties* обект съдържащ само конфигурациите на реалния обект, които по-късно при поискване на обекта ще послужат за създаването и конфигурирането на обект копие на оригиналния.

Модулите, които предоставят само factory имплементации за трансформация на определен тип обекти и не предоставят контекст имплементация, т.е. за тях не можем да говорим като за доставчици на именуващи услуги наричаме доставчици на семантични обекти. Семантичните обекти са обекти, с чиято семантика са запознати само object и state factory имплементациите, които ги съпровождат.

Важна характеристика на доставчиците на семантични обекти е, че могат да работят независимо един от друг и независимо от доставчиците на услуги за именуване.

2.7. Механизъм за намиране и създаване на factory инстанции в JNDI

Всички factory типове, които разгледахме се задават като конфигурации на контекстната среда при създаване на *InitialContext* инстанция. Всяка конфигурация съдържа пълното име на класа на имплементацията за конкретния factory тип (изключение прави единствено url context factory типът, чието име не е зададено директно, а се конструира в зависимост от url схемата, която удовлетворява). Всеки

път, когато JNDI средата (framework) трябва да намери и създаде инстанция от даден factory тип се опитва да зареди класа на factory имплементацията с context classloader инстанцията, намираща се в нишката, която изпълнява jndi операцията. В общия случай това е classloader–ът на компонент, който изпълнява jndi операцията. От тук идва и ограничението поставено от JNDI средата – всички factory имплементации, които даден клиент ще използва в изпълняваните от него jndi операции да са достъпни за classloader-а му.

2.8. Сървър за Java EE (Enterprise Edition) приложения

Най-широка употреба JNDI функционалността намира в сървърите за Java EE (Enterprise Edition) приложения. Според Java EE спецификацията приложенията използват JNDI услуги, за да намират необходимите им dbpool, connector, jms, ejb, и други ресурси. Приложенията, които работят на един Java EE сървър се поддържат от контейнер (container) компоненти, които им предоставят подходяща среда за работа и достъп до необходимите им ресурси. От своя страна контейнерите използват услугите на по базов слой компоненти за създаване на classloader-и, стартиране на java нишки, комуникация в разпределена сървър система и други. Базовите компоненти могат да използват различни библиотеки – за създаване на записи на файловата система или обработка на изключения.

За да гарантират изолация между отделните семантични слоеве от компоненти, а и между отделните компоненти в един слой съвременните сървър системи използват отделен classloader за всеки компонент, т.е. всяко едно приложение има собствен classloader, който може да зарежда класове само от клас ресурсите (code base) на приложението или от родителския (parent) си classloader. Ако един компонент иска да използва класове, които се намират в клас ресурсите на друг компонент, classloader-ът на първия компонент трябва да има референция към classloader-а на втория. Можем да усложним още малко йерархията на classloader-ите в сървър системата като наложим условие classloader на компонент от един слой да има референция само и единствено към публичната част от клас ресурсите на компонент от по-базов слой и то само при компоненти от два съседни слоя [5].

2.8.1. Проблеми при интеграцията с JNDI

На една сървър система може да има хиляди работещи приложения, които също да са организирани в йерархия - едни предлагат услуги за други и ги публикуват посредством JNDI средата. Съвсем нормално е на една такава сървър система да има няколко доставчика за именуващи услуги – разграничени според семантичния слой, в който се намират или според характеристиките на услугата, която предлагат (jndi услуга в разпределена система, услуга, която съхранява обектите в паметта или пък в релационна база); както и множество доставчици на семантични обекти (едно приложение предоставя услуга на друго като асоциира обект имплементиращ услугата с уникално име, така че всеки, който иска да се възползва от тази услуга за в бъдеще може да я потърси по име).

Точно тук идва и проблемът с ограничението наложено от JNDI спецификацията класовете на всички необходими factory имплементации да са достъпни за classloader-a на клиента, който изпълнява jndi операцията. Вероятно това ограничение е било поносимо преди 8 години, когато е била последната редакция на JNDI спецификацията и когато в сървърите за приложения е имало само един доставчик на именуващи услуги, който евентуално е предлагал поддръжка на стандартните референции в jndi.

С развитието на сървърите за приложения обаче JNDI услугите се усложняват и разнообразяват; разработчиците на приложения стават все по изобретателни в начините за кооперация с JNDI средата и с други приложения; въвежда се изолация между отделните компоненти на ниво classloader. Като резултат спазването на ограничението за зареждане на jndi factory имплементации от classloader-a на клиента става невъзможно.

Как се справят сървърите за приложения със ситуацията?

2.8.2. Частични решения

- Нарушават изолацията на classloader-ите на компонентите.
- Имплементират своя среда подобна на JNDI, която има различен механизъм за откриване и зареждане на factory имплементации, като по този начин губят универсалността на достъпа до именуващи услуги. Клиентите на техните

услуги не могат да ги използват посредством JNDI API, а трябва да използват тяхно специфично API.

- Забраняват използването на доставчици на семантични обекти извън приложението, с което се разпространяват.

Целта на настоящата разработка е да предложи алтернативен начин за намиране и зареждане на jndi factory имплементации.

Идентифицираме следните задачи:

- Механизмът за създаване на factory инстанции да не зависи от classloader-а на клиента.
- Предложеното решение да е напълно съвместимо с JNDI средата, т.е. клиентите да продължат да използват JNDI API за достъп до именуващи услуги.

3. Функционален анализ на проблемната област

Първо ще разгледаме подробно механизмът използван по подразбиране от JNDI средата за откриване и зареждане на factory имплементации, и сценариите, в които се използва. Ще анализираме начините да се припокрие стандартната функционалност на JNDI средата предоставени според SPI спецификацията. Ще предложим алтернативен механизъм за откриване и зареждане на factory имплементации на базата на направения анализ.

3.1. Стандартна интеграция на factory имплементации в JNDI средата

Ще разгледаме механизмите за интеграция на factory имплементации в JNDI средата, предоставени от SPI спецификацията.

3.1.1.NamingManager

Най-важният клас в JNDI SPI е *NamingManager*. Той е отговорен за създаването на контекст инстанции и инстанции на обекти записани като референции, както и за намирането на factory имплементации.

NamingManager класът е свързващото звено между JNDI средата и доставчиците на именуващи услуги и семантични обекти. Не може да се инстанциира директно. Има само една активна инстанция на NamingManager имплементацията в java виртуалната машина, която се достъпва посредством статичните методи на класа. На практика всяка заявка за създаване на начален контекст, асоцииране на обект с име и търсене на обект по име минава през *NamingManager* имплементацията.

Ще разгледаме пътя на заявката при всяка от тези операции.

3.1.2. Създаване на нов начален контекст

Клиентът използва един от конструкторите на *InitialContext* класа, за да получи достъп до съответната именуваща система посредством контекст имплементацията на доставчика на услугата. Клиентът е длъжен да зададе подходящи конфигурации на контекстната среда и чрез тях да укаже името на *InitialContextFactory* имплементацията, която да бъде използвана за създаване на контекст инстанция и информация за връзка с доставчика на jndi услугата (като адрес, порт и протокол за връзка). *InitialContext* класа извиква статичен метод на *NamingManager* имплементацията

```
NamingManager.getInitialContext(environment);
```

като в environment параметъра подава събраните конфигурации за контекст средата от различните възможни местонахождения (*Hashtable* в конструктора на *InitialContext*, *jndi.properties* файл, системен параметър на java процеса или конфигурация на applet, ако клиентът е applet приложение). Методът *getInitialContext* извлича конфигурацията за име на клас на *InitialContextFactory* имплементацията и се опитва да зареди инстанция от нея с контекстния classloader на нишката, в която се изпълнява операцията. Липсата на конфигурация за *InitialContextFactory*, както и неуспехът при зареждане на имплементацията водят до изключение (*NamingException*).

След успешното създаване на *InitialContext* инстанция клиентът може да премине към изпълнение на желаната jndi операция. Почти всички jndi операции приемат като параметър името на обекта или контекста, за който се отнасят. Това име може да е зададено с URL нотация, в който случай операцията трябва да бъде изпълнена от подходяща URL контекст имплементация. Задачата да намери подходящия контекст се пада на метода на *InitialContext* класа,

```
Context getURLOrDefaultInitCtx(String name),
```

който се извиква при всяка jndi операция извършена върху *InitialContext* инстанция. При извикване на този метод се проверява дали името участващо в операцията започва с URL схема. Ако е така се извиква,

```
NamingManager.getURLContext(scheme, environment),
```

чиято задача е да извлече префиксите на пакети зададени с конфигурацията *java.naming.factory.url.pkgs* и за всеки от тях да конструира име на клас по познатия вече шаблон:

<**префикс**> + "**.**" + <**url_схема**> + "**.**" + <**url_схема**>**URLContextFactory**

Където:

<**префикс**> е всеки от зададените пакетни префикси с конфигурацията *java.naming.factory.url.pkgs*. В края на списъка се добавя и префикса по подразбиране - *com.sun.jndi.url*.
<**url_схема**> е знаковият низ от началото на името до знака „**:**", ако съществува такъв.

NamingManager имплементацията прави опит последователно да зареди така получените имена на класове с контекстния classloader на нишката, в която се изпълнява операцията. Ако успее да създаде инстанция на *url context factory* тя ще бъде използвана за получаване на URL контекст, в противен случай ще се създаде инстанция на *InitialContextFactory* по описания вече начин. Така създаденият контекст се записва като поле на *InitialContext* инстанцията и се използва директно при следващи jndi операции.

NamingManager класът предлага начин за пренаписване на механизмът за намиране и зареждане на *InitialContextFactory* и *URLContextFactory* посредством регистрация на *InitialContextFactoryBuilder*.

```
NamingManager.setInitialContextFactoryBuilder(builder);
```

Методът *setInitialContextFactoryBuilder()* предоставя възможност за регистрирането на една единствена имплементация на интерфейса *InitialContextFactoryBuilder*. Единствено първото извикване на този метод води до успешна регистрация на *InitialContextFactoryBuilder*; всяко следващо води до изключение.

InitialContextFactoryBuilder интерфейсът има единствен метод,

```
public interface InitialContextFactoryBuilder {
```

```
public InitialContextFactory createInitialContextFactory(Hashtable environment);

}
```

в който доставчикът на jndi услуги може да пренапише механизмът на *NamingManager.getInitialContext()* и *NamingManager.getURLContext().*

Ако се регистрира *InitialContextFactoryBuilder* имплементация, всички заявки за намиране и зареждане на контекст factory имплементации, независимо от кой доставчик на jndi услуги са предоставени, се делегират към тази builder инстанция. Никой от стандартните механизми на JNDI средата за намиране на контекст и поддръжка на URL схеми няма да се изпълни в този случай.

Ключовият момент тук е, че само един единствен доставчик на именуващи услуги може да регистрира своя имплементация на *InitialContextFactoryBuilder*. Това може да е „най-главният" доставчик на услугата или просто първият, който успее да извика *setInitialContextFactoryBuilder()* метода. Това е още едно ограничение, с което текущата разработка трябва да се справи.

3.1.3. Именуване на обект

Върху така получената вече контекст имплементация може да се изпълни операция за асоцииране на обект с дадено име (bind или rebind). Доставчикът на услугата е задължен според JNDI SPI спецификацията да извика статичен метод на *NamingManager* класа:

```
Object getStateToBind(Object obj, Name name, Context nameCtx, Hashtable environment)
```

преди да се обработи/запише обекта получен като параметър от клиента. *NamingManager* имплементацията извлича от контекст средата конфигурацията "*java.naming.factory.state*", която съдържа списък от имена на класове на *StateFactory* имплементации разделени със символ „**:**". Опитва се да зареди *StateFactory* класовете, използвайки контекстния classloader на текущата нишка. Създава инстанция на всеки factory клас, който е успяла да зареди и извиква съответния му *getStateToBind(Object obj, Name name, Context nameCtx, Hashtable environment)* метод.

Всяка factory имплементация е длъжна да върне *null*, в случай че не може да обработи обекта. Така се обхождат всички factory имплементации записани в списъка докато не се стигне до резултат различен от *null* или прихващане на изключение (exception). В крайна сметка клиентският обект или се трансформира в следствие на намесата на подходяща state factory имплементация или остава непроменен.

NamingManager класът не предлага начин за пренаписване на механизмът за намиране и зареждане на *StateFactory* имплементации.

3.1.4. Търсене на обект по име

Операцията за търсенето на обект по име се нарича lookup и се изпълнява върху контекст инстанция предоставена от доставчик на jndi услуги. Доставчикът на услугата е задължен според JNDI SPI спецификацията да извика статичен метод на *NamingManager* класа:

```
Object getObjectInstance(Object refInfo, Name name, Context nameCtx, Hashtable environment)
```

преди да върне на клиента обекта получен като резултат от операцията. *NamingManager* имплементацията извлича от контекст средата конфигурацията "*java.naming.factory.object*", която съдържа списък от имена на класове на *ObjectFactory* имплементации разделени със символ „**:**". Опитва се да зареди *ObjectFactory* класовете, използвайки контекстния classloader на текущата нишка. Създава инстанция на всеки factory клас, който е успяла да зареди и извиква съответния му *getObjectInstance(Object refInfo, Name name, Context nameCtx, Hashtable environment)* метод.

Всяка factory имплементация е длъжна да върне *null*, в случай че не може да обработи обекта. Така се обхождат всички factory имплементации записани в списъка докато не се стигне до резултат различен от *null* или прихващане на изключение (exception). В крайна сметка обектът или се трансформира в следствие на намесата на подходяща object factory имплементация или се връща на клиента непроменен.

NamingManager класът предлага начин за пренаписване на механизма за намиране и зареждане на *ObjectFactory* имплементации посредством регистрация на *ObjectFactoryBuilder*.

```
NamingManager. setObjectFactoryBuilder(builder);
```

Методът *setObjectFactoryBuilder()* предоставя възможност за регистрирането на една единствена имплементация на интерфейса *ObjectFactoryBuilder*. Единствено първото извикване на този метод води до успешна регистрация на *ObjectFactoryBuilder*; всяко следващо води до изключение.

ObjectFactoryBuilder интерфейсът има единствен метод,

```
public interface ObjectFactoryBuilder {

    public ObjectFactory createObjectFactory(Object obj,Hashtable environment);

}
```

в който доставчикът на jndi услуги може да пренапише механизмът на *NamingManager.getObjectInstance()*.

Тук отново имаме ограничението, че само един единствен доставчик на семантични обекти може да регистрира своя имплементация на *ObjectFactoryBuilder*.

3.2. Налични ограничения в JNDI SPI

След като разгледахме пътя на основните jndi операции, ролята на NamingManager класа в тях и средствата предложени от JNDI SPI спецификацията за пренаписване на механизма за намиране и зареждане на factory имплементации можем да направим следната равносметка:

1. JNDI SPI предоставя възможност на точно един доставчик на именуващи услуги да използва собствен механизъм за намиране и зареждане на factory имплементации създаващи контексти.

2. JNDI SPI предоставя възможност на точно един доставчик на семантични обекти да използва собствен механизъм за намиране и зареждане на *ObjectFactory* имплементации.

3. JNDI SPI не предоставя възможност на доставчиците на семантични обекти да използват собствен механизъм за намиране и зареждане на *StateFactory* имплементации.

След като направихме равносметка на наличните средства за въздействие върху JNDI средата можем да формулираме напълно задачата, която си поставя настоящата разработка.

4. Дефиниране на функционалните изисквания към разработката

Настоящата дипломна работа цели да разработи система за регистрация и управление на доставчици на jndi услуги като модул надстройка на JNDI SPI, който да притежава следните характеристики:

- Да предостави начин за намиране и зареждане на factory имплементации независещ от използването на клиентския classloader, т.е. да освободи доставчиците на именуващи услуги и семантични обекти от ограничението – техните factory имплементации да са достъпни за клиентския classloader. Това решение би съвместило JNDI услугите и изолацията на компонентите в сървърите за приложения.

- Да предостави възможност на всеки доставчик на именуващи услуги и/или семантични обекти сам да избере механизмът за намиране и зареждане на factory имплементациите, които той предлага. По този начин ще се снеме ограничението, че само първият доставчик, който успее да регистрира factory builder в *NamingManager* класа има привилегията да пренапише този механизъм.

- Да предостави възможност на доставчиците на семантични обекти да използват собствен механизъм за намиране и зареждане на *StateFatory* имплементации. По този начин се компенсира липсата на метод в JNDI SPI за пренаписване на механизма за откриване и зареждане на *StateFatory* имплементации.

- Да се предостави възможност на клиентите на именуващи услуги да използват JNDI API, за връзка с доставчиците на тези услуги, т.е. разработваният модул трябва да е съвместим с JNDI SPI и достъпен посредством JNDI API.

5. Проектиране на решението

5.1. Изисквания за качество и съвместимост

- Разработваният модул трябва да е съвместим с JNDI SPI и достъпен посредством JNDI API

- Новата функционалност, която предоставя разработката, трябва да е напълно съвместима с наличната вече функционалност предоставена от JNDI средата.

- Качеството на услугата, която получава jndi доставчик не използващ функционалността на нашия модул трябва да е едно и също без значение дали системата ни за регистрация и управление на jndi доставчици е активна или не.

Така дефинираните изисквания за качество и съвместимост логично ни водят към процес на разработка, базиран на постоянно еволюиращи тестове (Test-driven development process).

5.2. Test-driven development process

Тестването е една основна част от процеса на разработка на всеки един продукт. „Test-driven development" е методология, която изтъква създаването на тестове като съществена и неразделна част от процеса на разработката на софтуер [6]. Тук не става дума обаче да се напишат тестове, които да проверят функционалност или качество на вече написан java код. Същественото в „Test-driven development" методологията е, че тестовете покриващи желаната функционалност се пишат преди да се започне имплементация за промяна на вече съществуваща или напълно нова функционалност. Така изпълнението на тестовете по време на процеса на разработка дава много бърза обратна връзка за текущото положение на разработвания модул.

„Test-driven development" процеса предполага създаването на автоматизирани тестове дефиниращи изискванията към разработвания модул преди да се започне реалната разработка на модула.

5.2.1. Автоматизирани тестове

Веднъж написани и автоматизирани, тестовете се изпълняват регулярно или по желание на разработчика, когато трябва да се гарантира цялост на функционалността след поправка на грешка в кода. Грануларността на тестовете може да бъде различна. Един пример е функционалното тестване, тип черна кутия (black box), където тестващия няма никаква идея за имплементацията на дадената система или модул; примерно тестване на публичните интерфейси на софтуерна библиотека. Тестовете могат да бъдат също така и на ниво клас (unit testing), когато целта е да се провери самата имплементация на даден клас или метод.

Тестовете трябва да се развиват непрекъснато, паралелно със софтуерната разработка [7]. Добавянето на нова функционалност или поправянето на грешка в кода на разработката трябва да се предхожда от добавяне на нов тестов сценарий.

Най-хубавото на автоматизираните тестове е, че не позволяват деградация в постигнатото качество и функционална завършеност на модула – възникналите проблеми стават видими незабавно и могат бързо да се отстранят.

Според изследванията на Ben Rometsch хващането на голяма част от дефектите още по време на разработката, подобрява многократно производителността на екипа от разработчици [8].

Поставен в различна среда от гледна точка на операционна система, хардуер, мрежова конфигурация и т.н. разработения софтуер може да не функционира според очакванията на клиента. Изпълнението на тестовете в този случай ще помогне за откриването на проблема и бързото му отстраняване.

Друг плюс на автоматизираните тестове е, че резултатите от тях могат да свидетелстват за нивото на качество и функционална цялост на даден софтуер пред крайните му потребители.

5.2.2. JUnit

Съвсем логично е тестовете за софтуер, който се разработва на Java да бъдат написани също на Java. По този начин тестовете стават допълнение към кода, който тестват, като при това осигуряват цялостност на интерфейсите му на ниво компилация

JUnit [9] е среда за автоматизирано тестване на Java приложения създадена от Erich Gamma и Kent Beck. Тя позволява дефинирането на тестове (unit tests) за градивните частици на даден продукт – класове и методи. Тестовете представляват Java програми, имащи за цел да проверят дали даден код функционира според очакванията.

JUnit тестовете могат да се организират в тестови групи включващи тестови случаи (test cases) и други тестови групи. Това позволява изпълнението само на определено ниво от тестовата йерархия, единичен тест или цяла тестова група.

JUnit се е утвърдил като стандартна среда за разработка и изпълнение на тестове (unit tests). Сравнително лесен е за използване и предоставя изключителна гъвкавост. В интернет пространството се предлагат множество разширения и допълнения (extensions) към JUnit.

Повечето среди за разработка имат добра интеграция с JUnit, което улеснява писането на тестове и визуализира резултатите веднага след изпълнението им. Eclipse средата, която сме избрали за настоящата разработка също има стандартна поддръжка за писане и изпълняване на JUnit тестове.

5.3. Дефиниране на тестовите сценарии

Ще дефинираме тестови сценарии за следните аспекти:

5.3.1. Функционална цялост

Тази група тестове има за цел да удостовери пълнота по отношение на функционалността, която сме заложили в разработката си. Най-добре е да тръгнем от моделиране на проблемния сценарий. В конкретния случай това е клиентско приложение, което не успява да създаде нов начален контекст, тъй като factory имплементацията предоставяща услугата не е достъпна за classloader-а на клиента.

За целта дефинираме модул доставчик на jndi услуги, който предоставя имплементация на контекст с модел на съхраняване на данните в паметта. Класовете, които включва този модул са:

test.thesis.jndi.provider.context.InMemoryContext - имплементира *javax.naming.Context* и моделира йерархична именуваща структура, използвайки хеш таблица за съхранение асоциациите име - > обект.

test.thesis.jndi.provider.context.InitialInMemoryContextFactory - имплементация на *javax.naming.spi.InitialContextFactory*, служи за създаване на начален контекст от тип *InMemoryContext*.

test.thesis.jndi.provider.context.ServiceSupportContext - *javax.naming.Context* имплементация предоставяща поддръжка на url схема "service: "; използва вътрешно *InMemoryContext* имплементацията за съхранение на данните.

test.thesis.jndi.provider.context.service.serviceURLContextFactory - имплементация на *javax.naming.spi.ObjectFactory*; създава *ServiceSupportContext* инстанция, ако името зададено в jndi операцията започва със "service: " схема.

Дефинираме поддръжка и за следните семантични обекти:

test.thesis.jndi.provider.semantics.book.Book - клас тип представящ обект, книга и нейните характеристики (автор, име, година на издаване, жанр); не подлежи на сериализация; за да се запише в именуващата система трябва се използва прилежащата му *StateFactory* имплементация; съответно при четене се използва *ObjectFactory* имплементация за създаване на обекта.

test.thesis.jndi.provider.semantics.book.BookStateFactory - имплементация на *javax.naming.spi.StateFactory*; извлича характеристиките на *Book* инстанцията и ги предава на именуващата система за съхранение.

test.thesis.jndi.provider.semantics.book.BookObjectFactory - имплементация на *javax.naming.spi.ObjectFactory*; по зададени характеристики на книгата създава обект *Book*.

test.thesis.jndi.provider.semantics.song.Song - клас тип представящ обект, песен и нейните характеристики (автор, име, година на издаване, жанр); не подлежи на сериализация; за да се запише в именуващата система трябва се използва прилежащата му *StateFactory* имплементация; съответно при четене се използва *ObjectFactory* имплементация за създаване на обекта.

test.thesis.jndi.provider.semantics.song.SongStateFactory - имплементация на *javax.naming.spi.StateFactory*; извлича характеристиките на *Song* инстанцията и ги предава на именуващата система за съхранение.

test.thesis.jndi.provider.semantics.song.SongObjectFactory - имплементация на *javax.naming.spi.ObjectFactory*; по зададени характеристики на книгата създава обект *Song*.

Необходими са ни два семантични обекта, за да можем да използваме единия за работа с *InMemoryContext*, а другия със *ServiceSupportContext*.

Дефинираме изолация на различните модули на classloader ниво по следния начин:

- Модул: Клиент – пакетира в себе си теста изпълняващ операция за нов начален контекст и последваща я lookup операция. Classloader-ът на този модул няма достъп до класовете на по-горе дефинираните factory имплементации.

- Модул: Доставчик на jndi услуги – пакетира в себе си контекст имплементациите и семантичните обекти заедно с прилежащите им factory имплементации.

Classloader-ите на тези два модула са напълно независими и нямат референции помежду си.

Клиент модулът е реализиран като JUnit *TestCase* - *test.thesis.jndi.provider.client.JNDITest*. Изпълнението му води до изключение, поради невъзможността на стандартната функционалност на JNDI средата да намери и зареди имплементацията на *InitialInMemoryContextFactory* и *serviceURLContextFactory* класовете.

След активиране на системата ни за регистрация и управление на доставчици на jndi услуги изпълнението на този тест трябва да премине успешно.

5.3.2. Съвместимост със стандартната функционалност на JNDI средата

Дефинираме тестови сценарии, за да проверим коректността на изпълнение на операциите за намиране на factory имплементации. В крайна сметка това, което променя нашата разработка е само механизма за намиране на factory имплементации, а резултатът от операцията трябва да е същият, какъвто е предвиден според спецификацията на операцията.

- Търсене на *InitialContextFactory*, при зададено име на класа на имплементацията като конфигурация на контекстната среда.
    - Ако можем да го намерим – > връщаме инстанция
    - В противен случай – > предизвикваме изключение
- Търсене на *UrlContextFactory*, при зададена схема в името подадено на jndi операцията и зададени конфигурации на контекстната среда за *InitialContextFactory* клас и префикси за url контекст пакети.
    - Ако можем да го намерим – > връщаме инстанция

    - В противен случай – > опитваме да намерим *InitialContextFactory* по име
        - Ако можем да го намерим – > връщаме инстанция
        - В противен случай – > предизвикваме изключение

- Търсене на *ObjectFactory*, при зададено име на класа на имплементацията като конфигурация на контекстната среда.
    - Ако можем да го намерим – > връщаме инстанция
    - В противен случай – > връщаме обекта непроменен
- Търсене на *StateFactory*, при зададено име на класа на имплементацията като конфигурация на контекстната среда.
    - Ако можем да го намерим – > връщаме инстанция
    - В противен случай – > връщаме обекта непроменен

Тези тестови сценарии са реализирани като JUnit *TestCase* – *test.thesis.jndi.correctness.FunctionalityTest*. Минават успешно със стандартната имплементация на JNDI средата и трябва да продължат да минават успешно след активиране на системата ни за регистрация и управление на доставчици на jndi услуги като надстройка на JNDI средата.

5.3.3. Оптималност и бързодействие

Дефинираме тестови сценарии, които ще ни помогнат да измерим ефективността на изпълнение на основните jndi операциите преди и след активиране на системата ни за регистрация и управление на доставчици на jndi услуги.

Какъв е смисълът да предлагаме решение, ако то колкото и да е функционално не е конкурентно по отношение на бързодействие и ефективност?

Целта ни ще бъде да разработим системата така, че ефективността на изпълнение на jndi операции да е еквивалентна или по-добра от ефективността постигната от стандартната имплементация на JNDI средата.

Разбира се трябва да бъдем и справедливи в критериите си, тъй като за разлика от стандартната имплементация на JNDI средата, която използва директно зареждане на factory класа с classloader-а на нишката, нашата система предоставя възможност на всеки доставчик да използва свой собствен механизъм. Така някой доставчик може да реши да използва четене от база данни в механизма си за намиране на factory имплементация, което несъмнено е по-бавно от стандартното зареждане на класа и създаване на негова инстанция.

Описание на тестовите сценарии:

- 1000 изпълнения на *new InitialContext()* + *unbind*(име) операция, с участие на *InitialContextFactory*
- 1000 изпълнения на *new InitialContext()* + *unbind*(схема) операция, с участие на URL context factory
- 1000 изпълнения на *bind()* операция с участие на *StateFactory*
- 1000 изпълнения на *lookup()* операция с участие на *ObjectFactory*

За първите два сценария се налага след изпълнение на new InitialContext заявката да извикаме някаква jndi операция с име за параметър, за да предизвикаме задействането на механизма за търсене на initial и/или url context factory имплементации. Избираме да изпълним unbind() операция (изключва обекта асоцииран със зададеното име от именуващата система), тъй като по спецификация не предизвиква изключение дори ако в система несъществува обект с търсеното име.

Тестовите сценарии са реализирани като JUnit *TestCase* – *test.thesis.jndi.performance.PerformanceTest.* Изпълняват се първо върху стандартната JNDI имплементация, после се активира системата за регистрация и управление на доставчици на jndi услуги, тестовете се изпълняват още веднъж и накрая се сравняват получените резултати. Резултатите от второто изпълнение трябва да са по-добри или еквивалентни на предходните.

След като дефинирахме тестовете съпътстващи разработвания модул можем да преминем към проектирането и имплементацията му.

5.4. Архитектурни концепции

Най-логичният отговор на въпроса как да получим jndi factory имплементация по име на клас, без да използваме контекстния classloader на нишката е: Да имаме готова инстанция на factory обекта. Само ако имаме готова инстанция на обект няма да се налага ние да правим зареждането на класа му и създаването на нова инстанция от този тип.

Архитектурно factory обектите позволяват подобен подход, тъй като сами по себе си те не носят желаната от клиента функционалност, а служат за създаване на обекти, които притежават тази функционалност. Според дефиницията на Factory method дизайн шаблона е необходима само една инстанция на factory обекта, която да създава обекти персонализирани спрямо параметрите подадени от конкретния клиент [10]. Следователно ще решим проблема с клиентския classloader, ако проектираме контейнер обект за factory имплементации, с възможност всеки доставчик на именуващи услуги и/или семантични обекти да регистрира в него инстанции на собствените си имплементации. Ако пазим тези инстанции асоциирани със съответните им клас имена в *Hashtable* структури разделени спрямо типа на factory имплементациите (initial context, object, state and url context) ще можем бързо и лесно да намерим инстанция на подходящата factory имплементация за конкретната клиентска заявка. При това без да се налага да правим зареждане на класа й. За доставчиците на услуги не би било проблем да заредят класовете на factory имплементациите, които предоставят – те или се намират в клас ресурсите на доставчика или classloader-а на доставчика има референция към тях.

Това решение би било удачно, но все още е доста ограничаващо за доставчиците на jndi услуги – те на практика нямат възможност да направят нищо друго освен при инициализацията си да създадат инстанция на всяка една factory имплементация, която предоставят и да я регистрират в нашия контейнер модул.

Поглеждайки към втората цел, на текущата разработка - всеки доставчик на именуващи услуги и/или семантични обекти сам да избере механизма за намиране и зареждане на factory имплементациите, които той предлага, става ясно, че контейнерът за factory инстанции е стъпка в правилната посока, но не достатъчна, за да ни отведе до целта.

След като искаме всеки един доставчик да „има мнение” по въпроса с намирането и създаването на factory инстанция ще се наложи да го „попитаме” в подходящия момент. Вместо нашият модул да регистрира factory имплементации, ще регистрира доставчици на jndi услуги. Така, когато ни трябва да намерим подходящата factory имплементация ще можем да попитаме всеки от доставчиците дали може да ни я осигури.

Ще използваме принципа на JNDI средата за трансформиране на обект участващ в jndi операция, а именно: извиква се съответният метод последователно на всяка намерена factory имплементация; ако текущата имплементация не може да трансформира обекта връща резултат *null* и това е знак да се продължи със следващата factory имплементация; търсенето спира, когато се получи резултат различен от *null* или се прихване изключение.

6. Реализация

6.1. Контейнер за доставчици на jndi услуги

Дефинираме интерфейс *Provider* с публични методи за получаване на всеки един тип factory.

```
public interface Provider {

   public InitialContextFactory getInitialContextFactory(Hashtable env) throws NamingException;

   public ObjectFactory getURLContextFactory(String schema, Hashtable env) throws

   NamingException;

   public ObjectFactory getObjectFactory(String name) throws NamingException;

   public StateFactory getStateFactory(String name) throws NamingException;

}
```

Доставчиците на именуващи услуги и/или семантични обекти трябва да реализират този интерфейс като при извикване на метод за получаване на factory инстанция могат да приложат свой механизъм за откриването и създаването й, в случай че имплементацията на поисканото factory е предоставена от тях. В противен случай методът трябва да върне като резултат *null*. Методът може да хвърли изключение само в случай че доставчикът е разпознал factory имплементацията като своя, но има технически трудности да направи инстанция от нея (клас ресурсът липсва, няма връзка към jndi сървъра, на който е публикувана услугата, и т.н.).

Дефинираме абстрактен клас *ProviderManager*, който ще служи за контейнер на Provider обектите. Класът използва Singleton дизайн шаблон. Използва се статично посредством *getInstance()* метод. Предоставя методи за регистрация и премахване на Provider инстанция, които трябва да се използват от доставчиците на jndi услуги и семантични обекти съответно, когато искат да се включат или изключат от процеса по създаване на factory инстанция.

```
public abstract class ProviderManager implements StateFactory {

    protected static ProviderManager instance = null;

    public static ProviderManager getInstance() throws NamingException;

    public abstract void registerProvider(Provider provider) throws NamingException;

    public abstract void unregisterProvider(Provider provider) throws NamingException;

}
```

Имплементацията на абстрактните методи *registerProvider()* и *unregisterProvider()* ще предостави наследникът на *ProviderManager* класа – *ProviderResolver*. Целта ни е да разделим функционалността на модула на публична и частна като предоставим само необходимата функционалност на доставчиците на jnd услуги за публично използване, а останалото скрием.

Дефинираме клас *ProviderResolver* наследник на *ProviderManager* класа. При зареждане на класа ще се изпълни статичен инициализатор, който създава инстанция на *ProviderResolver* класа и я записва като статично поле в родителския *ProviderManager* клас. От тук насетне всеки, който извика статичният метод *getInstance()* на *ProviderManager* класа ще получи singleton инстанция от тип *ProviderResolver* и ще работи с нея.

ProviderResolver класа ще служи за контейнер на доставчици на именуващи услуги и семантични обекти, и за комуникацията с тях.

```
public class ProviderResolver extends ProviderManager {

    private Hashtable providers = new Hashtable();

     public void registerProvider(Provider provider) throws NamingException;

     public void unregisterProvider(Provider provider) throws NamingException;

     public Object getStateToBind(Object obj, Name name, Context nameCtx, Hashtable env) throws
NamingException;

    public StateFactory findStateFactory(final String stateFactoryName) throws NamingException;

    public ObjectFactory findObjectFactory(final String objectFactoryName) throws NamingException;
```

```
    public InitialContextFactory findInitialContextFactory(Hashtable env) throws NamingException;

    public ObjectFactory findURLContextFactory(final String scheme, Hashtable env) throws
NamingException;

}
```

Инстанциите от тип *Provider*, които доставчиците регистрират се пазят в *Hashtable* структура – име на класа на доставчика срещу *Provider* инстанция. Освен методите за регистриране и премахване на доставчици класът предоставя и методи за комуникация с тях. Това са методи за търсене на тип factory по име и евентуално конфигурации на контекстната среда. Всеки един от методите: *findInitialContextFactory(), findObjectFactory(), findStateFactory(), findURLContextFactory()* взима списъка на регистрираните доставчици и за всеки от тях извиква съответния метод за търсене на тип factory - *getInitialContextFactory(), getObjectFactory(), getStateFactory(), getURLContextFactory().* Обхождането на доставчиците спира при получаване на резултат различен от *null* от поредния доставчик или прихващането на изключение. Ако никой от доставчиците не успее да върне инстанция на търсеното factory съответният findXXX метод връща *null*.

След като вече имаме концепция за комуникацията на нашия модул с доставчиците на именуващи услуги и семантични обекти трябва да се погрижим за интеграцията му с JNDI средата.

6.2. Интеграция с JNDI средата

Ще използваме предоставените от JNDI SPI начини да се пренапише механизмът за намиране и зареждане на *InitialContextFactory* и *ObjectFactory* имплементации чрез регистрация на factory builder имплементации.

Дефинираме клас *InitialContextFactoryBuilderImpl*, който реализира интерфейсът *InitialContextFactoryBuilder* от JNDI SPI.

```
public class InitialContextFactoryBuilderImpl implements InitialContextFactoryBuilder {

    public InitialContextFactory createInitialContextFactory(Hashtable env) throws NamingException;

}
```

След като регистрираме нашия *InitialContextFactoryBuilder* в JNDI средата, *NamingManager* имплементацията ще започне да препраща всички заявки за създаване на начален контекст към него посредством извикване на метод *createInitialContextFactory*.

Ако на този етап препратим заявката за намиране на factory към *ProviderResolver* имплементацията ни ще получим като резултат инстанция на *InitialContextFactory* според конфигурациите на контекст средата. *NamingManager*-а ще използва тази factory инстанция, за да създаде контекст обект. Този контекст ще бъде записан в полето *defaultInitContext* на *InitialContext* класа и при следваща jndi операция ще бъде използван директно. Това, което пропускаме тук е възможността при jndi операция да проверим името за URL схема и да намерим подходящия контекст, който да я обработи. В нашия случай дори да има URL схема в името участващо в операцията заявката ще се изпълни върху контекста създаден от *InitialContextFactory* имплементацията. А това само по себе си означава, че лишаваме JNDI системата, както и всички доставчици на именуващи услуги от функционалност за поддръжка на URL схеми, което е напълно неприемливо.

За да предложим поддръжка за URL схеми и контексти ще трябва да отложим намирането и използването на *InitialContextFactory* имплементацията в *InitialContextFactoryBuilderImpl*. Вместо това ще конструираме и ще върнем на *NamingManager*-а една наша имплементация на *InitialContextFactory*.

Дефинираме *DefaultInitialContextFactory* клас, който реализира интерфейса *InitialContextFactory* от JNDI SPI. За него също използваме дизайн шаблон Singleton тъй като ще ни е нужна само една инстанция на класа за java процес. Тази инстанция е достъпна статично чрез *getInstance()* метода.

```
public class DefaultInitialContextFactory implements InitialContextFactory {

   private static DefaultInitialContextFactory instance = null;

   public static InitialContextFactory getInstance();

   public Context getInitialContext(Hashtable environment) throws NamingException;

}
```

NamingManager-а ще използва нашата *DefaultInitialContextFactory* инстанция за създаване на контекст посредством метода *getInitialContext*. Знаейки, че този контекст по-късно ще бъде използван за изпълнение на jndi операции не можем да върнем истинска контекст имплементация, защото все още не знаем дали това трябва да бъде initial или url контекст имплементация. Трябва да върнем празен контекст, т.е. контекст лишен от jndi функционалност, който по-късно, при изпълнението на първата jndi операция да можем да заменим с инстанция на правилния initial или url контекст.

Дефинираме клас *DefaultInitialContext* като наследник на *InitialContext* класа от JNDI SPI.

Методът, който най-много ни интересува в *InitialContext* класа и искаме да се възползваме от него в *DefaultInitialContext* е *getURLOrDefaultInitCtx.* Този метод се извиква преди всяка една jndi операция изпълнена върху *InitialContext* класа и именно той прави проверката за URL схема в подаденото като параметър име. Методът е с „*protected*" ниво на достъп, т.е. може да се пренапише в наследниците на класа. Това и ще направим в *DefaultInitialContext* класа.

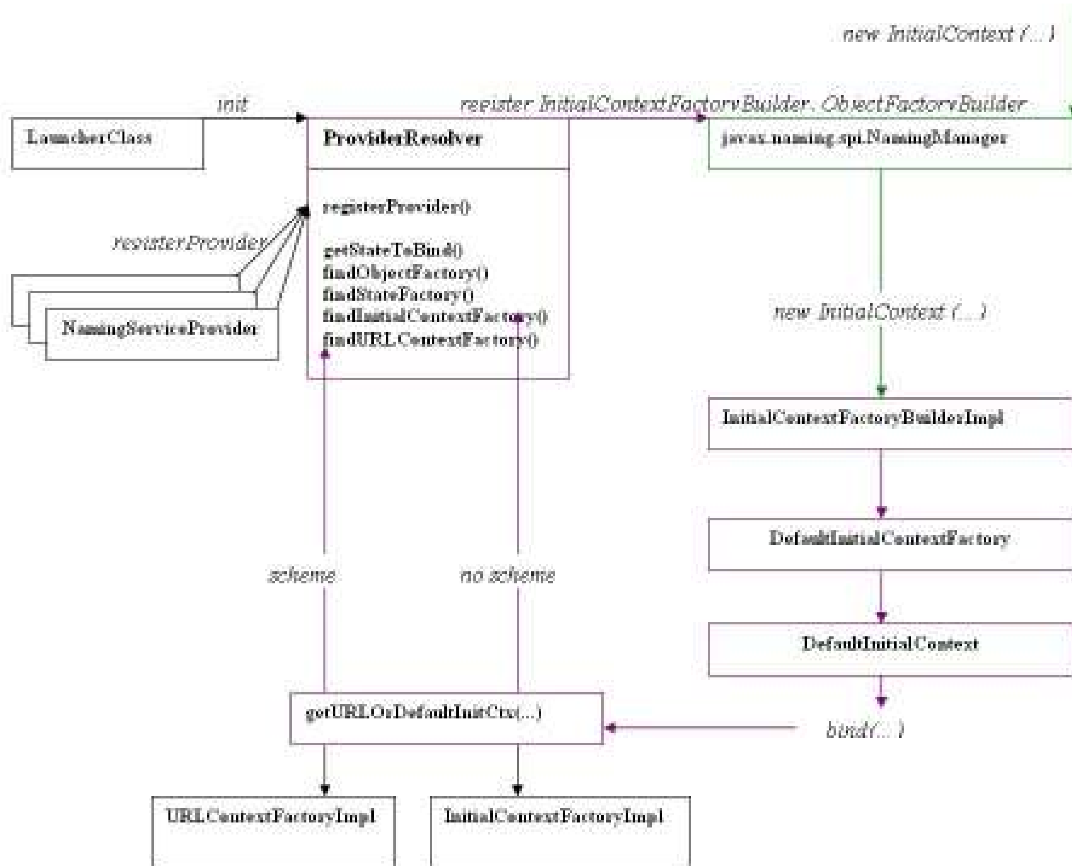
```
public class DefaultInitialContext extends InitialContext {

   protected Context getURLOrDefaultInitCtx(String name) throws NamingException;

   protected Context getDefaultInitCtx() throws NamingException;

   protected Context getURLOrDefaultInitCtx(Name name) throws NamingException;

}
```

getURLOrDefaultInitCtx() методът е точното място да попитаме *ProviderResolver* обекта, дали може да намери инстанция на factory имплементацията, която ни трябва. Първо проверяваме дали името започва с url схема. Ако да извикваме *findURLContextFactory* метода на *ProviderResolver* обекта, с параметри изолираната url схема и конфигурациите на контекстната среда (context environment). От своя страна *ProviderResolver*-а извиква синхронно *getURLContextFactory()* метода на регистрираните *Provider* инстанции. Претърсването спира, когато поредният извикан *Provider* върне резултат различен от *null* или предизвика изключение или когато и последният извикан *Provider* върне *null*. В случай на успешно намиране на

URLContextFactory, DefaultInitialContext-а го използва за създаване на контекст инстанция, върху която да се изпълни операцията.

При неуспех или липса на схема в името се прави опит да се намери подходяща *InitialContextFactory* имплементация. Извиква се *findInitialContextFactory* метода на *ProviderResolver* обекта, с параметър конфигурациите на контекстната среда (context environment). *ProviderResolver*-а извиква синхронно *getInitialContextFactory()* метода на регистрираните *Provider* инстанции като им подава името на *InitialContextFactory* имплементацията, която се търси. Претърсването спира, когато поредният извикан *Provider* върне резултат различен от *null* или предизвика изключение или когато и последният извикан *Provider* върне *null*. В случай на успешно намиране на *InitialContextFactory*, *DefaultInitialContext*-а го използва за създаване на контекст инстанция, върху която да се изпълни операцията. Също така го записва в полето си *defaultInitContext*, за ползване при следващи jndi операции. При неуспех на никоя от *Provider* инстанциите да намери търсената factory имплементация се прибягва до стандартния механизъм на *NamingManager* имплементацията, а именно *ProviderResolver*-а прави опит да зареди класа на търсеното factory с контекстния classloader на текущата нишка (current thread). При нов неуспех се стига до изключение (*NamingException*).

Схематично операцията за създаване на нов начален контекст с участие на *InitialContextFactory* и *URLContextFactory* имплементации може да се представи така:



Фиг. 5: *Схема на операцията създаване на нов начален контекст.*

До тук успяхме да въвлечем *ProviderResolver* обекта ни в намирането и създаването на initial и url context factory имплементации. Следва да направим същото и за останалите два factory типа.

Дефинираме клас *ObjectFactoryBuilderImpl*, който реализира интерфейсът *ObjectFactoryBuilder* от JNDI SPI. След като регистрираме нашия *ObjectFactoryBuilder* в JNDI средата, *NamingManager* имплементацията ще започне да препраща всички заявки за намиране и създаване на *ObjectFactory* инстанции към него посредством извикване на метод *createObjectFactory(Object refInfo, Hashtable env)*.

```
public class ObjectFactoryBuilderImpl implements ObjectFactoryBuilder {
```

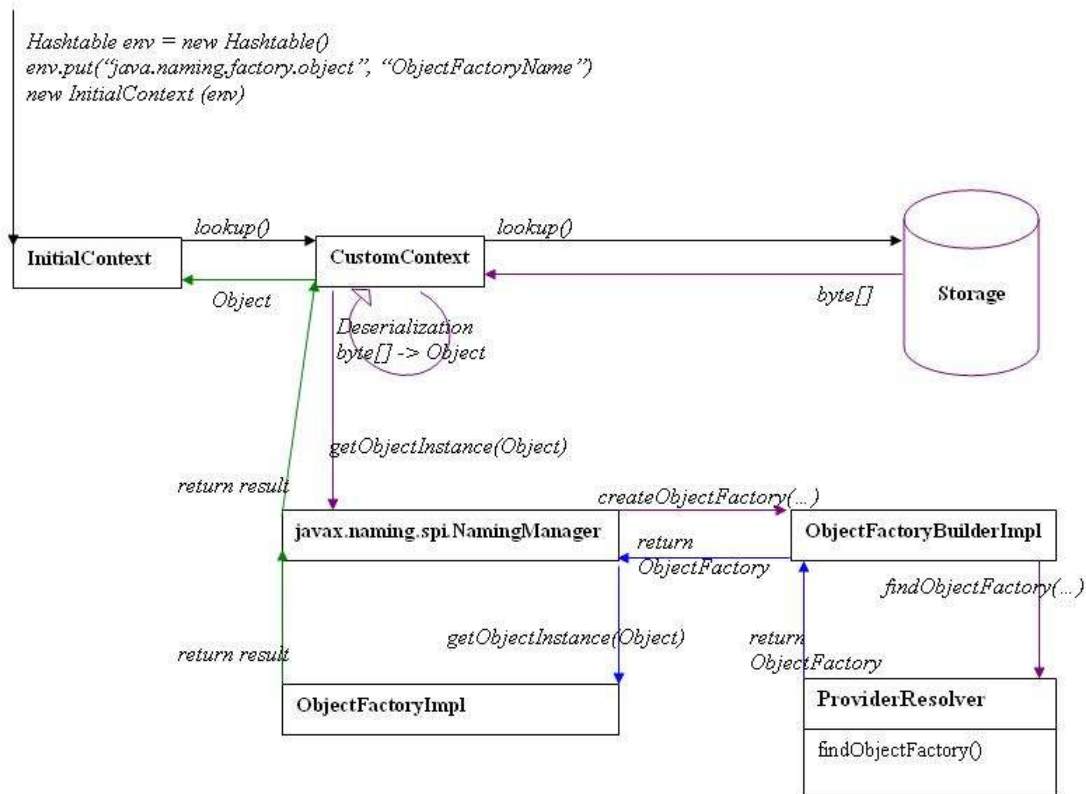
```
    public ObjectFactory createObjectFactory(Object refInfo, Hashtable env) throws NamingException
;

}
```

Методът *createObjectFactory* извлича от конфигурациите на контекст средата списъка с имена на *ObjectFactory* имплементации и последователно ги подава на *findObjectFactory()* метода на *ProviderResolver*-а. От своя страна *ProviderResolver*-а извиква синхронно *getObjectFactory()* метода на регистрираните Provider инстанции. Претърсването спира, когато поредният извикан *Provider* върне резултат различен от *null* или предизвика изключение или когато и последният извикан Provider върне *null*. В случай на успешно намиране на *ObjectFactory*, *NamingManager*-а го използва за трансформиране на обекта върнат като резултат от jndi операция преди той да се предаде на клиента. Ако не е намерена подходяща factory имплементация стандартната процедура на *NamingManager*-а е да върне обекта на клиента непроменен, но когато има регистриран *ObjectFactoryBuilder* той е длъжен да върне различен от *null* резултат дори когато няма подходяща *ObjectFactory* имплементация. В този случай ние трябва да върнем някаква наша имплементация на *ObjectFactory*, която просто да връща обекта непроменен вместо да го трансформира.

Дефинираме *DefaultObjectFactory* клас, който реализира интерфейса *ObjectFactory* и има единствен метод *getObjectInstance()*, който връща подадения като параметър обект, без да го променя.

```
public class DefaultObjectFactory implements ObjectFactory {

   public Object getObjectInstance(Object obj, Name name, Context ctx, Hashtable env) throws
Exception;

}
```

Схематично операцията за търсене на обект по име с участие на *ObjectFactory* имплементация, която трансформира обекта може да се представи така:



Фиг. 6: *Схема на операцията търсене на обект по име.*

До тук използвахме предоставените от JNDI SPI методи за пренаписване на механизма за намиране и създаване на context и object factory инстанции. За *StateFactory* имплементациите обаче не се предлага такава възможност. Единственият вариант за въздействие върху стандартния механизъм за откриване и създаване на *StateFactory* инстанции е *NamingManager*-а да извика метод на наш клас, който от своя страна да делегира заявката към *ProviderResolver*-а. Тъй като, в този случай нямаме възможност за регистрация на нещо като *StateFactoryBuilder* ще използваме наша *StateFactory* имплементация.

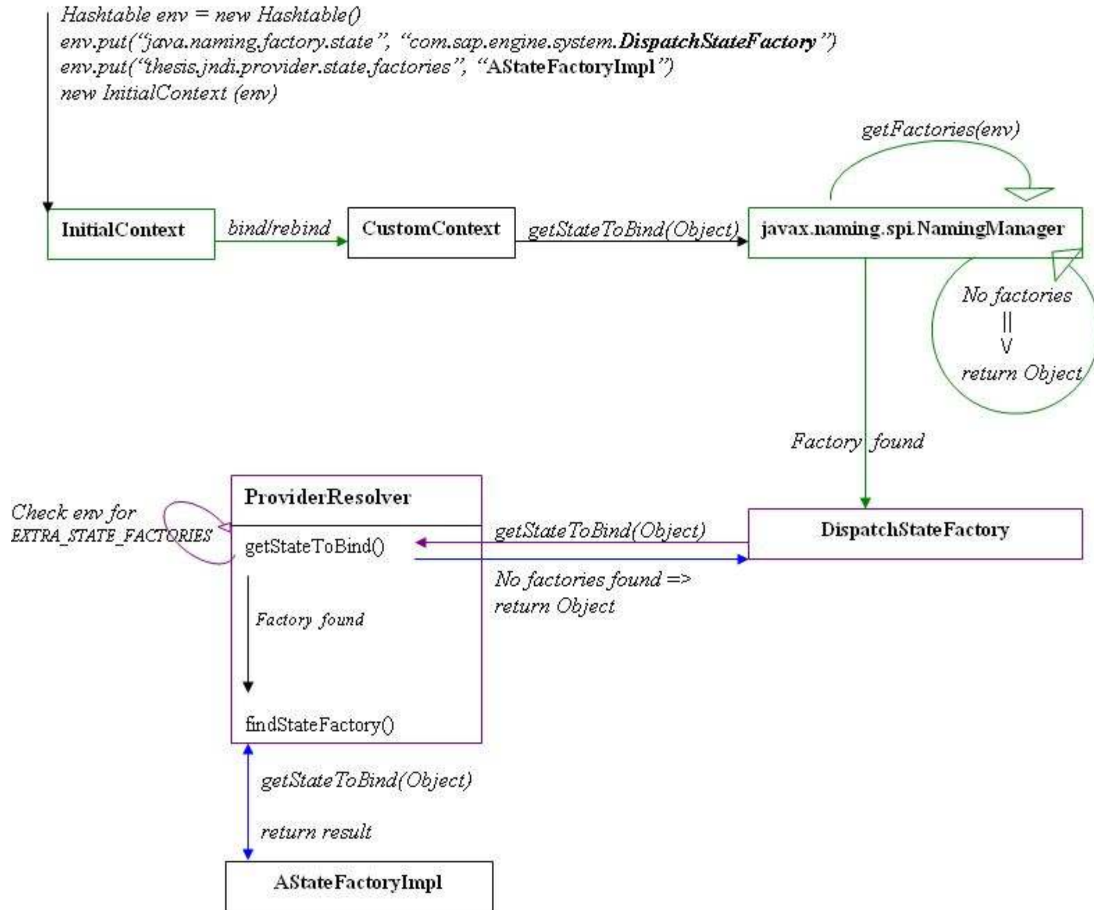
Дефинираме клас *DispatchStateFactory*, който реализира интерфейсът *StateFactory*.

```
public class DispatchStateFactory implements StateFactory {

   public Object getStateToBind(Object obj, Name name, Context ctx, Hashtable env) throws
NamingException ;
```

```
}
```

Когато *NamingManager*-ът трябва на намери *StateFactory* инстанция, взима списъка от имена на *StateFactory* класове дефинирани в конфигурацията „*java.naming.factory.state*" на контекстната среда и се опитва да зареди всяка от тях с контекстния classloader на нишката. Един от тези factory класове записани в конфигурацията е *DispatchStateFactory*. Ако някое factory преди него успее да обработи обекта, до извикване на *DispatchStateFactory* имплементацията няма да се стигне. Ако обаче никой от предишните factory класове не е бил успешно зареден или не е успял да разпознае обекта *NamingManager*-ът ще извика *getStateToBind()* метода на *DispatchStateFactory* класа и той ще препрати заявката към *ProviderResolver*-а. *ProviderResolver* класът ще вземе списъкът от *StateFactory* класове от конфигурацията на контекстната среда, ще изключи от него *DispatchStateFactory* записа и по познатия вече алгоритъм ще провери дали някой от регистрираните *Provider* обекти е в състояние да създаде подходяща factory инстанция. Ако се намери успешно *StateFactory* инстанция *getStateToBind()* методът и ще бъде извикан. В противен случай *null* резултат ще бъде върнат към *NamingManager*-а.

Схематично операцията за асоцииране на обект с име с участие на *StateFactory* имплементация, която трансформира обекта може да се представи така:



Фиг. 6: *Схема на операцията асоцииране на обект с име.*

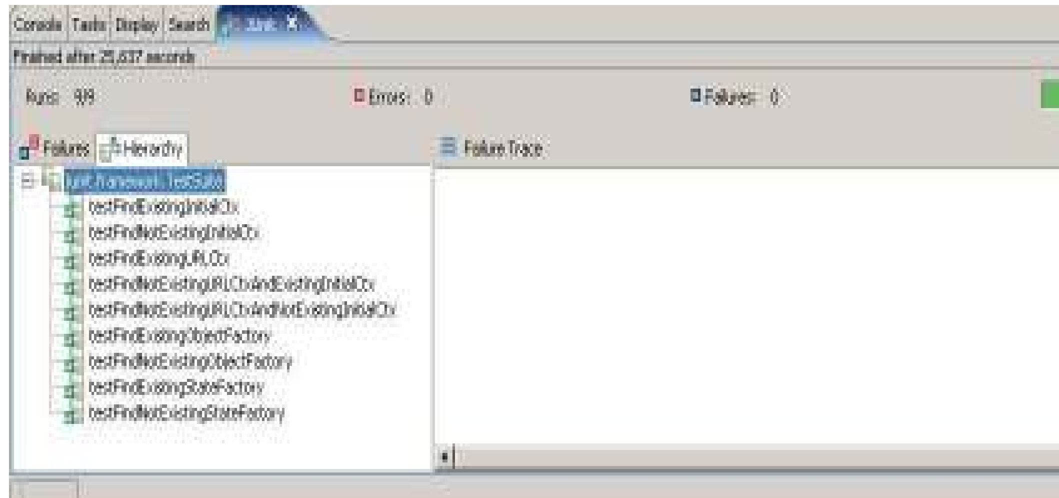
7. Проверка за качество и коректност на услугата

След като дефинирахме функционалността на разработвания модул и успяхме да включим наш механизъм в JNDI SPI за намиране и създаване на всички типове factory обекти е време да проверим качествата на модула по отношение на:

- Коректност на предложената функционалност – дали разработваният модул предлага същата базова функционалност на JNDI средата. Много е важно доставчиците желаещи да използват стандартния механизъм на JNDI средата за получаване на factory имплементации, т.е. нерегистрирали свои *Provider* инстанции в нашия *ProviderManager* да получат същото качество на услугата през разработвания модул, както, ако той не беше внедрен в JNDI системата.

  Изпълнението на теста за съвместимост със стандартната имплементация на JNDI средата е успешно и показва, че разработената система има едно и също поведение със стандартната имплементация по отношение на резултатите върнати при еднакви входни параметри и условия за изпълнение. Тестът е изпълнен в eclipse среда за разработка срещу JUnit библиотека версия 4.4.

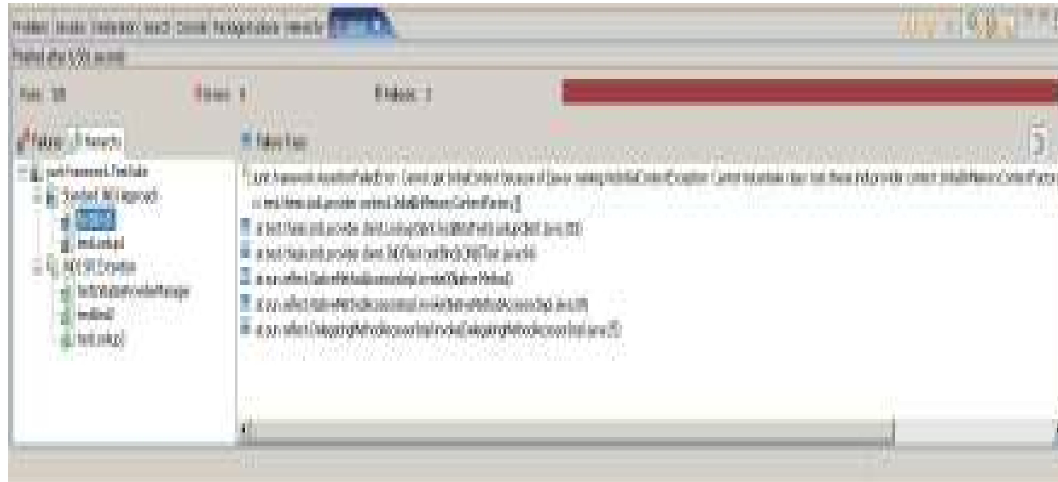
  Фигура 7 изобразява резултатите от теста.



Фиг. 7: *Резултати от теста за съвместимост на разработената система и стандартната JNDI функционалност.*

- Степен на покритие на поставените цели – дали сме успели да постигнем целите поставени пред разработвания модул.

  Тестовете покриващи предложената от разработката нова функционалност са успешно изпълнени. Използвана е стандартната интеграция на eclipse средата за разработка и JUnit библиотека 4.4. Фигура 8 изобразява резултатите от изпълнението на тестовете. Тук ясно се вижда, как стандартната JNDI имплементация не може да се справи, ако factory класовете не са достъпни за клиента – първата група тестове са неуспешни. Втората група тестове преминава успешно при същите условия, който факт свидетелства, че системата ни успешно се справя с този проблем.

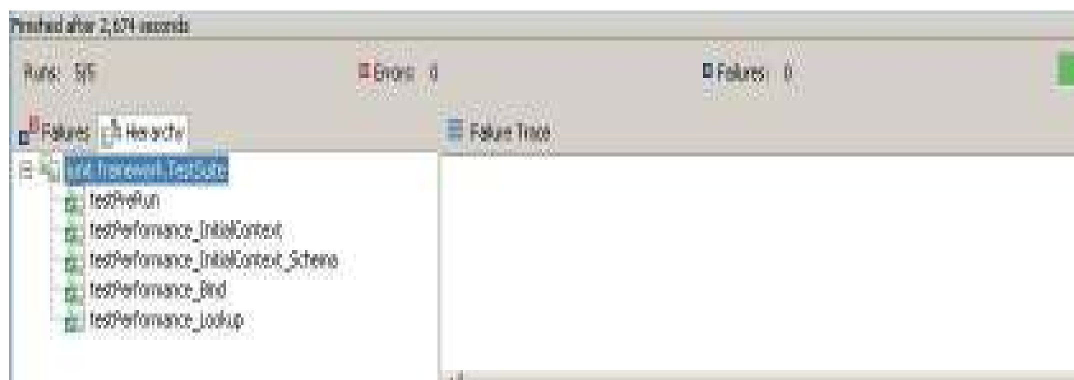


Фиг. 8: *Резултати от теста за пълнота на предложената функционалност.*

- Оптималност на операциите по отношение на бързодействие и консумация на памет.

При първото изпълнение на тестовете, проверяващи дали ефективността при намиране и зареждане на factory имплементации в предложеното решение е по-добра или поне същата в сравнение със стандартната JNDI имплементация, резултатите бяха негативни. Предложената система даде резултати (в милисекунди) с около 30% по-високи в сравнение със стандартната JNDI имплементация, т.е. jndi операциите, които тестваме се изпълняваха с около 30% по-бавно.

След направените оптимизации и подобрения описани в следващата глава резултатът се подобри значително. Фигура 9 изобразява резултатът от изпълнение на JUnit тестовете.



Фиг. 9: *Резултати от теста за бързодействие и ефективност.*

Най-добри резултати разработената системата постига при операциите за създаване на нов начален контекст.

Конкретните резултати от проведения тест са поместени в таблицата на фигура 10. Това са акумулираните и усреднени резултати от 10 последователни изпълнения.

Име на тест	**Брой операции**	**Резултат при стандартна JNDI среда**	**Резултат при разработената система**
InitialContext + Unbind	1000	**531**	**80**
InitialContext + Unbind + Scheme	1000	**431**	**120**
Rebind	1000	**160**	**120**
Lookup	1000	**150**	**90**

Фиг. 10: *Резултати от теста за бързодействие и ефективност.*

8. Подобрения и разширения

На база на резултатите от направените тестове можем да направим някои подобрения и оптимизации представени в следващите две части.

8.1. Функционални

Наличната функционалност заложена в системата за регистрация и управление на jndi доставчици покрива:

- 100% от стандартната функционалност на JNDI средата т.е., за клиентите и доставчиците на jndi услуги е напълно прозрачно, че между JNDI средата и имплементациите на именуващи услуги се е появил допълнителен слой. Ако доставчиците на JNDI услуги не използват новите възможности, които предлага нашият модул, т.е. не регистрират свои *Provider* имплементации в *ProviderManager*-а услугата, която получават от модула ни е същата като стандартната услуга на JNDI средата.

- 95% от функционалността, която сме си поставили за цел. Доставчиците на именуващи услуги и семантични обекти имат възможност да използват свой механизъм за откриване и зареждане на своите factory имплементации посредством регистрация в нашата система. По този начин освобождават клиентите си от задължението да имат достъп до класовете от имплементацията на услугата, която използват. Системата ни е интегрирана в JNDIсредата и достъпна за клиентите на именуващи услуги през стандартните JNDI API интерфейси.

  Единственият пропуск в предвидената функционалност е, че доставчиците на семантични обекти не могат да са сигурни, че ще имат възможност в 100% от случаите да използват собствен механизъм за намиране на *StateFactory* имплементацията си. Ако класа на търсената имплементация е достъпен за classloader-а на клиента и нашето *DispatchStateFactory* е описано след търсеното factory, *NamingManager*-а ще зареди factory-то със стандартния механизъм. Така

няма да получим възможност да делегираме търсенето на необходимото factory на *ProviderResolver*-а ни, а от там на регистрираните *Provider* имплементации. Другият проблем е, че докато се стигне до *DispatchStateFactory* имплементацията ни, *NamingManager*-а ще е изпробвал стандартния механизъм безуспешно върху всички предходни записи на factory имена губейки така доста време. За да се справим с тези два функционални недостатъка на разработвания модул ще дефинираме наша конфигурация на контекст средата - "*thesis.jndi.provider.state.factories*", която ще е достъпна като константа в *DispatchStateFactory* класа. Клиентите използващи дадени *StateFactory* имплементации ще могат да ги опишат в тази конфигурация на контекст средата, а в стандартната "*java.naming.factory.state*" да зададат само името на *DispatchStateFactory* класа. Така *DispatchStateFactory* имплементацията ще се зареди от *NamingManager*-а първа и ще може да делегира заявката за намиране на подходящо *StateFactory* на *ProviderResolver*-а ни. Той от своя страна ще вземе списъка от имена на factory имплементации от "*thesis.jndi.provider.state.factories*" конфигурацията на контекстната среда, ще добави към него останалите factory имплементации описани в „*java.naming.factory.state*" конфигурацията и за всеки запис от списъка ще извика *getStateFactory* метода на регистрираните *Provider*-и.

8.2. Оптимизационни

На база на извършените тестове можем да направим следните оптимизации:

- Най-бавната част от процеса по намиране на дадена factory имплементация е обхождането един по един на всички регистрирани *Provider* имплементации. Няколко последователни заявки за едно и също factory се изпълняват еднакво бавно. А всъщност е необходимо само първия път да обиколим *Provider* имплементациите, за да намерим тази, която може да осигури исканото factory и да кешираме резултата. Така всяка следваща заявка за това factory ще минава много бързо. Актуализация на кеша ще се налага само в случай на премахване на регистрирана *Provider* инстанция.

  ProviderResolver класа дефинира четири хеш таблици (Hashtable) по една за всеки вид factory и съхранява в тях името на търсеното factory или URL схема като ключ срещу името на класа на *Provider* имплементацията, която е успяла да го

предостави. Записваме името на класа на *Provider* обекта, защото да запишем самият обект означава да държим жива референция към classloader-а му. Ако classloader-а на компонента, който е регистрирал *Provider* обект се подмени (redeploy/restart) и компонента направи регистрация на нова *Provider* инстанция без първо да премахне старата с *unregisterProvider()* ще се получи classloader leak (изтичане на ресурси), защото ние все още държим и използваме старата инстанция на *Provider* обекта.

Съответно всички findYYYFactory методи се променят първо да проверяват дали има запис за *Provider*, който може да върне търсеното factory. Ако „да" по името на класа на *Provider*-а се намира имплементацията му. В противни случаи се преминава към познатия алгоритъм с „питането" на всяка един *Provider* дали може да намери инстанция на търсеното factory. Първият от *Provider*-ите, който успее да върне не *null* резултат се записва по име в таблицата за съответния тип factory.

unregisterProvider(Provider) метода се грижи да почисти записите в таблицата отнасящи се за *Provider* инстанцията, която бива премахната. Тъй като ключовете в таблицата са имена на factory имплементации, а стойностите имена на класове на *Provider* имплементации, *unregisterProvider* метода трябва да обходи цялата таблица, търсейки стойности съвпадащи с името на класа на премахнатия *Provider*. Но това е цена, която си заслужава да платим за сметка на оптимизацията във findYYYFactory методите.

9. Ръководство за употреба

9.1. Инициализация

Класовете от имплементацията на системата за регистрация и управление на jndi доставчици трябва да са достъпни за всички приложения и компоненти в рамките на java виртуалната машина. Това налага публикуването им в началото (root) на йерархията от classloader-и, т.е. тези класове трябва да са достъпни за родителския classloader на classloader инстанциите на останалите компоненти.

Необходима е инициализация на системата преди употреба.

Инициализацията се свежда до извикване на *init()* метода на *ProviderResolver* класа.

```
ProviderResolver.init();
```

Важно е инициализацията на системата да се случи преди който и да е доставчик на именуващи услуги да се е активирал, като по този начин се избягва възможността някой вече да е регистрирал *InitialContextFactoryBuilder* и *ObjectFactoryBuilder* в *NamingManager*-a.

Веднъж инициализирана системата е готова за използване.

9.2. Клиенти на jndi услуги

Тъй като съвместимостта с JNDI API беше основен аспект при дефинирането на системата ни за регистрация и управление на доставчици на jndi услуги, клиентите на тези услуги ще могат да се възползват от тях, както до сега през стандартните интерфейси на JNDI API. Голямо облекчение за тези клиенти носи и фактът, че не се налага повече класовете на имплементацията на jndi услугата да са достъпни за клиентския classloader.

9.3. Доставчици на jndi услуги

Доставчиците на именуващи услуги и/или семантични обекти трябва да имплементират *Provider* интерфейса като са свободни да заложат, какъвто искат механизъм за намиране и създаване на factory обекти от типовете, които предоставят. При активиране на услугата, най-често стартиране на приложението доставчик, инстанция на *Provider* имплементация трябва да се регистрира в *ProviderManager* класа.

```
ProviderManager.getInstance().registerProvider(new AnyProvider());
```

Важно е при деактивиране на услугата, най-често спиране на приложението доставчик, Provider имплементацията да се премахне от системата за управление на jndi доставчици. Това става посредством извикване на статичен метод *unregisterProvider* на *ProviderManager* имплементацията.

```
ProviderManager.getInstance().unregisterProvider(anyProviderInstance);
```

Метода *unregisterProvider* приема като аргумент инстанцията на Provider имплементацията, която е била регистрирана.

9.4. Миграция на вече съществуващи приложения

Вече съществуващи приложения доставчици на jndi услуги могат много лесно да се приспособят да работят с предложената система. Единственото условие е да регистрират своя имплементация на *Provider* интерфейса в *ProviderManager* класа.

Но дори и да не се възползват от функционалността предоставена от системата за регистрация и управление на доставчици на jndi услуги, старите приложения ще продължат да са достъпни и да работят, както до сега. Активирането на системата като надстройка на JNDI средата няма да попречи на изпълнението на стандартната функционалност в случаите, когато приложенията разчитат на нея.

10. Заключение

Реализираната система успешно премина подготвените тестовете гарантираци изпълнение на поставените цели и задачи, коректност на работа и съвместимост със стандартната функционалност на JNDI средата.

Резултатът от текущата разработка може да се използва успешно във всеки един по-мащабен продукт, където се налага изолация на приложенията на ниво classloader.

Проблемите, които текущата разработка има за цел да реши, е възможно в бъдеще да бъдат решени в самата JNDI платформа, но за съжаление текущата версия на спецификацията 1.2.1. не е променяна в последните 8 години. Дори новата Java EE 5 спецификация включва старата 1.2.1 версия на JNDI. В Java общността (community) няма информация дали и кога може да се очаква нова версия на спецификацията.

С развиването на Java EE сървър технологиите обаче все повече нараства нуждата от гъвкава и добре дефинирана интеграция между доставчиците на именуващи услуги и техните клиенти. По тази причина може да се заключи, че текущата разработка е реализирана в правилния момент от времето, когато от нея има реална нужда.

10.1. Възможни подобрения и бъдещи насоки

Полезно допълнение на системата за регистрация и управление на доставчици на jndi услуги би била автоматичната регистрация на доставчици.

В момента единственият начин един доставчик на jndi услуги да се интегрира със системата ни е като създаде собствена имплементация на *Provider* интерфейса и програмно я регистрира в системата. Това предполага, че приложението доставчик има знание за жизнения си цикъл – като минимум стартиране и спиране. Но ако приложението няма ясно обусловен жизнен цикъл или няма знание за него, в кой момент от време трябва да направи регистрацията на *Provider* имплементацията си?

А ако доставчика на jndi услуга е библиотека – набор от класове използвани от приложенията?

Всеки един компонент обаче, който живее в дадена сървър система има deployment descriptor, т.е. файл описващ различни конфигурации на компонента като: classloader референции към други компоненти, необходими ресурси, име, версия и др.

В JNDI SPI има дефиниран jndiprovider.properties файл, който се пакетира в пакета на контекст имплементацията на доставчика на услугата и се използва за откриване на *ObjectFactory* и *StateFactory* имплементации.

На подобен принцип всеки компонент може да опише в deployment descriptor файла си имената на класовете на различните типове factory имплементации, които пакетира в архива си. Инфраструктурата, която чете deployment descriptor файла и се грижи за внедряването на компонент във виртуалната машина на сървър системата може да генерира опростена *Provider* имплементация за този компонент и автоматично да я регистрира в разработената вече система. *Provider* имплементацията може да има таблици за типовете factory имплементации, които предлага. След създаване на classloader-а на клиента, с негова помощ се зарежда инстанция от всяка factory имплементация описана в deployment descriptor файла и се поставя в съответната таблица на *Provider* имплементацията.

По този начин компонентът доставчик на jndi услуги получава автоматична интеграция със системата ни, без да е необходимо да се правят промени по кода му. Това решение би било много полезно за компоненти/библиотеки предлагащи именуващи услуги или семантични обекти предоставени от външни софтуерни доставчици. В този случай не можем да променяме кода на компонентите, но за да са съвместими с конкретната сървър система за тях също се предвиждат deployment descriptor файлове.

11. Допълнения

11.1. Речник на използваните термини

Classloader	*Classloader обектите формират основата на платформата Java. Всеки един клас бива зареден от конкретен Classloader обект, имащ за задача да прочете дефиницията на класа и да инициализира Class обект от нея. След това – имайки Class обекта – можем да създаваме инстанции от конкретния тип. Classloader обектите формират йерархична структура на наследяване.*
Singleton pattern	*Singleton е шаблон предвиждащ създаване и използване на една единствена инстанция на обект от даден тип, която да координира използването на предлаганата функционалност в системата.*
Factory method pattern	*Factory method е шаблон отнасящ се за методи, които трябва да създадат обект без предварително да се знае конкретният тип на обекта.структура. Обикновено методът декларира връщана стойност от супер тип, а в процес на работа установява обект от кой подтип трябва да създаде.*

Design Patterns	*Набор от шаблони, предоставящи всеобщо известни и доказани решения на добре познати базови проблеми. Тези решения са доказали своята оптималност с течение на времето. Употребата на такива шаблони позволява много по-лесно общуване между различните разработчици, тъй като използвайки еднакви термини всички те общуват на „един език” – езика на шаблоните.*
Java Virtual Machine (JVM)	*Java виртуалната машина е средата, в която работят и се изпълняват програмите написани на езика Java. Нейната цел е да интерпретира кода на програмата, да го компилира до машинен код и да го изпълни. Повечето реализации имат 2 режима на работа – режим на интерпретиране и режим на изпълнение на машинен код. При първия виртуалната машина третира кода на програмата инструкция по инструкция, интерпретирайки всяка една от тях и изпълнявайки я. Втория режим превежда програмата на машинен език, използвайки различни оптимизации, за да се постигне подобряване на производителността.*

JUnit	*JUnit е среда за автоматизирано тестване на Java приложения създадена от Erich Gamma и Kent Beck. Тя позволява дефинирането на тестове (unit tests) за градивните частици на даден продукт – класове и методи.*
Test case	*Единичен тест, проверяващ коректността на конкретен метод от даден модул. Може да бъде групиран в групи от тестове.*
Test suite	*Група от тестове, позволяваща построяването на йерархични структури от тестове и групи от тестове. Всички тестове от групата се изпълняват последователно, проверявайки различни аспекти на правилната работа на един и същ модул*
Test-driven development process	*Методология, която изтъква създаването на тестове като важна и неразделна част от процеса на разработката на софтуер.*

12.Използвана литература

[1]. Todd Sundsted, "JNDI Overview, Part 1: An introduction to naming services," JavaWorld, 01/2000.

[2]. Jim Farley, William Crawford, Java Enterprise in a Nutshell, Second Eddition

[3]. Jeffry A. Borror, Java Principles of Object-Oriented Programming

[4]. Rosanna Lee, The JNDI Tutorial, Building Directory-Enabled Java Applications

[5]. Mick Jordan, Laurent Daynès, Grzegorz Czajkowski, Scaling J2EETM application servers with the Multi-tasking Virtual Machine, Software: Practice and Experience, Volume 36, Issue 6, 14 Feb 2006

[6]. Kent Beck, Test-driven Development: By Example, 2003

[7]. Elfriede Dustin, Jeff Rashka, John Paul, Automated Software Testing: Introduction, Management, and Performance, 1999

[8]. Ben Rometsch, 10 ways to increase your web development productivitywhilst playing Wii Tennis, March 2007

[9]. Andy Schneider, JUnit best practices: Techniques for building resilient, relocatable, multithreaded JUnit testsJavaWorld.com, 12/21/00

[10]. Sherif. M. Yacoub, Hany Hussein Ammar, Pattern-Oriented Analysis and Design: Composing Patterns to Design Software, 2004