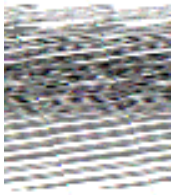


СОФИЙСКИ УНИВЕРСИТЕТ
“СВ. КЛИМЕНТ ОХРИДСКИ”

ФАКУЛТЕТ ПО МАТЕМАТИКА И ИНФОРМАТИКА

КАТЕДРА “ИНФОРМАЦИОННИ ТЕХНОЛОГИИ”

СПЕЦИАЛИЗАЦИЯ “ИНФОРМАЦИОННИ И ТЕЛЕКОМУКАЦИОННИ ТЕХНОЛОГИИ”

ДИПЛОМНА РАБОТА

Тема:

Компонент за трансформиране на електронни търговски документи към стандартен вътрешен XML формат

Дипломант:

Иво Емилов Жеков,

ФN№ 42649

Дипломен ръководител:

доц. Силвия Илиева

София, 2005 г.

Съдържание

Списък на фигурите

1 Въведение

През последните няколко десетилетия сме свидетели на всеобхватно разпространение на информационните технологии. Те намират все по-широко приложение в ежедневието на съвременното общество – от средства за развлечение до неотменна част на научноизследователския апарат. Но това бурно развитие поражда и редица проблеми.

Тук ще разгледаме един от тях в контекста на системите за обмен на бизнес документи, чиито основни функции са получаването на поръчки, потвърждаването или отхвърлянето им, както и последващото изпращане на фактури при успешен край на бизнес транзакцията.

1.1 Кратка анотация на проблема

Един от основните проблеми в комуникацията между бизнес системите – формата на входно/изходните данни – е до голяма степен обусловен от появата и налагането на Интернет. Глобалната мрежа е предизвикала бурното развитие и усъвършенстване на софтуерните системи за електронна комуникация между различните по вид и предназначение бизнес системи на търговските партньори. Предимствата на тези решения са многобройни и безспорни – оптимизиране на операциите, намаляване възможността за грешки, пълна автоматизация на процесите в отделната организация и при транзакциите между тях. Но оттук следва и основното предизвикателство – как да се проектират и реализират системите така, че да е възможно да се обменя информация между хетерогенни системи – и като технология на изработка (например в зависимост от използваните платформи като J2EE, .NET и т.н.), и като изисквания спрямо формата и начина на обмен на данните – по FTP, HTTP, BatchNet или в зависимост от протокола – SOAP, XML и т.н.

В резултат на задълбочени анализи ([1]) се изяснява, че най-подходящ формат за предаване на такъв вид данни е XML (eXtensible Markup Language). Но заедно с това се наблюдава и появата на много близки, но все пак разнородни и по-тясно специализирани видове XML. Освен това, за

съвместимост със стари системи, все още се поддържат и линейни файлови (flat file) формати.

1.2 Цел – унифициране формата на данните

Целта на настоящата дипломна работа е да се разработи помощно мидълуер (middleware) приложение, което да трансформира различни външни формати за бизнес документи в общ, вътрешен XML формат.

Такова решение ще улесни интеграцията между вече съществуващите решения. Също така негова основна цел е да капсулира детайлите при получаването на входните данни, както и последващото изпращане на резултата. Тази му функционалност ще позволи на софтуерните системи на отделните организации да се съсредоточат върху основната си функция, а именно специфичните за съответния бизнес процеси.

Като потенциални потребители на решението могат да бъдат определени два типа софтуерни решения. Първите са вече съществуващите такива, при които голяма част от работата по поддръжка и разширяване на функционалността се изразходва за свързване с външни системи, които предоставят нови услуги за бизнеса. Вторият вид са новите системи в областта на бизнес транзакциите, които се проектират като гъвкави и силно разширяеми решения.

Конкретните задачи, произтичащи от целта, са:

- Анализиране и изследване на най-използваните формати за бизнес поръчки. Приложението да разглежда XML форматите, но да е достатъчно гъвкаво, за да е възможно лесното добавяне и поддръжка и на други формати, в частност линейните текстови данни.
- Специфициране на най-подходящите общи части на XML форматите и дефиниране на общ вътрешен XML формат.
- Анализ и дизайн на мидълуер компонент, трансформиращ определени видове XML в дефинирания вече вътрешен XML формат.

- Реализация на мидълуер компонента, включващ и среда за модулно тестване на трансформирането на бизнес документи, получени в някой от познатите на системата формати.
- Разработка на примерно Интернет базирано приложение, позволяващо да се тества предложеното софтуерно решение.

Ограничаващи/облекчаващи условия:

- Приложението да използва свободен софтуер под формата на XML стандарти, бази данни и т.н.
- Приложението да се фокусира върху процеса на трансформиране и изследване на XML стандартите, а не толкова върху удобството на графичния интерфейс.

1.3 Структура на дипломната работа

В последващото изложение, в глава 2 “Обзор на проблемната област и теоретична постановка на решението” е направен увод в използваната терминология, като са дефинирани понятията мидълуер (точка 2.1) и XML (eXtensible markup Language) (точка 2.2). Също така са разгледани най-широко разпространените формати за електронни търговски документи (точка 2.3).

В глава 3 “Модел на решението” е направен обзор на предлаганото решение. Описанието е направено от няколко различни аспекта: структурата на базата данни (глава 4), вътрешният формат за представяне на търговските ордери (глава 5), дизайн и имплементиране на трансформиращия компонент (глава 6), както и разработената инфраструктура за тестване на приложението (глава 7).

В глава 8 “Преглед на съществуващи интеграционни решения в разглежданата област” са представени функционално подобни съществуващи решения.

В глава 9 “Заключение” е направен критичен анализ на разработеното приложение, като заедно с това са представени и бъдещите направления за развитие на разглежданата система.

В глава 10 “Използвана литература” е предоставен списъкът с използваните книги и интернет адреси с допълнителна литература.

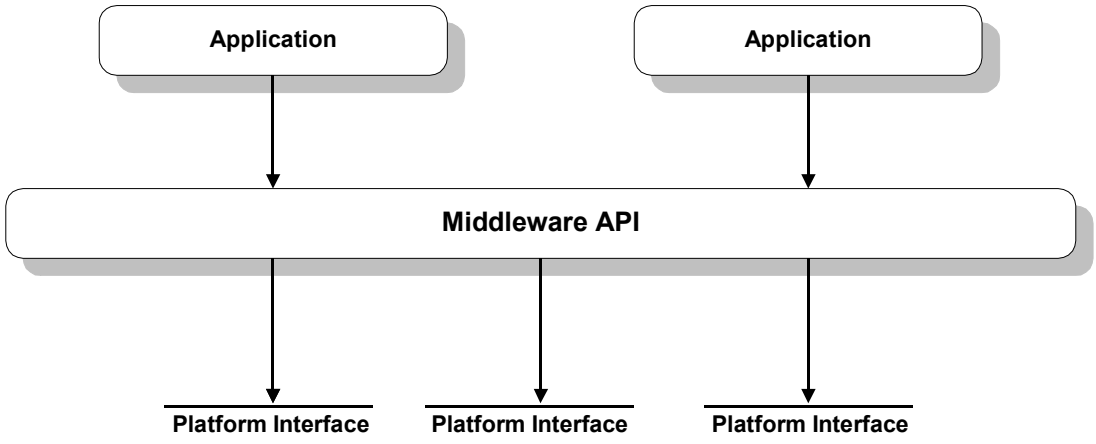
В глава 11 “Приложения” се намира описанието на придружаващия диск с програмния код на разработените системи и скриптовете за базата данни (Приложение А: Описание на придружаващия компакт диск), а също така и цялостното съдържание на вътрешния формат за представяне на търговските поръчки (Приложение Б: Вътрешен формат – BDI XSD).

2 Обзор на проблемната област и теоретична постановка на решението

Първоначално ще се спрем на теоретичната обосновка на дадения проблем, заедно с дефиниция на основните използвани понятия, за да може постепенно в техния контекст да се изяснят аспектите на предлаганото решение.

2.1 Мидълуер

Под понятието мидълуер (middleware) в съвременните информационни технологии се класифицира софтуер, който е отговорен за свързването на разнородни системи и *позволява множество процеси, работещи на една или повече машини да взаимодействат по мрежата* ([2]). Тази технология се е появила през 70–те години на миналия век като алтернатива на широко използваните дотогава самостоятелни (stand-alone) приложения в подкрепа на трансформирането им към клиент/сървър архитектурите. Най-добре развитите инициативи в тази област са Distributed Computing Environment (DCE) ([11]) на Open Software Foundation, Common Object Request Broker Architecture (CORBA) ([12]) на Object Management Group, COM/DCOM ([13]) архитектурата на Microsoft, както и Enterprise Java Beans (EJB) спецификацията ([14]) за J2EE (Java 2 Enterprise Edition) Application Server.



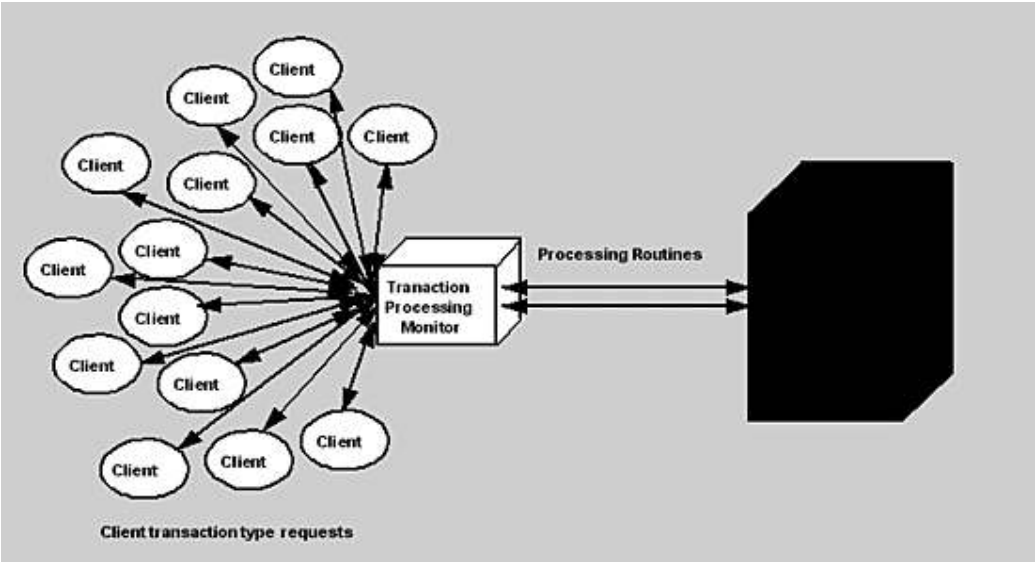
Фиг. 1 - Употреба на мидълуер компонентите

Според някои автори ([2]), както е показано и на Фиг. 1, *мидълуер компонентите са поредица от разпределен софтуер, който съществува между приложението и принадлежащата платформа на даден системен възел в мрежата.*

Основната идея на мидълуер компонентите е да се повиши нивото на абстракция при разпределените системи. Те предоставят функционален комплект от интерфейси за създаване на приложения (Application Programming Interfaces - API), които трябва да бъдат независими от мрежовите ресурси, надеждни и с гарантирана наличност, както и с *разширяем капацитет без загуба на функционалност* ([3]).

Има няколко основни вида мидълуер технологии според начина на използването им и функцията, която изпълняват в разпределените системи:

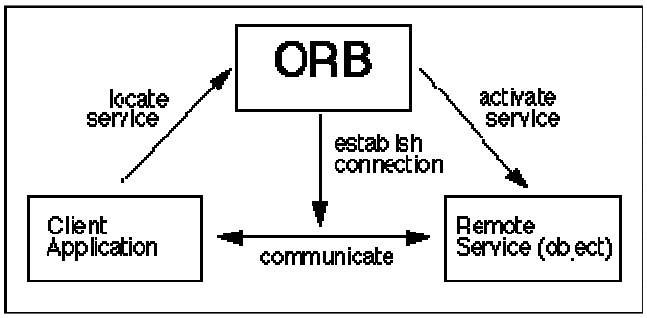
- Средства за управление на бизнес процеси - Transaction Processing (TP) monitors. Тази технология предоставя на разпределените клиент/сървър системи възможността по *ефикасен и надежден начин да изграждат, изпълняват и управляват транзакционни бизнес процеси* ([15]). Чрез нея могат да се предоставят услуги на множество клиенти чрез пренасочване на техните заявки (в зависимост от начина на последващата им обработка) към съответните сървърни входни точки (Фиг. 2).



Фиг. 2 – Мидълуер средства за управление на транзакции

Употребата на подобни приложения се явява ефективна от гледна точка на разходите алтернатива на надграждането на системите за управление на бази данни или на платформените ресурси, които биха могли да изпълняват подобна роля. Такъв вид решения се използват в системите за управление на данни, за достъп до мрежи, системите за сигурност, при резервации на самолетни билети и т.н.

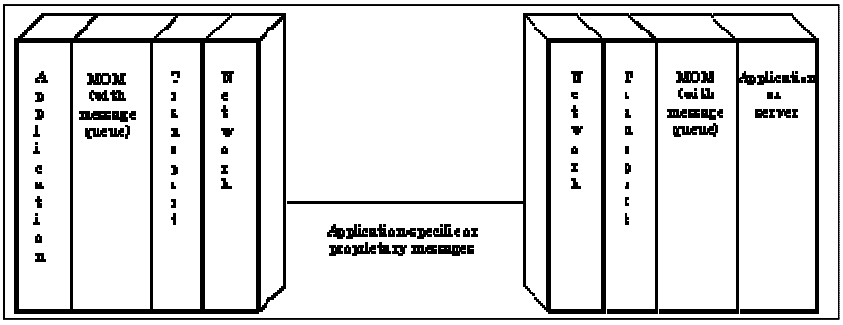
- Отдалечени процедурни извиквания – Remote Procedure Call (RPC), които позволяват на логиката на приложението да бъде мрежово разпределена и да не е зависима от физическото местоположение на даден ресурс. По същността си този вид мидълуер е може да се дефинира като *клиент/сървър инфраструктура, която увеличава платформената преносимост и гъвкавост на приложението, позволявайки му да бъде разпределено на множество различни операционни системи* ([16]). Също така подпомага намалява сложността при разработване на приложения, защото изолира разработчика от детайлите на различните операционни системи и мрежови протоколи. Първите подобни напълно функционални системи се появяват към края на 70-те и началото на 80-те години на миналия век.
- Посредник на извиквания към обекти – Object Request Broker (ORB), позволяващ компонентите, съставящи дадено приложение, да комуникират и обменят данни помежду си (Фиг. 3). По този начин се улеснява взаимодействието между разпределените системи, защото брокерите позволяват изграждането на приложения чрез съединяване на компоненти (в общия случай изработени от различни производители), които могат да си предават информация чрез посредника ([17]).



Фиг. 3 – Посредник за извиквания между обекти

Важно е да се отбележи, че детайлите при изграждането на съответният посредник не са от особено значение за разработчика, единственото, което има значение за него, е интерфейса на самите обекти.

- Мидълуер, ориентиран към прехвърлянето на данни между системите – Message-Oriented Middleware (MOM), схематично представен на Фиг. 4. Тези компоненти са аналогични на системите за изпращане на електронна поща в смисъла, че *са асинхронни и изискват получателите на съобщенията да могат да интерпретират тяхното съдържание по подходящ начин* ([18]).



Фиг. 4 – Мидълуер за изпращане и получаване на съобщения

Както ще видим в последствие, разработеното решение най-добре се вписва именно в контекста на последния вид мидълуер. Но заедно с това то предоставя само ограничен набор от характерните за мидълуера функции. Например не е независимо от мрежовите ресурси и не предоставя гаранции, че всяко входящо съобщение ще бъде получено от изходната бизнес система. Последното изискване е характерно за така наречените системи за обработка на опашки от съобщения – message queue (MQ), които представляват *временно хранилище за съобщенията в случай че*

предназначената за обработката им програма е заета или не е свързана с мрежата ([18]).

Основната цел на настоящата дипломна работа е да създаде ниво на абстракция между разнородните системи за обработка на търговски поръчки, и в този си контекст следва да бъде считано за мидълуер приложение.

2.2 Extensible Markup Language (XML)

XML (eXtensible Markup Language), както е известно, е абревиатура, означаваща “разширяем маркиращ език”. Но това в действителност не е език сам по себе си. Това е *метаезик, който се използва да създава други езици* ([4]). Чрез XML се създават структурирани, самоописващи се документи, които съответстват на комплект от правила, създадени за всеки специфичен език. XML осигурява съществуването на широко разнообразие от строго специфични езици. Като примерите могат да се посочат Mathematical Markup Language (MathML), Electronic Business XML (ebXML), Voice Markup Language (VXML) и други.

Едно от най-точните описания на терминът “маркиращ език” е *метод за предоставяне на мета - данни (информация за друго множество от данни)* ([5]). Освен маркиращите елементи, наричани още тагове (tags), които служат за представяне на данните, XML предоставя и структурирано съдържание под формата на елементи, атрибути, коментари и други, наричани общо възли (XML nodes). Този тип гъвкаво представяне на информацията позволява лесно да се изпращат и приемат данни, както и улеснява преобразуването им от един формат към друг.

Всеки XML език има своя собствена граматика, специфично множество от правила, определящи съдържанието и структурата на документите, написани на този език. Например, елемента <*price*> може да има смисъл в ebXML документ, но не и в MathML документ. Тъй като всеки език трябва да отговаря на дадени граматични правила, то XML осигурява улеснения за документиране на граматиката на специфичния език. Използвайки тези стандартни правила за валидация, всеки XML синтактичен анализатор може

да верифицира структурата на даден XML документ. Така стигаме до трите основни XML технологии при обработката на документи: проверка формата на данните (validation), четене - по-точно казано правене на синтактичен разбор (parsing), и превеждане в разбираем за системата формат (translation). Също така, с развитие на обектно ориентираното програмиране, все по-голямо разпространение получават и технологиите за двупосочно преобразуване структурата и съдържанието на XML документите към обекти в термините на съответния програмен език. Ще разгледаме накратко всяка една от тези технологии, за да се изясни тяхната употреба в настоящата дипломна работа.

2.2.1 Проверка формата на данните

Формата на предоставените данни може да бъде проверен посредством един от двата най-разпространени начина: чрез DTD (Document Type Definition) или XSD (XML Schema Definition).

Дефиницията на типа документ (DTD) е текстов файл, съдържащ описание на правилата относно структурата и съдържанието на XML документите. В него се изброяват коректните за дадения XML елементи, включително реда им на появяване и техните атрибути ([4]).

Въпреки че като първоначален опит за описание на структурата на XML документите DTD е доста успешен, той налага и множество ограничения. Основното е, че самата дефиниция на типа документ не е XML документ сама по себе си. Поради тази причина се налага тя да бъде обработвана по различен начин при четенето на документа, и освен това самата дефиниция подлежи по-трудно на проверка. Също така тези дефиниции не могат да се справят с усложняването на XML синтаксиса и в частност с конфликтите при използването на префикси на имената (namespaces) и при описанието на сложните връзки между елементите в документа и между документите като цяло. Поради тези причини World Wide Web Consortium (W3C) създава XML Schema (XSD) механизма, който трябва да замени гореописаните DTD.

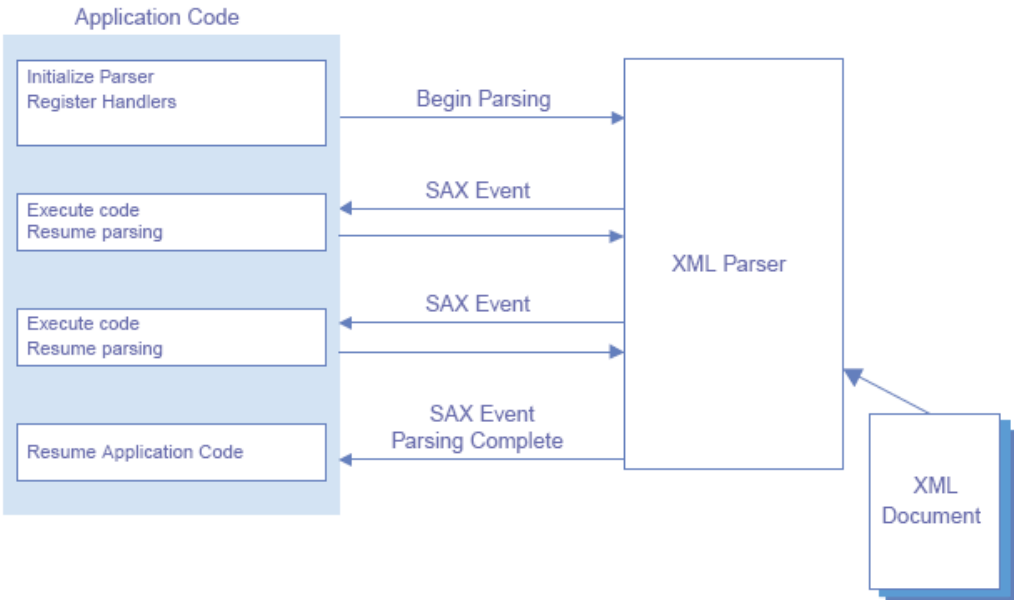
Най-точната им и ясна дефиниция ([4]) е че, *XML Schema Definition (XSD) е XML – базирана декларация на граматиката на XML документите*.

В настоящото приложение се използват и двата начина за проверка на постъпващите данни, в зависимост от специфичния входящ формат. Голяма част от текущите индустриални стандарти предоставят дефиниция на своя формат под формата на линеен текстов файл – DTD, което налага и нуждата от гъвкавост на приложението при валидиране на постъпващите данни.

2.2.2 Синтактичен разбор (четене) на данните

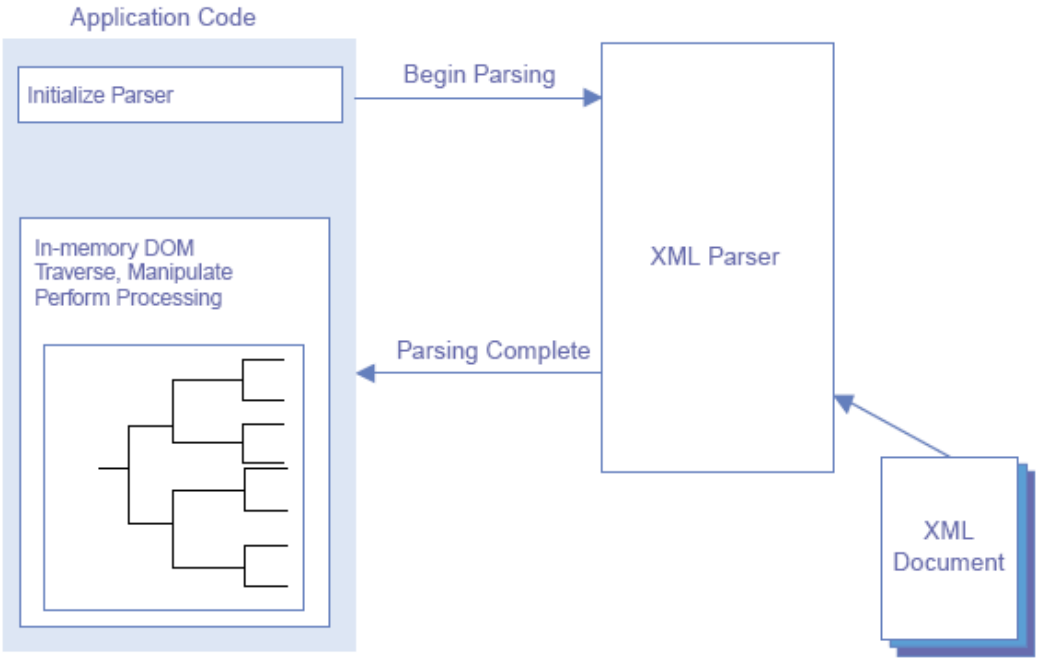
Преди да може да бъде валидиран и използван, всеки XML документ трябва да бъде прочетен. Това става с помощта на така наречените XML анализатори (XML parsers). Това са софтуерни компоненти, които могат да четат, а в повечето случаи и да валидират съдържанието на даден XML документ.

Голяма част от съществуващите XML анализатори поддържат и двата основни метода за прочитане на XML данни: SAX (Simple API for XML) - Фиг. 5 - и DOM (Document Object Model) - Фиг. 6.



Фиг. 5 - Употреба на SAX API

Първият метод е базиран на събития, получавани при четенето на XML документи, като например срещането на затварящ елемент. Това събитие достига до клиента, предоставяйки му информация в контекста на кой елемент е, кой е следващият и т.н. SAX е най-бързият начин за прочитане на данните, но се губи възможността да се оперира над документа като едно цяло, тъй като във всеки един момент разполагаме само с част от съдържанието. Като следствие от това ограничение е и липсата на директна поддръжка за родител - дете връзките в документа от самия XML анализатор.



Фиг. 6 - Употреба на DOM API

При алтернативната технология за прочитане на документ - DOM, XML анализатора прочита цялата информация в дървовидна структура в паметта. Приложението получава указател към корена на документа. Този метод е далеч по-лесен за имплементиране, но е и доста по-бавен и изискващ повече системна памет. Въпреки това има технологии, наследници на DOM, които позволяват употребата му и с големи входни данни.

В този аспект на XML технологиите в настоящата дипломна работа се използва DOM за прочитане на входящите данни от бизнес системите, защото е достатъчно бърз и надежден с оглед на поставените цели. Освен

това основната тежест на системата е при трансформирането на данните, за което се използват друг набор от технологии, които ще разгледаме по-подробно при анализа на решението.

2.2.3 Превеждане на данните

Едно от ключовите предимства на XML в сравнение с останалите формати за пренос на данни е възможността лесно да се преобразува даден формат в друг по общоприложим начин. Технологията, която позволява тази транслация, се нарича XSLT (eXtensible Stylesheet Language for Transformations) и дефинира *механизма за адресиране на XML данни и за трансформирането им в други форми* ([19]). Тя предоставя рамка за трансформиране структурата на даден документ, комбинирайки входящия XML с XSL stylesheet, за да продуцира изходящия документ.

Тази технология няма директно отношение към настоящата дипломна работа, затова няма да бъде разгледана в по-голяма дълбочина.

2.2.4 Обектна трансформация на данните

Развитието на езиците за програмирането и по-специално налагането на обектно ориентираните концепции доведе до създаването на технологии за представяне на XML данни чрез обекти. Тъй като настоящото мидълуер приложение е разработено чрез езикът Java, то ще се спрем на съответната технология в контекста на този език.

Въпреки че е сравнително нова, технологията JAXB (Java Architecture for XML Binding) бързо се налага като стандарт за преобразуване на XML към обекти и обратно. Тя предоставя програмен интерфейс, който позволява автоматично *двупосочно трансформиране между XML документи и Java обекти* ([20]). По дадена дефиниция на схема JAXB компилаторът може да генерира Java класове, които позволяват да се разработват приложения, способни да четат, манипулират и възпроизвеждат XML данни без да се грижат за детайлите при прочитане на входящия документ. Генерираният програмен код предоставя ниво на абстракция чрез публичен интерфейс,

който позволява достъп до XML съдържанието без да са нужни специфични знания за скритите на по-ниско ниво структури от данни.

Трябва да се отбележи, че JAXB архитектурата всъщност е следващо ниво от развитието на DOM и SAX синтактичните анализатори, като по този начин наследява много от техните функции и същевременно преодолява основните им недостатъци. Основни й характеристики са:

- Валидиране на данните - Всяко JAXB приложение следва стриктно дефиницията на съответния формат, и така не позволява създаването на невалидни Java обекти.

- Скорост - SAX е бърз благодарение на събитийно ориентирания си (event-driven) модел, и JAXB като негов наследник може да съгласува скоростта с механизма си за преобразуване на данните.

- Лесна употреба - JAXB компилаторът генерира код, който освобождава разработчикът от писането и проверяването на функции за прочитане формата на документа. Използвайки го, програмистът до голяма степен може да се концентрира върху бизнес логиката като до голяма степен се абстрахира от структурата на входящите данни.

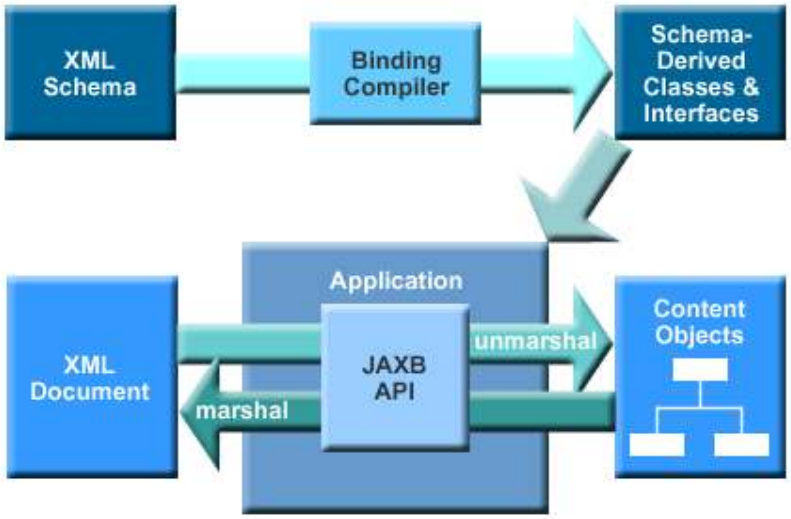
- Преобразуване типовете на данните – Типовете данни в XML автоматично се преобразуват в Java типове данни.

- Гъвкавост – Схемата за свързване между XML формата и съответните Java класове (т.нар. Binding schema) може да бъде променяна според нуждите на приложението.

- Разширяемост – Генерираните класове могат да бъдат разширявани (чрез наследяване) за добавяне на нови функции на приложението.

На Фиг. 7 е предоставена схема с описание на стъпките, които са нужни за конфигурирането и последващото преобразуване на входящият XML документ към съответните Java обекти, както и основните термини,

описващи всяко необходимо действие. Всъщност JAXB спецификацията описва и двата основни процеса: първоначалното конфигуриране, въз основата на което се генерират Java обектите и последващата им употреба от съответното приложение.



Фиг. 7 - Архитектура на JAXB

Основният входящ източник е схемата, описваща формата на входящия документ – XML Schema. Чрез нея JAXB компилаторът (означен по-горе като Binding compiler) получава информация какви Java класове и интерфейси да генерира – техните имена, полета, методи и връзките между тях.

От гледна точка на приложението има два основни процеса, които изискват използването на генерираните програмни единици. Първият е прочитането на входящите XML документи, което в термините на JAXB технологията се нарича ънмаршалинг (unmarshal). Това означава създаване на дървото от генерирани обекти, които представляват съдържанието и структурата на документа. Обектната йерархия не е на основата на DOM спецификацията, а е по-ефикасна при използването на системната памет. Вторият основен процес е преобразуването от Java обектния модел към XML документ. Това практически е обратния на гореописания процес. Трябва да се спомене, че и при двата процеса валидацията на данните може да се конфигурира, и е отговорност на програмиста да борави с тази опция.

Технологията JAXB е една от основните, използвани в настоящата дипломна работа. Чрез нея структурата на вътрешния формат се преобразува в Java класове, посредством които се създават документи във независимия интеграционен формат.

2.3 Индустриални стандарти за бизнес документи

Както вече беше споменато XML – базираните стандарти за описване на бизнес документи представляват една важна стъпка в развитието на електронния бизнес (e-business). Но за съжаление има твърде много такива формати, тъй като първоначалните усилия за стандартизация са имали склонността да бъдат по-скоро технически решения, породени от определени индустриални изисквания, или ориентирани към строго специфичен бизнес проект.

Тук ще се бъдат разгледани някои от по-широко разпространените XML стандарти.

2.3.1 RosettaNet

Организацията “RosettaNet” е основана с идеална цел. Тя представлява консорциум между влиятелни компании в областта на информационните технологии, електронните компоненти, телекомуникационни компании. Основната й цел е *да бъдат създадени и развити отворени стандарти за е-бизнес процеси, които да обхващат голям диапазон от промишлени отрасли. Тези стандарти трябва да формират общия език на електронната търговия* ([6]).

Основното понятие, на което се набляга при изготвянето на RosettaNet стандартите, е дефиниране интерфейса на партньорските процеси (PIP - Partner Interface Process). Три типа спецификации се изготвят за всеки съответен тип бизнес документ. Те са следните: описание на интерфейса на партньорските процеси, специфични указания за самия вид документ (Message Guideline) и DTD за проверка валидността на XML документа (при

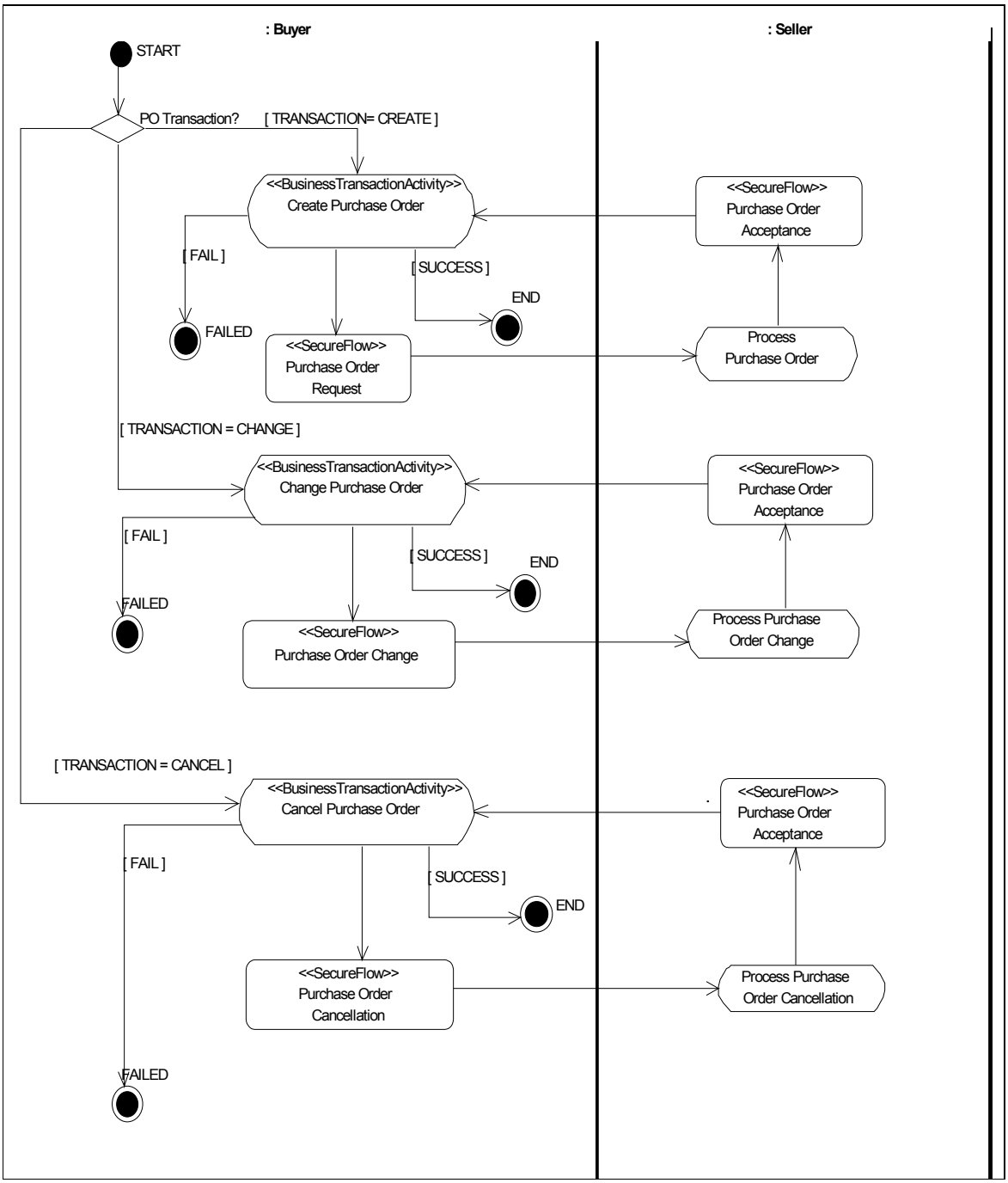
последните версии вече се използва XSD, какъвто е и случаят в настоящата дипломна работа). Спецификацията предоставя описание на процеса на размяната на съобщения, както и целта и ролите в съответния партньорски бизнес процес. Втория вид документ дефинира речника, структурата и позволените елементи за данни и съответните типове стойности за всяко съобщение в съответния бизнес процес. Дефиницията на типа документ, както вече беше изяснено по-горе (2.2.1), служи за проверка последователността на елементите, техните имена, композиция и атрибути.

В текущата дипломна работа се разглеждат основно документите за търговски поръчки, които в контекста на RosettaNet стандартите се обозначават като 3A4 PIP. Ще разгледаме версия 2.02.

Управлението на процеса на поръчки обхваща създаването, промяната и отменянето на поръчката. Купувачът или неговата организация първоначално създават документ за поръчка и изпращат заявка за изпълнението й към продавача. Той от своя страна потвърждава получаването на ордера, връщайки съответния бизнес документ. Тогава купувачът би могъл да инициира промяна или отменяне на поръчката. Има два типа документи, чрез които може да се промени заявката:

- Чрез оригиналният документ с промяна в съдържанието.
- Нов документ, който съдържа само промените на оригиналния документ. Разбира се, получавайки новия документ, трябва да може недвусмислено да се идентифицира първоначалната поръчка.

На Фиг. 8 е показана схемата на движение на документите при бизнес процесите от гледна точка на RosettaNet стандарта:



Фиг. 8 - Управление процесът на електронни поръчки при RosettaNet

В настоящата дипломна работа е осъществена трансформацията между RosettaNet 3A4 PIP и вътрешният за системата формат за търговски поръчки. В съпътстващият компакт диск е предоставена съответната дефиниция на 3A4 PIP схемата за валидация.

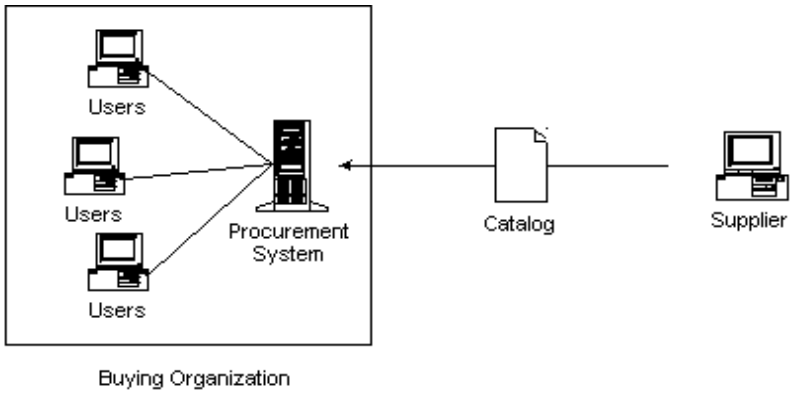
2.3.2 cXML

Стандартът cXML (Commerce eXtensible Markup Language) е отворен формат за описване на транзакциите при електронната търговия. Той позволява на купувачите, доставчиците и посредниците да комуникират помежду си, като използват само един, стандартен, отворен формат. Това е добре дефиниран, строен език, проектиран специално за бизнес-към-бизнес (B2B e-commerce) електроните портали, и е изборът на големи като обем търговци.

При cXML транзакциите се състоят от добре дефиниран процес на размяна на документи, които са текстови файлове в предварително дефиниран формат. Тези документи са аналогични на хартиените, които традиционно се използват в търговията.

Има три основни типа cXML документи:

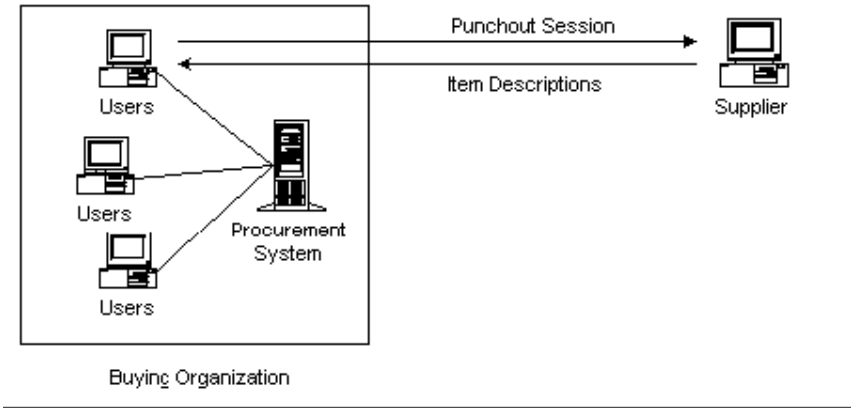
- Каталози (Фиг. 9) – това са файлове, които предоставят списък от продукти и услуги на купувачите. Чрез тях се описват продуктите и услугите, заедно с техните цени и други основни характеристики. Каталозите са основния комуникационен канал между търговските партньори, чрез тях купувачите получават актуална информация за предлаганите от дадения доставчик продукти и услуги.



Фиг. 9 - cXML каталог

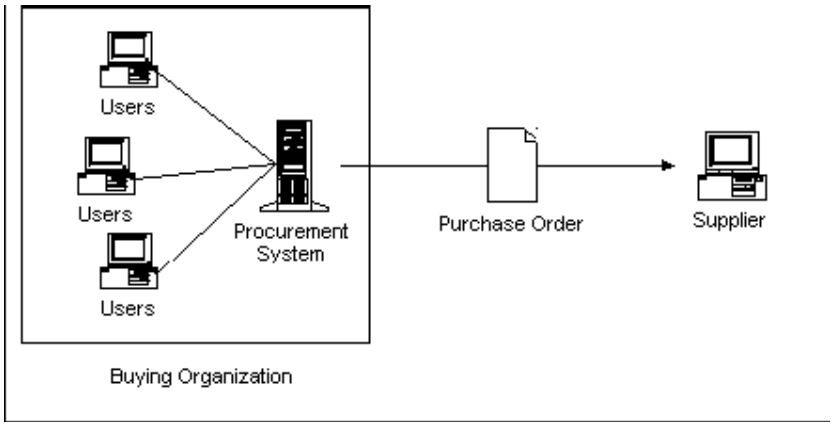
- Динамичен каталог (Punchout) – както подсказва и името този вид документ е аналог на статичните каталози. По своята същност тези

документи представляват страници в Интернет пространството с интерактивни каталози. Те самите комуникират със системите за доставка чрез cXML като по този начин улесняват клиентите на приложенията (Фиг. 10).



Фиг. 10 - cXML динамичен каталог

- Търговски ордери (Purchase Orders) – купувачите изпращат търговски поръчки към доставчиците, изисквайки изпълнение на даден договор (Фиг. 11).



Фиг. 11 - cXML търговски поръчки

В контекста да настоящата тема най - важен тук е именно последния вид документи. Той се дефинира като *заявка към доставчика да изпълни формалния договор, сключен с купувачът* ([7]). Специфично за cXML транзакциите е наличието на два подтипа търговски поръчки – изпращат се “заявки за поръчки” (OrderRequest) и съответно им се отвръща с “отговори

на поръчки" (OrderResponse). Първия вид е аналогичен на търговските поръчки при RosettaNet. Вторият вид се явява потвърждение, че е получена търговската поръчка. Но той сам по себе си не означава, че е поето задължение да се изпълни поръчката, а само удостоверява нейното получаване.

Друго важно разширение на cXML стандарта е възможността да се прибавят други документи към търговките документи, например асоциирани бележки, рисунки, факсове. За тази цел се използва MIME (Multipurpose Internet Mail Extensions) протокола. В такъв случай cXML документа съдържа указател към външния приложен документ, който се изпраща отделно, по електронната поща или чрез факс.

За настоящото приложение cXML, версия 1.1, се явява втората система, която се трансформира към вътрешния стандарт. Тя е интересна също така с факта, че позволява входящите системи да се валидират чрез DTD, описана в 2.2.1. Този документ е приложен в съпътстващия компакт диск.

2.3.3 xCBL

Абревиатурата xCBL означава XML Common Business Library (обща бизнес библиотека за XML). Представлява множество от XML документи и техните компоненти, безплатно предлагани на http://www.xcbl.org.

Последните версии на xCBL се предоставя само под формата на XML Schema Definition като форма на валидиране, но при предните версии са се използвали множество по-специфични формати като SOX (език за схеми на Commerce One) схема, W3C XML Schema, комплект от XDR схеми и други. Това многообразие, обусловено от по-силното обвързване с даден производител, е довело до не толкова широкото разпространение на този стандарт. Въпреки формата за структурна валидация синтаксисът на самите документи не е претърпял значими корекции.

Едно от най-големите достижения на xCBL е *възможността за подпомагане взаимодействието на разнородни системи (interoperability) при размяната на бизнес съобщения* ([8]). С последната версия, 4.0,

започва използването на езика за схеми XSDL, които се очертава като нова тенденция в развитието на електронната търговия. Също така е съгласуван и с инициативата Universal Business Language (UBL), която е друг опит в посока към унифициране на бизнес стандартите. Кратко описание на този все още нов формат е дадено в точка 8.3.

Този стандарт не се различава особено от другите в дефиницията на бизнес процесите и съответните им документи. Поддържат се следните основни вида документи:

- Поръчка (Order) – електронно съобщение, което инициира търговски ордер. За разглежданата версия това е Order документ (PurchaseOrder за xCBL, версия 2.x).

- Отговор на поръчка (OrderResponse) – съобщение, което е отговор на горния документ. За xCBL това е OrderResponse документ.

- Прекъсване на поръчка (OrderCancel) – съобщение за прекъсване на търговска поръчка. При xCBL това е или Order, или ChangeOrder документ.

- Промяна на поръчка (ChangeOrder) – документ, който съдържа промени на оригиналната поръчка или на предишен ChangeOrder документ.

Трансформация от cXML стандарта към вътрешния стандарт, използван от дипломната работа, не е имплементиран в настоящата версия на приложението. Това решение е продиктувано от липсата на съществени за целта на настоящата дипломна работа различия с разгледаните RosettaNet и cXML стандарти, както и от ограничения обем на дипломната работа.

2.3.4 OAGIS

Open Applications Group Integration Specification (OAGIS), създадена в края на 1995 година, е *отворена, напълно независима организация, фокусираща се върху нарастване на взаимозаменяемостта сред бизнес приложенията*

и създаването на стандартен бизнес език, който да поддържа тази цел ([9]).

За да е възможно да получи интеграция и сътрудничеството между коренно различни системи, OAGIS се стреми да изрази общата, хоризонтална архитектура, на която се подчиняват всички системи. Тя се нарича Business Object Document (BOD). Тези документи може да се характеризират като бизнес съобщения, които се разменят между софтуерните приложения или компоненти, между компаниите, между различните веригите за доставки и вътре в отделните такива. Тъй като BOD стандарта трябва да е независим от комуникационните канали – HTTP, SMTP (Simple Mail Transfer Protocol), SOAP (Simple Object Access Protocol), ebXML Transport and Routing – той, за разлика от вече описаните стандарти, може да предоставя статуса на комуникацията и различни кодове на грешки. BOD документите може да бъдат обрисувани като съвкупност от следните елементи:

- Област на приложението (Application Area) – служи за предаване на информация която може да бъде използвана от комуникационния канал да изпрати даденото съобщение.

- Област на данните (Data Area) – носи специфичния за бизнеса смисъл или данни, които се предават през BOD.

- Глаголи (Verbs) – идентифицират действията, които се извършват върху дадените “Съществителни” от BOD документа. Глаголите носят разширяемия смисъл на документите, за да може възможно най-пълно да се обхванат вертикалните характеристики на дадения бизнес.

- Съществителни (Nouns) - предоставят бизнес специфичните данни, т.е. дали съобщението е търговска поръчка (PurchaseOrder), поръчка за продаване (SalesOrder), определяне на пазарна цена (Quote), информация за доставка (Shipment) и т.н. Състоят се от “Компоненти”, описани по-долу.

- Компоненти (Components) – това са разширяемите, изграждащи блокове на дадено “Съществително”, например заглавната информация на дадена търговска поръчка (PurchaseOrder Header), информацията за отделен елемент от поръчката (PurchaseOrder Line) и т.н. Те, от своя страна, се състоят от “Полета” и “Сложни елементи”.

- Сложни елементи (Compounds) – основни елементи на BOD документите, които могат да се споделят от всички документи. Такива са например “Количество” (Quantity), “Общ размер” (Amount) и други.

- Полета (Fields) – елементите от най-ниско ниво, дефинирани в OAGIS. Това са фундаментални елементи като “Име” (Name), “Описание” (Description), които се използват за създаването на сложни елементи и компоненти.

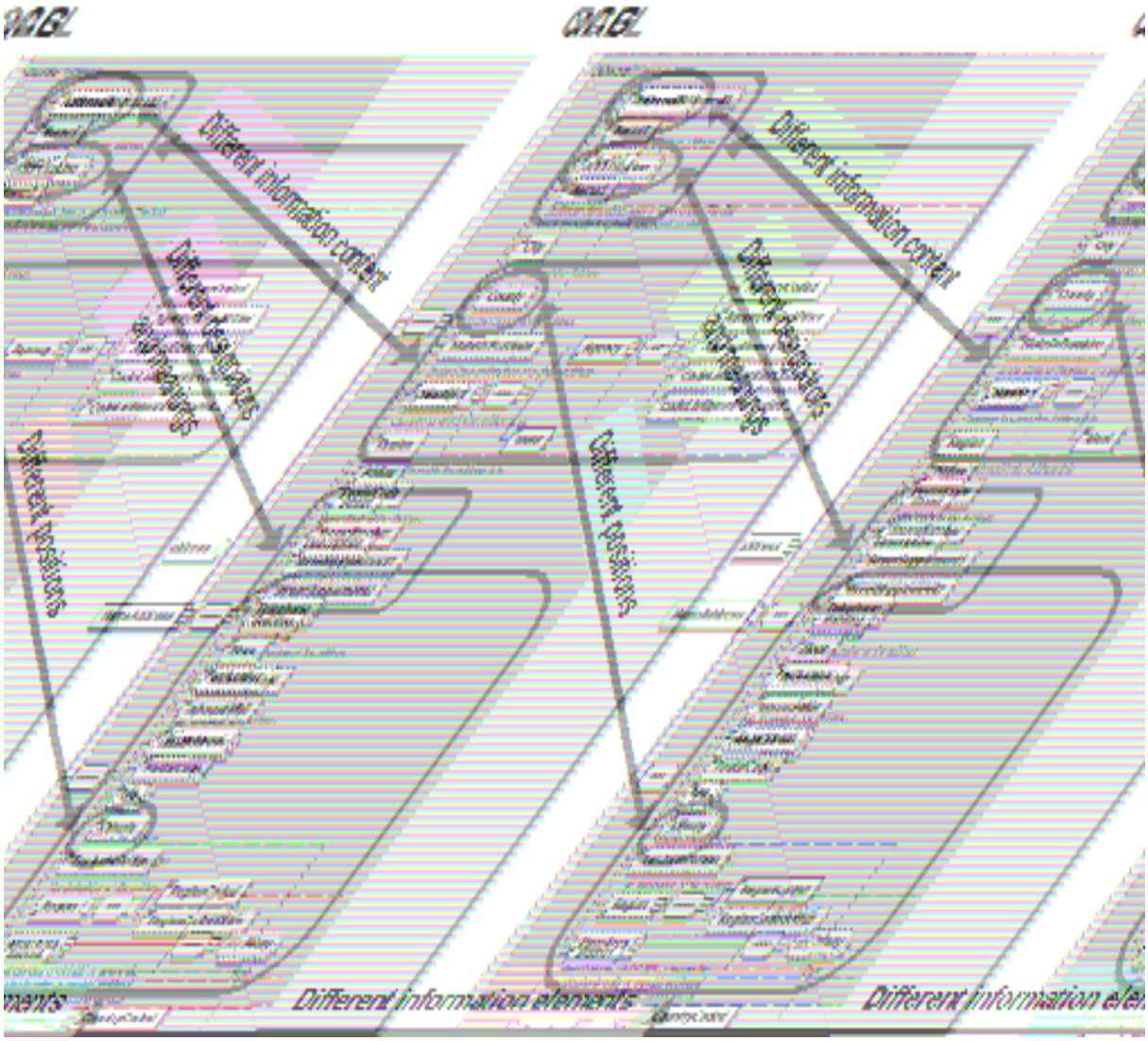
Трансформацията от OAGIS формата не е имплементирана в настоящата дипломна работа поради ограничения й обем.

2.3.5 Изводи

Трябва да се отбележи, че описаните по-горе формати съвсем не представляват пълно множество от използваните в момента стандарти при бизнес към бизнес (B2B) комуникацията. Има и други XML стандарти (например HR-XML), както и по-стари, линейни формати, познати под общото название EDI (Electronic Data Interchange) документи.

Въпреки това прави впечатление, че по същество всички разгледани стандарти описват по много сходен начин бизнес процеса на електронната търговия, и това е разбираемо, тъй като на практика процеса е един. Главната разлика идва от формата на описваните данни, който също е доста близък, и на места се различава само по последователността на елементите, както се вижда на долната диаграма (Фиг. 12). До голяма степен тези различия са обусловени от историческия произход на всеки един

формат, и по-точно от специфичния бизнес сегмент, към който е бил насочен при проектирането си.



Фиг. 12 - Разлики между XML формати

Горните проучвания доказват, че създаването на общ XML формат за бизнес транзакции, който да изолира приложенията от необходимостта да интерпретират всички съществуващи стандарти, е много важна дейност и ще намали до голяма степен себестойността на изработване на системи за комуникация между специфичните решения за електронен бизнес. Именно това е и целта на настоящата дипломна работа.

3 Модел на решението

Целта на настоящата дипломна работа е да се разработи помощно приложение, което да трансформира разнообразните индустриални формати за бизнес документи в общ, вътрешен формат. Такова решение трябва да позволява лесна интеграция между вече съществуващите стандарти, както и да подпомага добавянето на нови външни системи. Една от основните му цели е и да капсулира детайлите при получаването на входните данни, както и последващото изпращане на резултата, предоставяйки удобен интерфейс за системите, които ще бъдат разработени като потребители на неговите услуги.

Непосредствено от поставените цели следват двете основни характеристики на предлаганото решение:

- Разработване на приложението като мидълуер система. Налага се от изискването да предоставя ясно дефинирани интерфейси за системите, с които ще си взаимодейства. Това са на първо място входните системи. Те ще се грижат за фактическото получаване на търговските документи. За тях трябва да се предостави точно определена входна точка, за да могат да изпращат реалните данни на разглежданото преобразуващо приложение. На второ място това са системите за обработка на документите в термините на съответният бизнес. За тях е жизнено важно да получават данните в предварително договорен формат, който трябва да се променя много рядко, за да се минимизират разходите по тяхната поддръжка. Както вече бе споменато при теоретичната обосновка на проблема (и по-точно в точка 2.1) разглежданата система най-добре се вписва във вида мидълуер, ориентиран към прехвърлянето на данни между системите, без да предоставя цялостната му функционалност.

- Използването на XML за представяне на търговските документи. Налага се от съвременните тенденции като стандартен формат при предаване на данни, и затова използването на XML намира все по-

голямо приложение в съвременните софтуерни системи. Това се дължи както на удобния, йерархичен, лесно разширяем и разбираем синтаксис, така и на възможността да се валидира и верифицира формата и съдържанието на данните. Също така употребата му в настоящата система следва непосредствено от факта, че всички входни системи (с изключение на линейните файлови формати, които се поддържат само за съвместимост със стари системи) изпращат електронните документи в под формата на XML данни.

При така поставената постановка на задачата става ясно, че основната тежест при изработката на решението е определянето на общите части на изследваните формати, както и възможността да се направи вътрешната схема по начин, така че тя да бъде възможно най-лесно разширяема. Заедно с това се вижда, че пълно решаване на задачата е трудно постижимо поради огромното разнообразие от съществуващи стандарти за предаване на данни при електронния бизнес. Оттук произлизат и основните характеристики, с които трябва да се съобразява проектирането на предлаганото решение. Това са гъвкавост и лесна разширяемост в два основни аспекта – добавяне на други входни стандарти (освен имплементираните RosettaNet и cXML) и добавяне на други типове документи, които трябва да могат да се интерпретират, освен електронните поръчки. Трябва да се направи уточнението, че измежду разнообразните документи, използвани при бизнес към бизнес комуникацията, ордерите като тип документ е избран като най-подходящ за първоначалната версия на системата, защото е основополагащ елемент при е-бизнеса и на него се базират останалите типове документи като електронните фактури, потвържденията за получаване на поръчките и други. От своя страна RosettaNet и cXML са двата преобразувани формата поради факта, че те имат най-голямо разпространение (което с особена сила важи за RosettaNet) и като цяло представляват пълно множество от начини за валидация на XML: RosettaNet чрез XSD и cXMLчрез DTD.

За по-точно описание на създадената система тя може да се раздели на четири части, които, макар и свързани, предоставят различни гледни точки

към приложението. Това са базата от данни, вътрешния формат за представяне на търговските поръчки, изходният код на мидълуер приложението и средата за тестване (на отделните модули и на системата като цяло). Те ще бъдат представени в отделни точки от настоящото изложение, като едновременно с това ще се изяснява и начина, по който те си сътрудничат при изграждане на цялостното решение.

4 Дизайн на базата от данни

Като сървър за бази данни в настоящото приложение се използва PostgreSQL 7.3.1 поради факта, че това е безплатен софтуер, който поддържа минимума от функции, необходими на приложението. По-точно това са възможността сървъра да поддържа процедури, записани в базата и освен това да предлага поддръжка на изпълнението на няколко SQL DML (Data Manipulation Language) извиквания към базата в един и същ контекст, т.е. така наречените транзакции. За процедурите, записани в базата, при PostgreSQL съществува специален език, наречен PL/PG SQL.

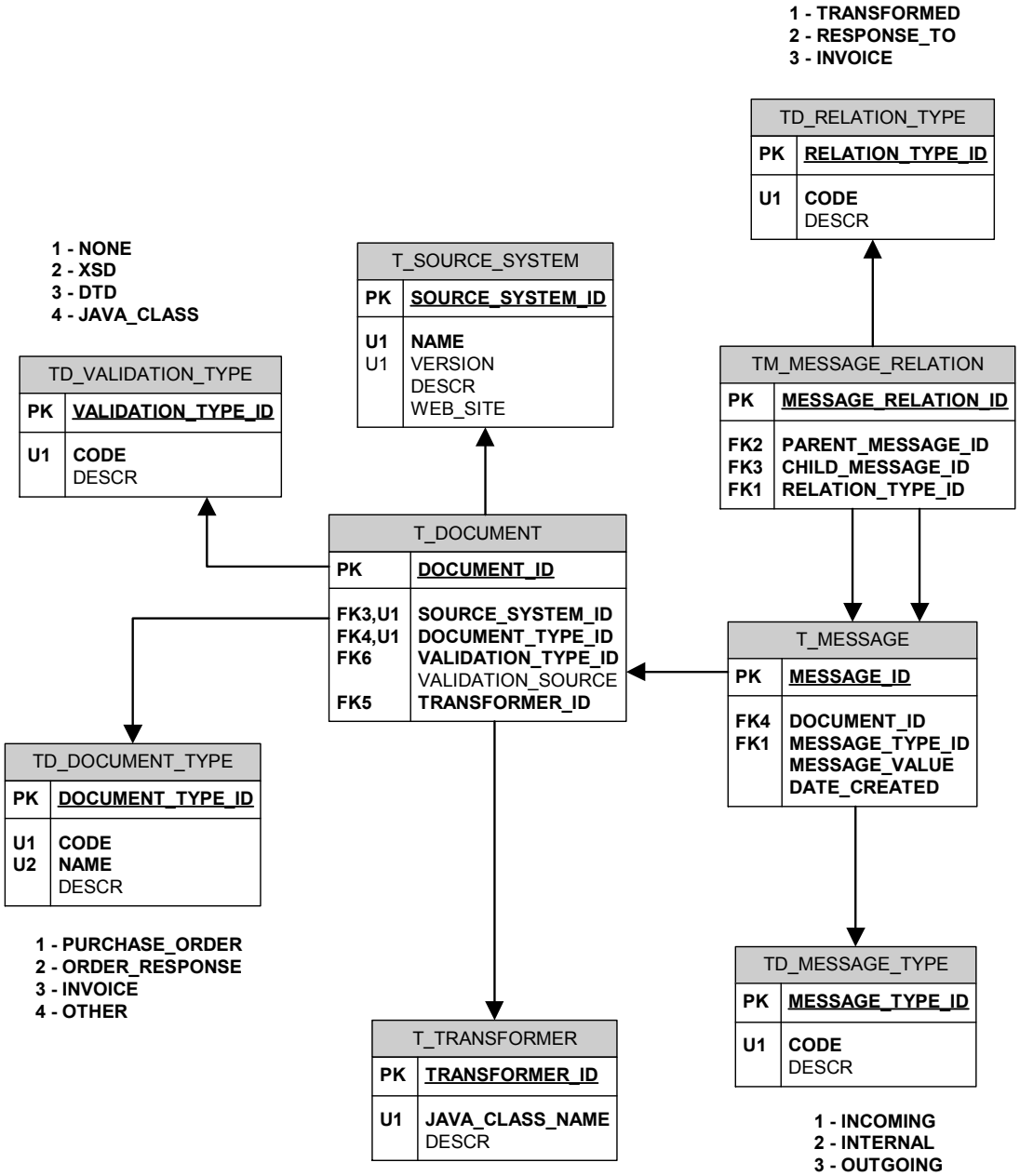
Всъщност, поради факта че PostgreSQL поддържа SQL99 стандарта, сървъра за бази данни би могъл лесно да бъде подменен с някой комерсиален, например Oracle, който би позволил скалируемост и лесно разширяване на базата. Тук трябва да се отбележи и технологията за достъп до бази от данни в Java, наречена JDBC (Java Database Connectivity), която позволява на разработчика да се изолира (до голяма степен) от спецификата на съответната база от данни, стига само тя да предоставя нужния JDBC интерфейс. Този аспект на приложението ще бъде разгледан по-подробно при представяне на самия програмен код.

По-долу е показан дизайнът на базата от данни. Основната тенденция при неговото проектиране е да е съобразен с духът на цялостната разработка, т.е. да е достатъчно гъвкав и лесно разширяем. Също така при проектирането на базата се спазват някои конвенции за именуване на обектите, както и за структурата на всяка таблица, които също са описани по-надолу в изложението. Смисълът на подобен вид конвенции за именуване (подобни са следвани и при проектиране на схемата, и при дизайна на Java приложението) помагат да се използва общ език между всички въвлечени в създаването на дадена система, и също така спомага за по-бързото интегриране на ресурси в рамките на дадена група софтуерни разработчици.

4.1 Таблици в базата данни

В тази точка ще разгледаме пълният списък от таблици, проектирани за нуждата на приложението, както и ще бъде разгледана конвенцията при образуване на имената им.

На Фиг. 13 е представен дизайна на базата данни:



Фиг. 13 - Дизайн на базата от данни

В настоящата база от данни съществуват три типа таблици:

- Таблици, представляващи рядко променяща се информация, т.нар. домейн (domain) таблици. Това са таблици, които не се променят при нормалното функциониране на цялостната система и съдържат множество от стойности, представляващо определен, фиксиран аспект на приложението. Такива таблици нормално се администрират през отделна система, при строго определени правила. Освен това при много приложения е прието данните от тези таблици да се съхраняват в паметта на програмата за бързодействие, разчитайки на тяхната неизменяемост. В настоящата система имената на тези таблици имат префикс “TD_”, което позволява лесното им разграничаване от останалите обекти в базата. В общия случай имат три колони: идентификатор – първичен ключ, код (също уникално поле) и описание, което се използва за изясняване съдържанието на съответния запис.

- Таблици, които представляват бизнес обекти (entity tables). Тези таблици отразяват движещите елементи на системата, които са динамични като структура и носят основната функционалност на приложението. Конвенцията тук е техните имена да са с префикс “T_”. Понеже изразяват динамичното поведение на системата при тях не може да се открие какъвто и да било шаблон. Но тъй като за настоящото приложение е прието като конвенция всяка таблица да има идентификатор на записа (първичен ключ), то и при таблиците от разглеждания вид се наблюдава такова поле.

- Таблици, които представляват връзки между бизнес обектите (many-to-many relations). Съдържат първичните ключове на двата участващи в релацията обекта. При настоящия дизайн техните имена започват с префикса “TM_”. Обикновено съдържат първичен ключ, външни ключове към обектите от релацията и понякога допълнителна информация като например датата на създаване на даденото отношение.

Също така трябва да се отбележи, че за по-голяма яснота в горната диаграма имената на колоните, които не могат да съдържат празни стойности (NOT NULL) са с удебелен шрифт.

Всяка от тези таблици ще бъде описана в семантичния й контекст, който допринася за функционалността на системата.

4.1.1 TD_VALIDATION_TYPE

В тази таблица се описват познатите на системата формати за валидация. Има следната структура:

- VALIDATION_TYPE_ID – числово поле, първичен ключ (primary key) за таблицата
- CODE – текстово, уникално поле, където се записват кодовете на типовете валидация
- DESCR – текстово, незадължително поле, където може да се запише описание на съответния тип валидация.

При текущата версия на системата тази таблица съдържа четири записа, като два типа валидация се използват от системата, а останалите са въведени с цел разширяемост. Двата типа валидация, които са приложими към XML, както вече бе споменато, са посредством схеми (CODE='XSD') и тип документ (CODE='DTD'). Другите два записа, които за момента не се поддържат, са документ без валидация (CODE='NONE') и Java клас, чрез който би могъл да се провери структурата на дадено входно за системата съобщение (CODE='JAVA_CLASS'). Последния тип валидация би могъл да се използва при проверката на постъпващи поръчки под формата на линейни текстови файлове.

4.1.2 TD_DOCUMENT_TYPE

Тук се описват типовете входни от гледна точка на системата документи. Структурата на таблицата е:

- DOCUMENT_TYPE_ID – числово поле, първичен ключ за таблицата
- CODE – текстово, уникално поле, където се записват кодовете на съответния документ
- NAME – текстово, уникално поле, представлява името на типа документ. Понеже записите от тази таблица се предлагат на потребителя за избор от тестовото Web приложение, то в това поле се записва името на типа документ по подходящ за визуализиране начин.
- DESCR – текстово, незадължително поле, където може да се запише описание на типа документ

Системата за интеграция на документи понастоящем трансформира единствено търговски поръчки като входящи съобщения, но в тази таблица са въведени още отговори на поръчки (CODE='ORDER_RESPONSE'), фактури (CODE='INVOICE') и други (CODE='OTHER').

4.1.3 TD_RELATION_TYPE

В тази таблица се описват връзките между съответните документи.

- RELATION_TYPE_ID – числово поле, първичен ключ за таблицата
- CODE – текстово, уникално поле, където се записват кодовете на връзките между документите, обработвани от системата.
- DESCR – текстово, незадължително поле, където може да се запише описание на съответната релация.

Настоящата система записва документите само като трансформирани (CODE='TRANSFORMED'), но при последващо развитие на приложението могат да се използват още множество релации, например отговор на (CODE='RESPONSE_TO'), фактура към (CODE='INVOICE_TO'), част от ('PART_OF'), когато документа е разделен на два или повече документи,

примерно поради желаната изходна дестинация на съобщенията, и други, в зависимост от функционалните изисквания към системата.

4.1.4 TD_MESSAGE_TYPE

Тук се записват типовете документи в зависимост от етапа им на обработка от гледна точка на приложението. Има следната структура:

- MESSAGE_TYPE_ID – числово поле, първичен ключ за таблицата
- CODE – текстово, уникално поле, където се записват кодовете на етапите от обработка на съобщенията.
- DESCR – текстово, незадължително поле, където може да се запише описание на съответния тип запис.

От гледна точка на системата постъпващите документи могат да бъдат в един от следните три стадия на обработка: постъпващи (CODE='INCOMING'), трансформирани във вътрешния стандарт (CODE='INTERNAL') и излизащи от системата (CODE='OUTGOING'). Третия формат все още не се използва, но за в бъдеще в него могат да се записват съобщения, които съдържат информация как да се прехвърли съответния документ към изходната система. Трябва да се отбележи, че тези три етапа представят "живота" (life-cycle) на документа в контекста на описваната система.

4.1.5 T_SOURCE_SYSTEM

В тази таблица се описват входните системи, данни от които могат да бъдат трансформирани от настоящото приложение. До известна степен и тя би могла да се разглежда като домейн таблица, защото промените в нея (добавяне на нови и премахване на вече излезли от употреба външни системи) ще се извършва през големи интервали от време. Но в настоящия етап от развитието на система тя е по-скоро динамична, защото има още много стандарти, които биха могли да се трансформират. Структурата на таблицата е следната:

- SOURCE_SYSTEM_ID – числово поле, първичен ключ за таблицата
- NAME – текстово поле, където се записва името на съответната система
- VERSION – текстово, незадължително поле, където може да се запише текущата версия на дадената система.
- DESCR – текстово, незадължително поле, съдържащо кратко описание на съответната версия на дадената система.
- WEB_SITE – текстово поле, съдържащо уеб адреса на дадената система

Полетата NAME и VERSION образуват уникален за таблицата ключ. Така проектирана таблицата позволява едновременното обработване на повече от една версия на даден формат, което е често наблюдавано явление в реалните софтуерни системи, тъй като за обратна съвместимост често се налага да се поддържат няколко по-стари версии. Още повече, че при описаните стандарти за обработка на електронни търговски документи разликите между версиите биха могли да бъдат много съществени (например по-старите версии да се проверяват синтактично чрез DTD, а съвременните – чрез схеми). Такъв дизайн води и до нормализация на базата данни.

В настоящия момент компонента за трансформиране подържа само две системи – RosettaNet, версия 02.02 и cXML, версия 1.0.

4.1.6 T_TRANSFORMER

Една от най-важните от функционална гледна точка таблици в системата. Служи за записване на Java класа, който практически извършва преобразуването от входящия към вътрешния стандарт. Обекти от дадения клас се създават динамично, използвайки Java Reflection API, което ще бъде разгледано впоследствие. Поради тази причина всички класове, които извършват дадена трансформация, трябва да наследяват един стандартен интерфейс, *org.fmi.bdi.transformer.ITransformer*, за да е възможно през

неговите методи да се извърши самото преобразуване на данните. Този аспект ще бъде разгледан по-обстойно при описанието на мидълуер компонента.

- TRANSFORMER_ID – числово поле, първичен ключ за таблицата
- JAVA_CLASS_NAME – текстово, задължително поле, където се записва пълното име (пакета и името на класа) на съответния преобразуващ Java клас
- DESCR – текстово, незадължително поле, съдържащо кратко описание на трансформиращия клас

За момента са имплементирани два преобразуващи класа за търговски поръчки – един за RosettaNet и другия за cXML.

4.1.7 T_DOCUMENT

Таблицата T_DOCUMENT играе основна роля в разглежданата база от данни. Тя служи за нормализиране структурата на базата и на практика репрезентира основните елементи, обработвани в системата – електронните документи.

- DOCUMENT_ID – числово поле, първичен ключ за таблицата
- SOURCE_SYSTEM_ID – числово, задължително поле, външен ключ към таблицата T_SOURCE_SYSTEM. Показва входната система на дадения документ
- DOCUMENT_TYPE_ID – числово, задължително поле, външен ключ към таблицата TD_DOCUMENT_TYPE. Показва типа на документа – поръчка, фактура, т.н.
- VALIDATION_TYPE_ID - числово, задължително поле, външен ключ към таблицата TD_VALIDATION_TYPE. Показва начина за проверка структурата на входния документ, ако изобщо документа подлежи на валидация

- VALIDATION_SOURCE – текстово, незадължително поле. Неговата семантика трябва да се разглежда в контекста на предходното поле – ако начина за валидация е чрез DTD или XSD, то тук се записва относителния път във файловата система към съответния източник на валидация. Ако проверката се извършва чрез Java клас, то тук се записва пълното му име, а ако не трябва да се проверява валидността, то това поле не съдържа никаква стойност

- TRANSFORMER_ID - числово, задължително поле, външен ключ към таблицата T_TRANSFORMER. Показва чрез кой клас ще се преобразува дадения документ

При текущия вариант на приложението се тази таблица съдържа 2 записа – първия представлява търговските поръчки от RosettaNet, а другия – тези от cXML.

4.1.8 T_MESSAGE

В таблицата T_MESSAGE се записват всички документи, преминали през интеграционната система. По този начин се съхранява информацията за целият поток от документи. Съхраняват се не само входящите съобщения, но също така и трансформираните, а за в бъдеще е възможно да се записват и изходните.

- MESSAGE_ID – числово поле, първичен ключ за таблицата

- DOSUMENT_ID – числово, задължително поле, външен ключ към таблицата T_DOCUMENT. Показва от какъв тип е дадения документ – дали е търговски ордер от RosettaNet, фактура от OAGIS и т.н.

- MESSAGE_TYPE_ID – числово, задължително поле, външен ключ към таблицата TD_MESSAGE_TYPE. Показва етапа от жизнения цикъл на съобщението от гледна точка на разглежданата система – дали е новопостъпил, във вътрешен формат или предстои да бъде изпратен към съответният компонент за бизнес обработка.

- MESSAGE_VALUE – текстово, задължително поле. Съдържа самото съобщение, което в зависимост от етапа му на обработка може да е XML или обикновен текст (ако е входно съобщение под формата на линейни текстови данни). Най-подходящия тип за това поле е CLOB (Character Large Object) SQL92 типа.

- DATE_CREATED – служебно поле, представлява датата на въвеждане на съответния запис, може да се използва за хронологична подредба на съобщенията

За в бъдеще тази таблица трябва да съхранява и изходните документи, както и да отразява допълнителни, междинни, но важни стадий от живота на документите в системата.

4.1.9 TM_MESSAGE_RELATION

Това е единствената таблица от вида “много-към-много”. Тя описва връзките между документите в системата.

- MESSAGE_RELATION_ID – числово поле, първичен ключ за таблицата

- PARENT_MESSAGE_ID – числово, задължително поле, външен ключ към таблицата T_MESSAGE. Указател към първия от участващите в релацията документи.

- CHILD_MESSAGE_ID – числово, задължително поле, външен ключ към таблицата T_MESSAGE. Служи за идентифициране на другия документ от релацията

- RELATION_TYPE_ID – числово, задължително поле. Съдържа вида на взаимоотношението между дадените съобщения

Въпреки че имената на колоните подсказват, че единия от документите е водещ в релацията, с по-голям приоритет, това на практика не е така. Двата документа са равнопоставени, имената на колоните спомагат само да се уточни посоката, в която се извършва действието – от родителя към детето.

В настоящия вариант се използва единствено релацията “трансформиране”, като родителя е входния документ, а за дете се счита преобразувания във вътрешния стандарт документ.

4.2 Допълнителни обекти

4.2.1 SEQ_MESSAGE_ID

Последователност от числа (на английски sequence), която се използва за задаване неповторяеми стойности на първичния ключ в таблицата T_MESSAGE. Широко използван похват в базите от данни, тъй като гарантира уникалност на стойностите в ключа при многонишков достъп, и от няколко различни транзакции. Има две основни операции, които могат да се извършват върху тези обекти: взимане на текущата стойност и генериране на следващата поред стойност.

4.2.2 Функцията STORE_MESSAGE

Записана в базата функция. Основната й цел е да запише даден документ в базата, като в същото време капсулира детайлите около самата операция. В дадения случай, в зависимост от входните си параметри, тя записва и релации за новото съобщение.

Съответната й спецификация е:

```
CREATE OR REPLACE FUNCTION store_message(INTEGER, INTEGER, TEXT, INTEGER,
INTEGER)
RETURNS INTEGER AS '
DECLARE

        documentId ALIAS FOR $1;
        msgTypeId ALIAS FOR $2;
        msgVal ALIAS FOR $3;

        -- optional, if NULL are passed it means this is a "parent" message
        parentMsgId ALIAS FOR $4;
        relTypeId ALIAS FOR $5;

        newMsgId INTEGER;
BEGIN

        SELECT INTO newMsgId nextval("public."SEQ_MESSAGE_ID"");

        INSERT INTO "T_MESSAGE" ("MESSAGE_ID", "DOCUMENT_ID",
```

```
"MESSAGE_TYPE_ID", "MESSAGE_VALUE")
        VALUES (newMsgId, documentId, msgTypeId, msgVal);

        IF parentMsgId IS NOT NULL and relTypeId IS NOT NULL THEN
                -- insert relationship
                INSERT INTO "TM_MESSAGE_RELATION" ("PARENT_MESSAGE_ID",
"CHILD_MESSAGE_ID", "RELATION_TYPE_ID")
                VALUES (parentMsgId, newMsgId, relTypeId);
        END IF;

        RETURN newMsgId;
END;
' LANGUAGE 'plpgsql';
```

Детайлите по имплементацията й за зависими от вида база от данни, като при PostgeSQL се използва неговия език за създаване на записани в базата функции и процедури – PL/PG SQL.

Кодът на тази функция, както и скриптовете за създаване и първоначално попълване на базата данни могат да бъдат намерени на съпровождащия компакт диск. За повече информация - Приложение А: Описание на придружаващия компакт диск.

5 Вътрешен формат за представяне на документи за търговски поръчки

Както вече бе подчертано основното предизвикателство при изготвяне на приложението е проектирането на вътрешния стандарт за търговски поръчки. За него са налице две основни изисквания: да представлява достатъчно пълно сечение от функционалността на разглежданите схеми и заедно с това да предоставя възможност за лесно добавяне на нови входящи бизнес формати. Второто условие най-добре може да се изпълни с използването на общи, абстрактни XML елементи, които сами по себе си нямат семантично значение, а служат като обвивка на входните елементи.

Проектираната схема също така трябва да позволява и вертикална разширяемост – добавяне на други видове електронни търговски документи като фактури, отговори на поръчки и т.н. Разбира се, самата специфика на тези документи е коренно различна, но въпреки това биха могли да се набележат общи елементи, които впоследствие да бъдат преструктурирани в отделни схеми (например дадена частична схема, съдържаща последователност от елементите количество, брой, обща цена, които във всички схеми ще се срещат в тази последователност). По този начин на родителските схеми ще е нужен само указател към дадените външни елементи, и, ако се налага промяна от бизнес гледна точка, тя ще бъде извършена на едно единствено, точно определено място.

Тук следва да се отбележи и Java технологията, използвана от приложението, за да може лесно и по общодостъпен начин да борави със схемата. Това е JAXB (Java API for XML Binding), която ще бъде обяснена по-детайлно при документирането кода на приложението, а теоретичната ѝ обосновка бе представена в глава 2.2.4.

Трябва да се обърне внимание и на конвенцията, която е съблюдавана при изграждане на вътрешния формат. Това е така наречената CamelCase номенклатура. Така например за елементите се използва UpperCamelCase (например <Header>), за атрибутите LowerCamelCase (например <… parentMsgID=”123”/>). За акронимите при елементи се използва също

UpperCamelCase (например <BDIDocument>), докато за атрибути се използва обратния стил – LowerCamelCase.

Важно е да се отбележи, че предоставените по-долу референции към полета от входните формата са съгласувани с XPath (XML Path Language) ([19]) стандарта. Това е набор от синтактични правила за дефиниране на части от XML документите. Най-общо той позволява използването на дефиниции на пътища, подобни на традиционните във файловите системи, заедно с няколко предефинирани функции и логически условия, като чрез тях може да се опише местоположението на всеки XML елемент. Сам по себе си XPath не е написан като XML, но е основен градивен елемент на отбелязаната по-горе (точка 2.2.3) XSLT технология. За нуждите на настоящото изложение е достатъчно да отбележим, че пътища, започващи с "/" означават абсолютен път от корена на текущото дърво, а ако започва с "//", то избира всички елементи, отговарящи на даденото условие, без значение от нивото, на което се намират в XML структурата. Атрибутите на елементите се обозначават с "@". За повече информация текущата спецификация може да бъде намерена на следният адрес в Интернет пространството: http://www.w3.org/TR/xpath.

В използваните детайлни фигури се съблюдава следната конвенция: плътните линии означават задължителен елемент (или списък от елементи), пунктираните линии – незадължителен, а числата в долният десен край на елементите показват броя (cardinality) на съответният елемент – колко пъти може да присъства в дадения списък.

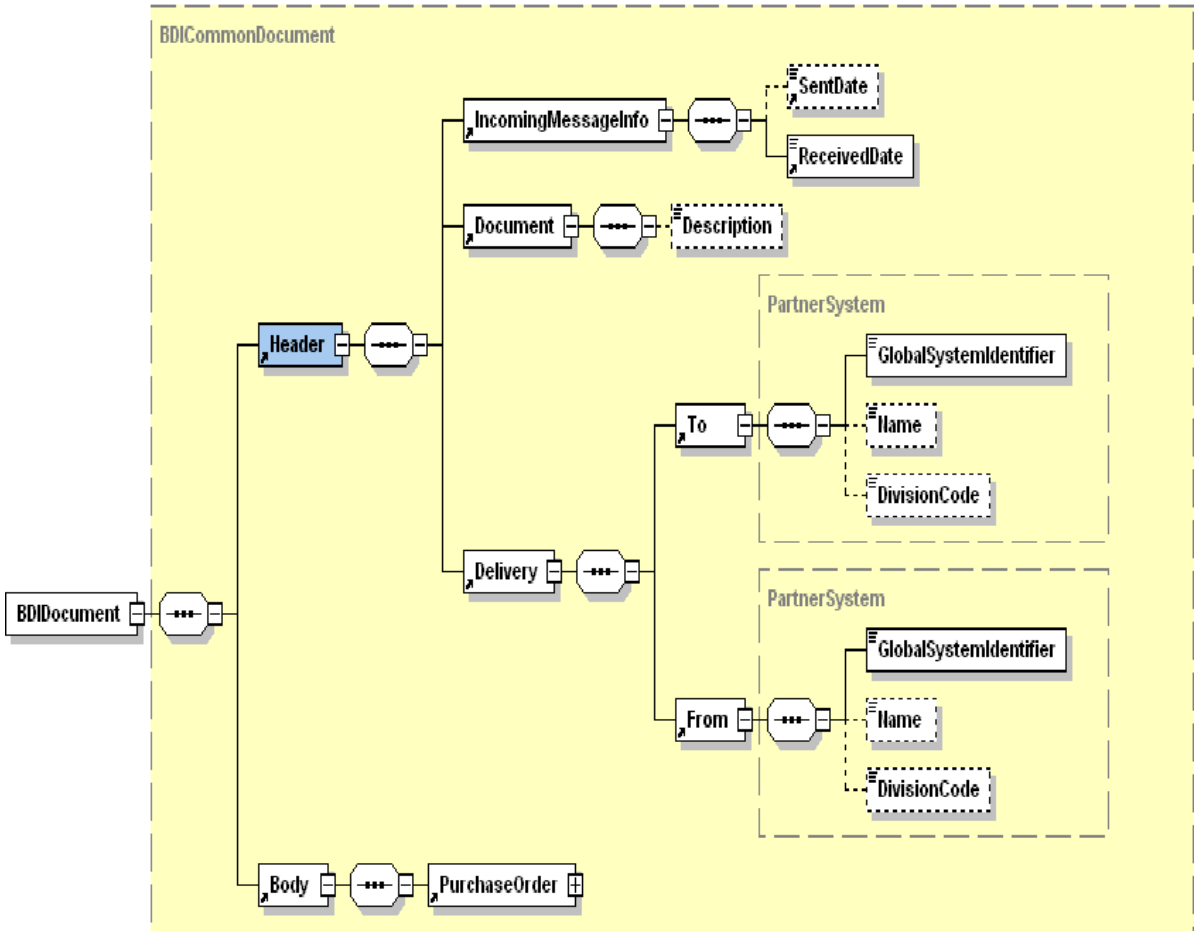
Нека отбележим, че главния (root) елемента на всички схеми в даденото приложение (съществуващата за търговски поръчки и бъдещите такива) ще е общ и ще се нарича ***BDIDocument*** (абревиатурата BDI идва от името на настоящата система – Business Document Integrator). Тя ще съдържа два поделемента: ***Header*** и ***Body***.

До края на настоящата точка ще разгледаме използваната схема, получена след детайлно изследване на част от съществуващите стандарти. Поради големия й обем в подробности ще се спрем само на описанието на най-важните й елементи.

Цялостното съдържание на вътрешната схема е предоставено в Приложение Б: Вътрешен формат – BDI XSD.

5.1 Елемента *Header*

Схематично представен на Фиг. 14. Съдържа информация за документа като цяло, и не е строго специфичен за търговските поръчки, а може да се използва и за другите видове електронни търговски документи.



Фиг. 14 - Елемент BDIDocument.Header

Има три поделемента: *IncomingMessageInfo*, *Document* и *Delivery*.

Първият е много важен от гледна точка на системата, тъй като той съдържа указател към входния документ – това е атрибута *parentMsgID*, който показва идентификатора на входния документ (или в термините на базата данни това е T_MESSAGE.MESSAGE_ID на входящия документ). Има

и атрибут - *sourceSystem* - който съдържа името на формата, в който е изпратен входящия документ. Елементите *SentDate* (незадължителен) и *ReceivedDate* специфицират съответно датата на изпращане на документа и датата, на която той е получен от интеграционната система, или в термините на базата данни това е полето T_MESSAGE.DATE_CREATE.

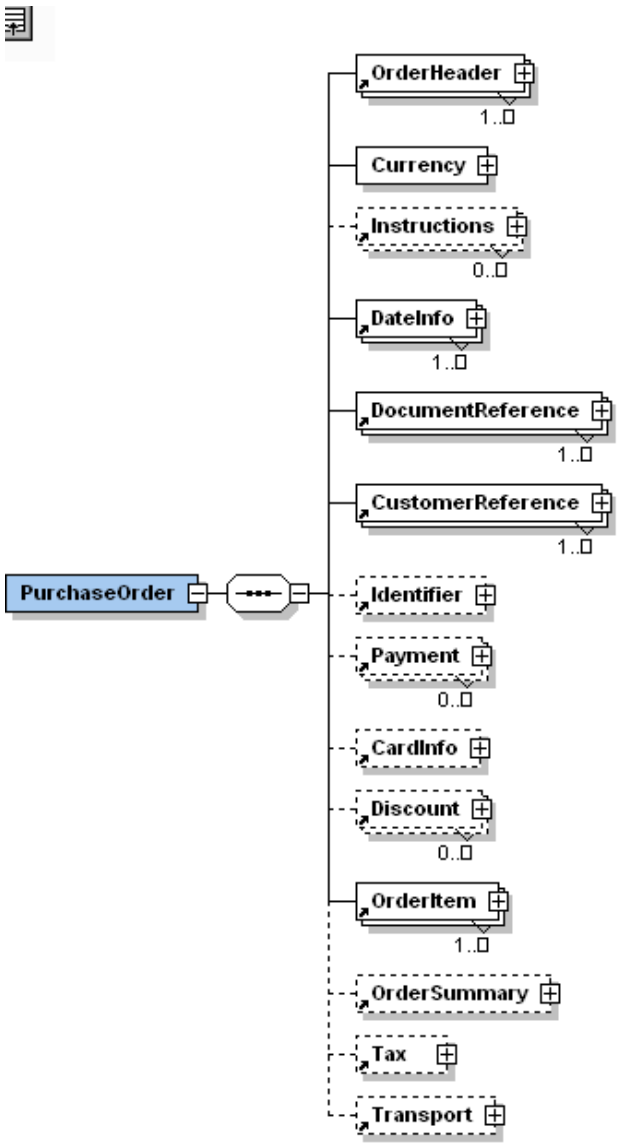
Информацията, съдържаща се в този елемент, е от съществено значение, защото чрез нея може да идентифицира еднозначно съответния входен документ, което от своя страна позволява да се осъществи и обратната комуникация, например връщането на електронна фактура. Така се удовлетворява и една от основните цели на всеки мидълуер компонент, а именно изискването да не се губи информация при трансформирането на данните.

Елемента *Document* специфицира типа на текущия документ (понастоящем винаги PurchaseOrder), както и уникален идентификатор от входящата система (атрибута *proprietaryDocID*). Наличието на този елемент се налага от факта, че самият *Header* елемент е с общо предназначение.

Елемента *Delivery* съдържа два елемента – *To* и *From* – които са от един и същ тип (PartnerSystem) от гледна точка на схемата. По този начин се набляга на еднаквата им синтактична структура. Но от друга страна те имат различна семантика – единия специфицира входящата система, а вторият – изходната, в контекста на текущия документ. Трябва да се отбележи, че тук под входяща система се има предвид не формата, а реалния бизнес партньор, инициирал поръчката – Dell, IBM, Microsoft, и други. Тези елементи са функционално еднакви с */cXML/Header/To/Credential* (*/cXML/Header/From/Credential*) при cXML, докато при трансформацията от RosettaNet формата това са следните два XML елемента - */Pip3A4PurchaseOrderRequest/toRole/PartnerRoleDescription/PartnerDescription /BusinessDescription* (*/Pip3A4PurchaseOrderRequest/fromRole/PartnerRoleDescription/PartnerDescription/BusinessDescription*).

5.2 Елемента ***Body.PurchaseOrder***

Съдържа информация, специфична за настоящата търговска поръчка. Носи основните от гледна точка на бизнеса данни: колко и какви продукти се поръчват, основна валута за разплащане, на кой клиента да се изпрати самата стока и на кой съответната фактура, и т.н.



Фиг. 15 - Елемент BDIDocument.Body.PurchaseOrder

Горната диаграма (Фиг. 15) представлява съответната подсхема, като елементите *Instructions*, *DocumentReference*, *CustomerReference* и *OrderItem* ще бъдат разгледани отделно поради по-съществения им значение.

OrderHeader елемента е задължителен, като може да присъства един или повече пъти. Основната му роля е да показва типът на търговската поръчка, който бива нов, променен или изтрит при cXML (това е полето */cXML/Request/OrderRequest/OrderRequestHeader@type).* При RosettaNet е */Pip3A4PurchaseOrderRequest/PurchaseOrder/GlobalPurchaseOrderTypeCode*. Съгласно RosettaNet стандарта търговските поръчки се делят на функционален принцип и могат да бъдат всестранен (“Blanket”), консигнационна поръчка (“Consigned order”), поръчка, която трябва само да се фактурира (“Do not ship/invoice only”), и други, специфицирани от съответния ограничен тип на схемата.

Currency елемента е от CurrencyType тип от гледна точка на схемата, който съдържа главната и алтернативната (ако има такава) валута за съответната поръчка като цяло, както и курса на обмяна между дадените валути. Той пряко се трансформира от съответния cXML елемент - /*cXML/Request/OrderRequest/OrderRequestHeader/Total/Money* (и атрибутите му *@currency* и *@alternateCurrency*). При RosettaNet това е */Pip3A4PurchaseOrderRequest/PurchaseOrder/totalAmount/FinancialAmount/GlobalCurrencyCode*, който е списък от стандартни три буквени ISO кодове.

Списъкът от елементи *DateInfo* показва важни дати, асоциирани с поръчката. Трябва да присъства един или повече пъти на ниво PurchaseOrder. Значимите кодове на дати са специфицирани като ограничени списъци в настоящата схема, по-точно в атрибута *@dateType*. Те могат да са следните: EAD – най-ранна дата на доставка, LAD – най-късна дата на доставка, ORD – дата на поръчване, RDD – изисквана дата на доставка, SBD – дата на изпращане. Използването на предефинирани ограничени типове в схемата е мощно средство при последващото й разширяване, защото само чрез добавяне на нов тип може да се получи значително функционално разширение на вътрешния стандарт.

Елемента *Identifier* служи за оторизация на клиента, който е изпратил поръчката. Съдържа името му, както и някой важни негови кодове в системата.

Елемента *Payment* показва начина на заплащане на съответната поръчка: по касов път, в брой, чрез банков трансфер и други. При RosettaNet за всеки елемент */Pip3A4PurchaseOrderRequest/PurchaseOrder/FinancingTerms/ PaymentTerms* се създава съответен *Payment* елемент при трансформиране към вътрешния формат на поръчките.

CardInfo е незадължителен и служи за съхраняване информация за кредитната карта, през която ще се извършва разплащането – номера, името на притежателя, датата на изтичане на картата. Съответните му елементи (заедно с атрибути им, които на практика се трансформират) са */cXML/Request/OrderRequest/OrderRequestHeader/Payment/PCard* при cXML и */Pip3A4PurchaseOrderRequest/PurchaseOrder/AccountDescription/CreditCard* при RosettaNet.

Незадължителния списъкът от елементи *Discount* съхранява информация за евентуалните отстъпки или специални сделки, които се полагат на даденият клиент. Чрез него се реагира на бизнес ситуациите, когато поради договорни задължения (например поради голям обем на сделките) се предоставя известна отстъпка. Такава информация се съдържа в RosettaNet */Pip3A4PurchaseOrderRequest/PurchaseOrder/FinancingTerms/PaymentTerms/ Discounts*.

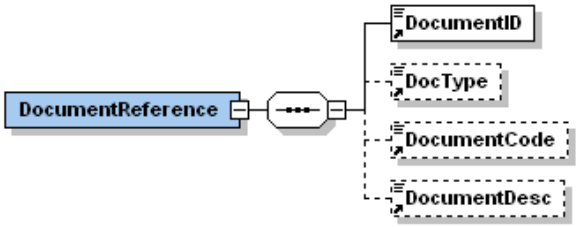
Следващите два елемента са приложими само на ниво търговски ордер като цяло, тъй като съдържат информация са цялостната поръчка. Първият е *OrderSummary* елемента, който се състои от поделементи, обозначаващи съответната цена в основната и алтернативната валута, както и общия брой на елементите. Вторият е *Tax* – това за дължимите такси за съответната поръчка, под формата на данъци, транспортни плащания и други. Може да бъдат като процентно съотношение от цената на цялата поръчка (елемента *TaxPercent*) или като нетна сума (елемента *TaxAmount*). Също така следва да се отбележи, че тук отново се използва CurrencyType за елемента *TaxCurrency*, което прави използването на валутите във вътрешния формат

гъвкаво и лесно за поддръжка и разширение. При cXML този елемент е */cXML/Request/OrderRequest/OrderRequestHeader/Tax*.

Последния елемент от разглежданата поредица е ***Transport*** елемента, който също така се използва и на ниво продукт. Той показва датата за транспортиране (елемента ***TransportDate***), начина (елемента ***TransportMode***) и друга подходяща информация. Подобни данни се наблюдават при RosettaNet формата, по-точно казано това е специализираният елемент */Pip3A4PurchaseOrderRequest/PurchaseOrder/requestedEvent/TransportationEvent*, заедно с прилежащите му подчинени елементи *DateStamp* и *GlobalTransportEventCode*.

5.3 Елемента ***DocumentReference***

Списъкът от ***DocumentReference*** елементите представлява колекция от бизнес свойства, които позволяват описването на множество документи (Фиг. 16). Чрез него се указват уникални за външните системи идентификатори на документите. Използва се както на ниво поръчка (където е задължителен, трябва да присъства поне един такъв елемент), така и на ниво отделен продукт.



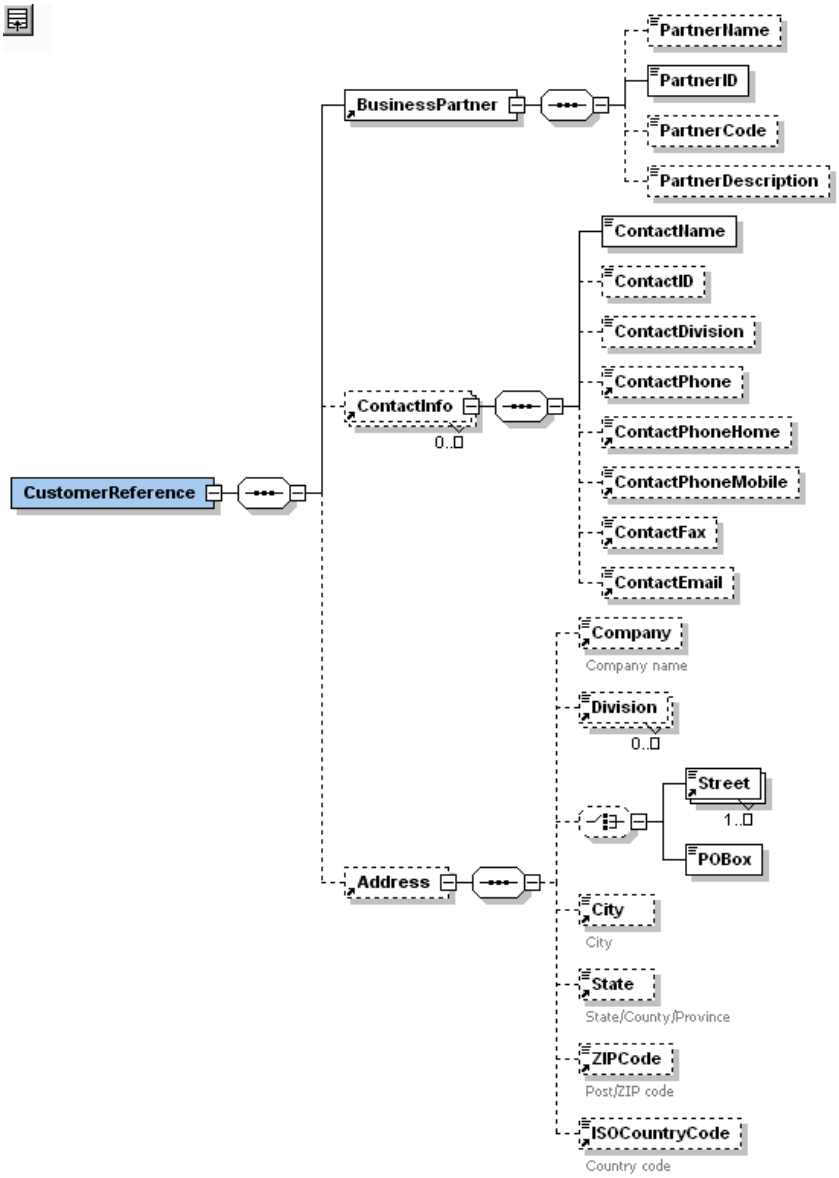
Фиг. 16 - Елемент DocumentReference

При cXML съответните елементи, които се трансформират, са */cXML/Request/OrderRequest/OrderRequestHeader/DocumentReference*, а при RosettaNet */Pip3A4PurchaseOrderRequest/PurchaseOrder/DocumentReference*.

5.4 Елемента *CustomerReference*

Описва пълното множество от свързаните с дадения търговски ордер бизнес единици.

Пълната му спецификация е показана на Фиг. 17:



Фиг. 17 - Елемент CustomerReference

Специфицирането на съответния тип бизнес контакт се постига чрез използване на ограничен тип кодове във вътрешната схема (чрез JAXB технологията за тези стойности автоматично се генерират Java константи).

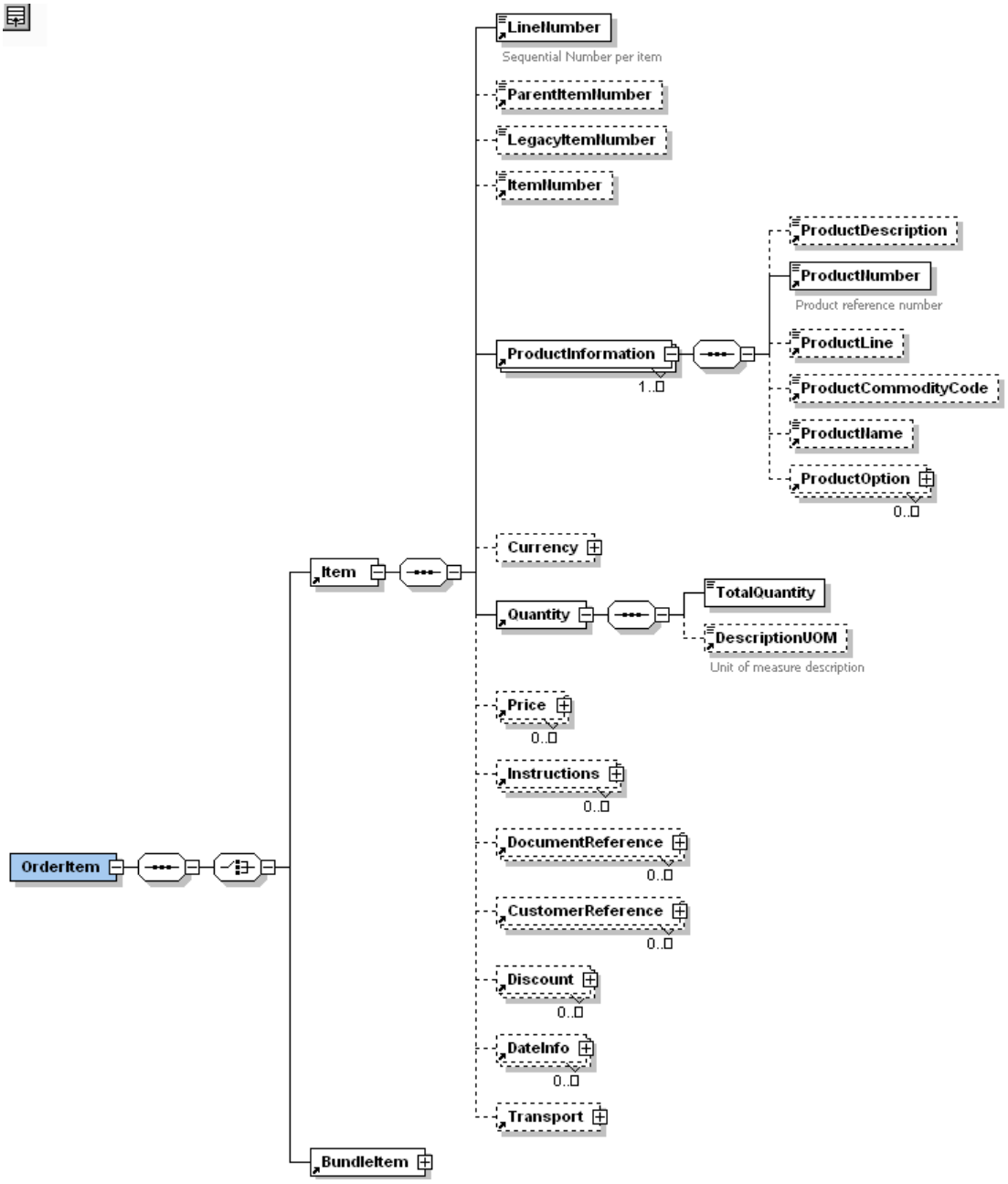
Така дефиниран съответният атрибут е *BusinessPartner@typeID*, който има една от следните стойности: SHP – бизнес лицето, на което следва да се доставят продуктите от поръчката, SLD – продаден на, BIL – на кой се изпраща сметката, INV – на кой да се изпрати фактурата, FBY – от кой е финансирана поръчката, INS – къде да се инсталира (в контекста на съответният бизнес), CBN – основен код на клиента в партньорската система, PUR - купувач, SUP - доставчик, VDR – бизнес единицата, изработила дадения продукт (подходящ главно на ниво артикул), PDR – партньорски идентификатор, от гледна точка на системата, която генерира поръчката.

Основната функционалност, както вече бе изяснено, се носи от поделемента *BusinessPatner.PartnerID*, в контекста на съответният тип на бизнес партньора. Незадължителните елемент *ContactInfo* и *Address* имат по-скоро второстепенно значение, те доставят допълнителна информация за текущия клиент. Но е възможно на даден етап от жизненият цикъл на поръчката да се окажат необходими за правилното й изпълнение, например при извършване на доставка адресът става от първостепенно значение.

При cXML стандарта се наблюдават единствено референции към бизнес партньор, на който трябва да се прати сметката за поръчката (*/cXML/Request/OrderRequest/OrderRequestHeader/BillTo*, код BIL при BDI) и също така къде да се достави поръчката (код SHP, елемента е */cXML/Request/OrderRequest/OrderRequestHeader/ShipTo*), докато при RosettaNet се наблюдава по-голямо подмножество от бизнес контакти: (*/Pip3A4PurchaseOrderRequest/PurchaseOrder/AccountDescription/financedBy/PartnerDescription*, код FBY), къде трябва да се достави поръчката като цяло (*/Pip3A4PurchaseOrderRequest/PurchaseOrder/shipTo/PartnerDescription*, код SHP), кой е отговорен за заплащане на цялостния търговски ордер (*/Pip3A4PurchaseOrderRequest/PurchaseOrder/AccountDescription/billTo/PartnerDescription*, код BIL), бизнес контакта на купувач, който инициира поръчката (*/Pip3A4PurchaseOrderRequest/PurchaseOrder/SecondaryBuyer/PartnerDescription*, код PUR).

5.5 Елемента *Body.PurchaseOrder.OrderItem*

Всяка търговска поръчка, независимо от вида й, съдържа един или повече артикула (Фиг. 18), тъй като те носят основната стойност от бизнес гледна точка като дефинират какви точно продукти или услуги желае да получи купувачът. Затова и всяка от разглежданите схеми съдържа елементи с подобен смисъл.



Фиг. 18 - Елемент Body.PurchaseOrder.OrderItem

Във вътрешният формат има два взаимноизключващи се елемента, които могат да представляват даден продукт. Това са елемента *Item*, който описва обикновен артикул и елемента *BundleItem*, който представлява няколко взаимносвързани части, които представляват един единствен продукт от гледна точка на клиента. Например това може да е даден модел лаптоп, който се разделя на памет, видео карта, твърд диск от гледна точка на производителят. Самият елемент *BundleItem* може да се разглежда като *OrderItem* елемент, защото и той съдържа списък от обикновени артикули, но под общо име. Затова ще бъде разгледан основно елемента *Item* като основна градивна единица, описваща даден продукт.

Съответните елементи, представляващи отделен артикул са както следва: /*cXML/Request/OrderRequest/ItemOut* при cXML и при RosettaNet формата: */Pip3A4PurchaseOrderRequest/PurchaseOrder/ProductLineItem*. По-надолу в изложението ще бъдат давани относителни пътища спрямо тези елементи.

Елемента *LineNumber* съдържа текущия номер на елемента в цялостния списък от продукти. Този елемент се генерира като пореден номер на текущо прочетеният елемент и няма директно представяне във входните формати.

Следващите три елемента (*ParentItemNumber*, *LegacyItemNumber* и *ItemNumber*) специфицират подредбата на елементите от гледна точка на входната система, както могат и да съдържат уникален идентификатор на продукта в партньорската система. Елемента *ParentItemNumber* се използва за обозначаване на зависимост между артикулите и има основополагащо значение при изграждане на така наречените “пакети” от продукти (разгледаните по-горе елементи *BundleItem*, където в общият случай всички артикули се явяват части на първият продукт от пакета). Използвайки този елемент е възможно да се описват формални (и неформални) зависимости между продуктите.

Задължителният списък от елементи *ProductInformation* носи основната каталожна информация за дадения продукт. Чрез познатата техника на използване на предефинирани в схемата константи за даден атрибут

(*@productCode* в случаят) се специфицират четири важни типа идентификатори в зависимост от вида на електронната обслужваща система: тази на производителят (код VPN – производствен номер на частта), на купувача (BPN и BIP – кодове на частта в системите на купувача) или на доставчика (код SPN). В контекста на стойността на този атрибут се интерпретира и задължителният елемент *ProductNumber*, който показва и стойността на самия код. Останалите елементи предоставят допълнителна информация за продукта; името му (*ProductName*), кратко описание (*ProductDescription*) и т.н. Списъкът от елементи *ProductOption* има по-специфично значение, той служи за специфициране на някой основни детайли за артикула. Например такива са локализацията при персоналните компютри. Съответните елементи при cXML са *ItemID/SupplierPartID* (трансформиран до код SUP) и *ItemDetail/ManufacturerPartID* (във вътрешния формат *ProductNumber*) и *ItemDetail/ManufacturerName* (съответно *ProductName*), които съставляват продуктовата информация с код VPN. При RosettaNet се наблюдава информация само от гледна точка на купувача (код BPN) и това е списъкът от елементи, свързан с данни за продукта - *ProductIdentification/PartnerProductIdentification* с поделементи съответно *ProprietaryProductIdentifier* (трансформиран към *ProductNumber*) и *GlobalPartnerClassificationCode* (съответно *ProductCommodityCode*).

Всеки артикул трябва да подлежи на някакъв вид количествено измерване - брой, тежест, обем - за да е известно какво количество изисква купувачът. Точно това е и семантиката на задължителният елемент *Quantity*. Чрез атрибута *unitOfMeasure* се специфицира вида количествена характеристика, а елемента *TotalQuantity* показва самото количество като неотрицателно число. Даденият елемент има значение и при определяне цялостната цена, понеже обикновено при търговските поръчки се дава само цената на единица продукт. Съответните елементи са както следва: атрибута *@quantity* и елемента *ItemDetail/UnitOfMeasure* при cXML, а при RosettaNet бизнес стандарта това са *OrderQuantity/requestedQuantity/ProductQuantity* и *GlobalProductUnitOfMeasureCode*. Трябва да се отбележи, че докато при cXML полето с типа на мярката е свободен текст, то при RosettaNet това е списък от възможни стойности, като например десет килограмов бидон (“10

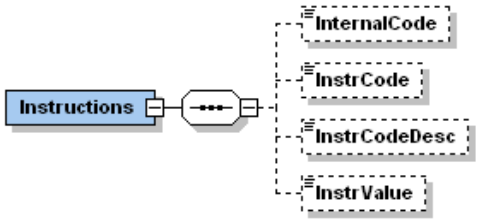
Kilogram Drum"), десет хиляди килограмов вагон ("10,000 Gallon Tankcar") и т.н. Като цяло RosettaNet схемата представя значително по-стройни ограничителни правила от cXML формата.

Друг важен елемент е списъкът от цени *Price*, който обаче не е задължителен, тъй като той може да се изчислява при обработката на поръчката, в зависимост от дължимите отстъпки. Също така, в общия случай, истинската цена остава скрита от купувача, и не може да се разчита на предоставената от него информация. Атрибута *@priceType* също е ограничен от самата схема и може да се наблюдават няколко негови стойности: цена с отстъпка от гледна точка на купувача (код BDP), цена с отстъпка от гледна точка на производителят (код VDP) и т.н. В този контекст се изчислява и стойността на елемента *PriceAmount*. Трансформираните елементи от входните формати са както следва: *ItemDetail/UnitPrice/Money* при cXML и *requestedUnitPrice/FinancialAmount/MonetaryAmount* при формата RosettaNet.

Останалите елементи, които описват даден артикул като *Currency*, *CustomerReference*, *Discount*, *DateInfo* няма да бъдат разглеждани в подробности, тъй като те имат същата структура като съответните елементи на ниво поръчка. Също така те носят и подобна семантика, но се отнасят за отделния продукт, а не за ордера като цяло. По този начин позволяват гъвкавост при образуването на поръчката, например даден артикул може да се заплати в различна валута от главната, дадени елементи може да се изпратят на друг бизнес партньор (или на друго подразделение на основния клиент) и т.н. Важно е да се спомене, че от бизнес гледна точка ако присъстват данни от входящата система на ниво отделен продукт, то те имат приоритет над съответните данни от поръчката като цяло.

5.6 Елемента *Instructions*

Елементът *Instructions* (Фиг. 19) ще е от изключителна важност при евентуално развитие на системата, тъй като той може да служи като универсална "обвивка" (wrapper) за всеки елемент от входните формати. За това основна роля играе неговата структура:



Фиг. 19 - Елемент Instructions

Тя позволява да се обхване всеки индивидуален елемент, чрез използване на познат на системата код (елемента *InternalCode*), а съответно елементите *InstrCode* и *InstrValue* съдържат стойностите от входящия документ. При този вариант на използване на елемента атрибута *@instrType* съдържа стойността “MARKER”. В настоящият вариант на вътрешния стандарт по този начин се добавя информация за даден вид такси (или данъци) на ниво артикул при трансформиране от RosettaNet. По този начин се избягва необходимостта вътрешната схема да е пълно сечение от всички бизнес формати, което е и практически невъзможно.

Този елемент обаче има и друга функция. Той служи за добавяне на важна информация в тялото на самия XML документи, която служи за проследяване на важни събития при трансформирането към вътрешният стандарт, а и при последващата обработка на документа. В зависимост от важността на събитието дадената инструкция може да има тип INF – служи за информация, WARN - предупреждение и ERR – грешка. В настоящия вариант на мидълуер компонента се среща само WARN инструкция, която се добавя, ако сборът от цените на отделните продукти не отговаря на цената в елемента *Body.PurchaseOrder.OrderSummary*. Тази проверка се извършва само при обработка на поръчки, пристигащи в cXML формат.

6 Мидълуер приложение за трансформиране на данни

В настоящата точка ще разгледаме разработеното мидълуер приложение за трансформиране на входящите електронни бизнес документи.

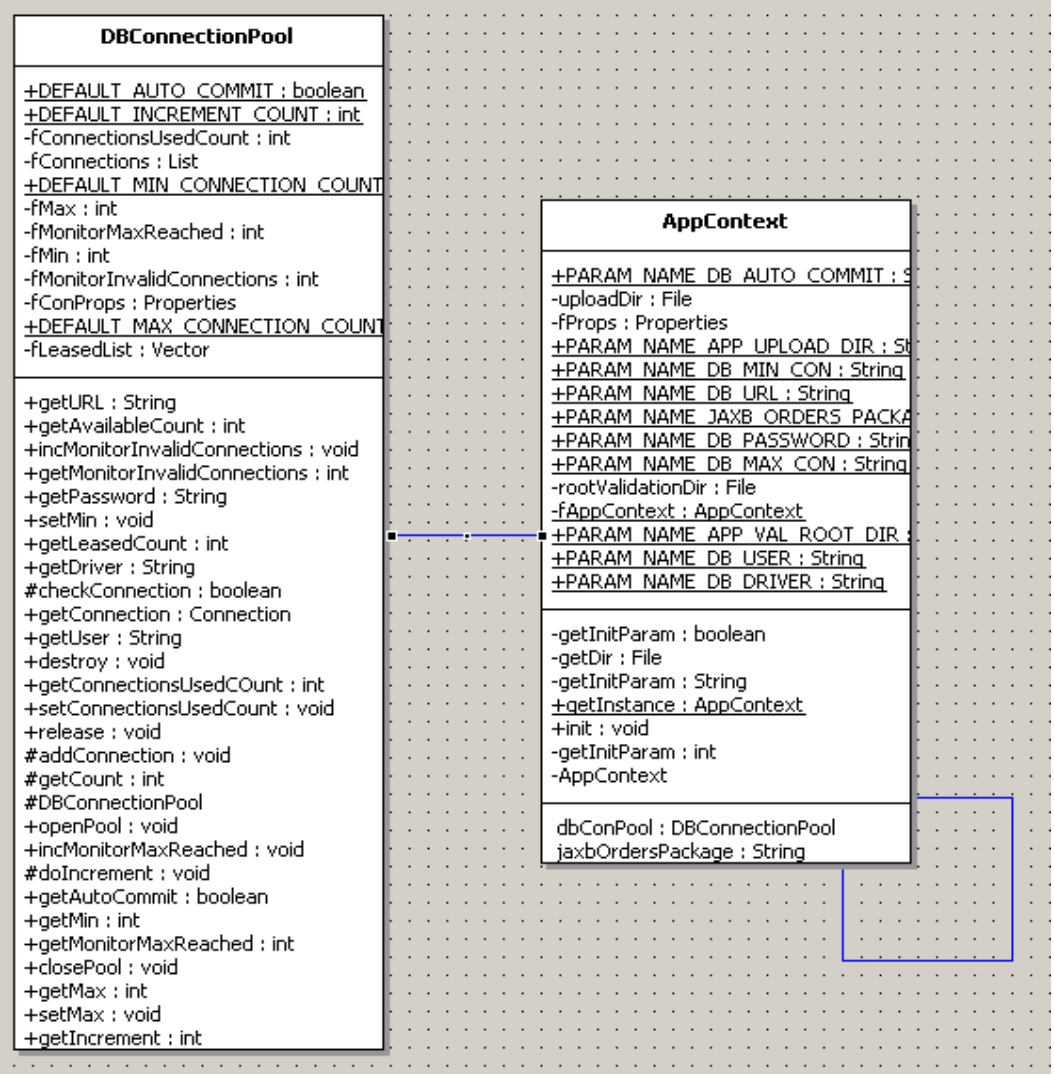
Като програмен език за изработка на решението е избран Java, версия 1.4, заради платформената си независимост, което е от особено важно значение за мидълуер приложенията. Причината за това се обуславя от факта, че не винаги при създаването на даден компонент е предварително ясно на каква точно операционна среда ще работи за в бъдеще, и следователно платформената независимост се явява основно изискване. Затова и Java като език за програмиране е използван предимно в клиент - сървър приложенията, и като следствие от този факт - и при развитието на мидълуер архитектурите. От основно значение при изборът на Java е факта, че езикът позволява изключително лесен и до голяма степен стандартизиран начин за достъп до бази данни. Обработката на XML също е много добре развита, въпреки че съществуват множество програмни интерфейси (APIs), създадени с тази цел. Но трябва да се отбележи, че повечето от тях са проектирани с ясна целенасоченост като по този начин се минимизира възможността разработчика да се “изгуби” в XML технологиите.

Диаграмите при излагане на програмния код са според стандарта UML (Unified Modeling Language) и са генерирани с помощта на добавката (plug-in) UML Explorer на Together за най-популярната Java платформата за разработване на приложения – Eclipse, версия 3.0.

Конвенцията за именуване на класовете, променливите и методите е стандартната, разработена от Sun. Известни модификации има единствено при използване на отварящата скоба (“{”), която се слага в началото на новия ред при сигнатурата на методи, при използването на логически условия, цикли и т.н.

6.1 Архитектура на приложението за работа с базата данни

На Фиг. 20 е показана UML диаграмата на основните използвани в приложението класове за обработване на системната конфигурация и достъп до съответната база данни.



Фиг. 20 - Класове за работа с базата данни

Класът *org.fmi.bdi.helpers.AppContext* съхранява основната конфигурация на системата. Поради факта, че приложението трябва да се инициализира само веднъж, при създаването на този клас се използва т.нар. модел на единствения обект (Singleton pattern). Този шаблон гарантира, че *може да*

има една и само една инстанция на даден клас в системата ([10]). Основното му приложение е когато се борави с единични външни обекти, например принтер, база данни или конфигурация (както е в настоящия случай). Постига се чрез следният програмен фрагмент:

```
public static AppContext getInstance()
{
        if (fAppContext == null)
        {
                synchronized (AppContext.class)
                {
                        if (fAppContext == null)
                        {
                                fAppContext = new AppContext();
                        }
                }
        }

        return fAppContext;

}

private AppContext()
{

}
```

При инициализирането си (извикване на *init()* метода) единствения контекстен обект на приложението се грижи за прочитането на всички необходими параметри, голяма част от които са свързани с базата данни – адресът й, името на потребителя, паролата и т.н. – и създава списък за общо ползване (pool) на връзки към базата, т.е. строго специализиран обект от Java класа *org.fmi.bdi.helpers.DBConnectionPool*. Описаният похват е широко разпространен, тъй като съдейства за оптимизиране работата с базите данни. Основната причина е, че постоянното отваряне и затваряне на конекции към базата отнема много системни ресурси и затова е по-добре да се държат отворени малко на брой връзки към базата, които да са директно използваеми от приложението.

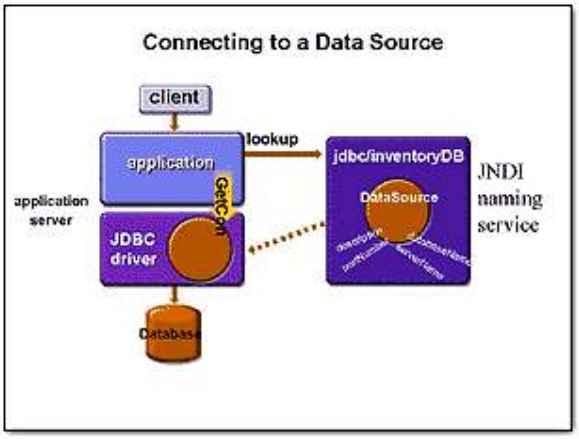
Самият клас *org.fmi.bdi.helpers.DBConnectionPool* предлага на своите клиенти синхронизирани услуги за достъп до капсулираните вътрешно връзки към базата данни: взимане и връщане на съответния обект за достъп до базата, свиване или разширяване на списъка и други подобни. Тъй като този клас се създава при инициализирането на единствения за приложението контекстен обект, то и той има само една инстанция, което

позволява вътрешно, скрито от своите клиентите, да управлява колекцията си от конекции към базата, като по този начин е възможно да се намали рискът от получаване на т.нар. “мъртви хватки” (dead lock) при управление на транзакциите в базата данни, както и да се минимизира възможността клиентите да не следват установения начин за достъп до базата – взимане на конекция, използването й и последващо връщане в списъка. Откриването и отстраняването на такива проблеми по принцип е много трудоемко и затова трябва да се отчита още при първоначалното проектиране на системата.

В настоящото приложение се използва технологията JDBC (Java Database Connectivity), която е общоприет индустриален стандарт за независима от производителя връзка между Java като програмен език и множество сървъри за бази данни. Основното й предимство е, че позволява спецификите на използвания сървър да бъдат скрити от разработчика. Това съдейства за по-бърз цикъл на програмиране и заедно с това улеснява подмяната на базата, стига производителят да има разработен сървърен Java управляващ интерфейс, отговарящ на JDBC API спецификацията (JDBC driver).

JDBC позволява да се извършват следните три основни операции (Фиг. 21):

- създаване на връзка към базата (в общият случай това може да е и друг табличен източник на данни)
- изпращане на SQL заявки (търсене, създаване на нови записи и промяна на съществуващи, както и команди за създаване, промяна или изтриване на обекти в базата данни)
- обработване на получените резултати



Фиг. 21 - Употреба на JDBC

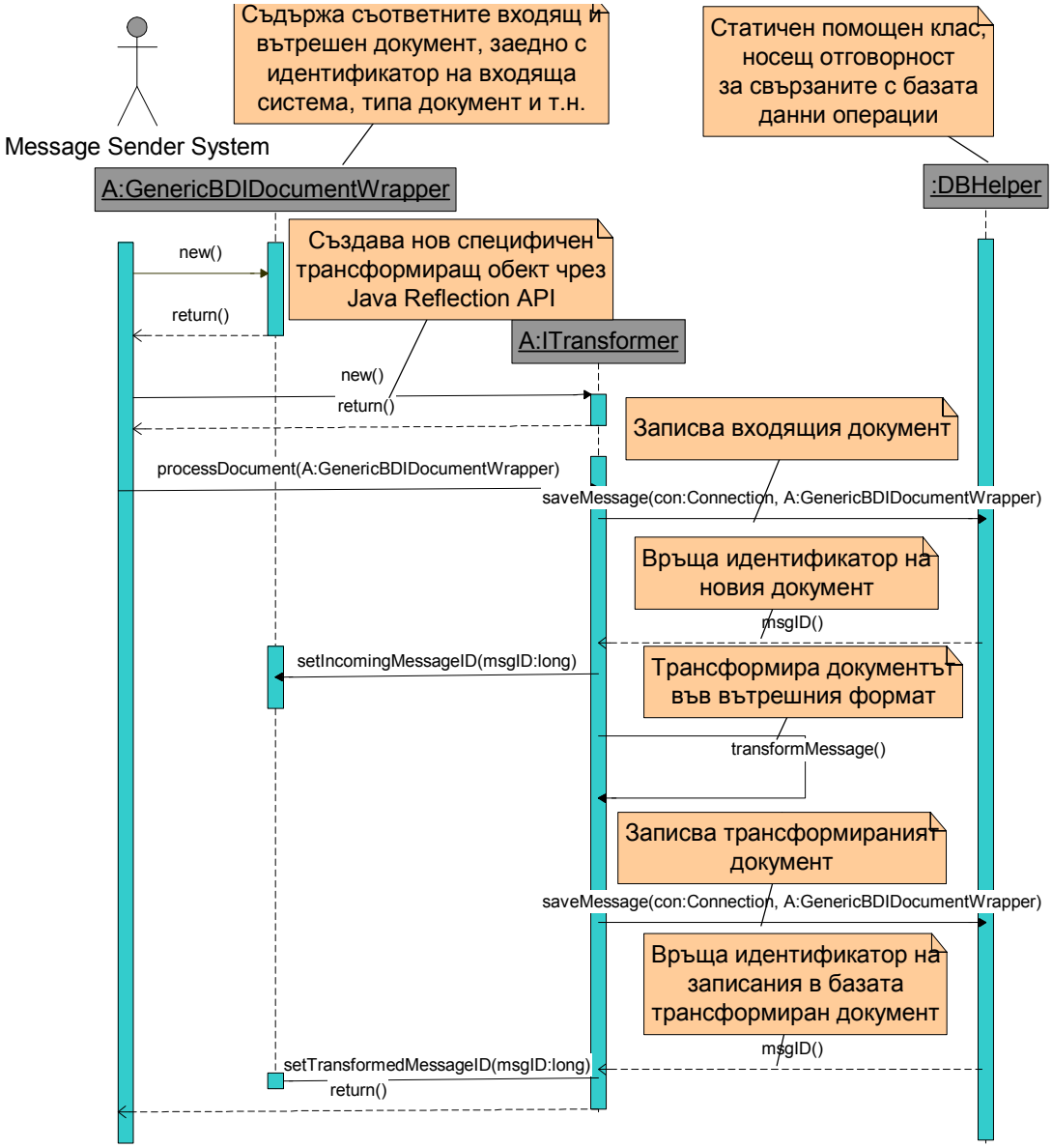
Самите операции с базата данни като записване на документ (входящ или вече трансформиран), получаване на документ по неговия уникален идентификатор и т.н. са капсулирани в Java класа *org.fmi.bdi.db.DBHelper*. Това е и основното място, където се наблюдава употребата на JDBC за достъп до базата данни.

6.2 Обработване и трансформиране на входящи документи

Системата е гъвкаво проектирана по такъв начин, че да е възможно да обработва и трансформира всевъзможни видове търговски документи. Но засега единствено търговски поръчки, пристигащи като XML файлове (във cXML или RosettaNet формат), са имплементирани. Затова в частност ще се спрем на техният "жизнен цикъл" в системата, като по този начин най-ясно ще се открояят съставните елементи на трансформиращото приложение, както и връзките между тях.

На Фиг. 22 е представена диаграма на обработването на едно съобщение от системата за интеграция на документи. В термините на UML това е така наречената диаграма от последователност на предаване на съобщения (sequence diagram). Тя трябва да се разглежда в контекста на извикването на методи (заедно със съответните им параметри) между главните действащи лица в системата (в случая представени като обекти,

принадлежащи на съответният Java клас), както и с кратко описание на всеки метод.



Фиг. 22 - Жизнен цикъл на документ в приложението

Нека първо отбележим, че под “Система за изпращане на съобщения” (Message Sender System) трябва да се разбира вътрешната система, която получава електронен документ, и поради съдържанието му и/или спецификата на входящият канал получава информация за формата на документа (cXML, RosettaNet, xCBL и т.н.), както и за типа му (търговска

поръчка, отговор на поръчка, електронна фактура и т.н.). Нейно задължение е запазването на входящите документи в опашка като то този начин следва да се инициира обработването им от системата.

В следващите подточки ще бъдат детайлно разгледани съответните действия от гледна точка на приложението. Първоначално ще се спрем на една спомагателна, но много важна функционалност, която спомага преобразуването на данните във вътрешния формат.

6.2.1 Представяне на вътрешния BDI XML формат като JAXB обекти

Използваната технология за генериране на Java обекти, представляващи даден XML документ, е описана по-подробно в точка 2.2.4, тук ще я разгледаме в контекста на трансформиращия мидълуер компонент.

Както се изясни основният източник за генериране на обектната обвивка на вътрешния формат е схемата на приложението, разгледана в глава 5. Но заедно с това JAXB спецификацията позволява да се дефинира специална схема (т.нар. Binding schema), чрез която може да се конфигурира генерирането на програмния код. Например посредством тази схема се разрешават колизиите с имената на XML елементите, т.е. ако два XML елемента имат еднакво име, то за един от съответните класове трябва да се специфицира уникално име, за да е възможно последващото компилиране до двоичен код. В настоящото приложение се използва следната схема от разгледания тип:

```
<jaxb:bindings
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
   xmlns:xjc="http://java.sun.com/xml/ns/jaxb/xjc"
   extensionPrefixes="xjc"
   version="1.0">

   <jaxb:bindings xmlns:xs="http://www.w3.org/2001/XMLSchema"
      schemaLocation="../../target/common_format/CommonStandard_PurchaseOrder_v1_0.xsd"
      node="/xs:schema">

      <jaxb:globalBindings generateIsSetMethod="true">
         <xjc:serializable uid="47110815"/>
         <xjc:superClass name="org.fmi.bdi.helpers.BaseJAXBDoc"/>

<!-- xs:decimal not used anymore
         <jaxb:javaType name="java.math.BigDecimal" xmlType="xs:decimal"
            parseMethod="org.fmi.bdi.helpers.AmountConverter.parseAmount"
            printMethod="org.fmi.bdi.helpers.AmountConverter.printAmount"/>
-->
      </jaxb:globalBindings>
```

```
  </jaxb:bindings>
</jaxb:bindings>
```

Единственото й приложение за настоящата система е да специфицира общ клас, които е родител на всички генерирани JAXB класове. Това е класът *org.fmi.bdi.helpers.BaseJAXBDoc*. Тъй като този клас не се прегенерира при евентуална промяна на схемата, то негова отговорност е да предоставя основната, обща функционалност за всички JAXB класове, а именно общата логика за прочитане от входящата XML структура (unmarshal) и последващото преобразуване от Java обектната йерархия към XML (marshalling).

Самото генериране на Java класовете става с помощта на компилатора XJC, който е част от стандартните JAXB библиотеки. Той има множество опции, но най-важната от гледна точка на приложението е името на пакета, в който се генерират класовете. В случая на търговски поръчки това е *org.fmi.bdi.commonstandard.orders.jaxb*, което е конфигурационен параметър на скрипта за изграждане на приложението. За тази цел се използва инструмента Maven (http://maven.apache.org/), която е с отворен код и представлява спомагателно средство за управление, тестване и документиране на проекти. Той се базира на по-стария, но широко разпространен инструмент Ant (http://ant.apache.org/).

6.2.2 Прочитане и валидиране на входният документ

Системата, която се грижи за приемане на документи от външните партньори, инициира създаване на обект от сървъра за приложения, който носи отговорността да продължи обработката на документа. Тя подава цялата налична информация за документа – идентификатор на формата, типа, както и самото му съдържание. Задължение на трансформиращият компонент е да получи тази информация, да провери коректността на формата на входящите данни, както и да създаде впоследствие обект, който да съхранява цялата налична информация. Съответният Java клас е *org.fmi.bdi.document.GenericBDIDocumentWrapper*.

Този клас капсулира всички детайли, свързани със специфичния документ – типа му, входната система, типа валидация, както и съдържанието на търговската поръчка. При инициализацията си той зарежда данните от базата, използвайки за уникален ключ идентификатора на входната система, както и специфичния тип на документа. Предварително се проверява валидността на подадените параметри и се прочита съдържанието на полученото съобщение. В частност ще разгледаме случая когато входния документ е XML, тъй като върху него е поставено ударението на приложението.

Класовете, които служат за прочитане и проверка формата на входния документ, се намират в пакета *org.fmi.bdi.parser*. На Фиг. 23 е съответната UML диаграма на класовете от пакета:

Фиг. 23 - Пакета org.fmi.bdi.parser

В класът *ValidationParser* е съсредоточена общата функционалност на използваните XML анализатори. Той инициализира общите им части и съдържа указател към колекцията, където се съхраняват грешките, възникнали в процеса на проверка формата на документа. Наследниците му – *XSDValidationParser* и *DTDValidationParser* – са отговорни за действителната проверката на формата съответно въз основа на схема или тип дефиниция. Както вече бе показано (точка 4.1.1) типът проверка се фиксира предварително на базата на входната система и типа съобщение, а изборът кой точно валидатор да се използва се извършва в *org.fmi.bdi.document.GenericBDIDocumentWrapper* чрез следният програмен фрагмент:

```
switch (fValidationTypeID)
{
        case IValidationType.DTD:
                DTDValidationParser dtdParser = new
DTDValidationParser(valSource);
                fIncomingDocument = dtdParser.parse(uploadedXML);
                break;
        case IValidationType.XSD:
                XSDValidationParser xsdParser = new
XSDValidationParser("file:/" + valSource);
                fIncomingDocument = xsdParser.parse(uploadedXML);
                break;
        default:
                throw new BDIException("Validation type: '"
                                                                +
fValidationTypeCode +"' not implemented.");
}
```

Вътрешно парсерите използват широко разпространените библиотеки с отворен код от Apache Software Foundation (http://www.apache.org/) – Xalan и Xerces. Те напълно отговарят на разгледаните стандарти за обработка на XML (точка 2.2) , и на практика са си извоювали лидерска позиция сред множеството библиотеки за обработка на XML в Java.

Важно е да се отбележи, че в обектите от разглеждания клас (*GenericBDIDocumentWrapper*) се съхранява информация не само за входящия документ, а и за трансформирания, която се попълва в процеса на обработка на документа. По този начин се улеснява функционално приложението, а също така се поддържа и връзката входен документ-трансформиран и на обектно ниво, а не само в базата данни.

6.2.3 Трансформиране към вътрешния XML формат

В разглежданото мидълуер приложение терминът “трансформиране” следва да се разглежда в смисълът на преобразуване структурата на входните документи към вътрешния XML стандарт. И тъй като основната цел при проектирането на решението е да се позволи неговата лесна разширяемост то е нужно да се избере съответен похват за конфигуриране на тази операция.

През фазата на проектиране бяха разгледани два възможни варианта. Първият предвиждаше в базата от данни за всяка уникална комбинация - входна система/версия/тип на съобщението – да съществуват необходимите записи, представящи чрез XPath биективна трансформация между входящият документ и вътрешния стандарт. Но това решение има няколко важни недостатъка. Основните са че не е лесно приложимо към входящи документи в линеен формат, а също така даже и за XML документи е трудно да се имплементира каквато и да била допълнителна функционалност, различна от самото XML преобразуване. Затова бе избран вторият вариант, а именно в базата да се записва името на Java класа, който е отговорен за дадената трансформация. Единственото ограничение е, че този клас следва да наследява един вътрешен Java интерфейс.

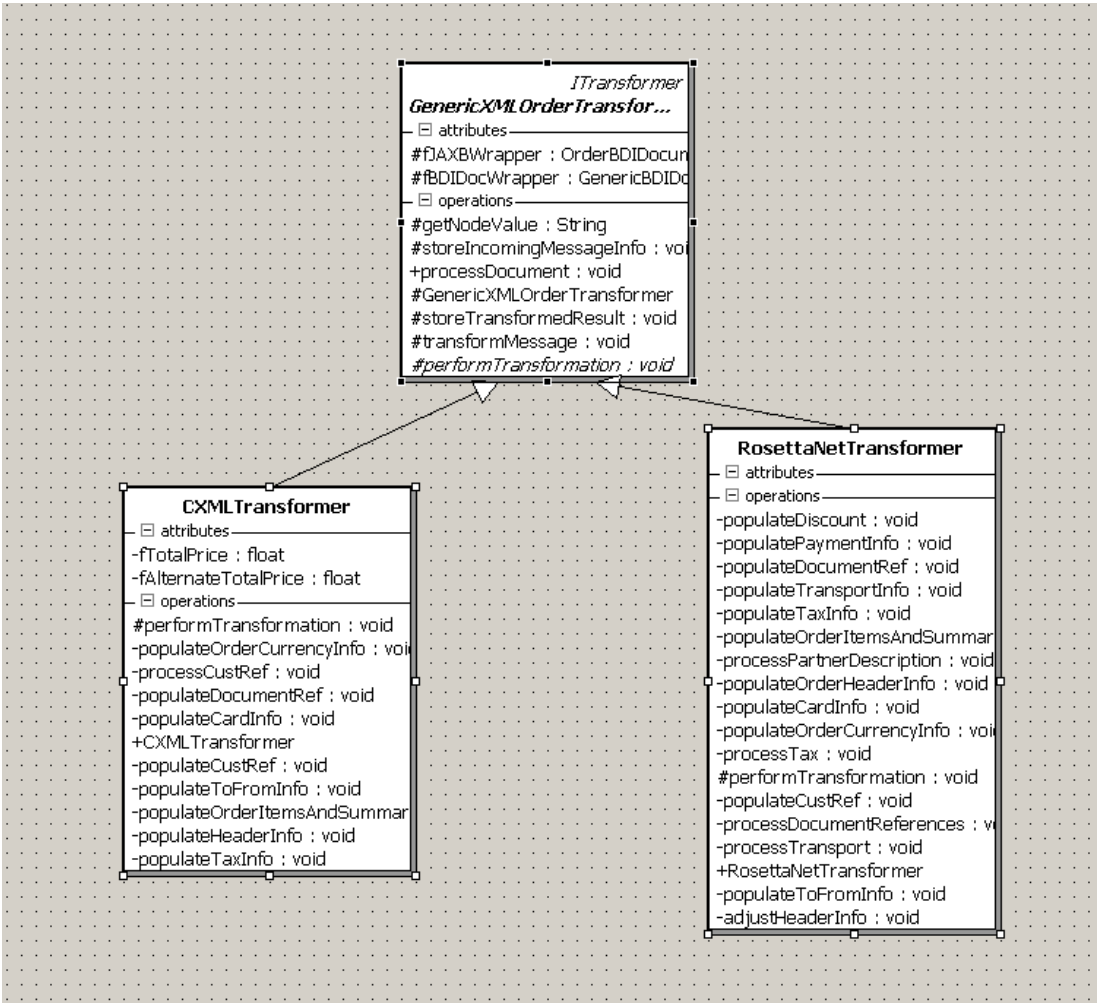
За да е възможно изпълнението на горната задача основна роля изпълнява Java технологията за динамично създаване на обекти, позната като Java Reflection API.

Технологията Java Reflection е позната още от първата версия (1.0) на езика. Поддръжката й е вградена в основите на виртуалната машина (JVM) и е една от ключовите характеристики на езика, която не присъства в основата на езици като C, C++, Pascal. Разглежданият програмен интерфейс позволява да се получава динамично (по време на изпълнение на програмата) информацията за полетата, методите и конструкторите на заредените класове, както и да се създават обекти само по пълното име на класа им и сигнатурата на някой от конструкторите им. Тази функционалност, макар и много мощна, трябва да се използва в разумни граници, защото отнема много системи ресурси като памет и процесорно

време, като по този начин забавя самото приложение. Но в настоящето приложение, както вече бе подчертано, първостепенно значение е отделено на гъвкавостта, не толкова на бързодействието. Още повече че за в бъдеще е възможно да се управлява списък от такива обекти, вместо да се създават всеки път, като по този начин ще се минимизира отрицателния ефект за производителността на системата.

Основните класове, които поддържат Java Reflection, са *java.lang.Class* и класовете от пакета *java.lang.reflect*, които предоставят обекти - обвивки за методи, полета, конструктори и други.

В настоящата система таблицата T_TRANSFORMER (разгледана в точка 4.1.6) съдейства за използването на гореописаната технология, тъй като в нея се съхранява пълното име на Java класа за преобразуване, който трябва да наследява вътрешния интерфейс *org.fmi.bdi.transformer.ITransformer* и да има празен конструктор. В момента са разработени трансформатори за RosettaNet PIP 3A4 и за cXML стандартите, на Фиг. 24 е съответната диаграма на класовете:



Фиг. 24 - Пакета org.fmi.bdi.transformer.order

За да е възможно динамичното инстанциране (през Java Reflection) и последващото използване на обекти от трансформиращите класове е необходимо всички да имат метод с предварително дефинирано име, за да знае мидълуер компонента кой метод да извика на неизвестния по време на компилиране обект. За тази цел всички преобразуващи класове (не само тези за търговски документи) трябва да наследяват Java интерфейса *ITransformer* и да имплементират метода му *processDocument()*. Заедно с това, в процеса на разработване на класовете за преобразуване на търговски поръчки, се откроиха още общи функционални части от обработката на документите. За тази цел е създаде абстрактния Java клас *org.fmi.bdi.transformer.order.GenericXMLOrderTransformer*. Основната му роля е да капсулира така намерените общи характеристики при обработката на

търговските поръчки и да гарантира схематизирания по-горе (Фиг. 22) цикъл на обработка на дадено съобщение. Именно в него, чрез имплементация на метода от интерфейса, се извършва първоначалното записване на входящия документ, запазването на новополучения идентификатор от базата, както и финалното записване на трансформирания документ. По този начин за класовете надолу по йерархията остава единствено да се грижат за самото преобразуване, т.е. да имплементират абстрактния метод *performTransformation()*. Ето и самият програмен код на метода *processDocument()*:

```
public void processDocument(Object doc) throws BDIException {
        fBDIDocWrapper = (GenericBDIDocumentWrapper) doc;

        AppContext app = AppContext.getInstance();
        DBConnectionPool pool = app.getDbConPool();

        Connection con = pool.getConnection();
        try {
                // first incoming document must be stored into the database
                int incomingMsgID = DBHelper.saveMessage(con, fBDIDocWrapper);
                fBDIDocWrapper.setIncomingMessageID(incomingMsgID);

                // transforms to internal format
                transformMessage();

                //finally the result must be stored too
                int transformedMsgID = DBHelper.saveMessage(con, fBDIDocWrapper);
                fBDIDocWrapper.setTransformedMessageID(transformedMsgID);
                con.commit();
        } catch (Exception e) {
                try {
                        con.rollback();
                } catch (SQLException sqle) {
                        throw new BDIException("Unexpected error occured: "
                                        + sqle.getMessage(), sqle);
                }

                if (e instanceof BDIException)
                        throw (BDIException) e;
                else
                        throw new BDIException("Unexpected error occured: "
                                        + e.getMessage(), e);

        } finally {
                pool.release(con);
        }
}
```

Преобразуващите класове (понастоящем за RosettaNet това е класът *RosettaNetTransformer*, а за cXML – *CXMLTransformer*, и двата от Java пакета *org.fmi.bdi.transformer.order*), наследявайки гореописания клас, гарантират желаното от дизайн гледна точка развитие на документа в рамките на

системата. Така тези два класа се грижат само за преобразуване на съответните входящи документи, и като такива имат много близко функционално поведение. От тяхна гледна точка входните данни вече са валидирани и са достъпни под формата на Java обект, представляващ дървовиден XML документ. За самото прочитане на необходимите данни се използва XPath (който, разбира се, е различен при двата формата) и по-точно неговата интерпретация при парсера на Apache. При трансформиране на данните се използват и разгледаните (точка 6.2.1) JAXB обекти, за да се трансформира дадения документ във вътрешния XML формат.

Тук трябва да спомена и *org.fmi.bdi.document.order.OrderBDIDocument* класа. По принцип всяка промяна на вътрешната XML схема налага повторно компилиране на съответните JAXB обекти. Затова поддръжката на директното им използване в приложението може да се окаже твърде трудоемко за в бъдеще. Именно тук се намесват обектите от класа *OrderBDIDocument*, които скриват детайлите при взаимодействието с JAXB технологията като по този начин изолират до голяма степен приложението от бъдещи промени, тъй като съответните модификации могат да бъдат направени на едно централизирано място. До известна степен този клас може да се счита за “фасада” на преобразуването, тъй като имплементира съответния широко разпространен шаблон в Java – Facade Pattern.

С това разглеждането на начина на разработка на основното приложение на системата завърши и следва да се спрем на проверката на функционалността му.

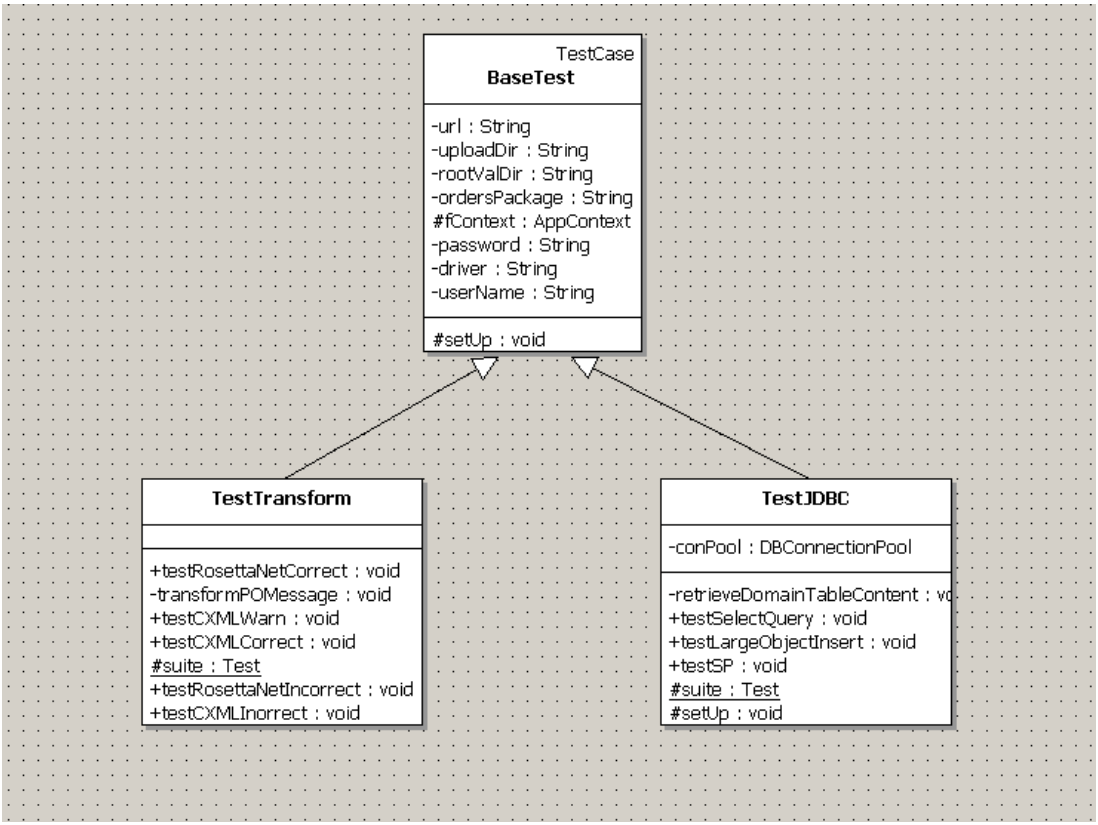
7 Среда за тестване на приложението

Разработването на специално приложение за тестване на настоящата дипломна работа се оказа от първостепенно значение, тъй като няма имплементирани истински системи – клиенти на този мидълуер компонент. Затова е нужно да се симулира тяхната функционалност чрез уеб базирано, самостоятелно приложение. То служи за проверка бизнес изискванията, възложени при проектирането на трансформиращия компонент. Чрез него се цели единствено проверката на изходните данни на системата при първоначално зададен вход, без значение от вътрешната логика на продукта.

Но както и всяко приложение, преди да бъде верифицирана цялостната му функционалност е нужно първо да се потвърди правилното действие (и взаимодействие) на различните му модули. За тази цел първо ще се спрем на съответните тестове, разработени паралелно със самото приложение.

7.1 Модулни тестове

Под термина “модулен тест” (unit test) обикновено се разбира тестване на ниво интерфейс на всеки клас от приложението. В случая на системи, базирани на Java, основната архитектура за проверка на модулите е JUnit (http://www.junit.org/). Основната й характеристика е, че е лесна за ползване, но заедно с това достатъчна гъвкава за нуждите на повечето приложения. Тя позволява и автоматизиране на тестването, като по този начин гарантира стабилността на всички версии на дадена система.



Фиг. 25 - Архитектура за модулно тестване

В приложение има специален Java пакет, *org.fmi.bdi.tests* (Фиг. 25), чиято роля е да осигури проверката на функционалността на основните обособени модули. Базовият клас е *org.fmi.bdi.tests.BaseTest*. Негова отговорност е да осигури поддръжката на тестовия фреймуърк и поради тази причина той наследява основният JUnit клас *junit.framework.TestCase*. Заедно с това той има за задача да инициализира приложението, предоставяйки му нужните параметри като парола и потребител за базата данни, класа с драйвъра и други. Това става чрез интерфейсния метод *setUp()*, който се извиква точно преди извикването на тестовите методи. Тестовите методи от своя страна се определят динамично от JUnit архитектурата, чрез вече разгледаната Java Reflection технология. Изискванията към тях са да са видими за всички (в термините на Java - public), да са без аргументи и името на метода да започва с *test*.

В настоящото приложения са имплементирани два такива тестови класа за основните модули: един за тестване на връзката към базата данни и един

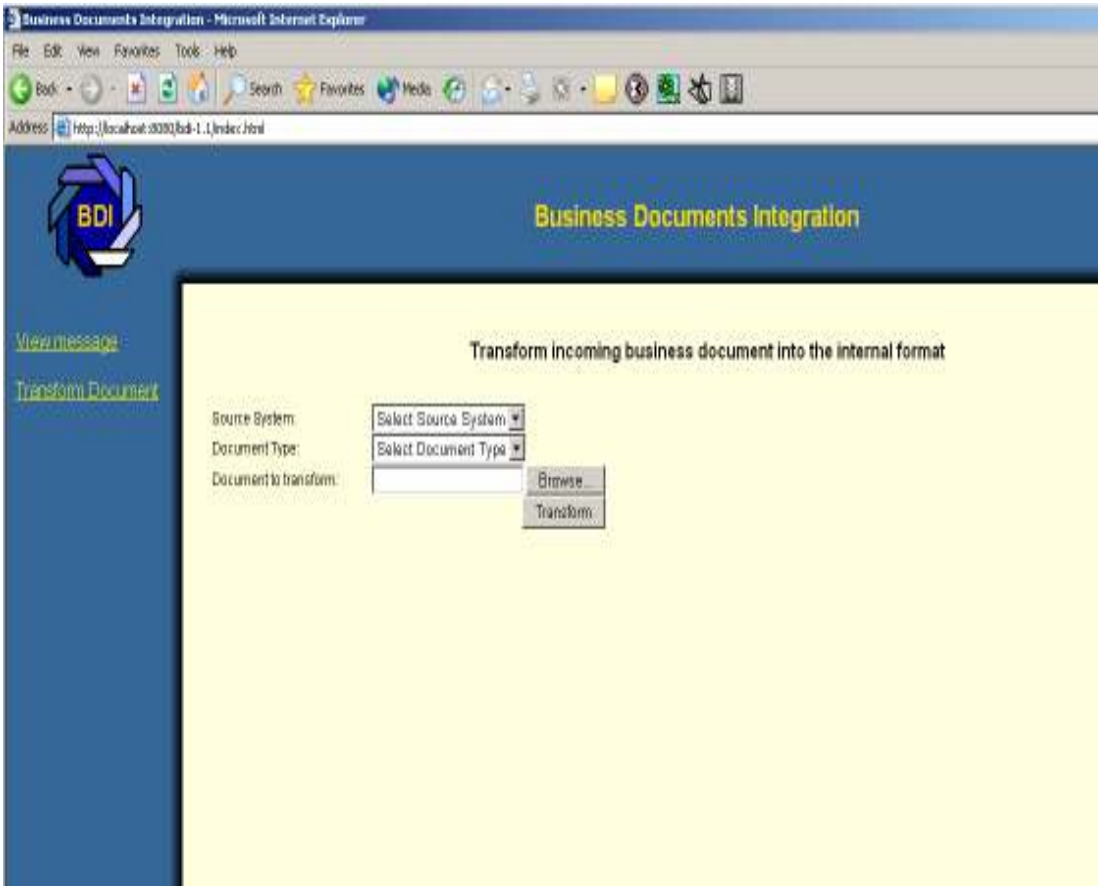
за валидиране и трансформиране на входните данни, съответно *org.fmi.bdi.tests.TestTransform* и *org.fmi.bdi.tests.TestJDBC*.

7.2 Системни (интеграционни) тестове

За проверка функционалността на системата като цяло е нужно да се извършат два типа проверки. Първата е от потребителска гледна точка, т.е. да се погледне на системата като на “черна кутия” (от английски black box). При този подход е важно единствено съответствието между подадените входни данни и очаквания изход, без да е нужно познаването на самата система. Предимствата му са, че позволява тестване от гледна точка на функционалността, защото самия проверяващ (възможно е това да е и автоматизиран процес) следи единствено за спазване на потребителските изисквания, поставени при специфицирането на продукта. Втория подход е познат като структурен, или още като “бяла кутия” (white box). Тук се предполага, че тестващия има поглед върху структурата на програмата. Предимството е, че по този начин е възможно да се избере пълен набор тестови данни, така че да се покрият всички възможни разклонения на приложението.

При настоящата система, за тестване на цялостната функционалност е изработено отделно Web базирано приложение, което позволява да се симулира входния канал на системата (чрез подаване на тестови файлове с поръчки), както и да се верифицира изходът на интеграционния компонент. Използваната технология е Servlet/JSP, която е основната платформа за генериране на динамично Web съдържание при Java.

Следва да се отбележи, че главното улеснение на клиентите е възможността директно да качват файла с търговската поръчка на сървъра, вместо да се налага да копират съдържанието му в някое текстово поле. Това се оказа и основното предизвикателство при разработването на приложението, тъй като начина за извършване на тази операция не е стандартизиран при използваната версия на Servlet/JSP програмния интерфейс, а отговорността за неговото имплементиране е оставена изцяло на разработчика.



Фиг. 26 - Web приложение за функционално тестване на приложението

Споменатото Web приложение притежава две основни функционалности: показване съдържанието на даден документ по идентификатора от таблицата T_MESSAGE, както и подаване на входен документ чрез специфициране на входящата система, съответния тип на документа и съдържанието му като път във файловата система (Фиг. 26) и последваща проверка на резултата. Този интерфейс е достатъчен за тестване на входа и изходът на система, защото позволява да се валидира резултатът от трансформацията на дадено множество реални документи. Също така чрез него се удостоверява обработката на грешките в приложението, защото могат да се проверят и случаите когато системата не може (а и не трябва) да преобразува даден документ.

В настоящия момент Web приложението се обслужва от Jakarta Tomcat Web сървър, версия 4.1.18 (http://jakarta.apache.org/tomcat/index.html), която поддържа спецификациите Servlet 2.3 и JSP 1.2. Но тъй като тестовото

приложение е разработено в съответствие с Servlet/JSP спецификациите, то не зависи от специфичния Web контейнер и лесно би могло да се използва на който и да е Web server, стига той да поддържа Web Archive спецификацията.

8 Преглед на съществуващи интеграционни решения в разглежданата област

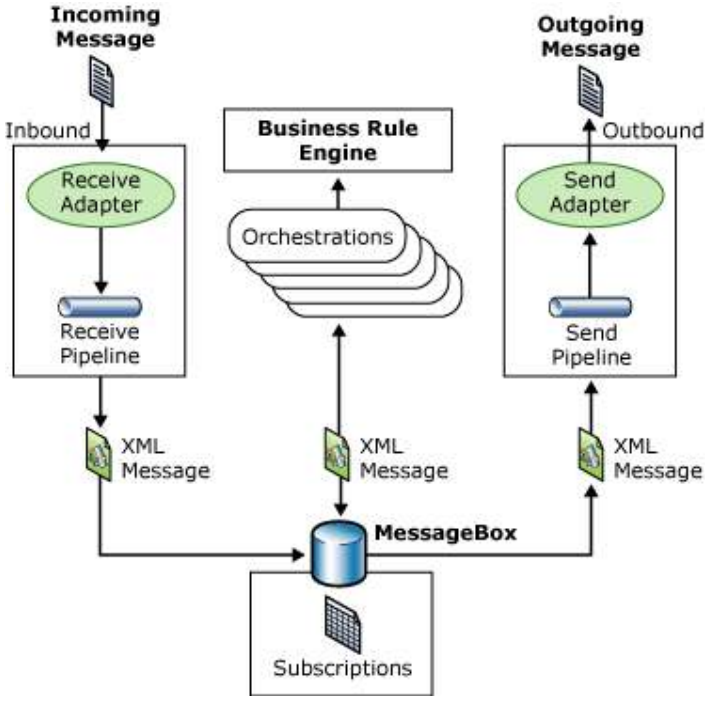
Ще разгледаме интеграционните платформи на едни от най-големите софтуерни компании, участващи в пазарния сегмент за консолидиране при бизнес към бизнес системите, а именно SAP и Microsoft. Разбира се, има и други подобни решения, но до известна степен може да се твърде, че тези две компании са едни от пионерите в разглежданата област, и при това с устойчив пазарен дял и в наши дни. Поради тази причина те са избрани като обекти за сравнение с предлаганото решение.

8.1 Microsoft BizTalk Server

Продуктът на Microsoft® - BizTalk® Server – е популярно решение за интегриране на организациите на бизнес партньорите. Идеята му е да създаде бизнес процеси, които да комбинират отделни (и разнородни) приложения в единно цяло. Той позволява, посредством използването на потребителски визуален интерфейс, да се създават и моделират бизнес процеси, които използват услуги от дадените приложения. Последната му версия - BizTalk® Server 2004 – за разлика от предходните вече не се базира на Microsoft COM архитектурата, а използва по-модерната .NET архитектура.

Следващата диаграма (Фиг. 27) показва техническата архитектура на разглеждания продукт. Той се състои от т.нар. адаптери за получаване и изпращане на съобщения. Те се грижат за получаване и изпращане на документи, било то през Интернет, като ги четат от дадено място по мрежата и в последствие ги изпращат на съответния клиент. Има също така и т.нар. поточни линии (pipelines), където функционално се проверяват и преобразуват данните. Те могат да съдържат компоненти за конвертиране на съобщенията в XML, проверка на цифровия им подпис и т.н. Важни елементи също така са компонентите за организиране (orchestrations), както и основния двигател на системата (business rules engine). Те се използват от бизнес анализаторите чрез графики да дефинират условия, цикли и други детайли от бизнес поведението. Всеки такъв компонент си има т.нар.

абонамент (subscription) в сървъра, за да бъде информиран какъв вид документи са интересни за него. Последният, но не по значение елемент от BizTalk сървъра е базата от данни където се пазят съобщенията. Тя се нарича MessageBox, разработена на Microsoft SQL Server.

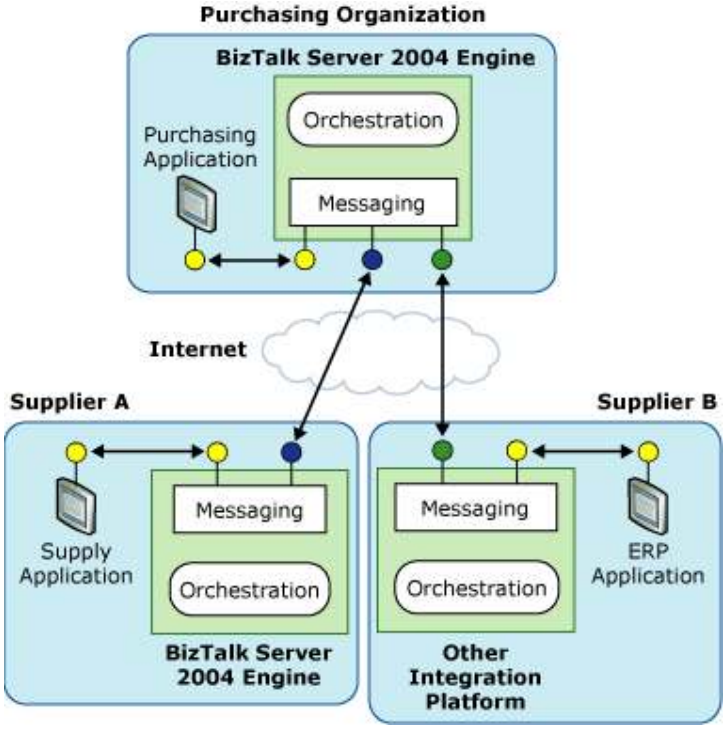


Фиг. 27 - Архитектура на BizTalk сървъра

Има два основни сценария при използването му като интеграционна платформа:

- За свързване на приложения вътре в дадена организация, обикновено обозначавано с термина Enterprise Application Integration (EAI)
- За консолидиране на приложения в различни организации, познато като business-to-business (B2B) integration

Втория вид се доближава по-плътно до поставените цели на настоящата диплома работа, затова и ще бъде разгледан накратко.



Фиг. 28 - B2B функционалноста на BizTalk сървъра

Приложенията на Фиг. 28 се свързват по следния начин:

- Организацията купувач използва инстанция на Microsoft® BizTalk® Server 2004, която взаимодейства с две организации – доставчици
- Доставчикът А също използва BizTalk Server 2004, като по този начин предоставя индиректен достъп до собственото си приложение
- Доставчикът Б използва интеграционна платформа от друг производител и се свързва към BizTalk Server 2004 сървъра на купувача чрез, например, Web услуги. Той работи по същия бизнес процес като другите доставчици, така че получава съответната дефиниция на процеса от сървъра и продължава по стандартните канали на комуникация

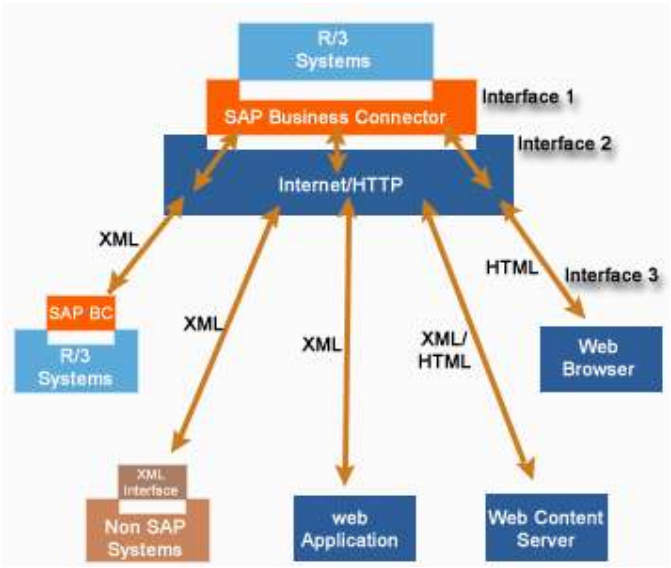
Настоящето приложение най-добре се вписва в контекста на описаните поточни линии, и по точно се доближава до компонента за преобразуване на XML формати, познат като BizTalk Mapper. Той предоставя графичен

интерфейс за директно свързване на стойностите в двете схеми (входящата и желаната след трансформацията), използвайки XSLT технологията. За по-сложни преобразувания се използват т.нар. функциоиди (functoid). Това са последователности от изпълним код, които могат да дефинират произволно сложни преобразувания.

8.2 SAP Business Connector

Този продукт е платформата на SAP за свързване между неговите потребители и външни системи от други производители. Системата може да предоставя данните към SAP приложения в няколко стандарта: IDoc-XML или BAPI-XML. Това са XML разширения за IDoc и BAPI (Business Programming Application Interface), които се използват за комуникация със SAP системите. Също така SAP Business Connector използва и XML xCBL документ, за да улесни обмяната на информация с външни системи, в частност MarketSet пазарите. Той позволява интеграция с R/3 системите посредством отворена и не патентована (non-proprietary) технология.

На Фиг. 29 е представена XML базираната комуникация:



Фиг. 29 - Комуникация чрез SAP Business Connector

Разглеждания SAP Business Connector представлява прибавено XML ниво на абстракция между вътрешните R/3 структури от данни и протоколи и

външните системи. За тази цел чрез специално разработен от SAP RFC (Remote Function Calls) протокол се конвертира XML (или HTML), така че да не е необходимо наличието на специфичен SAP софтуер при клиента.

При този компонент в настоящите версии има вградена поддръжка на форматите IDoc-XML и RFC-XML, а в случаите когато трябва да се използват други формати могат да се използват графични инструменти за изграждане на трансформациите и за имплементиране на собствената логика.

8.3 Сходни системи и формати за електронни документи

Трябва да се отбележи, че съществуват и други подобни системи, например продукта на IBM – WebSphere Business Integration Message Broker, на които няма да се спираме допълнително поради сходните им характеристики. Следва да се отбележи обаче наличието на сходни формати, чиято цел е да унифицират процеса на обмен на документи между търговските партньори. Основната им цел е да разрешат проблемите с разработването и поддръжката на многочислени версии от общи бизнес документи като търговски поръчки и фактури и същевременно наличието на множество адаптери за трансформиране на тези стандарти.

Първият се нарича Universal Business Language (UBL) – Универсален Бизнес език. Целта му е да помогне за решаването на гореописаните проблеми дефинирайки XML формат с общо приложение, който може да бъде разширяван с цел да отговори на нуждите на специфичните индустрии. Той предоставя следните улеснения:

- Библиотека от XML схеми за компоненти от данни, които могат да се използват навсякъде, например “Address”, “Item”, “Payment”

- Малко подмножество XML схеми за общоприети бизнес документи, например “Order,” “Dispatch Advice” and “Invoice”, които могат да се използват в цялата бизнес верига: от поръчката до изпращане на фактурата.

- Поддръжка промяната на UBL от гледна точка на специфичните търговски взаимоотношения

Друг подобен проект е спецификацията ebXML (electronic business XML), който е насочен към унифициране не само формата, но и процеса на търговия. Той обаче е доста по-сложен и тежък за експлоатация и управление стандарт.

Трябва да се отбележи обаче, че въпреки добрите идеи, които са залегнали при проектирането на тези стандарти самата им интерпретация от отделните бизнес организации ги прави трудно интегрируеми, тъй като всеки бизнес ги интерпретира в тясно специализирана насока.

9 Заключение

Разгледаната система предоставя една добра основа за развитие на единен компонент за преобразуване на електронни документи от партньорски организации. При евентуалното й развитие, с добавянето на други входящи формати като SAP IDOC (XML и линеен формат), EDI, а също и чрез обработването на електронни фактури, потвърждения на поръчки и т.н. все по-ясно ще започне да се очертава предимствата от използването на една лека, но заедно с това гъвкава, отворена и силно поддаваща се на конфигуриране архитектура.

Заедно с това обаче и на този етап от развитието й могат да се набележат някои проблемни области, които изискват по-сериозно внимание. Основните подобрения в системата трябва да се извършат за увеличаване на бързодействието й и времето й за отговор, тъй като това е потенциално "тясно място" за приложението. Например едно възможно решение е обработката на документите да става асинхронно, т.е. да се запише входящия документ, да се върне отговор на клиента за успешното му получаване и след това последващото му преобразуване да се извършва отделно, в друга нишка (Thread в термините на Java) от системата. Също така е твърде вероятно при вграждането на други типове електронни документи да се открият еднакви XML елементи, а даже и последователности от такива елементи. За тази цел най-добре ще е съответните схеми да бъдат разбити на подсхеми, а главните да съдържат указател (в термините на XML схемите) към тях. Друго евентуално подобрение може да се приложи на ниво "сигурност на системата". Тъй като самите поръчки могат да съдържат поверителна информация (например номерът на кредитната карта, през която ще се извършва разплащането, цената с отстъпка на даде артикул и други), то е нужно за в бъдеще документите да се записват кодирани в базата, за да се избегне евентуален нежелан достъп до тази информация.

Разбира се, настоящата дипломна работа не може да предостави пълната бизнес функционалност на една индустриална система, но тя и не отстъпва

по възможности за трансформация, тъй като за тази цел се използва пълната функционалност на езика за програмиране Java. Също така трябва да се отбележи, че вътрешния стандарт е отворен, разширяем и не е обвързан тясно с нито една комерсиална архитектура, за разлика от използваните технологии при BizTalk сървъра например. Това са и основните предимства на разгледания мидълуер компонент, който трябва да се запазят при евентуалното му развитие.

10 Използвана литература

[1] Luis Ennser, Pietro Leo, Tamas Meszaros, Eric Valade, *"The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications"*, IBM RedBooks, September 2000: 31-33

[2] Philip Bernstein, *"Middleware: A Model for Distributed Services."* Communications of the ACM 39, 2 (February 1996): 86-97

[3] Richard Schreiber, *"Middleware Demystified."* Datamation 41, 6 (April 1, 1995): 41-45.

[4] Kurt Gabrick, David Weiss (2002), *"J2EE and XML Development"*, Manning Publications Co., ISBN: 1-930110-30-8

[5] Mark Birbeck, Jason Diamond, Jon Ducket, *"Professional XML 2nd Edition"*, Wrox Press Ltd., ISBN: 1861005059

[6] RosettaNet - http://www.rosettanet.org/RosettaNet/

[7] cXML - http://www.cxml.org/

[8] xCBL - http://www.xcbl.org/

[9] OAGIS - http://www.openapplications.org/

[10] James Cooper (1998), *"The Design Patterns Java Companion"*: 31-33

[11] Distributed Computing Environment - http://www.opengroup.org/dce/

[12] Common Object Request Broker Architecture - http://www.corba.org/

[13] Microsoft COM - http://www.microsoft.com/com

[14] Enterprise Java Beans - http://java.sun.com/products/ejb/

[15] Dickman, "*Two-Tier Versus Three-Tier Applications*" Informationweek 553 (November 13, 1995): 74-80

[16] Birrell, Nelson, "*Implementing Remote Procedure Calls*", ACM Transactions on Computer Systems 2, 1 (February 1984): 39-59

[17] Andrew Wade, "*Distributed Client-Server Databases*", Object Magazine 4, 1 (April 1994): 47-52

[18] Steve Steinke, "*Middleware Meets the Network*", LAN: The Network Solutions Magazine 10, 13 (December 1995): 56

[19] XSLT - http://www.w3.org/TR/xslt

[20] JAXB - http://java.sun.com/xml/jaxb/about.html

11 Приложения

В тази глава са предоставени инструкции за използване на прилежащия диск, както и съдържанието на разработената вътрешна схема.

11.1 Приложение А: Описание на придружаващия компакт диск

На съпътстващия компакт диск са предоставени програмния код на приложението, съответната документация и инсталациите на използвания софтуер.

11.1.1 Инсталации на използвания софтуер

В поддиректорията */Install* се намират инсталациите на използвания външен софтуер. По-долу е дадено кратко описание.

В настоящата дипломна работа се използва PostgreSQL 7.3.1. Alpha 1 for Windows за база данни, която няма опции къде да се инсталира, а използва директорията *C:\Program Files\PostgreSQL*. След инсталирането й следва да се редактират следните параметри от конфигурационния й файл (*Start Menu -> Programs -> PostgreSQL -> Adjust PostgreSQL Configuration File*): **tcpip_socket** трябва да се укаже на **true** (за да позволява базата връзки от BDI приложението) и **autocommit** на **false** (за да се управляват външно транзакциите). След стартиране на базата (*Start -> Programs -> PostgreSQL -> Utilities)* следва да се създаде административен потребител за базата: **createuser -a -d -P postgres**. След това и BDI схемата в базата от данни: **createdb -O postgres BDI**. Като следваща стъпка трябва да се създаде и езика за записани процедури, това става чрез командата: **createlang plpgsql BDI**. За да се запишат първоначалните данни в така създадената схема е нужно да се стартира интерактивния терминал и там да изпълним следната команда: **psql BDI postgres**. Инициализационните скриптове се намират в директорията */Source/db/BDI_scripts*. Те трябва да се запишат в директорията *C:\Program Files\PostgreSQL\BDI_scripts* и след това да се

изпълни командата **\i BDI_scripts_all.sql**. С това базата от данни е конфигурирана и готова за използване от приложението.

Тъй като системата е разработена на програмния език Java то в приложения компакт диск има инсталациите на Java Development Kit (JDK), версии 1.3 и 1.4. Те се намират в директория *Install/Java*. Приложението работи и със двете версии.

Като Web контейнер за тестовото приложение се използва Jakarta Tomcat, версия 4.1. Тя следва да се разархивира в директория по избор и след това да се нагласи конфигурационния параметър **JAVA_HOME**, в стартиращия файл *bin/catalina.bat*.

В директорията *Install/Maven* се намира инструмента Maven, който се използва за компилиране, автоматично тестване и изграждане на тестовото Web приложение. Най-общо казано той е следващо ниво в развитието на широко разпространения инструмент Ant.

Настоящата система е разработена на средата Eclipse, версия 3.0. Това е безплатна и силно разширяема среда, тъй като може да бъде интегрирана с множество добавки (plug-in). Инсталацията й се намира в директория *Install/Eclipse*, заедно с добавка от Together за генериране на UML диаграми. Важно при компилиране на проекта е да бъде зададена променливата **MAVEN_REPO**, която трябва да сочи към следната директория: **%MAVEN_INSTALL_DIR%/repository**. Такива променливи на средата в Eclipse се задават чрез менюто *Window -> Preferences -> Java -> Build Path -> Classpath Variables*.

11.1.2 Програмен код на системата и прилежащите приложения

В поддиректорията */Source* е предоставен изходният код на така представеното приложение.

В поддиректория */db/BDI_scripts* се намират скриптовете за генериране базата от данни, един за генериране на обектите (*BDI_DDL.sql*), един за

попълване на т.нар. домейн таблици (*populateDomainTables.sql*) и един за първоначално конфигуриране на разгледаните XML формати (*populateSourceSystems.sql*).

Директорията */Source/bdi* съдържа кодът на самия мидълуер компонент, заедно с файловете, което се изискват от инструмента Maven за неговото компилиране, тестване и деплойване (deployment). Това са *build.properties*, *maven.xml* и *project.xml*, които съдържат конфигурационните параметри на приложението като името на пакета, в който ще се генерират JAXB обектите, схемата на вътрешния BDI формат, директорията, в която се компилират резултатните класове и т.н. В директорията *bdi/lib* се намират външните библиотеки, например драйвъра за базата, Apache парсерите и други, а в директорията *bdi/src/java* е самият програмен код. Вътрешната XSD схема може да бъде намерена в *bdi/src/common_format* (също така тя е приложена и в Приложение Б: Вътрешен формат – BDI XSD). В поддиректорията *bdi/src/validation* се намират външните схеми за валидация, пътят до които, както вече бе отбелязано, се конфигурира през базата.

Третият проект, специфициран тук, се използва за тестови нужди и представлява разгледаното в точка 7.2 Web приложение.

За улеснение самото Web приложение е предоставено и като Web Archive (*Source/bdi-1.1.war*).

11.1.3 Документация на проекта

В поддиректорията */Documentation* може да се намери съпътстващата документация, включително и настоящия документ. В поддиректорията *Info* е предоставена част от разгледаната информацията за външните XML стандарти, самият XML език и други справочници, цитирани в за обосновка на настоящата дипломна работа. В директорията *Personal* се намират документи, Visio диаграми и картинки, изготвени по време на разработката на приложението.

11.2 Приложение Б: Вътрешен формат – BDI XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified">
	<xs:element name="BDIDocument" type="BDICommonDocument"/>
	<xs:complexType name="BDICommonDocument">
		<xs:annotation>
			<xs:documentation>root element of BDI Order Request document</xs:documentation>
		</xs:annotation>
		<xs:sequence>
			<xs:element ref="Header"/>
			<xs:element ref="Body"/>
		</xs:sequence>
	</xs:complexType>
	<xs:element name="Header">
		<xs:complexType mixed="false">
			<xs:annotation>
				<xs:documentation>Header Part of BDI Order Request
document</xs:documentation>
			</xs:annotation>
			<xs:sequence>
				<xs:element ref="IncomingMessageInfo"/>
				<xs:element ref="Document"/>
				<xs:element ref="Delivery"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="Document">
		<xs:complexType mixed="false">
			<xs:sequence>
				<xs:element name="Description" type="xs:string" minOccurs="0"/>
			</xs:sequence>
			<xs:attribute name="docType" type="xs:string" use="required">
				<xs:annotation>
					<xs:documentation>Classification of Message</xs:documentation>
				</xs:annotation>
			</xs:attribute>
			<xs:attribute name="proprietaryDocID" type="xs:string" use="required">
				<xs:annotation>
					<xs:documentation>Proprietary Identification</xs:documentation>
				</xs:annotation>
			</xs:attribute>
		</xs:complexType>
	</xs:element>
	<xs:element name="Delivery">
		<xs:complexType mixed="false">
			<xs:sequence>
				<xs:element ref="To"/>
				<xs:element ref="From"/>
			</xs:sequence>
		</xs:complexType>
	</xs:element>
	<xs:element name="IncomingMessageInfo">
		<xs:complexType mixed="false">
			<xs:sequence>
				<xs:element ref="SentDate" minOccurs="0">
					<xs:annotation>
						<xs:documentation>Date when the document was sent,
equals to receive date if does not exist for this specific type of document</xs:documentation>
					</xs:annotation>
				</xs:element>
				<xs:element ref="ReceivedDate">
					<xs:annotation>
						<xs:documentation>Date when the document was received in
BDI, could be T_MESSAGE.DATE_CREATED </xs:documentation>
					</xs:annotation>
				</xs:element>
			</xs:sequence>
			<xs:attribute name="sourceSystem" type="xs:string" use="required">
				<xs:annotation>
					<xs:documentation>The name of the system, which has sent the
document, e.g. RosettaNet, cXML, etc</xs:documentation>
```

```
                                        </xs:annotation>
                                </xs:attribute>
                                <xs:attribute name="parentMsgID" type="xs:int" use="required">
                                        <xs:annotation>
                                                <xs:documentation>The ID from T_MESSAGE table of incoming
document</xs:documentation>
                                        </xs:annotation>
                                </xs:attribute>
                        </xs:complexType>
                </xs:element>
                <xs:element name="ReceivedDate" type="xs:string"/>
                <xs:element name="SentDate" type="xs:string"/>
                <!-- End of Message Elements -->
                <xs:element name="To" type="PartnerSystem"/>
                <xs:element name="From" type="PartnerSystem"/>
                <xs:complexType name="PartnerSystem">
                        <xs:annotation>
                                <xs:documentation>Identifies the system, which sent or receive a bussiness
document</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                                <xs:element name="GlobalSystemIdentifier" type="xs:string"/>
                                <xs:element name="Name" type="xs:string" minOccurs="0"/>
                                <xs:element name="DivisionCode" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                </xs:complexType>
                <!--
                        Body elements
                -->
                <xs:element name="Body">
                        <xs:complexType mixed="false">
                                <xs:annotation>
                                        <xs:documentation>Body Part of BDI Order Request
document</xs:documentation>
                                </xs:annotation>
                                <xs:sequence>
                                        <xs:element ref="PurchaseOrder"/>
                                </xs:sequence>
                        </xs:complexType>
                </xs:element>
                <xs:element name="PurchaseOrder">
                        <xs:complexType mixed="false">
                                <xs:annotation>
                                        <xs:documentation>root element of Order</xs:documentation>
                                </xs:annotation>
                                <xs:sequence>
                                        <xs:element ref="OrderHeader" maxOccurs="unbounded"/>
                                        <xs:element name="Currency" type="CurrencyType"/>
                                        <xs:element ref="Instructions" minOccurs="0" maxOccurs="unbounded"/>
                                        <xs:element ref="DateInfo" maxOccurs="unbounded"/>
                                        <xs:element ref="DocumentReference" maxOccurs="unbounded"/>
                                        <xs:element ref="CustomerReference" maxOccurs="unbounded"/>
                                        <xs:element ref="Identifier" minOccurs="0"/>
                                        <xs:element ref="Payment" minOccurs="0" maxOccurs="unbounded"/>
                                        <xs:element ref="CardInfo" minOccurs="0"/>
                                        <xs:element ref="Discount" minOccurs="0" maxOccurs="unbounded"/>
                                        <xs:element ref="OrderItem" maxOccurs="unbounded"/>
                                        <xs:element ref="OrderSummary" minOccurs="0"/>
                                        <xs:element ref="Tax" minOccurs="0"/>
                                        <xs:element ref="Transport" minOccurs="0"/>
                                </xs:sequence>
                        </xs:complexType>
                </xs:element>
                <!--
                        Customer Reference elements
                -->
                <xs:element name="CustomerReference">
                        <xs:complexType>
                                <xs:annotation>
                                        <xs:documentation>Delivery Information: Ship-to, Invoice-to, Sold-to, Bill-
to</xs:documentation>
                                </xs:annotation>
                                <xs:sequence>
                                        <xs:element ref="BusinessPartner"/>
                                        <xs:element ref="ContactInfo" minOccurs="0" maxOccurs="unbounded"/>
                                        <xs:element ref="Address" minOccurs="0"/>
```

```
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="BusinessPartner">
                    <xs:complexType>
                        <xs:annotation>
                            <xs:documentation>Customer Number</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                            <xs:element name="PartnerName" type="xs:string" minOccurs="0"/>
                            <xs:element name="PartnerID" type="xs:string"/>
                            <xs:element name="PartnerCode" type="xs:string" minOccurs="0"/>
                            <xs:element name="PartnerDescription" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="typeID" use="required">
                            <xs:simpleType>
                                <xs:annotation>
                                    <xs:documentation>SHP = Ship-to; SLD = Sold-to; BIL = Bill-
to; INV = Invoice-to; FBY = Financed By; INS = Installed At; CBN = Customer Base Number;PUR = Purchaser;SUP =
Supplier; VDR = Vendor; PTR = Partnership ID; </xs:documentation>
                                </xs:annotation>
                                <xs:restriction base="xs:NMTOKEN">
                                    <xs:enumeration value="SHP"/>
                                    <xs:enumeration value="SLD"/>
                                    <xs:enumeration value="BIL"/>
                                    <xs:enumeration value="INV"/>
                                    <xs:enumeration value="FBY"/>
                                    <xs:enumeration value="INS"/>
                                    <xs:enumeration value="CBN"/>
                                    <xs:enumeration value="PUR"/>
                                    <xs:enumeration value="SUP"/>
                                    <xs:enumeration value="VDR"/>
                                    <xs:enumeration value="PTR"/>
                                </xs:restriction>
                            </xs:simpleType>
                        </xs:attribute>
                    </xs:complexType>
                </xs:element>
                <!-- Start of Address elements -->
                <xs:element name="Address">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element ref="Company" minOccurs="0"/>
                            <xs:element ref="Division" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:choice minOccurs="0">
                                <xs:element ref="Street" maxOccurs="unbounded"/>
                                <xs:element name="POBox" type="xs:string"/>
                            </xs:choice>
                            <xs:element ref="City" minOccurs="0"/>
                            <xs:element ref="State" minOccurs="0"/>
                            <xs:element ref="ZIPCode" minOccurs="0"/>
                            <xs:element ref="ISOCountryCode" minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Company" type="xs:string">
                    <xs:annotation>
                        <xs:documentation>Company name</xs:documentation>
                    </xs:annotation>
                </xs:element>
                <xs:element name="Division" type="xs:string"/>
                <xs:element name="ISOCountryCode">
                    <xs:annotation>
                        <xs:documentation>Country code</xs:documentation>
                    </xs:annotation>
                    <xs:simpleType>
                        <xs:restriction base="xs:string">
                            <xs:minLength value="0"/>
                            <xs:maxLength value="3"/>
                        </xs:restriction>
                    </xs:simpleType>
                </xs:element>
                <xs:element name="State" type="xs:string">
                    <xs:annotation>
                        <xs:documentation>State/County/Province</xs:documentation>
                    </xs:annotation>
```

```
		</xs:element>
		<xs:element name="Street" type="xs:string"/>
		<xs:element name="ZIPCode">
			<xs:annotation>
				<xs:documentation>Post/ZIP code</xs:documentation>
			</xs:annotation>
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:maxLength value="12"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<xs:element name="City" type="xs:string">
			<xs:annotation>
				<xs:documentation>City</xs:documentation>
			</xs:annotation>
		</xs:element>
		<!-- End of Address elements -->
		<xs:element name="ContactInfo">
			<xs:complexType>
				<xs:annotation>
					<xs:documentation>Common contact information</xs:documentation>
				</xs:annotation>
				<xs:sequence>
					<xs:element name="ContactName" type="xs:string"/>
					<xs:element name="ContactID" type="xs:string" minOccurs="0"/>
					<xs:element name="ContactDivision" type="xs:string" minOccurs="0"/>
					<xs:element ref="ContactPhone" minOccurs="0"/>
					<xs:element ref="ContactPhoneHome" minOccurs="0"/>
					<xs:element ref="ContactPhoneMobile" minOccurs="0"/>
					<xs:element ref="ContactFax" minOccurs="0"/>
					<xs:element ref="ContactEmail" minOccurs="0"/>
				</xs:sequence>
			</xs:complexType>
		</xs:element>
		<xs:element name="ContactEmail">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:maxLength value="192"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<xs:element name="ContactFax">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:maxLength value="25"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<xs:element name="ContactPhone">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:maxLength value="25"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<xs:element name="ContactPhoneHome">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:maxLength value="25"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<xs:element name="ContactPhoneMobile">
			<xs:simpleType>
				<xs:restriction base="xs:string">
					<xs:maxLength value="25"/>
				</xs:restriction>
			</xs:simpleType>
		</xs:element>
		<!--
		End of Customer Reference elements
-->
		<!--
		Instruction elements
-->
```

```
		<xs:element name="Instructions">
			<xs:complexType>
				<xs:annotation>
					<xs:documentation>Specific instructions, which could be added during then
processing of the message.
		Could be at header or at item level</xs:documentation>
				</xs:annotation>
				<xs:sequence>
					<xs:element name="InternalCode" type="xs:string" minOccurs="0"/>
					<xs:element name="InstrCode" type="xs:string" minOccurs="0"/>
					<xs:element name="InstrCodeDesc" type="xs:string" minOccurs="0"/>
					<xs:element name="InstrValue" type="xs:string" minOccurs="0"/>
				</xs:sequence>
				<xs:attribute name="instrType" use="required">
					<xs:simpleType>
						<xs:annotation>
							<xs:documentation>INF = Info; WARN=Warning instructions;
ERR = Error Instructions;MARKER= Marking Comments; </xs:documentation>
						</xs:annotation>
						<xs:restriction base="xs:NMTOKEN">
							<xs:enumeration value="INF"/>
							<xs:enumeration value="WARN"/>
							<xs:enumeration value="ERR"/>
							<xs:enumeration value="MARKER"/>
						</xs:restriction>
					</xs:simpleType>
				</xs:attribute>
			</xs:complexType>
		</xs:element>
		<!--
		End of Instruction elements
-->
		<!--
		Order Header, info element
-->
		<xs:element name="OrderHeader">
			<xs:complexType>
				<xs:annotation>
					<xs:documentation>Header Details</xs:documentation>
				</xs:annotation>
				<xs:sequence>
					<xs:element ref="BusinessDesc" minOccurs="0"/>
					<xs:element ref="BusinessID" minOccurs="0"/>
				</xs:sequence>
				<xs:attribute name="orderType" type="xs:string" use="required"/>
			</xs:complexType>
		</xs:element>
		<xs:element name="BusinessDesc" type="xs:string"/>
		<xs:element name="BusinessID" type="xs:string"/>
		<!--
		End of Order Header
-->
		<!--
		Start of Credit Card info
-->
		<xs:element name="CardInfo">
			<xs:complexType>
				<xs:sequence>
					<xs:element name="CardType" type="xs:string" minOccurs="0"/>
					<xs:element name="CardTypeOther" type="xs:string" minOccurs="0"/>
					<xs:element name="CardNum" type="xs:string"/>
					<xs:element name="CardExpirationDate" type="xs:string" minOccurs="0"/>
					<xs:element name="AllocatedAmount" type="xs:float" minOccurs="0"/>
					<xs:element name="CardAuthCode" type="xs:string" minOccurs="0"/>
					<xs:element name="CardAuthInfo" type="xs:string" minOccurs="0"/>
					<xs:element name="CardHolderName" type="xs:string" minOccurs="0"/>
				</xs:sequence>
			</xs:complexType>
		</xs:element>
		<!--
		End of Credit Card info
-->
		<!--
		Start of Tax element
-->
		<xs:element name="Tax">
```

```
                    <xs:complexType>
                        <xs:annotation>
                            <xs:documentation>Tax Details</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                            <xs:element name="TaxPercent" type="xs:float" minOccurs="0"/>
                            <xs:element name="TaxCurrency" type="CurrencyType" minOccurs="0"/>
                            <xs:element name="TaxAmount" type="xs:double" minOccurs="0"/>
                            <xs:element name="TaxDescription" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <!--
                End of Tax element
-->
                <!--
                Start of Currency element
-->
                <xs:complexType name="CurrencyType">
                    <xs:annotation>
                        <xs:documentation>Common Currency details</xs:documentation>
                    </xs:annotation>
                    <xs:sequence>
                        <xs:element name="PrimaryCurrency" type="ISOCurrency"/>
                        <xs:element name="SecondaryCurrency" type="ISOCurrency" minOccurs="0"/>
                        <xs:element ref="ExchangeRate" minOccurs="0"/>
                    </xs:sequence>
                </xs:complexType>
                <xs:simpleType name="ISOCurrency">
                    <xs:annotation>
                        <xs:documentation>3 digit ISO currency Code</xs:documentation>
                    </xs:annotation>
                    <xs:restriction base="xs:string">
                        <xs:minLength value="1"/>
                        <xs:maxLength value="3"/>
                    </xs:restriction>
                </xs:simpleType>
                <xs:element name="ExchangeRate" type="xs:float"/>
                <!--
                End of Currency element
-->
                <!--
                Start of DocumentReference element
-->
                <xs:element name="DocumentReference">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element ref="DocumentID"/>
                            <xs:element ref="DocType" minOccurs="0"/>
                            <xs:element ref="DocumentCode" minOccurs="0"/>
                            <xs:element ref="DocumentDesc" minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="DocumentDesc" type="xs:string"/>
                <xs:element name="DocumentID" type="xs:string"/>
                <xs:element name="DocType" type="xs:string"/>
                <xs:element name="DocumentCode" type="xs:string"/>
                <!--
                End of DocumentReference element
-->
                <!--
                Start of Identifier element
-->
                <xs:element name="Identifier">
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="UserLogin" type="xs:string" minOccurs="0"/>
                            <xs:element name="UserIdentifier" type="xs:string" minOccurs="0"/>
                            <xs:element name="EmployeeIdentifier" type="xs:string" minOccurs="0"/>
                            <xs:element name="RequestorIdentifier" type="xs:string" minOccurs="0"/>
                            <xs:element name="BuyerApproverIdentifier" type="xs:string" minOccurs="0"/>
                            <xs:element name="BuyerApproverCompanyName" type="xs:string"
minOccurs="0"/>
                            <xs:element name="VendorApproverIdentifier" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
```

```
                </xs:complexType>
        </xs:element>
        <!--
        End of Identifier element
-->
        <!--
        Start of Payment element
-->
        <xs:element name="Payment">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="PaymentType" type="xs:string"/>
                                <xs:element name="PaymenTermCode" type="xs:string" minOccurs="0"/>
                                <xs:element name="PaymenTerm" type="xs:string" minOccurs="0"/>
                                <xs:element name="PaymenTermOther" type="xs:string" minOccurs="0"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <!--
        End of Payment element
-->
        <!--
        Start of Transport element
-->
        <xs:element name="Transport">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element ref="TransportDate"/>
                                <xs:element ref="TransportMode"/>
                                <xs:element ref="TransportModeOther" minOccurs="0"/>
                                <xs:element ref="TransportMean" minOccurs="0"/>
                                <xs:element ref="TransportMeanOther" minOccurs="0"/>
                                <xs:element ref="TransportContractNumber" minOccurs="0"/>
                                <xs:element ref="TransportInstructions" minOccurs="0"/>
                        </xs:sequence>
                </xs:complexType>
        </xs:element>
        <xs:element name="TransportDate" type="xs:string"/>
        <xs:element name="TransportMode" type="xs:string"/>
        <xs:element name="TransportModeOther" type="xs:string"/>
        <xs:element name="TransportMean" type="xs:string"/>
        <xs:element name="TransportMeanOther" type="xs:string"/>
        <xs:element name="TransportContractNumber" type="xs:string"/>
        <xs:element name="TransportInstructions" type="xs:string"/>
        <!--
        End of Transport element
-->
        <!--
        Start of DateInfo element
        -->
        <xs:element name="DateInfo">
                <xs:complexType>
                        <xs:annotation>
                                <xs:documentation>Date information</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                                <xs:element ref="DateTimeUTC"/>
                                <xs:element ref="DateDesc" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="dateType" use="required">
                                <xs:annotation>
                                        <xs:documentation>Date type</xs:documentation>
                                </xs:annotation>
                                <xs:simpleType>
                                        <xs:annotation>
                                                <xs:documentation>EAD = earliest arrival date; LAD = Latest
arrival date; ORD = order date; RDD = requested delivery date; SBD = ship-by date </xs:documentation>
                                        </xs:annotation>
                                        <xs:restriction base="xs:NMTOKEN">
                                                <xs:enumeration value="EAD"/>
                                                <xs:enumeration value="LAD"/>
                                                <xs:enumeration value="ORD"/>
                                                <xs:enumeration value="RDD"/>
                                                <xs:enumeration value="SBD"/>
                                        </xs:restriction>
                                </xs:simpleType>
```

```
                                </xs:attribute>
                        </xs:complexType>
                </xs:element>
                <xs:element name="DateTimeUTC">
                        <xs:annotation>
                                <xs:documentation>Date in UTC format e.g. YYYYMMDD hhmmss +/-
zzzz</xs:documentation>
                        </xs:annotation>
                        <xs:simpleType>
                                <xs:restriction base="xs:string"/>
                        </xs:simpleType>
                </xs:element>
                <xs:element name="DateDesc" type="xs:string">
                        <xs:annotation>
                                <xs:documentation>Description of date</xs:documentation>
                        </xs:annotation>
                </xs:element>
                <!--
                        End of DateTime
                -->
                <!-- Start of Discount element -->
                <xs:element name="Discount">
                        <xs:complexType>
                                <xs:annotation>
                                        <xs:documentation>Discount / special deal information DiscounType OPG = opg
dept code; DIS = discount</xs:documentation>
                                </xs:annotation>
                                <xs:sequence>
                                        <xs:element ref="DiscountCode" minOccurs="0"/>
                                        <xs:element ref="DiscountID" minOccurs="0"/>
                                        <xs:element ref="DiscountAmount" minOccurs="0"/>
                                        <xs:element ref="DiscountPercent" minOccurs="0"/>
                                        <xs:element name="DiscountDaysDuration" type="xs:int" minOccurs="0"/>
                                        <xs:element ref="DiscountDueDate" minOccurs="0"/>
                                </xs:sequence>
                                <xs:attribute name="discountType" use="required">
                                        <xs:annotation>
                                                <xs:documentation>Discount type</xs:documentation>
                                        </xs:annotation>
                                        <xs:simpleType>
                                                <xs:annotation>
                                                        <xs:documentation>REQ = discount requested; GRT =
discount granted;</xs:documentation>
                                                </xs:annotation>
                                                <xs:restriction base="xs:NMTOKEN">
                                                        <xs:enumeration value="REQ"/>
                                                        <xs:enumeration value="GRT"/>
                                                </xs:restriction>
                                        </xs:simpleType>
                                </xs:attribute>
                        </xs:complexType>
                </xs:element>
                <xs:element name="DiscountID" type="xs:string">
                        <xs:annotation>
                                <xs:documentation>The M-code for the OPG</xs:documentation>
                        </xs:annotation>
                </xs:element>
                <xs:element name="DiscountAmount" type="xs:float"/>
                <xs:element name="DiscountCode" type="xs:string">
                        <xs:annotation>
                                <xs:documentation>Description of discount</xs:documentation>
                        </xs:annotation>
                </xs:element>
                <xs:element name="DiscountDueDate" type="xs:string"/>
                <xs:element name="DiscountPercent" type="xs:float">
                        <xs:annotation>
                                <xs:documentation>Discount Amount expressed as percentage</xs:documentation>
                        </xs:annotation>
                </xs:element>
                <!--
                        End of Discount element
                -->
                <!--
                        Start of OrderSummary
                -->
                <xs:element name="OrderSummary">
```

```
                    <xs:complexType>
                        <xs:sequence>
                            <xs:element name="PrimaryCurrencyTotal" type="xs:float" minOccurs="0"/>
                            <xs:element name="SecondaryCurrencyTotal" type="xs:float" minOccurs="0"/>
                            <xs:element name="TotalLinesCount" type="xs:int" minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <!--
                    End of OrderSummary
                -->
                <!--
                    Start of OrderItem element
                -->
                <xs:element name="OrderItem">
                    <xs:complexType>
                        <xs:annotation>
                            <xs:documentation>Line Item</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                            <xs:choice>
                                <xs:element ref="Item"/>
                                <xs:element ref="BundleItem"/>
                            </xs:choice>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <!-- Item elemnt -->
                <xs:element name="Item">
                    <xs:complexType>
                        <xs:annotation>
                            <xs:documentation>Item Information for simple Item structure</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                            <xs:element ref="LineNumber"/>
                            <xs:element ref="ParentItemNumber" minOccurs="0"/>
                            <xs:element name="LegacyItemNumber" type="xs:string" minOccurs="0"/>
                            <xs:element ref="ItemNumber" minOccurs="0"/>
                            <xs:element ref="ProductInformation" maxOccurs="unbounded"/>
                            <xs:element name="Currency" type="CurrencyType" minOccurs="0"/>
                            <xs:element ref="Quantity"/>
                            <xs:element ref="Price" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="Instructions" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="DocumentReference" minOccurs="0"
maxOccurs="unbounded"/>
                            <xs:element ref="CustomerReference" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="Discount" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="DateInfo" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="Transport" minOccurs="0"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="ItemNumber" type="xs:string"/>
                <xs:element name="LineNumber" type="xs:int">
                    <xs:annotation>
                        <xs:documentation>Sequential Number per item</xs:documentation>
                    </xs:annotation>
                </xs:element>
                <xs:element name="ParentItemNumber" type="xs:int"/>
                <!-- Price Elements -->
                <xs:element name="Price">
                    <xs:complexType>
                        <xs:annotation>
                            <xs:documentation>Customer Price</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                            <xs:element ref="PriceDescription" minOccurs="0"/>
                            <xs:element ref="PriceAmount"/>
                        </xs:sequence>
                        <xs:attribute name="priceType">
                            <xs:simpleType>
                                <xs:annotation>
                                    <xs:documentation>BDP = Buyer discounted price; VDP =
Vendor discounted price, DSP = discounted price (from storefront)</xs:documentation>
                                </xs:annotation>
                                <xs:restriction base="xs:NMTOKEN">
```

```
                                                <xs:enumeration value="BDP"/>
                                                <xs:enumeration value="VDP"/>
                                                <xs:enumeration value="DSP"/>
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
        <xs:element name="PriceAmount" type="xs:float"/>
        <xs:element name="PriceDescription" type="xs:string"/>
        <!-- end of Price elements -->
        <!-- Quantity elements -->
        <xs:element name="Quantity">
                <xs:complexType>
                        <xs:sequence>
                                <xs:element name="TotalQuantity" type="xs:int"/>
                                <xs:element ref="DescriptionUOM" minOccurs="0"/>
                        </xs:sequence>
                        <xs:attribute name="unitOfMeasure" type="xs:string">
                                <xs:annotation>
                                        <xs:documentation>Unit of measure code</xs:documentation>
                                </xs:annotation>
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
        <xs:element name="DescriptionUOM" type="xs:string">
                <xs:annotation>
                        <xs:documentation>Unit of measure description</xs:documentation>
                </xs:annotation>
        </xs:element>
        <!-- end of Quantity elements -->
        <!-- ProductInformation elements -->
        <xs:element name="ProductInformation">
                <xs:complexType>
                        <xs:annotation>
                                <xs:documentation>Product details (e.g. code, description
etc.)</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                                <xs:element ref="ProductDescription" minOccurs="0"/>
                                <xs:element ref="ProductNumber"/>
                                <xs:element ref="ProductLine" minOccurs="0"/>
                                <xs:element ref="ProductCommodityCode" minOccurs="0"/>
                                <xs:element ref="ProductName" minOccurs="0"/>
                                <xs:element ref="ProductOption" minOccurs="0" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="productCode" use="required">
                                <xs:simpleType>
                                        <xs:annotation>
                                                <xs:documentation>VPN = Vendor Part number; BPN = Buyer
Part number; BIP = Buyer Internal Product; SPN = Supplier Part Number</xs:documentation>
                                        </xs:annotation>
                                        <xs:restriction base="xs:NMTOKEN">
                                                <xs:enumeration value="VPN"/>
                                                <xs:enumeration value="BPN"/>
                                                <xs:enumeration value="BIP"/>
                                                <xs:enumeration value="SPN"/>
                                        </xs:restriction>
                                </xs:simpleType>
                        </xs:attribute>
                </xs:complexType>
        </xs:element>
        <xs:element name="ProductLine" type="xs:string"/>
        <xs:element name="ProductNumber" type="xs:string">
                <xs:annotation>
                        <xs:documentation>Product reference number</xs:documentation>
                </xs:annotation>
        </xs:element>
        <xs:element name="ProductCommodityCode" type="xs:string"/>
        <xs:element name="ProductName" type="xs:string"/>
        <xs:element name="ProductDescription" type="xs:string"/>
        <xs:element name="ProductOption">
                <xs:complexType>
                        <xs:annotation>
                                <xs:documentation>Product Options</xs:documentation>
                        </xs:annotation>
```

```
                        <xs:sequence>
                            <xs:element ref="OptionDescription" minOccurs="0"/>
                            <xs:element ref="Option"/>
                        </xs:sequence>
                    </xs:complexType>
                </xs:element>
                <xs:element name="Option" type="xs:string"/>
                <xs:element name="OptionDescription" type="xs:string"/>
                <!-- End of ProductInformation elements -->
                <!-- BundleItem elements -->
                <xs:element name="BundleItem">
                    <xs:complexType>
                        <xs:annotation>
                            <xs:documentation>Information for bundled Items</xs:documentation>
                        </xs:annotation>
                        <xs:sequence>
                            <xs:element ref="ProductInformation" maxOccurs="unbounded"/>
                            <xs:element ref="Quantity"/>
                            <xs:element ref="Price" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="DateInfo" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="Instructions" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="CustomerReference" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="Discount" minOccurs="0" maxOccurs="unbounded"/>
                            <xs:element ref="Item" maxOccurs="unbounded"/>
                        </xs:sequence>
                        <xs:attribute name="bundleItemNumber" type="xs:string" use="required"/>
                        <xs:attribute name="parentItemNumber" type="xs:string"/>
                        <xs:attribute name="itemType" type="xs:string"/>
                    </xs:complexType>
                </xs:element>
                <!--
                        End of BundleItem
                -->
                <!--
                        End of OrderItem elements
                -->
</xs:schema>
```