Софийски университет „Св. Климент Охридски”
Факултет по математика и информатика
Катедра „Информационни технологии”

ДИПЛОМНА РАБОТА

на тема

„Генератор на справки и SQL процесор”

Дипломант: **Илиян Михайлов Димов**
Специалност: **Разпределени системи и мобилни технологии**
Факултетен номер: **М-21524**

Научен ръководител:
доц. Боян Паскалев Бончев

гр. София, юни, 2007 г.

СЪДЪРЖАНИЕ

1. Предназначение на продукта и характеристики на наличните продукти на пазара.

Преди да характеризираме подробно възможностите на продукта ще пристъпим към описание на средата в която той може да се използва. Обикновено на пазара съществуват много пазарни субекти, които комуникират по между си. Тези субекти могат да бъдат търговци, доставчици, клиенти и други. Общото за всички от тях е, че те разполгат с данни за пазара в конкретната област от него, която обслужват. Другото общо нещо е, че те също така се нуждаят от допълнително информация за да могат да изпълняват своите функции. Така например търговците могат да се нуждаят от информация свързана с търсенето от страна на клиентите на дадени продукти или пък наличното количество продукция, което всеки доставчик може да им достави. Доставчиците пък от своя страна могат да се нуждаят от информация свързана с възможните начини и цени на доставка на даден продукт от дадено място по каналите за дистрибуция. Клиентите също могат да се нуждаят от информация свързана с възможните места, от които биха могли да закупят даден продукт. И така ако трябва да обобщим горните случаи, то трябва да кажем че за всички от тях е характерно това, че първо данните трябва да бъдат извлечени от съответните източници, консолидирани по някакъв начин и след това да бъдат анализирани, за да може да извлечем подходящата информация от тях. Освен този модел на обработка на данните съществуват още и много други като например обработка на данни в реално време, вземане на решения в реално време и други такива, на които ние няма да се спираме в детайли. И така нека се върнем на горният разглеждан модел. Графично той може да се представи и разгледа по следният начин :

Анализирайки горната графика от пръв поглед става ясно, че за да може да извлечем данните трябва да разполагаме с необходимите технически средсва за да извършим това. За да знаем какви точно технически средства да използваме трябва да знаем начинът по който данните са съхранени. Най - често данните се съхраняват в бази от данни, за да могат лесно да бъдат достъпвани, редактирани, трансформирани и анализирани. Като свободна единица на пазара всеки субект има правото да избира сам средствата, които да използва за да съхрани своите данни, както и да избере сам начините за достъп до тях, които да предостави на останалите. В резултат на това се получава така, че за да се извлекат данните трябва да се вземе под внимание фактът, че средата не е хомогенна. Това е една от причите, поради която е решено в проекта да се използва Java. Освен, че средата не е хомогенна трябва да се вземе в предвид и това, че тя може да се изменя, което от своя страна налага структурата на проекта на бъде гъвкава и адаптивна към тези изменения. От изброените до тук неща следва да се направи изводът, че целта на проекта се явява именно възможността да се предостави начин на потребителя да достъпва и обработва данните, от които се нуждае. Най – честата форма, под която потребителят иска да разглежда данните се явява някакъв вид справка, чието използване е насочено към задоволяване на конкретна нужда от информация. Освен да използва справки, потребителят трябва има възможност да създава свои собствени такива, чиято употреба е свързана със задаволяване на някакви конкретни нови нужди.

От всички налични на пазара продукти няма такъв, който да ни предложи средства с които да генерираме справки използвайки едновременно данните от различни източници. Именно поради тази причина е създадена тази система, която има за цел да осигури цялостен подход и решение свързано с лесното задаволяване на нуждите на потребителите при изграждането на справки.

2. Описание на използваните технически средства

Като основно средство за разработка на продукта е използван Eclipse 3.1 , чиито добри характеристики са напълно достатъчни за разработването на продукта. Освен това са използвани и съответните бази данни, които продуктът поддържа – MS SQL, My SQL, HSQL за тестването на продукта, а също така DBVisualizer – за разглеждане на съдържанието на базата данни и XML Spy – за разглеждане на XML файловете и схемите.

Описание на използваните библиотеки:

1. Библиотеки за достъп до бази данни.

Име на библиотеката	Предназначение на библиотеката
hsqldb-1_7_3_3.jar	Осигурява достъп до hsql базата данни.
msbase_2000_SP3.jar *mssqlserver_2000_SP3.jar* *msutil_2000_SP3.jar*	Осигурява достъп до ms sql базата данни.
mysql-connector-3_1_12.jar	Осигурява достъп до my sql базата данни.

2. Библитеки за парсване на xml файлове.

Име на библиотеката	Предназначение на библиотеката
xercesImpl.jar *xml-apis.jar*	Осигурява средства за парсване на xml файлове посредством DOM парсер.

3. Описание на продукта.

Като встъпителна част при разясняването на продукта трябва да се вземат в предвид функциите, които той предоставя на крайния потребител. За тази цел предварително трябва да се изяснят средствата, които той може да използва за генерирането на справката и стъпките, които трябва да бъдат използвани, за да бъде извършено това. Имайки в предвид, че справката може да бъде доста комплексна, това означава, че средствата използвани за нейното описание също ще бъдат доста комплексни. Изхождайки от този факт се стига до заключението, че оптималният вариант за представянето на справката е тя да бъде организирана в някакъв workflow. Оттук се подразбира и става ясно, че колкото по - сложна е справката толкова по - комплексен ще бъде workflow-а т.е., че има възможност да се генерира произволно сложна справка стига да може да бъде генериран workflow-а за нея. Друг извод, който се налага от всичко казано е, че за да може да се създаде или опише една справка, трябва да се познават добре средствата, с които тя ще бъде реализирана във workflow - а т.е. че възможностите на workflow - а за описание на справката пряко рефлектират върху съдържанието на самата справка. За да се осигури гъвкав workflow трябва да се вземе в предвид с какви средства ще бъде реализиран той и доколко ще може да отговаря на изискванията за разширяемост, гъвкавост и адаптивност. Освен всичко друго workflow - ът трябва да може да бъде четен и променян от потребителя в зависимост от нуждите на справката. Именно поради тази причина като основно средство за описание на workflow - а е избран езика XML. Предимството на XML като средство за описание на workflow - а е свързано с това, че той е лесен за четене, писане и модифициране от човека. Освен това той е разширяем посредством използването на namespace-и и лесен за структуриране, което го прави подхоящ за изброените по - горе цели.

3.1 Описание на средствата предоставяни от продукта.

Избора на XML като средство за описание на workflow - а предполага използването на тагове за да бъде извършено това. Таговете могат да се разделят на различни видове в зависимост от тяхното предназначение. Като цяло таговете служат за описание на логиката необходима за генериране на справката или пък за описание на част от данните необходими за реализацията на логиката. Таговете могат да съдържат в себе си други тагове, което позволя да се разширява лесно тяхната логика и комплексност, а също така да се запазва до известна степен и обратна съвместимост. От казаното до тук следва изводът, че таговете фактически представляват средствата предоставяни от продукта за описание и генериране на справката. По - долу са представени различните тагове и тяхното възможно предназначение.

- *transaction* – позволява групирането на тагове в транзакция. Предимството на транзакцията е, че или всички тагове се изпълняват наведнъж като едно цяло или нито един таг не се изпълнява.

- *statement* – позволява групирането на тагове в рамките на една и съща област. По този начин се осигурява ефективно управление на видимостта на таговете.

- *if* – позволява разклоняване на workflow - а в зависимост от някакво условие. Използва се когато се налага динамично да изберем стъпките необходими за изпълнението на workflow - а.

- *var* – позволява временно съхраняване на някаква информация, която е необходима за изпълнението на workflow-а.

- *setvar* – позволява установяването на стойност във var, която да бъде съхранена временно.

- *viewvar* – позволява временно съхранение на някаква информация в структуриран вид под формата на view, която е необходима за изпълнението на workflow-а.

- *binaryvar* – позволява временно съхранение на някаква бинарна информация, която е необходима за изпълнението на workflow - а.

- *setbinaryvar* – позволява установяването на стойност във binaryvar, която да бъде съхранена временно.

- *array* – позволява временно съхраняване на някаква информация в структуриран вид под формата на масив, която е необходима за изпълнението на workflow-а.

- *setarrayelement* – позволява установяването на стойност на даден елемент в масив.

- *addarrayelement* – позволява добавянето на елемент в масив.

- *removearrayelement* – позволява отстраняването на даден елемент от масив.

- *map* – позволява временно съхраняване на някаква информация в структуриран вид под формата на хеш-таблица, която е необходима за изпълнението на workflow-а.

- *addmapentry* – позволява добавянето на entry в хеш таблицата.

- *removemapentry* – позволява премахването на entry от хеш таблицата.

- *foreach* – позволява итерирането върху някакви структурирани данни като например view.

- *while* – позволява построяването на цикъл необходим за изпълнението на по-сложни задачи свързани с workflow - а.

- *procedure* – позволява декларирането на процедура, необходима за извършване на някакво сложно действие. Веднъж декларирана, процедурата може да бъде викана многократно в workflowa-а.

- *procedureCall* – позволява викането на процедура.

- *provider* – позволява декларирането на данните за осъществяване на връзка към база данни.

- *view* – позволява декларирането на view, необходимо за описанието на начина по който данните ще бъдат извлечени от източниците им и обработени в последствие. Веднъж декларирано, view - то може да бъде викано многократно с цел получаването на актуални данни винаги, когато това е необходимо.

- *viewCall* – позволява викането на view.

- *compositeview* – позволява деклариране на compositeview, необходимо за описанието на начина по който данните ще бъдат извлечени от източниците им и обработени в последствие. Характерно за compositeview е, че то обикновенно извършва операции върху view - та, които са характерни за множества. Веднъж декларирано, compositeview - то може да бъде викано многократно с цел получаването на актуални данни винаги, когато това е необходимо.

- *compositeviewCall* – позволява викането на compositeview.

- *insertview* – позволява декларирането на insertview. Характерното за insertview е, че то е отговорно за добавянето на всички получени данни към някакъв конкретен източник. Източникът трябва да бъде локално view.

- *insertviewCall* – позволява викането на insertview.

- *updateview* – позволява декларирането на updateview. Характерното за updateview е, че всички данни получени от него служат се модифицирането или актуализирането на някакъв източник. Източникът трябва да бъде локално view.

- *updateviewCall* – позволява викането на updateview.

- *deleteview* – позволява декларирането на deleteview. Характерното за deleteview е, че чрез него се отстранява някакви данни от някакъв източник. Източникът трябва да бъде локално view.

- *deleteviewCall* – позволява викането на deleteview.

- *dbinsertview* – позволява декларирането на dbinsertview. Характерно за dbinsertview е, че всички данни получени от него се добавят в някакъв източник. Източникът трябва да бъде таблица в базата данни.

- *dbinsertviewCall* – позволява викането на dbinsertview.

- *dbupdateview* – позволява декларирането на dbupdateview. Характерно за dbupdateview е, че всички данни получени от него служат за модифицирането или актуализирането на някакъв източник. Източникът трябва да бъде таблица в базата данни.

- *dbupdateviewCall* – позволява викането на dbupdateview.

- *dbdeleteview* – позволява декларирането на dbdeleteview. Характерно за dbdeleteview е, че чрез него се отстранява някакви данни от някакъв източник. Източникът трябва да бъде таблица в базата данни.

- *dbdeleteviewCall* – позволява викането на dbdeleteview.

- *xmlvar* – позволява временно съхраняване на някаква информация под формата на xml документ, която е необходима за изпълнението на workflow - а.

- *xmlvarload* – позволява зареждането на xml документа от някакъв външен източник.

- *xmlvarstore* – позволява съхранението на xml документа на някакъв външен източник.

- *xmlinsertview* – позволява декларирането на xmlinsertview. Характерно за xmlinsertview е, че всички данни получени от него се добавят към някакъв източник. Източникът трябва да бъде xml променлива.

- *xmlinsertviewCall* – позволява викането на xmlinsertview.

- *xmlupdateview* – позволява декларирането на xmlupdateview. Характерно за xmlupdateview е, че всички данни получени от него служат за модифицирането или актуализирането на някакъв източник. Източникът трябва да бъде xml променлива.

- *xmlupdateviewCall* – позволява викането на xmlupdateview.

- *xmldeleteview* – позволява декларирането на xmldeleteview. Характерно за xmldeleteview е, че чрез него се отстраняват някакви данни от някакъв източник. Източникът трябва да бъде xml променлива.

- *xmldeleteviewCall* – позволява викането на xmldeleteview.

3.2 Описание на възможностите предоставяни от продукта.

След прегледа на средствата предоставяни от продукта е съвсем естествено да следва анализ за неговите възможности. Като цяло се забелязва възможността на продукта да работи с различни източници на данни. От тези източници на данни могат да се изтриват, добавят, модифицират и извличат данни, според нуждите на потребителя. Важна характеристика, която не се забелява лесно на пръв поглед е възможността да се направи комплексна обработка на данните преди те бъдат използвани по тяхното предназначение. А именно - възможността да се организира самостоятелно обработката под формата на някакъв workflow. Друга характеристика, която не може да не остане незабелязана е възможността да се организират последователностите от действия в транзакции. Възможността да използваме транзакции разширява значително спектъра от действия, които могат да бъдат извършвани. Последното и най – съществено предимство е възможността да се работи едновременно с повече от един източник на данни. Тази функционалност е използвана особено сполучливо от тагове извършващи действия имащи връзка с някоя от традиционните операции свързани с бази данни – изтриване, модифициране, извличане или добавяне на данни. При тези тагове на потребителя се дава възможност да работи едновременно с тяколко таблици, всяка от които може да принадлежи на различна база данни. Характерното при този начин на работа е, че комплексността на заявката не се усложнява значително, а за сметка на това фукционалната област на практика става неограничена.

След като таговете на продукта бяха представени и анализирани подробно, идва време да се погледне по - глобално на самия workflow. Изборът на xml като език и средство за описанието на workflow - а съвсем не е случаен, както стана ясно в предходния раздел. Допълнително предимство освен модулността и разширяемостта са, че отделни компоненти от workflow - а или целия workflow могат да бъдат имплементирани и последователно внедрени в цялата система. В този случай xml - ът играе ролята на интерфейс, който единствено описва какво трябва трябва да бъде извършено, а имплементацията определя как точно то ще бъде извършено.

4. Дизайн и изисквания към системата.

Дизайнът на даден продукт представлява най - сериозната част от неговата разработка. Причините за това са няколко :

1. По време на дизайнът се налага да бъдат определени една голяма част от характеристиките, които ще притежава продуктът и до каква степен ще се осигури поддръжката на всяка една от тях.

2. Налага се често да се прави компромис с част от характеристиките на продукти, за сметка на други понеже се получава така, че някои характеристики си противоречат взаимно.

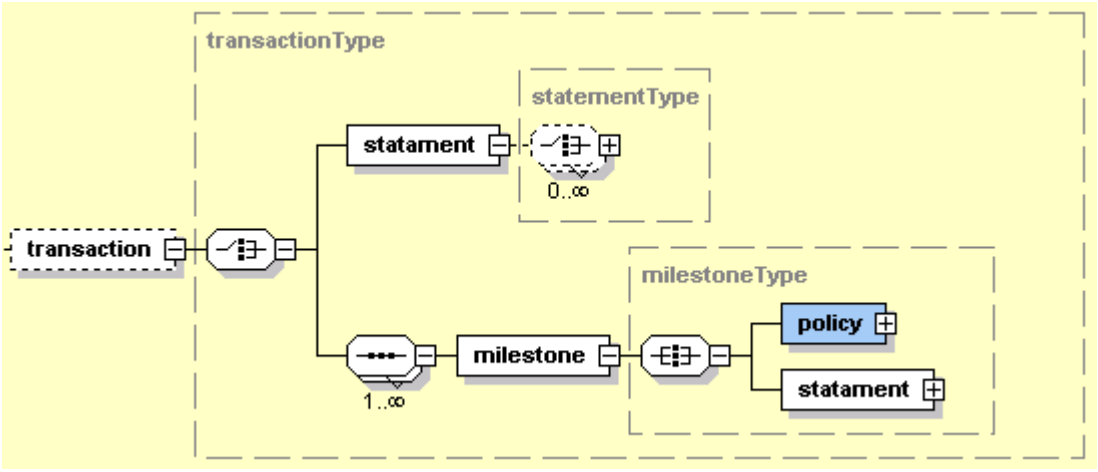
3. Налага се планиране на цикъла и метода на разработка, както и средствата необходими за това.

Може да се каже, че от програмистка гледна точка дизайнът е най - важната част от проектирането на една система, понеже именно той определя като цяло архитектурата по която ще бъде изграден продуктът. Всяка архитектура трябва да включва в себе си възможности за разширяемост, модулност, гъвкавост и сигурност. Балансът между тези характеристики определя цялостния облик на крайния продукт, както и неговата фукционалност.

4.1 Дизайн на компонентите използвани в системата.

Избора на xml като език за описание на workflow-а предоставя лесен начин за обособяването и разграничаването на компонентите в системата. На практика се получава така, че фактически някои от таговете са избрани да бъдат използвани като компоненти. Различните компоненти могат да имат различно предназначение като същото на практика важи и за таговете. При описанието на компонентите ще бъде разгледана детайлно тяхната структура и ще бъдат описани техните възможности и характеристики. Освен това ще бъде описано детайлно как точно е имплементиран всеки един от компонентите. Колкото по – сложен е един компонент толкова по - голямо внимание ще му бъде отделено. Друга важна особеност на компонентите е възможността те да съдържат други компоненти.

4.1.1 *transaction* - компонент



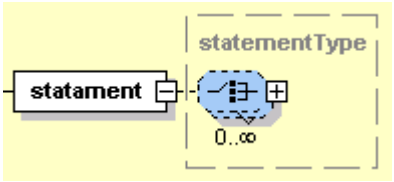
Използва се за групирането на тагове в транзакция. Предимството на транзакцията е, че или всички тагове се изпълняват наведнъж като едно цяло или нито един таг не се изпълнява.

Основна характеристика на транзакцията е възможността тя да се състои от някои подчасти наречени milestone-и. Всеки milestone може да бъде разглеждан като подчаст на транзакцията. Характерно за него е, че ако той не

може да бъде изпълнен поради някаква причина, това не води непременно до отхвърляне на цялата транзакция. Може да се случи така, че само определени части да бъдат отхвърлени, а транзакцията като цяло да завърши успешно. Фактически това какво точно да се случи в тези извънредни ситуации се определя от policy тага. В зависимост от него фактически може да се игнорира резултатът от грешка в дадена част на транзакцията, да се прекрати транзакцията или да се игнорира грешката в транзакцията и да се продължи нейното изпълнение от някакво друго място.

Transaction компонентът фактически представлява един от най – сложните компоненти в цялата система. Неговата комплексност се състои в това, че той трябва да може да възстанови данните преди изпълнението на транзакцията, в случай, че транзакцията не може да завърши успешно. За тази цел се налага да се използва най – общо казано допълнителен контекст, в който да се съхранява всяка една промяна, която се прави в рамките на транзакцията. В случай, че тразакцията не завърши успешно се прави възстановяне на оригиналния контекст, използвайки информацията от допълнителния такъв. Един от големите проблеми, които могат да възникнат в случая е, че всеки компонент в рамките на транзакцията трябва да актуализира, както допълнителния контекст със неговата стара стойност, така и актуалният контекст. Актуализирането на допълнителния контекст в случай, че компонентът е извън транзакция е безмислено. Поради това се налага всеки компонент да знае дали се намира в рамките на някаква транзакция или не.

4.1.2 *statement* - компонент

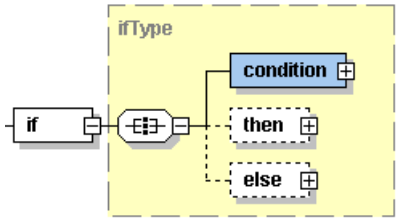


Позволява групирането на тагове в рамките на една и съща област. По този начин се осигурява ефективно управление на видимост на таговете.

Характерното за този компонент е, че той може да съдържа абсолютно всеки един друг компонент. В този смисъл той се явява нещо като контейнер. Всички компоненти, които се намират в контейнера могат да бъдат викани само от други компоненти които се намират в същият контейнер. Това е много ефикасно средство за управление на компонентите, както и ресурсите използвани от тях. След приключване на изпълнението на statement компонента всички ресурси, които са били заделени в него от другите компоненти се освобождават. Пример за такъв ресурс се явява оперативната памет.

Реализацията на statement компонента само по себе си представлявя проста задача. Причината за това се явява това, че той не изпълнява някаква конкретна функционалност. Главната функция на този компонент е да създаде и извика всеки един от компонентите, които се намират в рамките на неговия обхват.

4.1.3 *if* - компонент

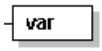


Позволява разклоняване на workflow - а в зависимост от някакво условие. Използва се когато се налага динамично да изберем стъпките необходими за изпълнението на workflow-а.

Особеното за този компонент е начинът по който се изчислява condition тага.
За да се приеме даден condition за истина, то неговата булева стойност трябва да е true. В противен случай, когато стойността връщана от condition тага е false или null то условието не се смята за удовлетворено. При положение, че условието е удовлетворено се продължава с изпълнението на then елемента. В противен случай – с else елемента.

Реализацията на if компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.4 *var* - компонент



Позволява временното съхраняване на някаква информация, която е необходима за изпълнението на workflow-а.

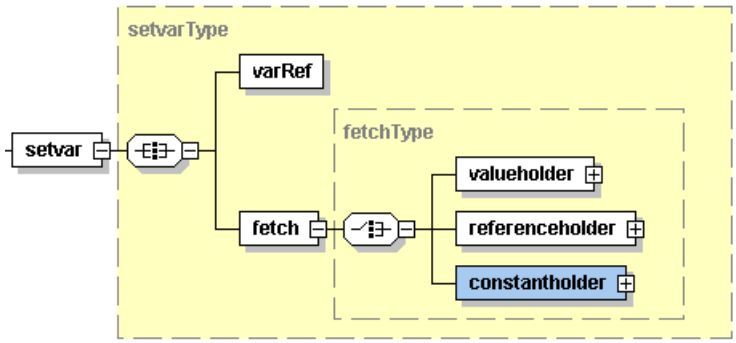
Областта на видимост на променливата е ограничена в рамките на **statement** елемента в който е декларирана. Тагът **var** може да съдържа следните атрибути:

Име на атрибута	Предназначение на атрибута
ID	Задава уникален идентификатор на променливата.
name	Задава името на променливата.
label	Задава описание на променливата.
type	Задава тип на променливата. Променливата може да има някой от следните типове - *CHAR*, *VARCHAR*, *LONGVARCHAR, NUMERIC*, *DECIMAL, BIT*, *TINYINT*, *SMALLINT*, *INTEGER*, *BIGINT*, *REAL*, *FLOAT*, *DOUBLE*, *DATE*, *TIME*, *TIMESTAMP*.

Характерна особеност на променливата е начинът на нейното съхранение. Освен в оригиналния контекст, нейната предходна стойност се съхранява и в допълнителния контекс в случай, че var тагът е част от някаква по – голяма транзакция. Това се прави с цел транзакцията да може да възстанови всички стойности, които е модифицирала.

Реализацията на var компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.5 *setvar* - компонент

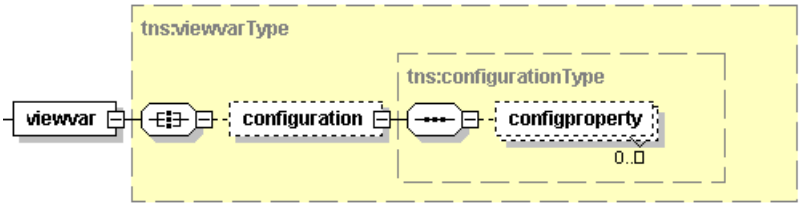


Позволява установяването на стойност във **var**, която да бъде съхранена временно.

Характерна особеност на този компонент е начинът, по който се указва променливата където да бъде съхранен даден обект, както и начинът, по който той ще бъде извлечен. С помощта на **varRef** се дава възможност да се реферира създадена вече променлива където да се запише обекта. За извличането на обекта се използва друг помощен компонент. Той е представен чрез **fetch** елемента и дава възможност за извличане на обекта посредством изчислението на някакъв израз (**valueholder**), посредством използването на някаква референция (**referenceholder**) или посредством използването на някаква константа (**constantholder**).

Реализацията на **setvar** компонента е значително опростена използвайки **fetch** компонента, който агрегира в себе си доста комплексни и сложни функции.

4.1.6 *viewvar* - компонент



Позволява временното съхраняване на някаква информация в структуриран вид под формата на view, която е необходима за изпълнението на workflow-а.

Областта на видимост на променливата е ограничена в рамките на **statement** елемента в който е декларирана. Тагът **viewvar** може да съдържа следните атрибути:

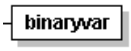
Име на атрибута	**Предназначение на атрибута**
ID	Задава уникален идентификатор на променливата.
name	Задава името на променливата.
label	Задава описание на променливата.

Характерна особеност на променливата е начинът на нейното съхранение. Освен в оригиналния контекст, нейната предходна стойност се съхранява и в допълнителния контекс в случай, че **viewvar** тагът е част от някаква по – голяма транзакция. Това се прави с цел транзакцията да може да възтанови всички стойности, които е модифицирала.

Реализацията на **viewvar** компонента се явява комплексна задача, която налага използването на допълнителна информация за нейното осъществяване. Тази информация се задава с помощта на **configuration** тага. Поради факта, че във **viewvar** може да се съдържат изключително голямо количество от данни, неминуемо възниква проблем с тяхното ефективно съхраняване и използване. Подходите използвани за справяне с проблема могат да бъдат различни. При единият подход данните се запиват върху диска и се прочитат от там, когато има нужда от тях. При другият подход данните остават в паметта. И двата подхода имат своите големи предимства и недостатъци, което налага търсенето на компромисен вариант, които да обедини всички положителни страни на горните два варианта. При този компромисен вариант се дава възможност да се кешират само определена част от данните, а останалите данни да бъдат записвани и прочитани от диска в случай на необходимост. Това каква част да бъде кеширана се задава от потребителя използвайки някои от конфигурационните параметри, декларирани с помощта на **configparameter**

тага. В случай, че потребителят не зададе стойност на някои от конфигурационните параметри, се използват подразбиращи се стойности.

4.1.7 *binaryvar* - компонент



Позволява временното съхраняване на някаква бинарна информация, която е необходима за изпълнението на workflow-а.

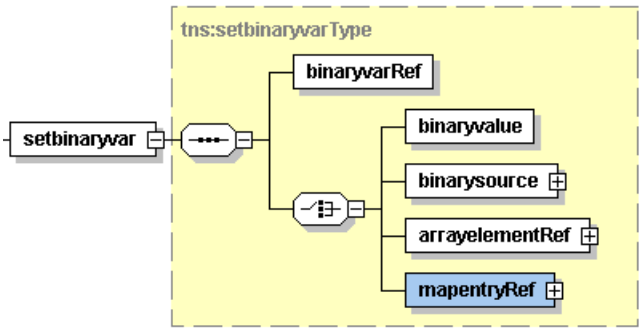
Областта на видимост на променливата е ограничена в рамките на **statement** елемента в който е декларирана. Тагът **binaryvar** може да съдържа следните атрибути:

Име на атрибута	Предназначение на атрибута
ID	Задава уникален идентификатор на променливата.
name	Задава името на променливата.
label	Задава описание на променливата.
binarytype	Задава тип на променливата. Променливата може да има някой от следните типове – BINARY, VARBINARY или LONGVARBINARY.

Характерна особеност на променливата е начинът на нейното съхранение. Освен в оригиналния контекст, нейната предходна стойност се съхранява и в допълнителния контекс в случай, че **binaryvar** тагът е част от някаква по – голяма транзакция. Това се прави с цел транзакцията да може да възтанови всички стойности, които е модифицирала.

Реализацията на **binaryvar** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.8 *setbinaryvar* - компонент

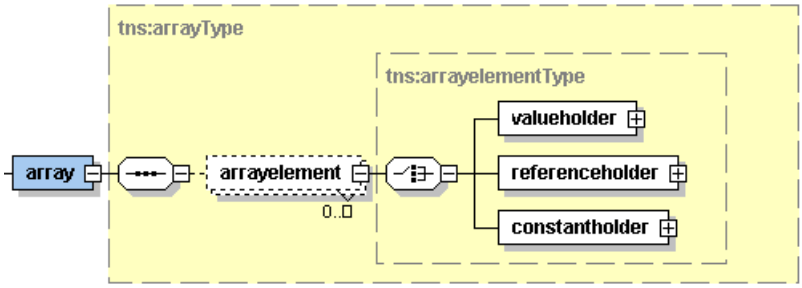


Позволява установяването на стойност във **binaryvar**, която да бъде съхранена временно.

Характерна особеност на този компонент е начинът по който се указва променливата, където да бъде съхранен даден обект, както и начинът по който той ще бъде извлечен. С помощта на **binaryvarRef** се дава възможност да се реферира създадена вече променлива където да се запише обекта. За извличането на обекта се използва някой от другите помощни компоненти.
Така например посредством **binaryvalue** може да извлечем декларирана стойност във **workflow** - а, а посредсвом **binarysource** може да бъде извлечена стойност дефинирана в някакъв външен файл. Използвайки **arrayelementRef** пък може да реферираме стойност на елемент от масив, а използвайки **mapentryRef** може да реферираме стойността на **entry** от **map**.

Реализацията на **setbinaryvar** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.9 *array* - компонент



Позволява временно съхраняване на някаква информация в структуриран вид под формата на масив, която е необходима за изпълнението на workflow-а.

Областта на видимост на променливата е ограничена в рамките на **statement** елемента в който е декларирана. Тагът **array** може да съдържа следните атрибути:

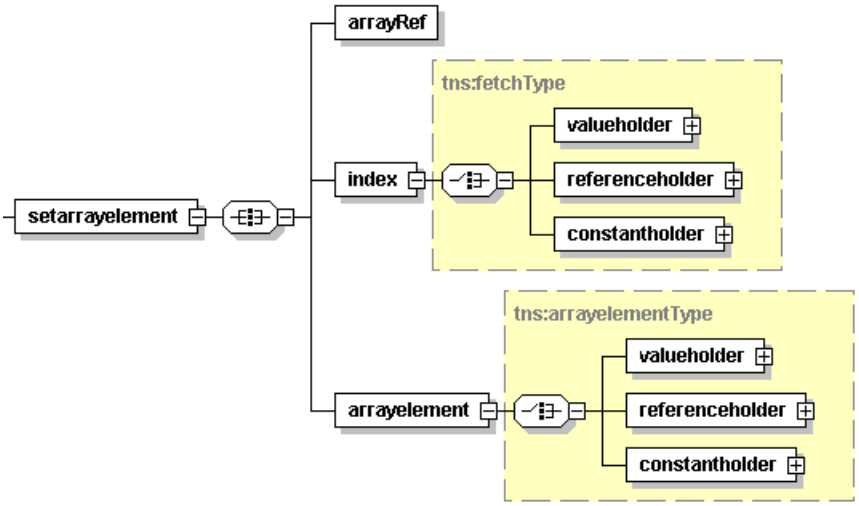
Име на атрибута	**Предназначение на атрибута**
ID	Задава уникален идентификатор на променливата.
name	Задава името на променливата.
label	Задава описание на променливата.

Характерна особеност на променливата е начинът на нейното съхранение. Освен в оригиналния контекст, нейната предходна стойност се съхранява и в допълнителния контекс в случай, че array тагът е част от някаква по – голяма транзакция. Това се прави с цел транзакцията да може да възтанови всички стойности, които е модифицирала.

Заедно с декларирането на array тага, може да укажем и arrayelement-ите, които той ще съдържа. Характерното за всеки arrayelement е, че той може да извлича стойността си с помощта на valueholder, referenceholder или constantholder.

Реализацията на array компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.10 *setarrayelement* - компонент

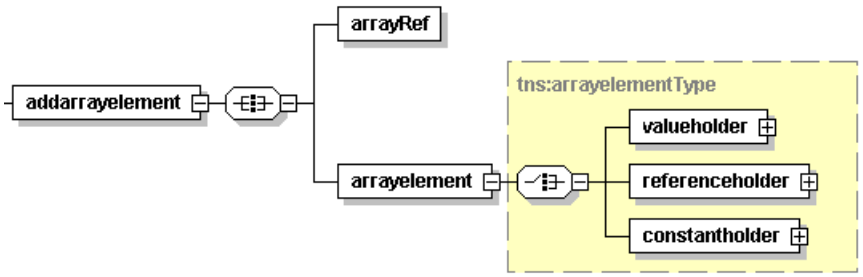


Позволява установяването на стойност в даден елемент от масив.

Характерна особеност на този компонент е начинът, по който се указва масива и индекса където да бъде съхранен даден обект, както и начинът по който те ще бъде извлечени. С помощта на **arrayRef** се дава възможност да се реферира създаден вече масив, а с помощта на **index** се дава възможност да се определи индекс от масива. Съдържанието на елемента **arrayelement** определя стойността, която ще бъде съхранена.

Реализацията на **setarrayelement** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.11 *addarrayelement* - компонент

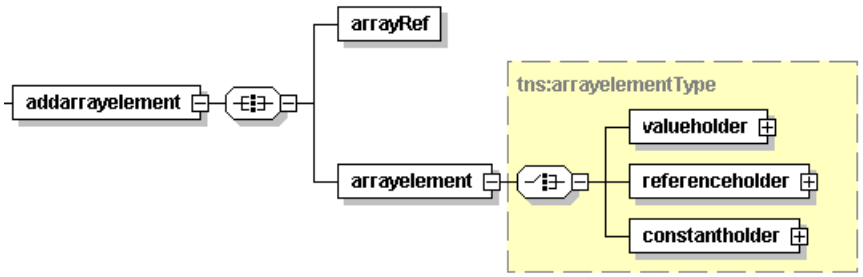


Позволява добавянето на елемент в края на масива.

Характерна особеност на този компонент е начинът по който се указва масива. С помощта на **arrayRef** се дава възможност да се реферира създаден вече масив. Съдържаниете на елемента **arrayelement** определя стойността, която ще бъде съхранена.

Реализацията на **addarrayelement** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.12 *removearrayelement* - компонент

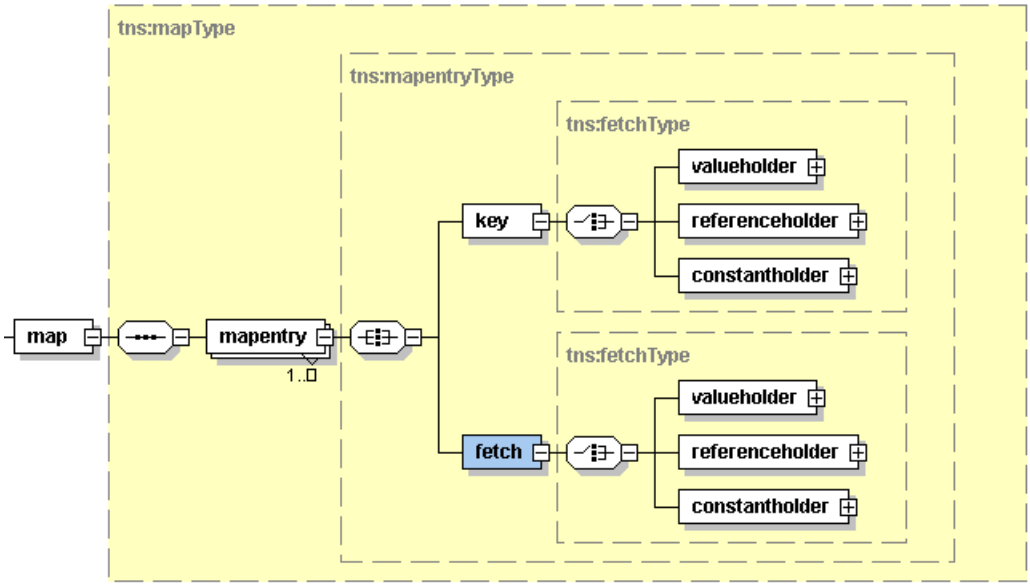


Позволява отстраняването на даден елемент от масива.

Характерна особеност на този компонент е начина, по който се указва масива и индекса на елемента, който ще бъде отстранен. С помощта на **arrayRef** се дава възможност да се реферира създаден вече масив, а с помощта на **index** се дава възможност да се определи индекс от масива.

Реализацията на **removearrayelement** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.13 *map* - компонент



Позволява временно съхраняване на някаква информация в структуриран вид под формата на хеш-таблица, която е необходима за изпълнението на workflow-а.

Областта на видимост на променливата е ограничена в рамките на statement елемента, в който е декларирана. Тагът map може да съдържа следните атрибути:

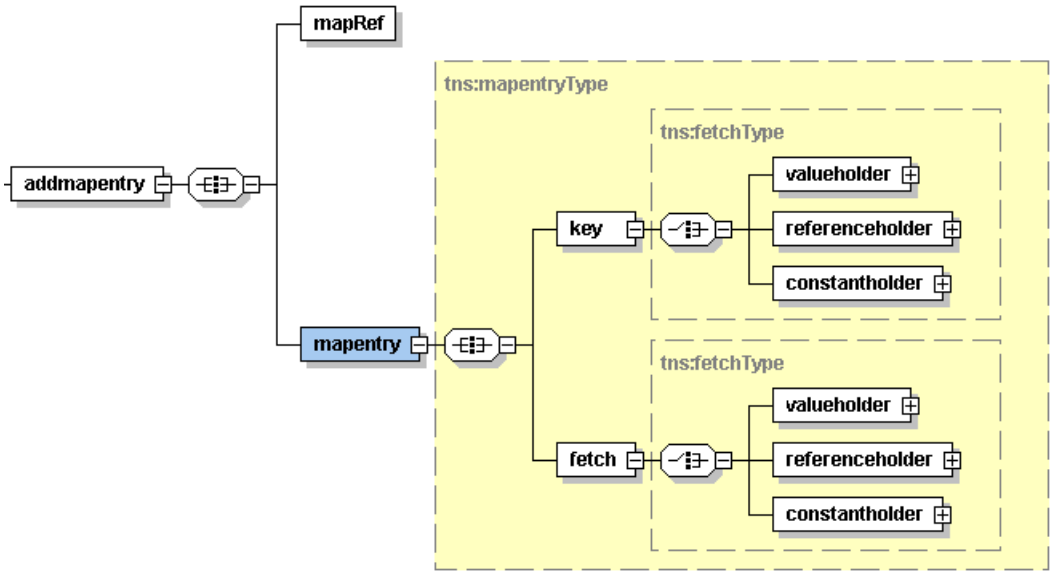
Име на атрибута	**Предназначение на атрибута**
ID	Задава уникален идентификатор на променливата.
name	Задава името на променливата.
label	Задава описание на променливата.

Характерна особеност на променливата е начинът на нейното съхранение. Освен в оригиналния контекст, нейната предходна стойност се съхранява и в допълнителния контекс в случай, че map тагът е част от някаква по – голяма транзакция. Това се прави с цел транзакцията да може да възтанови всички стойности, които е модифицирала.

Заедно с декларирането на map тагът, може да укажем и mapentry-итата, които той ще съдържа. Характерното за всяко mapentry е, че то съдържа key под който се съхранява дадена стойност извличена с помощта на fetch елемента.

Реализацията на map компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.14 *addmapentry* - компонент

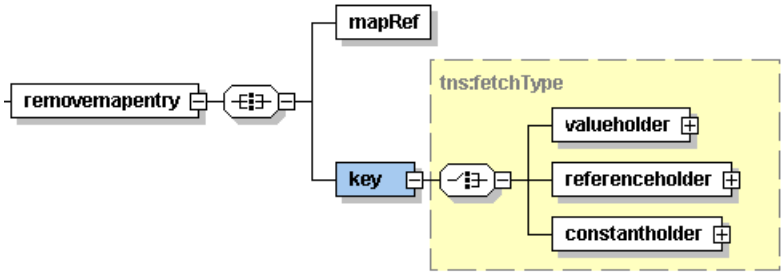


Позволява добавянето на **mapentry** в хеш таблицата.

Характерна особеност на този компонент е начина по който се указва хеш – таблицата, като за целта се използва **mapRef**, който дава възможност да се реферира създадена вече хеш – таблица. Съдържанието на **mapentry** определя данните, които ще бъдат добавени.

Реализацията на **addmapentry** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.15 *removemapentry* - компонент

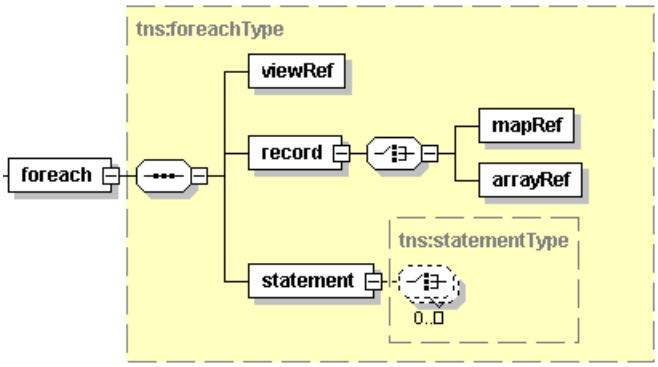


Позволява премахването на **mapentry** от хеш - таблицата.

Характерна особеност на този компонент е начина по който се указва хеш – таблицата, като за целта се използва **mapRef**, който дава възможност да се реферира създадена вече хеш – таблица. Стойността на тагът **key** определя ключа, който ще бъде премахнат от хеш – таблицата.

Реализацията на **removemapentry** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.16 *foreach* - компонент

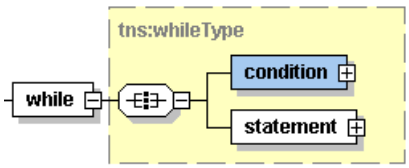


Позволява итерирането върху данни структурирани под формата на **view**.

Характерна особеност на този компонент е начина по който се указва **view**-то, като за целта се използва **viewRef**, който дава възможност да се реферира създадено вече **view**. Друга характерна особеност представлява тага **record**, който позволява да се записват под формата на **map** или **array** данните прочетени от текущият запис на **view**-то. За указването на **map** или **record** се използват **mapRef** или **arrayRef**. Тагът **statement** се използва за групиране на компонентите, които ще бъдат изпълнявани циклично. Итерирането върху структурираните данни продължава докато бъдат обходени всички данни.

Реализацията на **foreach** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.17 *while* - компонент

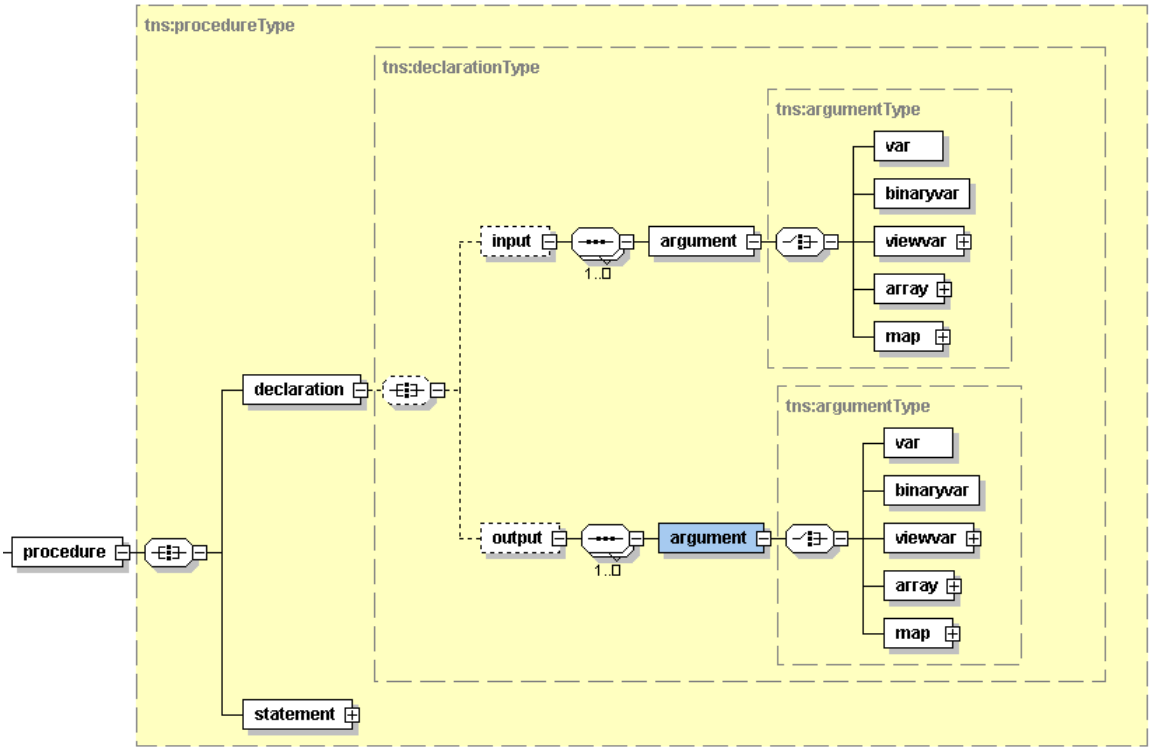


Позволява построяването на цикъл необходим за изпълнението на по-сложни задачи свързани с workflow-а.

Характерна особеност на този компонент е, че той се състои от два други тага : **condition** и **statement**. Тагът **condition** служи за изчисление на края на цикъла, докато тагът **statement** служи за групирането на таговете, които участват в цикъла.

Реализацията на **while** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

4.1.18 *procedure* - компонент



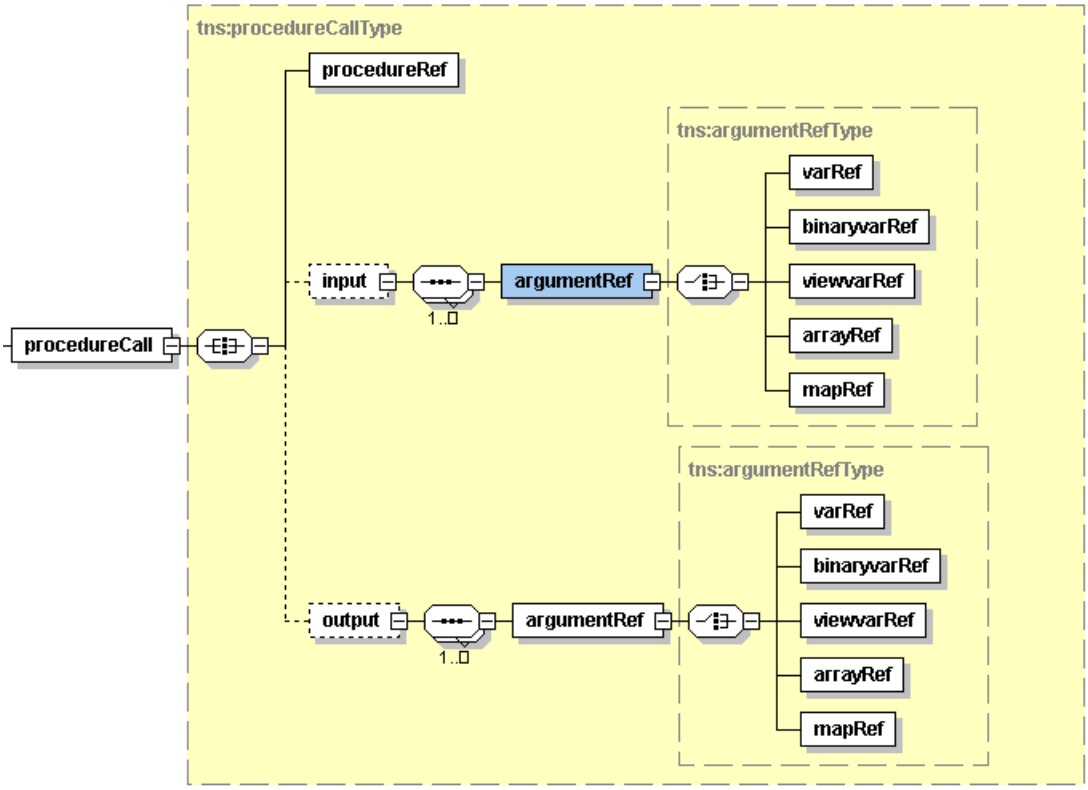
Позволява декларирането на процедура необходима за извършването на някакво сложно действие. Веднъж декларирана, процедурата може да бъде викана многократно в workflowa-а.

Характерно за този компонент е, че той се състои от две части. Едната част е декларативна и се представя с помощта на **declaration** елемента. От своя страна **declaration** елемента се състои от два други тага : **input** и **output**. С помощта на **input** елемента се декларират входящите аргументи, които приема процедурата при нейното извикване. С помощта на **output** елемента се декларират изходящите аргументи, които се връщат от процедурата след нейното извикване.

Реализацията на **procedure** компонента се явява комплексна задача и протича на няколко фази. При първата фаза се декларират входящите и изходящите параметри, след което на втора фаза се подават входящите стойности на всеки един **input** аргумент. След това на се изпълнява същността на процедурата представена чрез тага **statement** и най – накрая се връща резултат под формата на **output** аргументи. За една процедура не е задължително да има входящи или изходящи аргументи. Това се определя от потребителя в зависимост от неговата представа за това какво трябва да върши процедурата.

Друг проблем, които би могъл да възникне е начинът на предаване на параметрите. Прието е параметрите да се предават по стойност, тъй като в противен случай границата между входящи и изходящи параметри се размива.

4.1.19 *procedureCall* - компонент

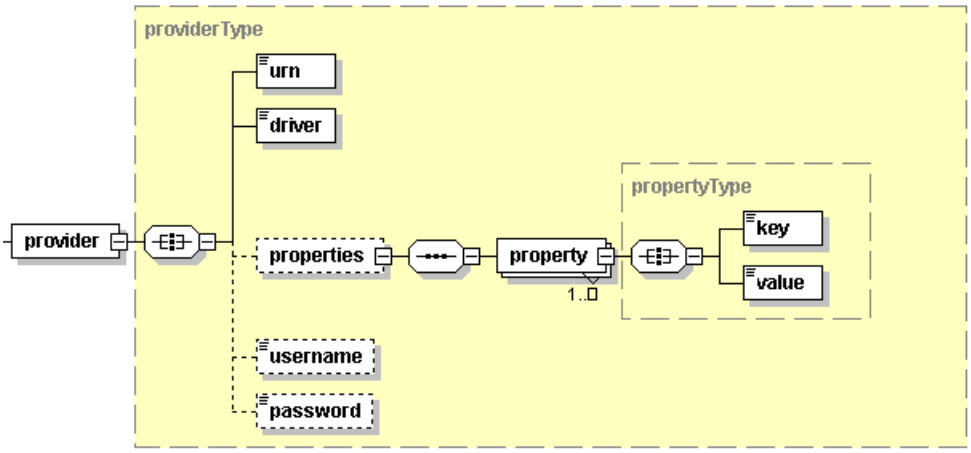


Позволява викането на процедура.

Характерно за този компонент е, че той се състои от три части. Първата част е **procedureRef** тага и позволява реферирането на процедура, която вече е била декларирана. Втората част е **input** тага и служи за описание на входящите параметрите с които ще се вика процедурата. Начинът по който се задава стойността на всеки един входящ параметър е посредством тага **argumentRef**, който съдържа ID атрибут с идентификатора от дефиницията на входящия аргумент, както и вътрешен елемент представляващ рефенция към някоя от възможните типове входящи променливи – **varRef**, **binaryRef**, **viewvarRef**, **arrayRef** или **mapRef**. Третата част на **procedure** елемента представлява **output** тага, който задава изходящите аргументи от процедурата. Начинът по който се задава стойността на всеки един изходящ параметър е посредством тага **argumentRef**, който съдържа ID атрибут с идентификатора от дефиницията на изходящия аргумент, както и вътрешен елемент представляващ рефенция към някоя от възможните типове изходящи променливи – **varRef**, **binaryRef**, **viewvarRef**, **arrayRef** или **mapRef**.

Реализацията на **procedureCall** компонента се явява комплексна задача, която засяга цялостното поведение на **workflow**-а. Възможността да се извикват процедури разширява значително възможностите на **workflow**-а и позволява да се групират, а също така систематизират често повтаряни действия под формата на процедури. Друго нещо на което трябва да се обърне внимание е липсата на контекст при извикването на процедурата. Единствено резултатът от процедурата представен с помощта на изходящи променливи се записва в съществуващия контекст. Това е от съществено значение за поддържане и осъществяване на нормалните политики свързани с управлението на транзакции.

4.1.20 *provider* - компонент

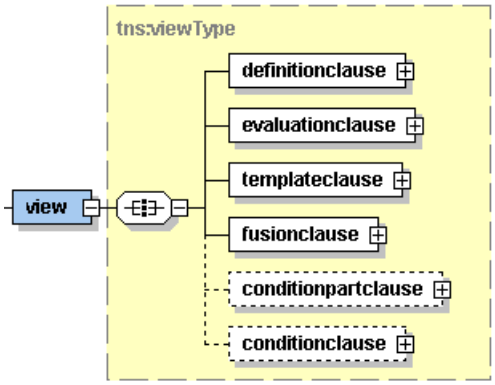


Позволява декларирането на данните за осъществяване на връзка към база данни.

Характерна особеност на този компонент е, че той се състои от няколко други тага : **urn, driver, properties, username и password.** Тагът **urn** е задължителен и служи за дефиниране на връзка към базата данни. Друг задължителен таг е **driver**, който служи за определяне на драйвер, който да се използва за изграждане на връзката към базата данни. Тагът **properties** е друг незадължителен таг, който може да служи за дефинирането на някои допълнителни **property** под формата на **key** и **value** необходими за осъществяване на връзката. Таговете **username** и **password** не са задължителни и служат съответно за дефиниране на потребителско име и парола необходими за идентифициране пред базата данни.

Реализацията на **provider** компонента се явява изключително проста и няма нужда от допълнителни обяснения.

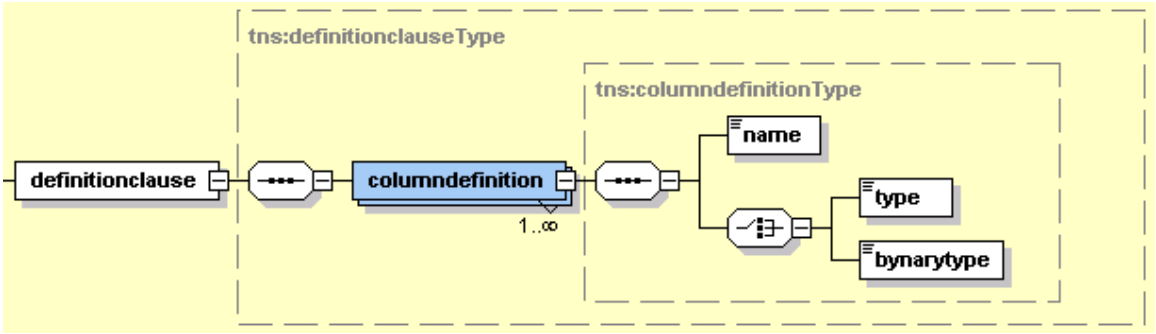
4.1.21 *view* - компонент



Позволява декларирането на **view**, необходимо за описание на начина, по който данните ще бъдат извлечени от източниците им и обработени в последствие. Веднъж декларирано, **view** - то може да бъде викано многократно с цел получаването на актуални данни винаги, когато това е необходимо.

Всяко едно **view** се състои от клаузи. Клаузите могат да бъдат : **definitionclause**, **evaluationclause**, **templateclause**, **fusionclause**, **conditionpartclause** и **conditionclause**.

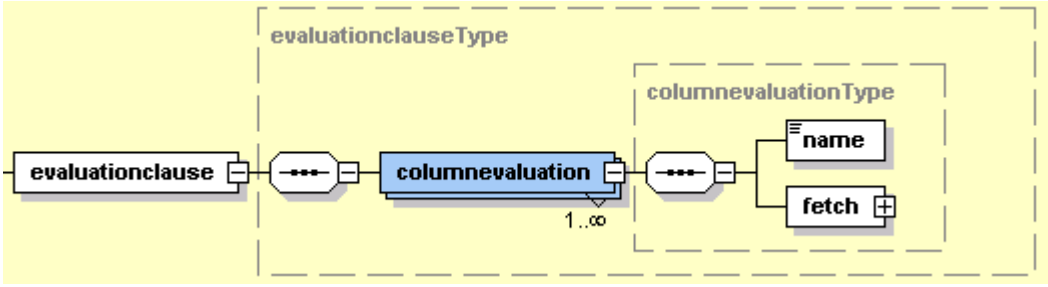
- **definitionclause**



Целта на тази клауза е да се опишат колоните от които се състои **view**-то, както и типът на всяка колона.

Всяка колона може да има за тип **type** или **binarytype**. Тагът **type** може да има някоя от следните стойности – CHAR, VARCHAR, LONGVARCHAR, NUMERIC, DECIMAL, BIT, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DATE, TIME или TIMESTAMP. Тагът **binarytype** може да има някоя от следните стойности - BINARY, VARBINARY или LONGVARBINARY.

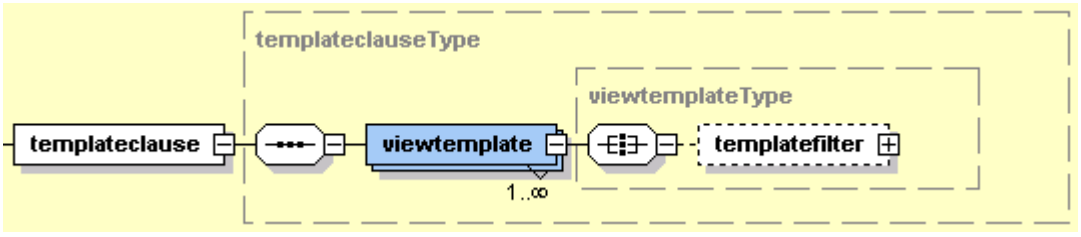
- **evaluationclause**



Целта на тази клауза е да се опишат изчисленията, които е необходимо да бъдат извършени, за да се получи стойността на всяка една колона.

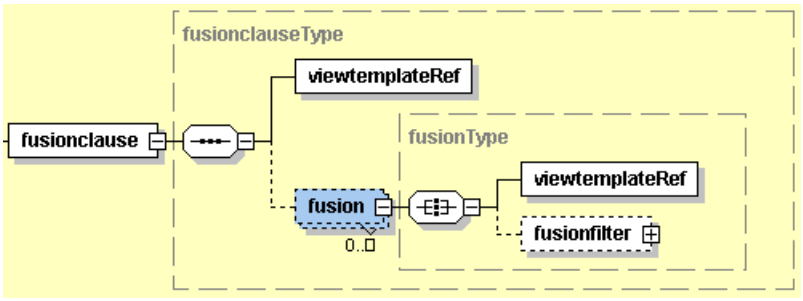
Всяко едно изчисление, което трябва да бъде извършено се задава посредством тагът **columndefinition**. Последният трябва да съдържа следните тагове – **name** и **fetch**. Тагът **name** определя колоната, за която се отнасят изчисленията, а тагът **fetch** определя как точно ще бъде извлечена и изчислена стойността.

- **templateclause**



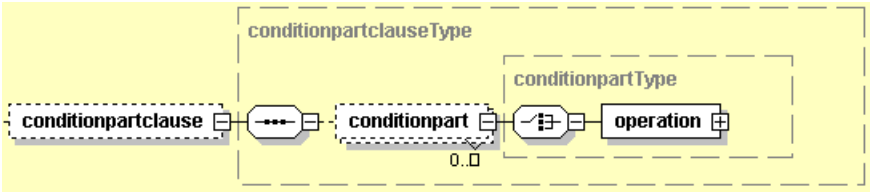
Целта на тази клауза е да се опишат **viewtemplate** - ите, които ще бъдат използвани. Целта на всеки един **viewtemplate** е да бъде използван като контейнер за данни и следователно от всеки един **viewtemplate** могат да бъдат извличани данни, които да участват в необходимите изчисления. Реалните данни, които ще участват в изчисленията се асоциират по време на извикване на **view**-то с компонента **viewCall**. Преди да участват за създаването на **view**-то във **fusionclause**, **viewtemplate** може да бъде филтриран посредством използването на тагът **temlpatefilter**.

- **fusionclause**



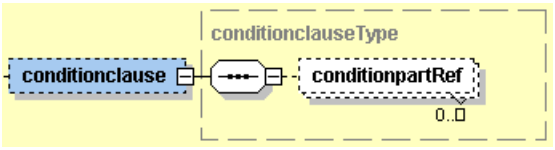
Целта на тази клауза е да се слеят по определен начин данните от различни източници. Сливането на източниците може да се осъществи по няколко възможни начина. Избора на конкретен начин, който да бъде използван зависи от *join* атрибута на **fusion** елемента. Той може да приема една от следните възможни стойности : INNER JOIN, LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN или CROSS JOIN. За някои от тези сливания може да се използва елемента **fusionfilter**, който се грижи да филтрира предварително и отдели само онези данни, които е необходимо да бъдат сляти.

- **conditionpartclause**



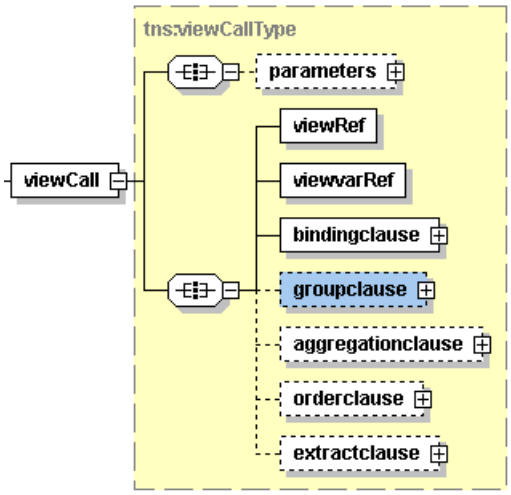
Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart** - ове на които трябва да отговаря даден запис, за да бъде добавен. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.22 *viewCall* - компонент



Позволява извикването на view.

Всяко едно **viewCall** се състои от различни тагове. Някои от таговете като **viewRef**, **viewvarRef** и **bindingclause** са задължителни, докато други от тях като parameters, groupclause, aggregationclause, orderclause и extractclause са допълнителни.

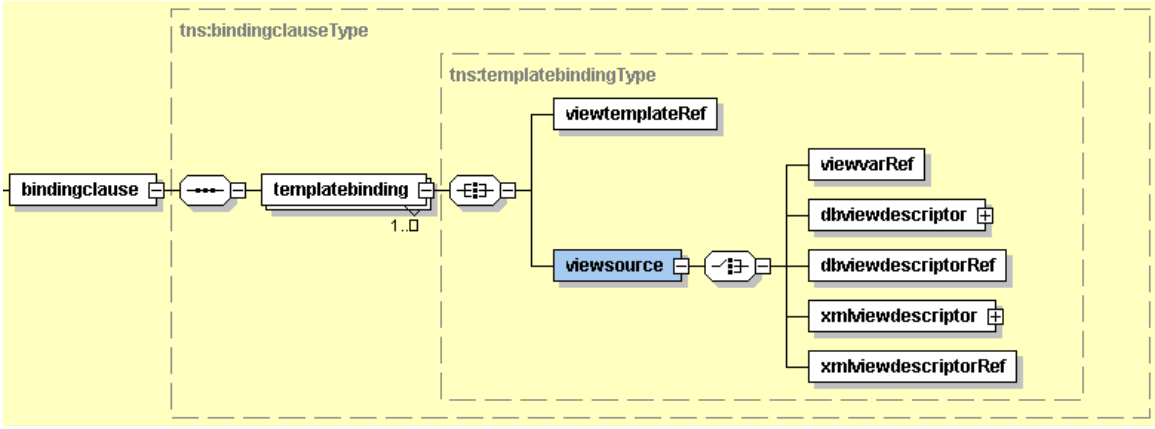
- **viewRef**

  Целта на този таг е да реферира декларирано вече **view**.

- **viewvarRef**

  Целта на този таг е да реферира декларирано вече **viewvar**, където ще бъде записан резултата.

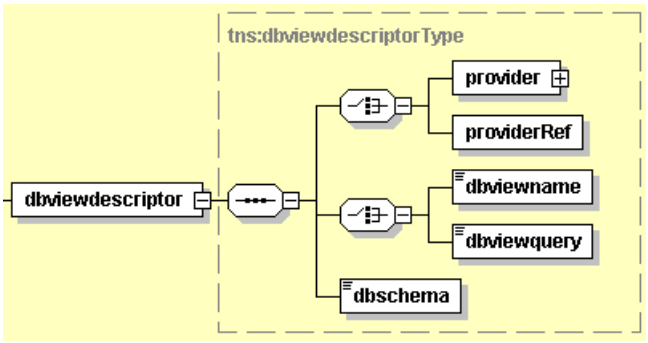
- **bindingclause**



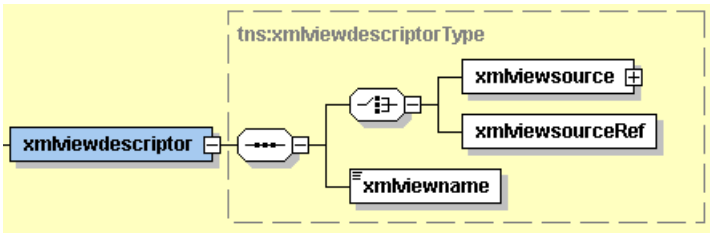
Целта на този клауза е да свържат **viewtemplate** - ите, които ще бъдат използвани при обработката на данните за **view** - то. Целта на всеки един **viewtemplate** е да бъде използван като контейнер за данни и следователно от всеки един **viewtemplate** могат да бъдат извличани данни, които да участват в необходимите изчисления. Реалните данни, които ще участват в изчисленията се асоциират използвайки някоя от възможните опции – **viewvarRef**, **dbviewdescriptor**, **dbviewdescrioptorRef**, **xmlviewdescriptor** или **xmlviewdescriptorRef**.

Име на тага	**Предназначение**
viewvarRef	Реферира създадена вече **viewvar**, която да се използва като източник на данни.
dbviewdescriptor	Указва таблица или изглед в база данни, който да бъде използван за източник на данните.
dbviewdescrioptorRef	Реферира деклариран вече **dbviewdescriptor**, който да се използва като източник на данни.
xmlviewdescriptor	Указва xml документ, част от данните на който да бъдат представени под формата на таблица, която да бъде използвана за източник на данните.
xmlviewdescriptorRef	Реферира деклариран вече **xmlviewdescriptor**, които да се използва като източник на данни.

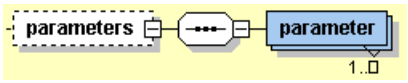
Структурата на **dbviewdescriptor** има следния вид:



Структурата на **xmlviewdescriptor** има следния вид:

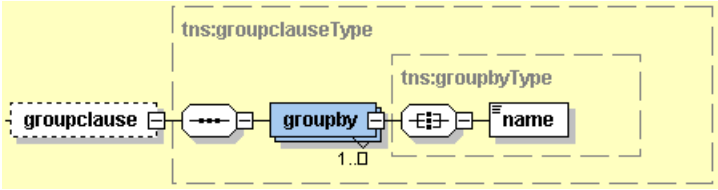


- **parameters**



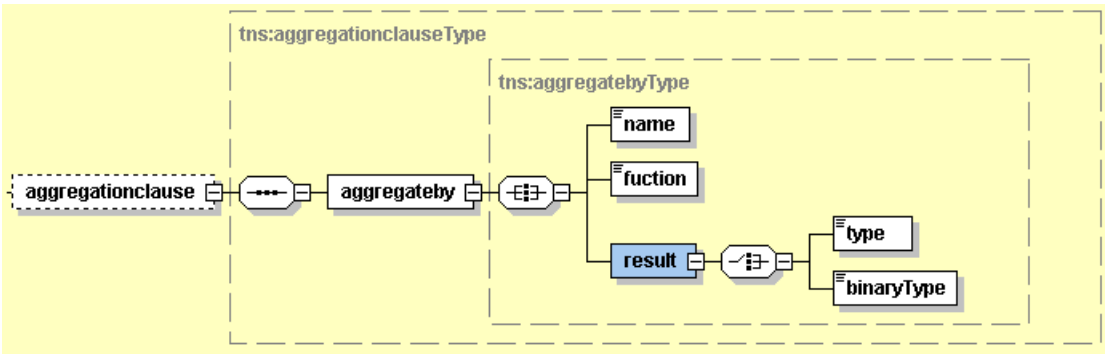
Целта на този таг е да събере всички **parameter**, които се подават при извикването на **viewCall**. декларирано вече **viewvar**, където ще бъде записан резултата.

- **groupclause**



Целта на тази клауза е да укаже как ще бъдат групирани данните от **view** елемента. Това става с помощта на тага **groupby**, който съдържа таг name указващ името на колона, която може да участва в групирането.

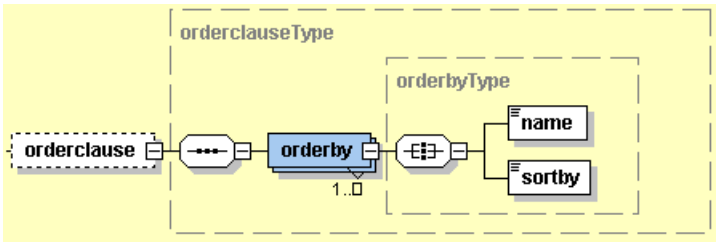
- **aggregationclause**



Целта на тази клауза е да укаже как ще бъдат агрегирани данните от view елемента. Това става с помощта на тага aggregateby, който съдържа таговете :

- **name** - указва колоната, която ще се използва за извършване групирането.
- **function** - указва функцията, която ще се използва за групиране на данните. Тя може да бъде една от следните - MIN, MAX, SUM, MULTIPLY, AVERAGE или COUNT.
- **result** - определя типа на данните, които ще се получат във view-то.

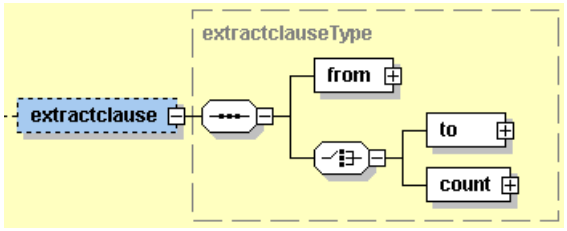
- **orderclause**



Целта на тази клауза е да укаже как ще бъдат сортирани данните от **view** елемента. Това става с помощта на тага orderby, който съдържа таговете:

- **name** - указва колоната по която ще се извършва сортирането.
- **sortby** - указва начинът по който ще бъдат сортирани данните. Може да бъде ASC или DESC.

- **extractclause**

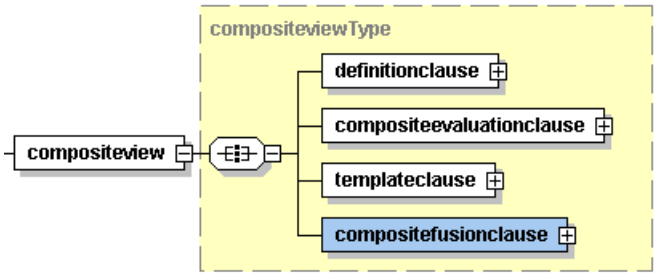


Целта на тази клауза е да укаже как ще бъдат извлечени данните от **view** елемента. Това става с помощта на таговете :

- **from** - определя началния номер на ред, от който ще започва извличането на данните.
- **to** – определя крайния номер на ред, до който ще бъдат извличани данните.
- **count** – определя максималния брой на редовете, които могат да бъдат извлечени.

Реализацията на **viewCall** компонента се явява изключително комплексна задача, която включва в себе си много други подзадачи. Детайлно обяснение за това как точно е имплементиран компонента ще бъде дадено при обясняване на реализацията.

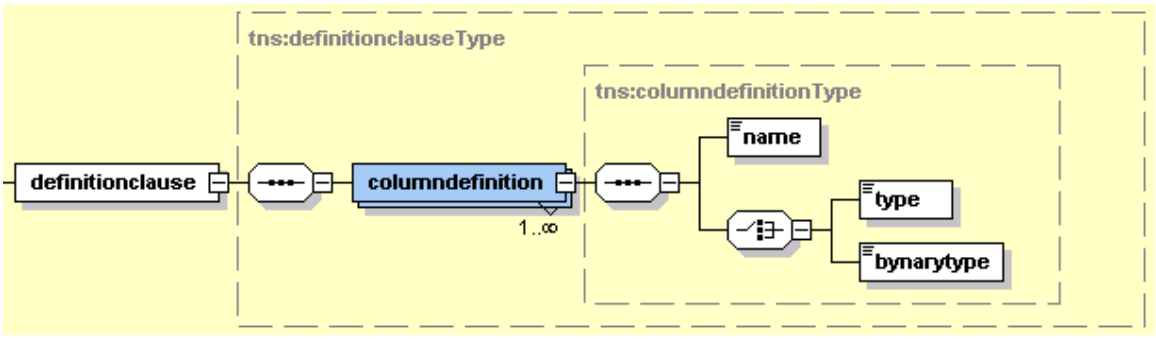
4.1.23 *compositeview* – компонент



Позволява деклариране на **compositeview**, необходимо за описанието на начина, по който данните ще бъдат извлечени от източниците им и обработени в последствие. Характерното за compositeview е, че то обикновенно извършва върху view-та операции, които са характерни за множества. Веднъж декларирано, compositeview-то може да бъде викано многократно с цел получаването на актуални данни винаги, когато това е необходимо.

Всяко едно **view** се състои от клаузи. Клаузите могат да бъдат : **definitionclause**, **compositeevaluationclause**, **templateclause** и **compositefusionclause**.

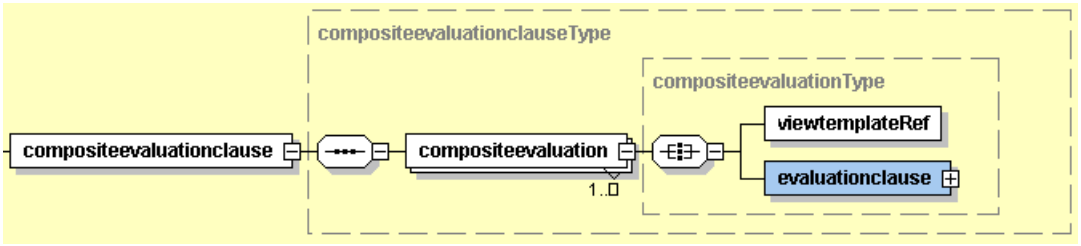
- **definitionclause**



Целта на тази клауза е да се опишат колоните от които се състои **view**-то, както и типът на всяка колона.

Всяка колона може да има за тип **type** или **binarytype**. Тагът **type** може да има някоя от следните стойности – CHAR, VARCHAR, LONGVARCHAR, NUMERIC, DECIMAL, BIT, TINYINT, SMALLINT, INTEGER, BIGINT, REAL, FLOAT, DOUBLE, DATE, TIME или TIMESTAMP. Тагът **binarytype** може да има някоя от следните стойности - BINARY, VARBINARY или LONGVARBINARY.

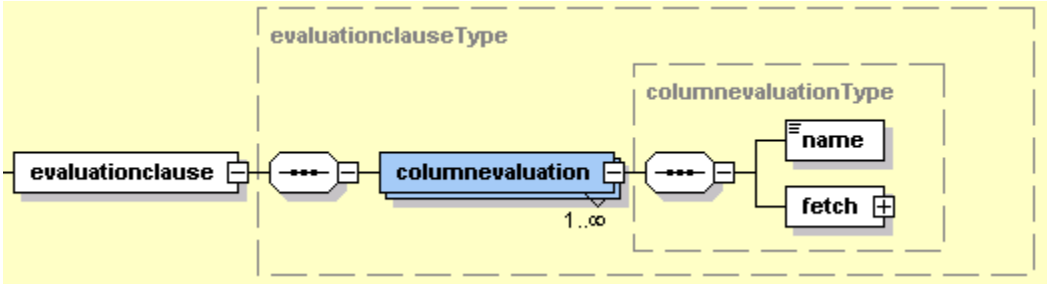
- **compositeevaluationclause**



Целта на тази клауза е да се опишат всички изчисленията, които е необходимо да бъдат извършени спрямо конкретен **viewtemplate,** за да се определят стойностите му във всяка една колона от **compositeview**-то. Както се подразбира за всеки **viewtemplate** тези изчисления могат да бъдат различни и поради тази причина **compositeevaluation** тага съдържа за всеки **viewtemplateRef** отделна **evaluationclause.**

  - **viewtemplateRef** – реферира деклариран **viewtemplate**, който да се използва при извършване на изчисленията.

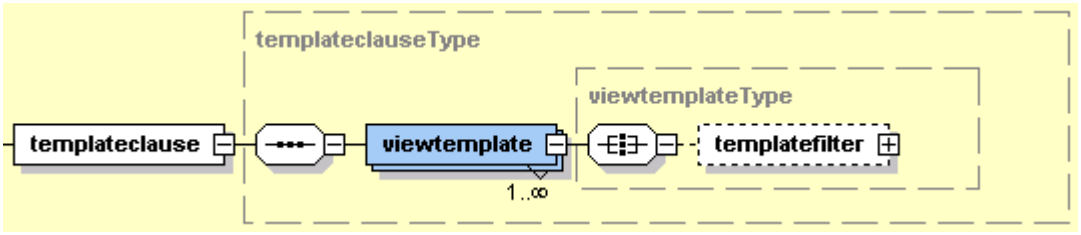
- **evaluationclause**



Целта на тази клауза е да се опишат изчисленията, които е необходимо да бъдат извършени за да се получи стойността на всяка една колона.

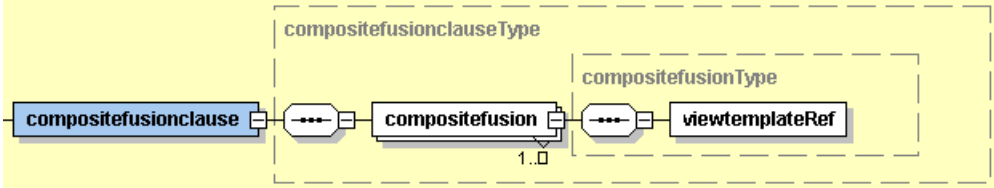
Всяко едно изчисление, което трябва да бъде извършено се задава посредством тагът **columndefinition**. Последният трябва да съдържа следните тагове – **name** и **fetch**. Тагът **name** определя колоната за която се отнасят изчисленията, а тагът **fetch** определя как точно ще бъде извлечена и изчислена стойността.

- **templateclause**



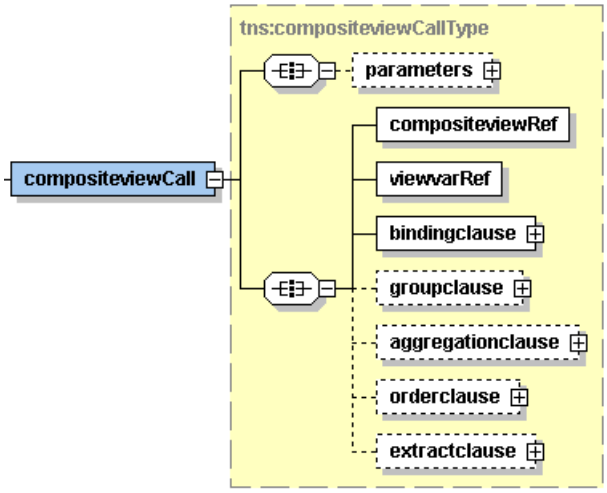
Целта на тази клауза е да се опишат **viewtemplate** – ите, които ще бъдат използвани. Целта на всеки един **viewtemplate** е да бъде използван като контейнер за данни и следователно от всеки един **viewtemplate** могат да бъдат извличани данни, които да участват в необходимите изчисления. Реалните данни, които ще участват в изчисленията се асоциират по време на извикване на **compositeview**-то с компонента **compositeviewCall**. Преди да участват за създаването на **compositeview**-то в **compositefusionclause**, **viewtemplate** може да бъде филтриран посредством използването на тагът **temlpatefilter**.

- **compositefusionclause**



Целта на тази клауза е да се слеят по определен начин данните от различни източници. Сливането на източниците може да се осъществи по няколко възможни начина. Избора на конкретен начин, който да бъде използван зависи от *compose* атрибута на **compositefusion** елемента. Той може да приема една от следните възможни стойности : UNION, INTERSECT, INTERSECT ALL, MINUS или MINUS ALL.

4.1.24 *compositeviewCall* - компонент



Позволява извикването на **compositeview**.

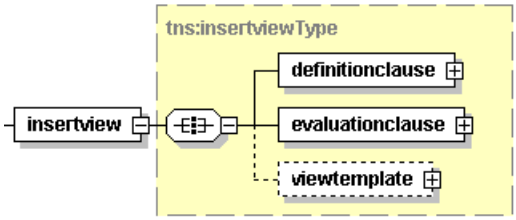
Всяко едно **compositeviewCall** се състои от различни тагове. Някои от таговете като **compositeviewRef**, **viewvarRef** и **bindingclause** са задължителни, докато други от тях като **parameters**, **groupclause**, **aggregationclause**, **orderclause** и **extractclause** са допълнителни.

Предназначението на всички **compositeviewCall** тагове е идентично с това на **viewCall** таговете. Единственото нещо което е различно това е **compositeviewRef** тага, който е заменил тага **viewRef**. Смисълът на **compositeviewRef** тага е да реферира декларирано вече **compositeview**.

Реализацията на **compositeviewCall** компонента се явява изключително комплексна задача, която включва в себе си много други подзадачи. Детайлно

обяснение за това как точно е имплементиран компонента ще бъде дадено при обясняване на реализацията.

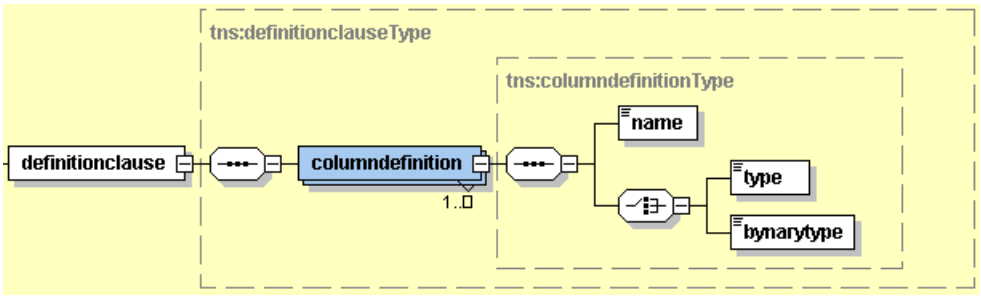
4.1.25 *insertview* - компонент



Позволява деклариране на **insertview**. Характерното за **insertview** е, че то е отговорно за добавянето на всички получени данни към някакъв конкретен източник. Източникът трябва да бъде локално view.

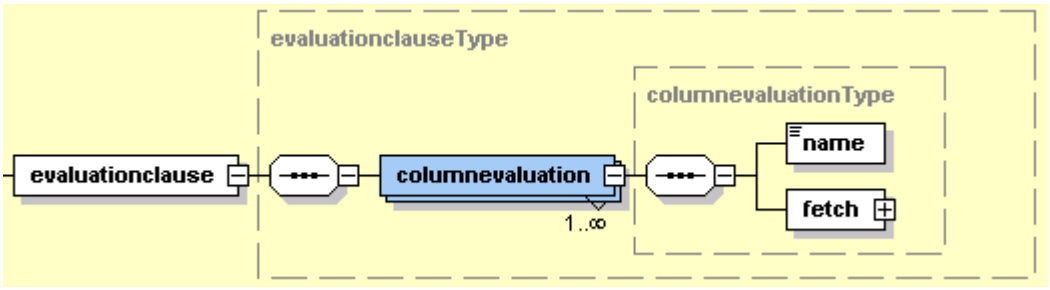
Всяко **insertview** се състои от тагове. Някои от таговете като **definitionclause** и **evaluationclause** са задължителни, докато други от тях като **viewtemplate** са допълнителни.

- **definitionclause**



Целта на тази клауза е да се дефинират колоните, в които ще се добавят данни посредством използването на **columndefinition**. Всяка **columndefinition** си има **name** и **type** или **binarytype**. Тагът **name** определя името на колоната, докато тагът **type** или **binarytype** определя типа на колоната.

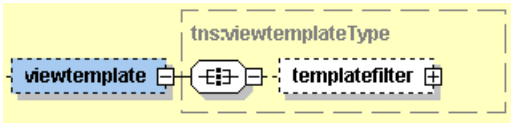
- **evaluationclause**



Целта на тази клауза е да се опишат изчисленията, които е необходимо да бъдат извършени за да се получи стойността на всяка една колона, която ще бъде добавяна.

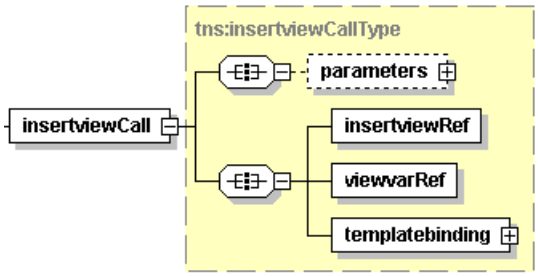
Всяко едно изчисление, което трябва да бъде извършено се задава посредством тага **columndefinition**. Последният трябва да съдържа следните тагове – **name** и **fetch**. Тагът **name** определя колоната за която се отнасят изчисленията, а тагът **fetch** определя как точно ще бъде извлечена и изчислена стойността.

- **viewtemplate**



Може да се представи като контейнер с данни, част от чието съдържание ще бъде добавено към източника. Контейнерът с данни може от друга страна да участва в извличането и пресмятането на нови данни, които да бъдат добавени към източника.

4.1.26 *insertviewCall* - компонент

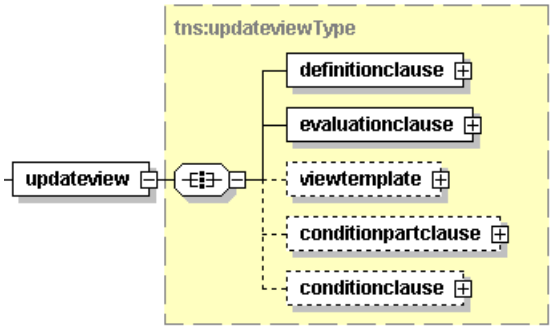


Позволява извикването на **insertview**.

Всяко **insertviewCall** се състои от тагове. Някои от таговете като **insertviewRef, viewvarRef** и **templatebinding** са задължителни, докато други от тях като **parameters** са допълнителни.

Име на тага	**Предназначение на тага**
insertviewRef	Реферира декларирано вече **insertview**.
viewvarRef	Реферира декларирана вече **viewvar**, където да се добавят данните.
templatebinding	Подава данните необходими за извършване на вмъкването.
parameters	Може да съдържа допълнителни параметри.

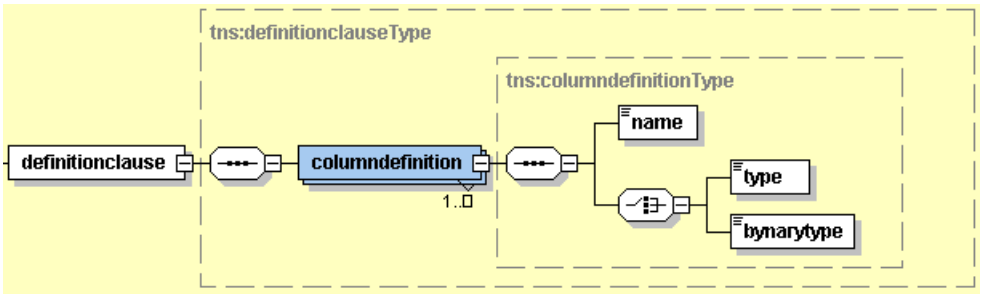
4.1.27 *updateview* - компонент



Позволява деклариране на **updateview**. Характерното за **updateview** е, че всички данни получени от него служат се модифицирането или актуализирането на някакъв източник. Източникът трябва да бъде локално view.

Всяко **updateview** се състои от тагове. Някои от таговете като **definitionclause** и **evaluationclause** са задължителни, докато други от тях като **viewtemplate**, **conditionpartclause** и **conditionpart** са допълнителни.

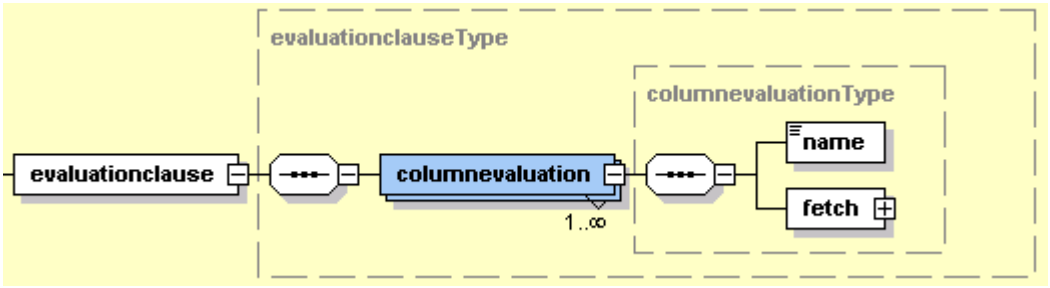
- **definitionclause**



Целта на тази клауза е да се дефинират колоните, в които ще се актуализират или променят данни посредством използването на

columndefinition. Всяка **columndefinition** си има **name** и **type** или **binarytype**. Тагът **name** определя името на колоната, докато тагът **type** или **binarytype** определя типа на колоната.

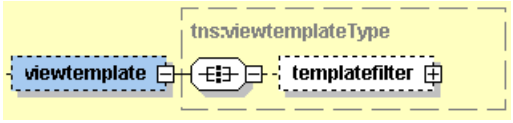
- **evaluationclause**



Целта на тази клауза е да се опишат изчисленията, които е необходимо да бъдат извършени за да се получи стойността на всяка една колона, която ще бъде актуализирана.

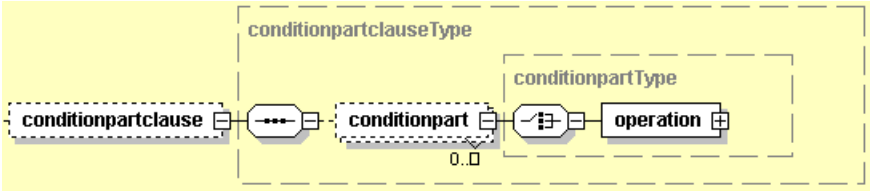
Всяко едно изчисление, което трябва да бъде извършено се задава посредством тага **columndefinition**. Последният трябва да съдържа следните тагове – **name** и **fetch**. Тагът **name** определя колоната, за която се отнасят изчисленията, а тагът **fetch** определя как точно ще бъде извлечена и изчислена стойността.

- **viewtemplate**



Може да се представи като контейнер с данни, част от чието съдържание ще послужи за актуализиране на съдържанието източника. Контейнерът с данни може от друга страна да участва в извличането и пресмятането на нови данни, които да бъдат използвани за актуализиране източника.

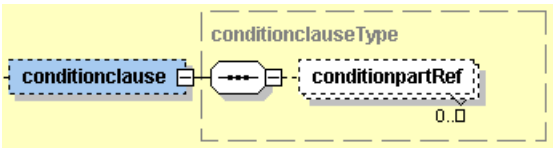
- **conditionpartclause**



Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията

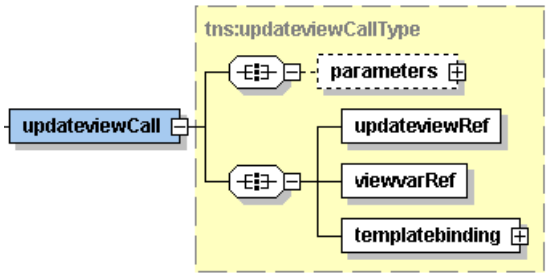
във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart**-ове, на които трябва да отговаря даден запис за да бъде актуализиран. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.28 *updateviewCall* - компонент

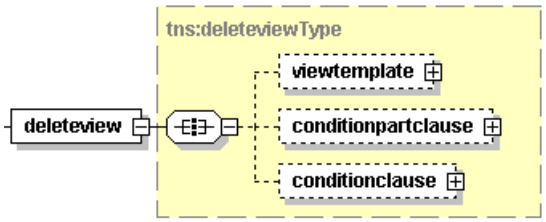


Позволява викането на **updateview**.

Всяко **updateviewCall** се състои от тагове. Някои от таговете като **updateviewRef, viewvarRef** и **templatebinding** са задължителни, докато други от тях като **parameters** са допълнителни.

Име на тага	**Предназначение на тага**
updateviewRef	Реферира декларирано вече **updateview**.
viewvarRef	Реферира декларирана вече **viewvar**, където да се актуализират данните.
templatebinding	Подава данните необходими за извършване на актуализирането.
parameters	Може да съдържа допълнителни параметри.

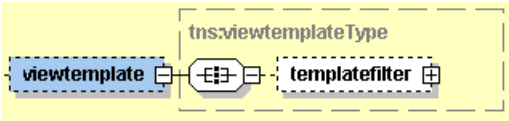
4.1.29 *deleteview* - компонент



Позволява деклариране на deleteview. Характерното за deleteview е, че чрез него се отстранява някакви данни от някакъв източник. Източникът трябва да бъде локално view.

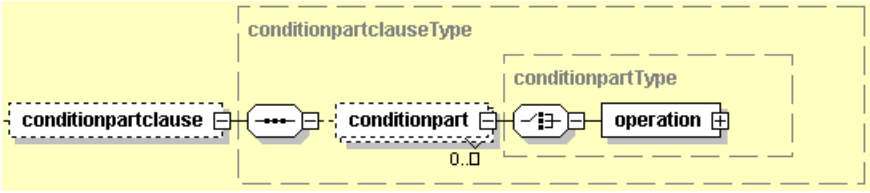
Всяко **deleteview** се състои от тагове. Таговете **viewtemplate**, **conditionpartclause** и **conditionclause** са допълнителни.

- **viewtemplate**



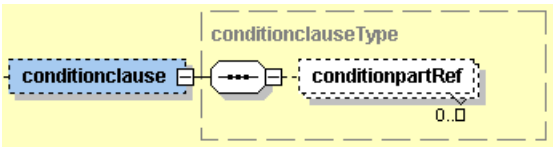
Може да се представи като контейнер с данни, част от чието съдържание ще послужи за изтриване на съдържанието източника. Контейнерът с данни може от друга страна да участва в извличането на данни за **conditionclause-**та.

- **conditionpartclause**



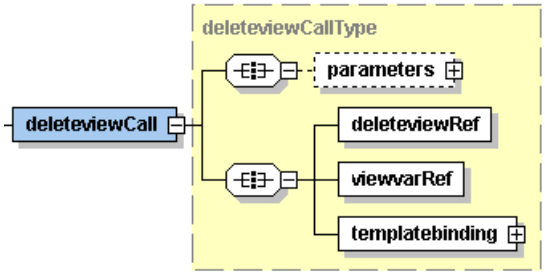
Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart**-ове, на които трябва да отговаря даден запис, за да бъде изтрит. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.30 *deleteviewCall* - компонент

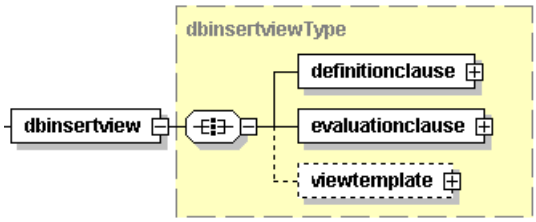


Позволява викането на **deleteview**.

Всяко **deleteviewCall** се състои от тагове. Някои от таговете като **deleteviewRef, viewvarRef** и **templatebinding** са задължителни, докато други от тях като **parameters** са допълнителни.

Име на тага	**Предназначение на тага**
deleteviewRef	Реферира декларирано вече **deleteview**.
viewvarRef	Реферира декларирана вече **viewvar**, от където да се изтрият данните.
templatebinding	Подава данните необходими за извършване на изтриването.
parameters	Може да съдържа допълнителни параметри.

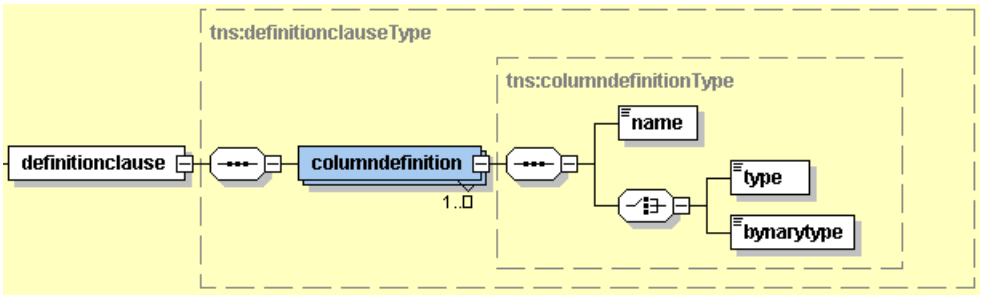
4.1.31 *dbinsertview* - компонент



Позволява деклариране на **dbinsertview**. Характерното за **dbinsertview** е, че всички данни получени от него се добавят в някакъв източник. Източникът трябва да бъде таблица в базата данни.

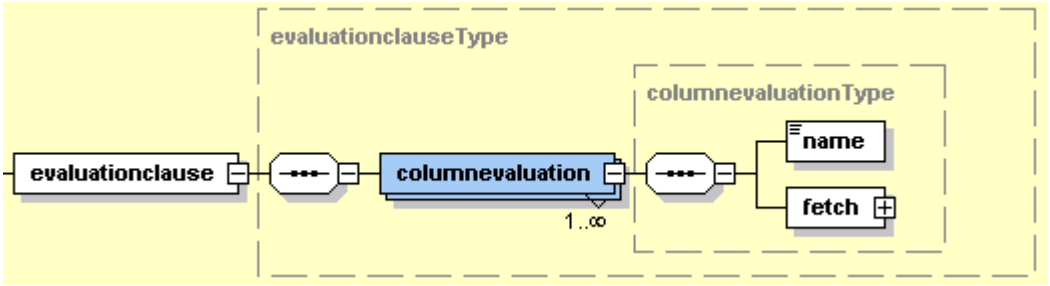
Всяко **dbinsertview** се състои от тагове. Някои от таговете като **definitionclause** и **evaluationclause** са задължителни, докато други от тях като **viewtemplate** са допълнителни.

- **definitionclause**



Целта на тази клауза е да се дефинират колоните в които ще се добавят данни посредством използването на **columndefinition**. Всяка **columndefinition** си има **name** и **type** или **binarytype**. Тагът **name** определя името на колоната, докато тагът **type** или **binarytype** определя типа на колоната.

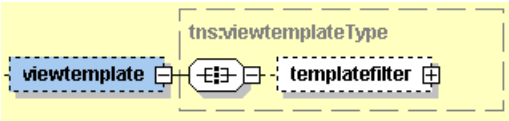
- **evaluationclause**



Целта на тази клауза е да се опишат изчисленията, които е необходимо да бъдат извършени за да се получи стойността на всяка една колона, която ще бъде добавяна.

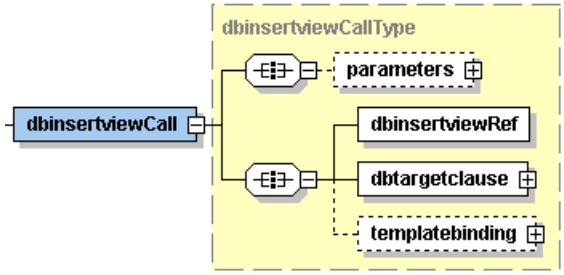
Всяко едно изчисление, което трябва да бъде извършено се задава посредством тагът **columndefinition**. Последният трябва да съдържа следните тагове – **name** и **fetch**. Тагът **name** определя колоната за която се отнасят изчисленията, а тагът **fetch** определя как точно ще бъде извлечена и изчислена стойността.

- **viewtemplate**



Може да се представи като контейнер с данни, част от чието съдържание ще бъде добавено към източника. Контейнерът с данни може от друга страна да участва в извличането и пресмятането на нови данни, които да бъдат добавени към източника.

4.1.32 *dbinsertviewCall* - компонент

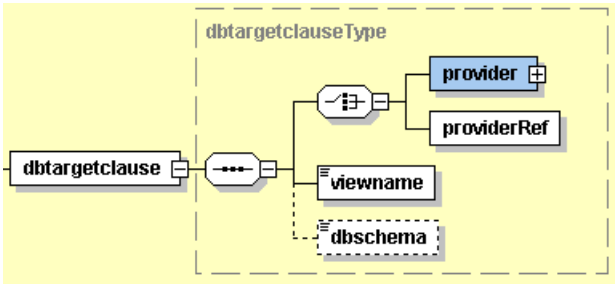


Позволява извикването на **dbinsertview**.

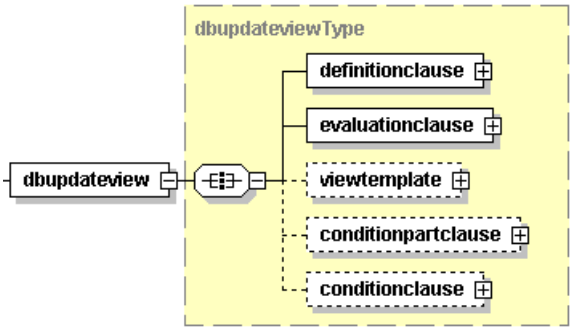
Всяко **dbinsertviewCall** се състои от тагове. Някои от таговете като **dbinsertviewRef** и **dbtargetclause** са задължителни, докато други от тях като **parameters** и **templatebinding** са допълнителни.

Име на тага	**Предназначение на тага**
dbinsertviewRef	Реферира декларирано вече **dbinsertview**.
dbtargetclause	Реферира външна таблица в база данни.
templatebinding	Подава данните необходими за извършване на вмъкването.
parameters	Може да съдържа допълнителни параметри.

Структурата на **dbtargetclause** има следният вид :



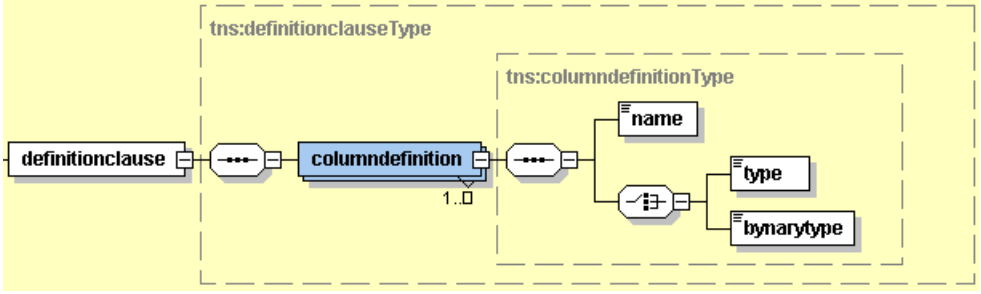
4.1.33 *dbupdateview* - компонент



Позволява деклариране на **dbupdateview**. Характерното за **dbupdateview** е, че всички данни получени от него служат за модифицирането или актуализирането на някакъв източник. Източникът трябва да бъде таблица в базата данни.

Всяко **dbupdateview** се състои от тагове. Някои от таговете като **definitionclause** и **evaluationclause** са задължителни, докато други от тях като **viewtemplate, conditionpartclause** и **conditionclause** са допълнителни.

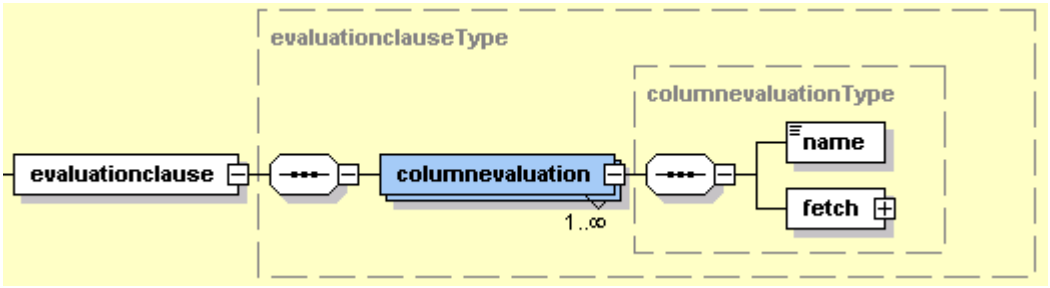
- **definitionclause**



Целта на тази клауза е да се дефинират колоните, в които ще се актуализират или променят данни посредством използването на

columndefinition. Всяка **columndefinition** си има **name** и **type** или **binarytype**. Тагът **name** определя името на колоната, докато тагът **type** или **binarytype** определя типа на колоната.

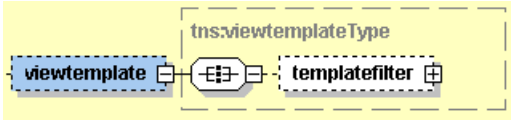
- **evaluationclause**



Целта на тази клауза е да се опишат изчисленията, които е необходимо да бъдат извършени за да се получи стойността на всяка една колона, която ще бъде актуализирана.

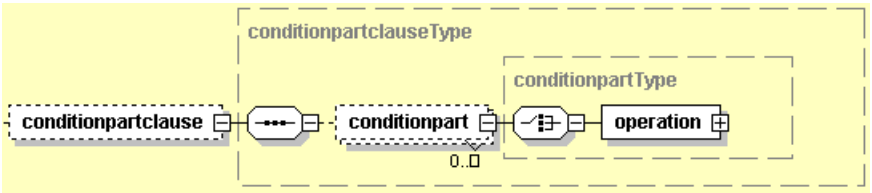
Всяко едно изчисление, което трябва да бъде извършено се задава посредством тага **columndefinition**. Последният трябва да съдържа следните тагове – **name** и **fetch**. Тагът **name** определя колоната за която се отнасят изчисленията, а тагът **fetch** определя как точно ще бъде извлечена и изчислена стойността.

- **viewtemplate**



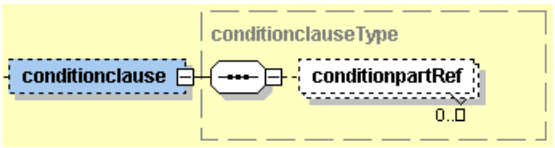
Може да се представи като контейнер с данни, част от чието съдържание ще послужи за актуализиране на съдържанието на източника. Контейнерът с данни може от друга страна да участва в извличането и пресмятането на нови данни, които да бъдат използвани за актуализиране източника.

- **conditionpartclause**



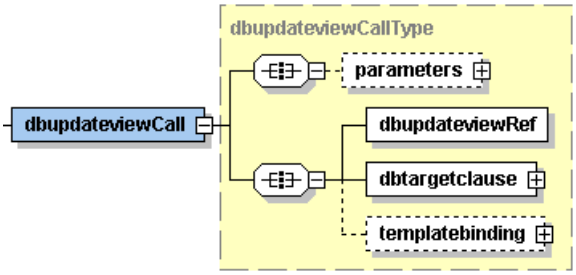
Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart**-ове, на които трябва да отговаря даден запис за да бъде актуализиран. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.34 *dbupdateviewCall* - компонент

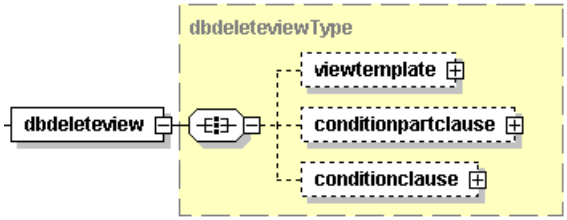


Позволява викането на **dbupdateview**.

Всяко **dbupdateviewCall** се състои от тагове. Някои от таговете като **dbupdateviewRef** и **dbtargetclause** са задължителни, докато други от тях като **parameters** и **templatebinding** са допълнителни.

Име на тага	**Предназначение на тага**
dbupdateviewRef	Реферира декларирано вече **dbupdateview**.
dbtargetclause	Реферира външна таблица в база данни.
templatebinding	Подава данните необходими за извършване на актуализирането.
parameters	Може да съдържа допълнителни параметри.

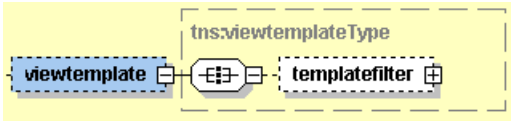
4.1.35 *dbdeleteview* - компонент



Позволява декларирането на **dbdeleteview**. Характерното за **dbdeleteview** е, че чрез него се отстранява някакви данни от някакъв източник. Източникът трябва да бъде таблица в базата данни.

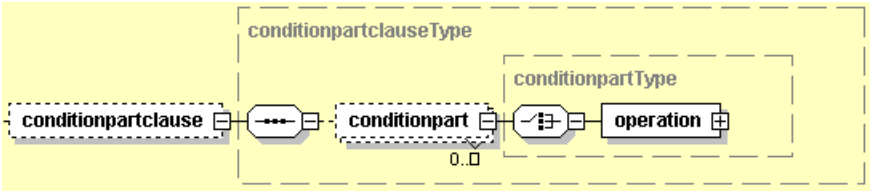
Всяко **dbdeleteview** се състои от тагове. Всички тагове на **dbdeleteview** се явяват допълнителни **- viewtemplate, conditionpartclause** и **conditionclause**.

- **viewtemplate**



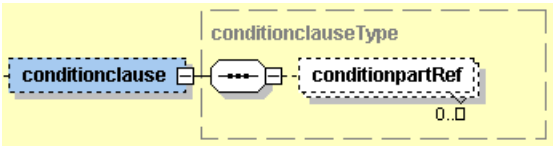
Може да се представи като контейнер с данни, чието съдържание ще бъде използвано за извличането и пресмятането на нови данни, които да бъдат послужат за определяне на записите, които да бъдат изтрити.

- **conditionpartclause**



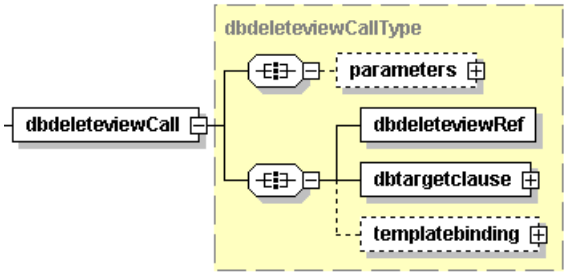
Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart**-ове, на които трябва да отговаря даден запис за да бъде изтрит. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.36 *dbdeleteviewCall* - компонент



Позволява викането на **dbdeleteview**.

Всяко **dbdeleteviewCall** се състои от тагове. Някои от таговете като **dbupdateviewRef** и **dbtargetclause** са задължителни, докато други от тях като **parameters** и **templatebinding** са допълнителни.

Име на тага	**Предназначение на тага**
dbdeleteviewRef	Реферира декларирано вече **dbdeleteview**.
dbtargetclause	Реферира външна таблица в база данни.
templatebinding	Подава данните необходими за извършване на изтриването.
parameters	Може да съдържа допълнителни параметри.

4.1.37 *xmlvar* - компонент

Позволява временно съхраняване на някаква информация под формата на xml документ, която е необходима за изпълнението на workflow - а.

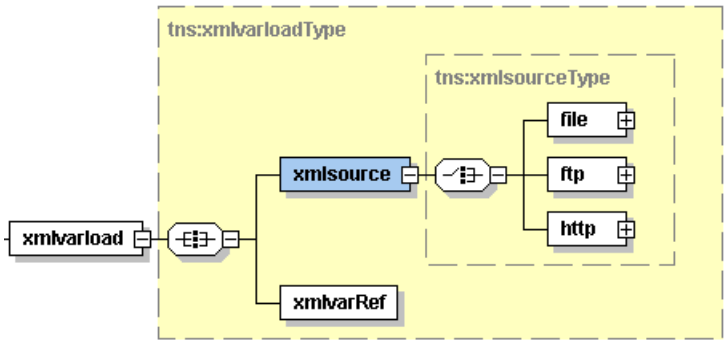
Областта на видимост на променливата е ограничена в рамките на **statement** елемента в който е декларирана. Тагът **xmlvar** може да съдържа следните атрибути:

Име на атрибута	**Предназначение на атрибута**
ID	Задава уникален идентификатор на променливата.
name	Задава името на променливата.
label	Задава описание на променливата.

Характерна особеност на променливата е начинът на нейното съхранение. Освен в оригиналния контекст, нейната предходна стойност се съхранява и в помощният контекс в случай, че **xmlvar** тагът е част от някаква по – голяма транзакция. Това се прави с цел транзакцията да може да възтанови всички стойности, които е модифицирала.

Реализацията на **xmlvar** компонента се явява изключително проста и няма нужда от допълнителни обяснения. Важно е да се спомене тук, че xml документа се съхранява в оперативната памет, което от своя страна може да доведе до проблеми, когато се налага работата с големи документи.

4.1.38 *xmlvarload* - компонент

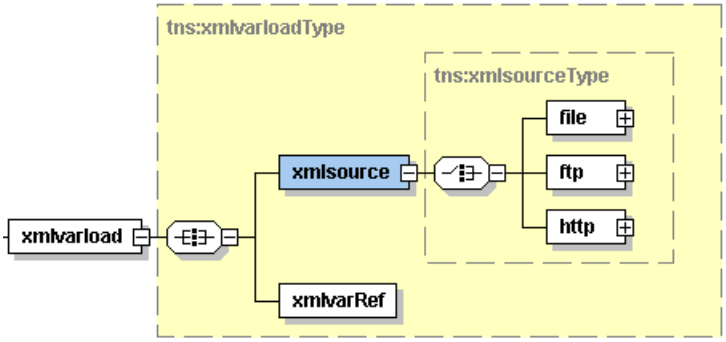


Позволява зареждането на xml документа от някакъв външен източник.

Характерна особеност на този компонент е начинът по който се указва променливата където да бъде съхранен xml документа обект, както и начинът по който той ще бъде извлечен. С помощта на **xmlvarRef** се дава възможност да се реферира създадена вече променлива, където да се запише обекта. За извличането на обекта се използва друг помощен компонент. Той е представен чрез **xmlsource** елемента и дава възможност за зареждане на документа посредством указването на локален файл (**file**), посредством използването на ftp за достъпване на файла (**ftp**) или посредством използването на http за достъпване на файла (**http**).

Реализацията на **xmlvarload** компонента е значително опростена използвайки **xmlsource** компонента, който в зависимост от своята имплементация може да агрегира в себе си доста комплексни и сложни функции.

4.1.39 *xmlvarstore* - компонент

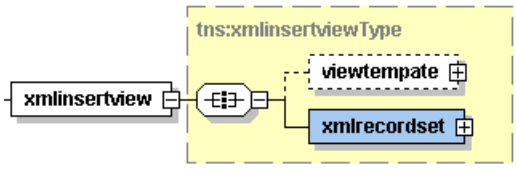


Позволява съхранението на xml документа на някакъв външен източник.

Характерна особеност на този компонент е начинът, по който се указва променливата от където ще бъде прочетен xml документа обект, както и начинът по който той ще бъде записан. С помощта на **xmlvarRef** се дава възможност да се реферира създадена вече променлива от където да бъде прочетен документа. За съхранението на обекта се използва друг помощен компонент. Той е представен чрез **xmlsource** елемента и дава възможност за записване на документа посредством указването на локален файл (**file**), посредством указването на ftp – сървър, където да бъде записан файла (**ftp**) или посредством указването на http – сървър, където да бъде записан файла (**http**).

Реализацията на **xmlvarstore** компонента значително се опростява използвайки **xmlsource** компонента, който в зависимост от своята имплементация може да агрегира в себе си доста комплексни и сложни функции.

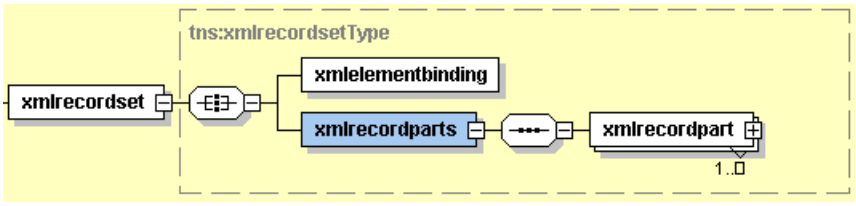
4.1.40 *xmlinsertview* - компонент



Позволява деклариране на **xmlinsertview**. Характерното за **xmlinsertview** е, че всички данни получени от него се добавят на данни в някакъв източник. Източникът трябва да бъде xml променлива.

Всяко **insertview** се състои от тагове. Някои от таговете като **xmlrecordset** са задължителни, докато други от тях като **viewtemplate** са допълнителни.

- **xmlrecordset**



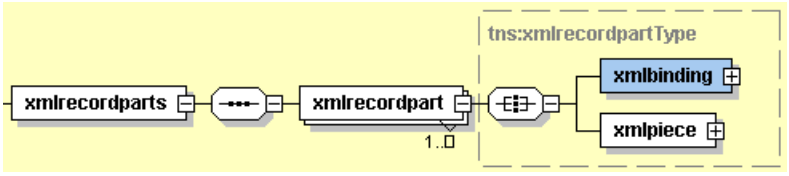
Целта на този таг е да опише начинът на образуване на всеки един нов запис, който ще бъде добавян. Тъй като записът ще бъде вмъкван в xml документ, логично е той самият да бъде представен като xml таг. За тази цел всеки **xmlrecordset** се състои от **xmlelementbinding** и **xmlrecordparts**.

- **xmlelementbinding**

  Служи за задаване на името на нов главен елемент, в който ще бъдат поместени данните от записа. Може да притежава следните атрибути.

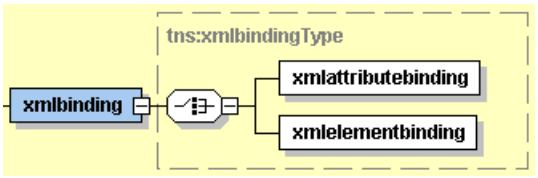
Име на атрибута	Предназначение
namespace	Задава namespace-а на главния елемент.
prefix	Задава prefix-а на главния елемент.
name	Задава името на главния елемент.
xmlbindingpolicy	Задава начинът на свързване на главния елемент спрямо областта в която ще бъде поместен. Може да бъде PREPEND, APPEND или REPLACE.

- **xmlrecordparts**



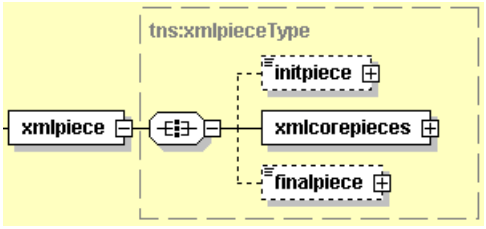
Служи за описание на съдържанието на главния елемент посредством използването на **xmlrecordpart** – ове, които представят различните части на записа. Целта на всеки **xmlrecordpart** е да добави определена част от записа към главния елемент, използвайки необходимите xml средства. За тази цел всеки **xmlrecordpart** се състои от **xmlbinding** и **xmlpiece**.

- ✓ **xmlbinding**



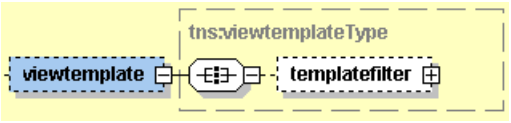
Описва как да се извърши добавянето на определена част към главният елемент. Частта може да се добави или като нов атрибут към главният елемент или като нов поделемент на главния елемент. Изборът на решение в случая зависи от изискванията към самия xml документ, като например схемата за валидиране.

- ✓ **xmlpiece**



Определя съдържанието, което съответната част може да добави. Съдържанието само по себе си се генерира използвайки елементите **initpiece**, **xmlcorepieces** и **finalpiece**. По този начин се осигурява по - голяма гъвкавост и възможност за обработка на данните преди те да бъдат добавени към главния запис.

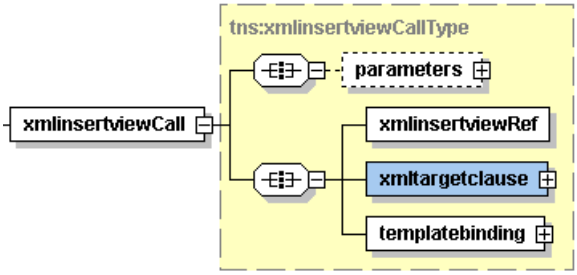
- **viewtemplate**



Може да се представи като контейнер с данни, част от чието съдържание ще бъде добавено към източника. Контейнерът с данни може от друга страна да участва в извличането и пресмятането на нови данни, които да бъдат добавени към източника.

Реализацията на **xmlinsertview** компонента се явява изключително комплексна задача, която включва в себе си много други подзадачи. Причината за това се корени в самото естество на задача. Детайлно обяснение за това как точно е имплементиран компонента ще бъде дадено при обясняване на реализацията.

4.1.41 *xmlinsertviewCall* - компонент



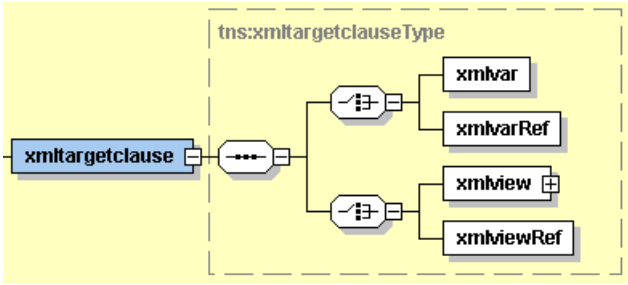
Позволява извикването на xmlinsertview.

Всяко едно **xmlinsertviewCall** се състои от различни тагове. Някои от таговете като **xmlinsertviewRef**, xmltargetclause и templatebinding са задължителни, докато други от тях като parameters са допълнителни.

- **xmlinsertviewRef**

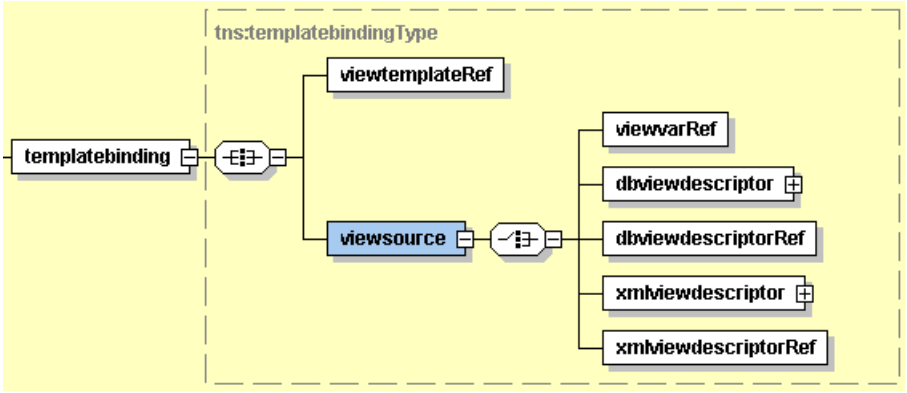
Целта на този таг е да реферира декларирано вече xmlinsertview.

- **xmltargetclause**



Целта на тази клауза е да дефинира променливата в която ще бъдат добавени данните посредством xmlvar или xmlvarRef, както и изгледа към тази променлива посредством използването на xmlview или xmlviewRef.

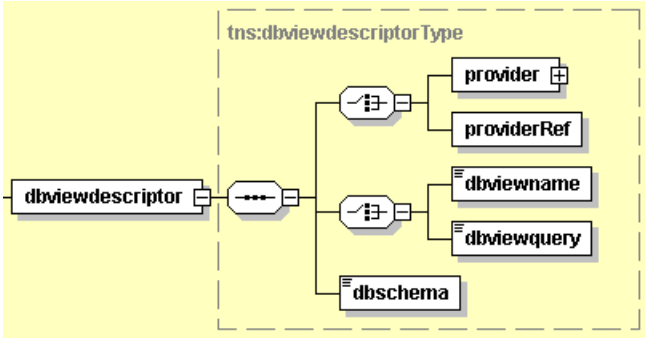
- **templatebinding**



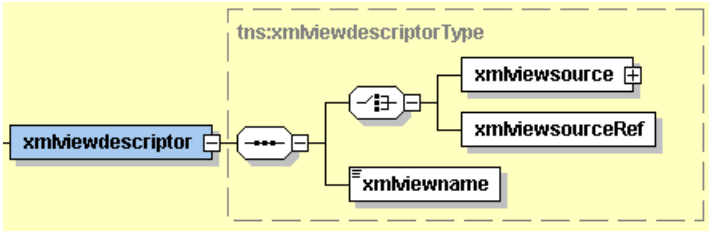
Целта на този таг е да свърже конкретен **viewtemplate**, който ще бъде използван при обработката на данните за **xmlinsertview** - то. Целта на всеки един viewtemplate е да бъде използван като контейнер от който да бъдат извличани данни, които да участват в необходимите изчисления. Реалните данни, които ще участват в изчисленията се асоциират използвайки някоя от възможните опции – **viewvarRef**, **dbviewdescriptor**, **dbviewdescrioptorRef**, **xmlviewdescriptor** или **xmlviewdescriptorRef**.

Име на тага	Предназначение
viewvarRef	Реферира създадена вече viewvar, която да се използва като източник на данни.
dbviewdescriptor	Указва таблица или изглед в база данни, който да бъде използван за източник на данните.
dbviewdescrioptorRef	Реферира деклариран вече dbviewdescriptor, който да се използва като източник на данни.
xmlviewdescriptor	Указва xml документ, част от данните на който да бъдат представени под формата на таблица, която да бъде използвана за източник на данните.
xmlviewdescriptorRef	Реферира деклариран вече xmlviewdescriptor, които да се използва като източник на данни.

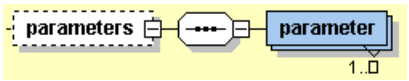
Структурата на **dbviewdescriptor** има следния вид:



Структурата на **xmlviewdescriptor** има следния вид:



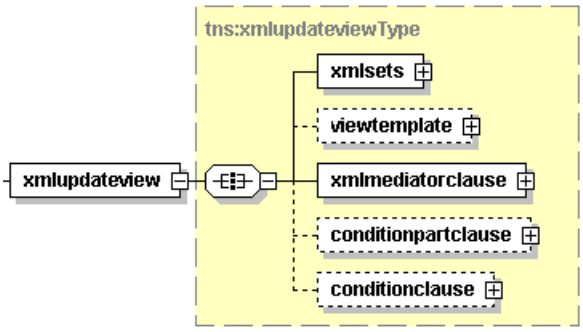
- **parameters**



Целта на този таг е да събере всички parameter, които се подават при извикването на **xmlinsertviewCall**.

Реализацията на **xmlinsertviewCall** компонента се явява изключително комплексна задача, която включва в себе си много други подзадачи. Детайлно обяснение за това как точно е имплементиран компонента ще бъде дадено при обясняване на реализацията.

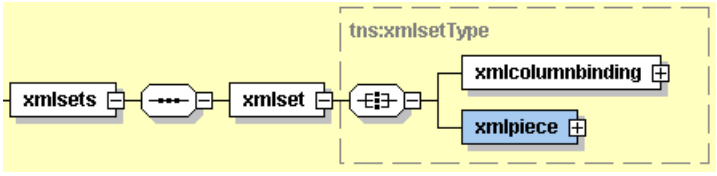
4.1.42 *xmlupdateview* - компонент



Позволява декларирането на **xmlupdateview**. Характерното за **xmlupdateview** е, че всички данни получени от него служат за модифицирането или актуализирането на някакъв източник. Източникът трябва да бъде xml променлива.

Всяко **xmlupdateview** се състои от тагове. Някои от таговете като **xmlsets** и **xmlmediatorclause** са задължителни, докато други от тях като **viewtemplate, conditionpartclause** и **conditionclause** са допълнителни.

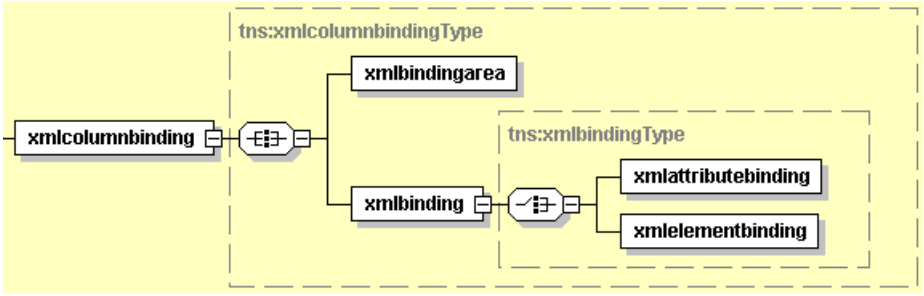
- **xmlsets**



Служи за описание на актуализациите, които трябва да бъдат извършени върху всеки един запис отговарящ на условията от **conditionclause**.

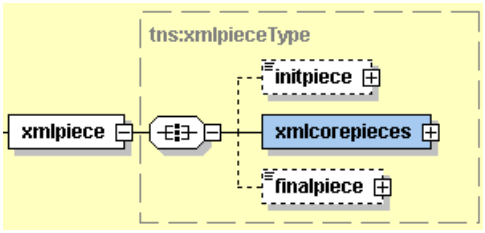
Всяка актуализация се задава посредством **xmlset** тага, който от своя страна се състои от таговете **xmlcolumnbinding** и **xmlpiece**.

  - **xmlcolumnbinding**



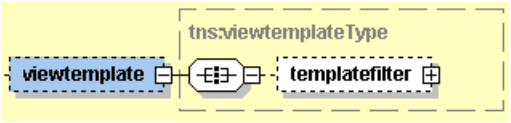
Служи за описание на мястото на актуализация и това как точно да бъде извършена тя. Състои се от таговете **xmlbindingarea** и **xmlbinding**. Посредством **xmlbindingarea** се задава поделемент на главния елемент, който трябва да бъде актуализиран. Посредством **xmlbinding** се определя как ще бъде модифициран поделемента. Модифицирането може да стане или чрез използването на атрибут или чрез използването на елемент, чието съдържание да бъде променено използвайки стойността от **xmlpiece**.

- **xmlpiece**



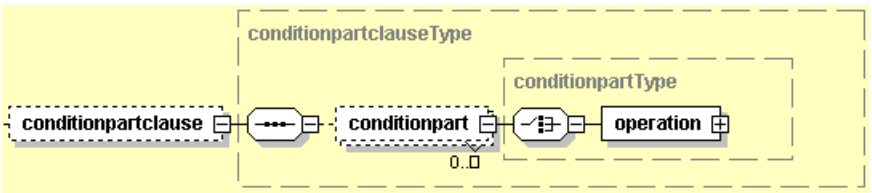
Определя съдържанието, което се добавя на мястото указано от **xmlcolumnbinding**. Съдържанието само по себе си се генерира използвайки елементите **initpiece**, **xmlcorepieces** и **finalpiece**. По този начин се осигурява по - голяма гъвкавост и възможност за обработка на данните преди те да бъдат добавени.

- **viewtemplate**



Може да се представи като контейнер с данни, част от чието съдържание ще послужи за актуализиране на съдържанието източника. Контейнерът с данни може от друга страна да участва в извличането и пресмятането на нови данни, които да бъдат използвани за актуализиране на източника.

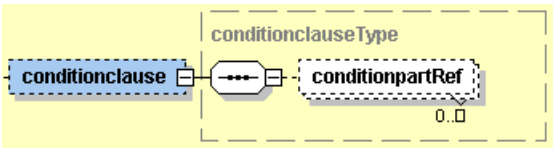
- **conditionpartclause**



Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията

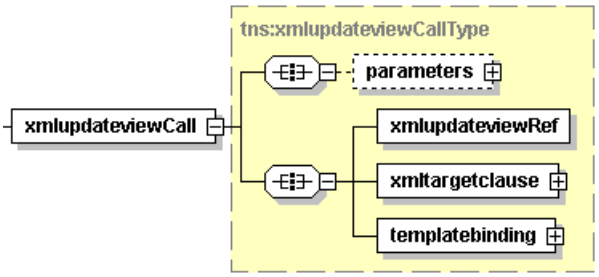
във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart**-ове, на които трябва да отговаря даден запис за да бъде актуализиран. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.43 *xmlupdateviewCall* - компонент



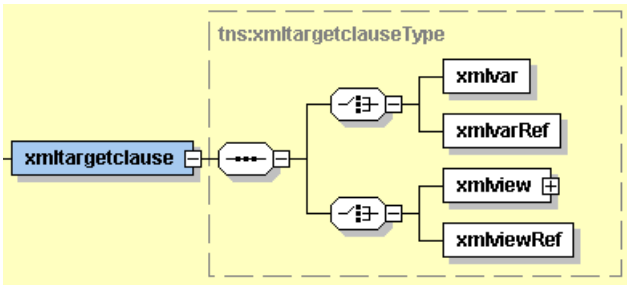
Позволява извикването на **xmlupdateview**.

Всяко едно **xmlupdateviewCall** се състои от различни тагове. Някои от таговете като **xmlupdateviewRef**, **xmltargetclause** и **templatebinding** са задължителни, докато други от тях като **parameters** са допълнителни.

- **xmlupdateviewRef**

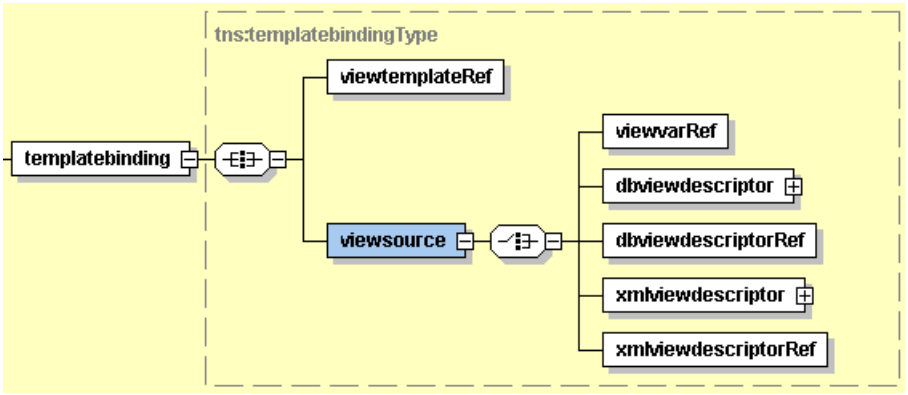
Целта на този таг е да реферира декларирано вече **xmlupdateview**.

- **xmltargetclause**



Целта на тази клауза е да дефинира променливата, в която ще бъдат актуализирани данните посредством **xmlvar** или **xmlvarRef**, както и изгледа към тази променлива посредством използването на **xmlview** или **xmlviewRef**.

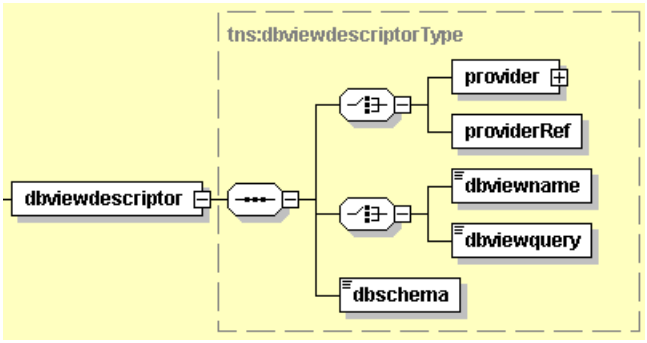
- **templatebinding**



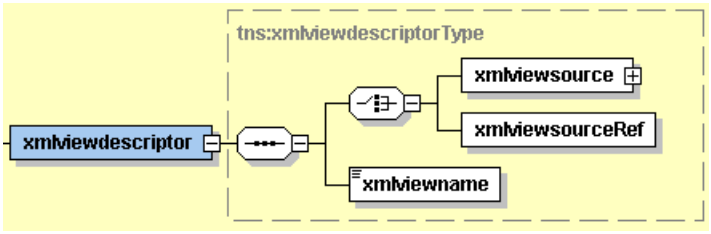
Целта на този таг е да свърже конкретен **viewtemplate**, които ще бъде използван при обработката на данните за **xmlinsertview** - то. Целта на всеки един **viewtemplate** е да бъде използван като контейнер от който да бъдат извличани данни, които да участват в необходимите изчисления. Реалните данни, които ще участват в изчисленията се асоциират използвайки някоя от възможните опции – **viewvarRef**, **dbviewdescriptor**, **dbviewdescrioptorRef**, **xmlviewdescriptor** или **xmlviewdescriptorRef**.

Име на тага	Предназначение
viewvarRef	Реферира създадена вече **viewvar**, която да се използва като източник на данни.
dbviewdescriptor	Указва таблица или изглед в база данни, който да бъде използван за източник на данните.
dbviewdescrioptorRef	Реферира деклариран вече **dbviewdescriptor**, който да се използва като източник на данни.
xmlviewdescriptor	Указва xml документ, част от данните на който да бъдат представени под формата на таблица, която да бъде използвана за източник на данните.
xmlviewdescriptorRef	Реферира деклариран вече **xmlviewdescriptor**, които да се използва като източник на данни.

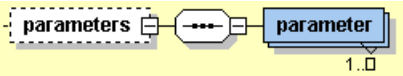
Структурата на **dbviewdescriptor** има следния вид:



Структурата на **xmlviewdescriptor** има следния вид:



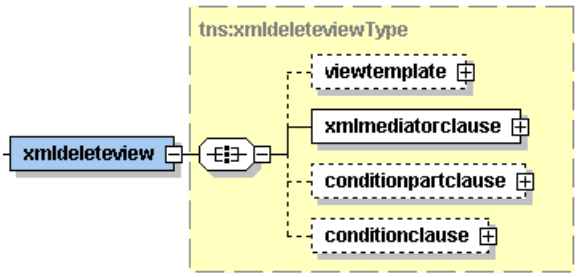
- **parameters**



Целта на този таг е да събере всички **parameter**, които се подават при извикването на **xmlupdateviewCall**.

Реализацията на **xmlupdateviewCall** компонента се явява изключително комплексна задача, която включва в себе си много други подзадачи. Детайлно обяснение за това как точно е имплементиран компонента ще бъде дадено при обясняване на реализацията.

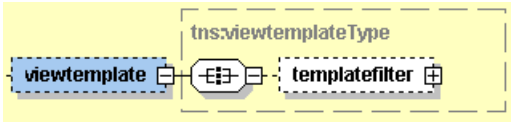
4.1.44 *xmldeleteview* - компонент



Позволява декларирането на **xmldeleteview**. Характерното за **xmldeleteview** е, че чрез него се отстранява някакви данни от някакъв източник. Източникът трябва да бъде xml променлива.

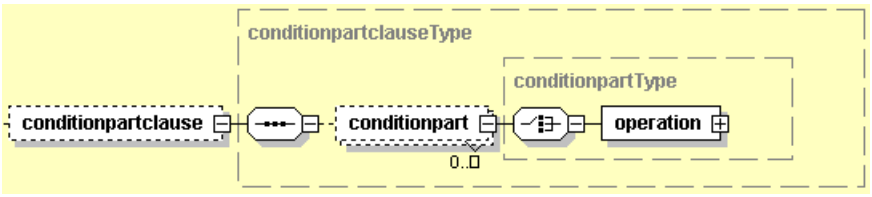
Всяко **xmldeleteview** се състои от тагове. Някои от таговете като **xmlmediatorclause** са задължителни, докато други от тях като **viewtemplate, conditionpartclause** и **conditionclause** са допълнителни.

- **viewtemplate**



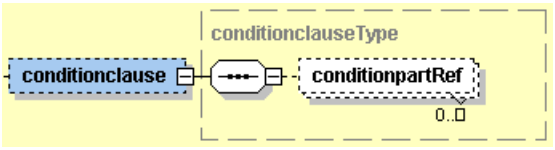
Може да се представи като контейнер с данни, чието съдържание ще бъде използвано за извличането и пресмятането на нови данни, които да послужат за определяне на записите, които да бъдат изтрити.

- **conditionpartclause**



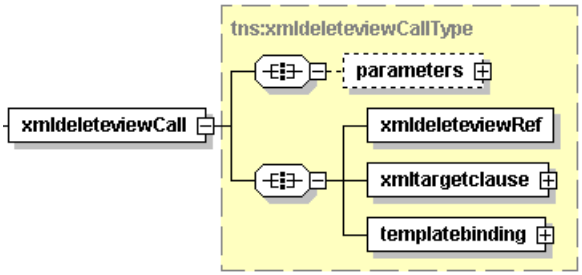
Целта на тази клауза е да се дефинират **conditionpart** – ове, които могат да участват в **conditionclause**. Без значение колко сложна е операцията във всеки един **conditionpart**, резултатът от нея трябва винаги да е булева стойност.

- **conditionclause**



Целта на тази клауза е да се реферират посредством **conditionpartRef** всички **conditionpart**-ове, на които трябва да отговаря даден запис за да бъде изтрит. За да е удовлетворен даден **conditionpart**, той трябва да връща като резултат булева истина.

4.1.45 *xmldeleteviewCall* - компонент



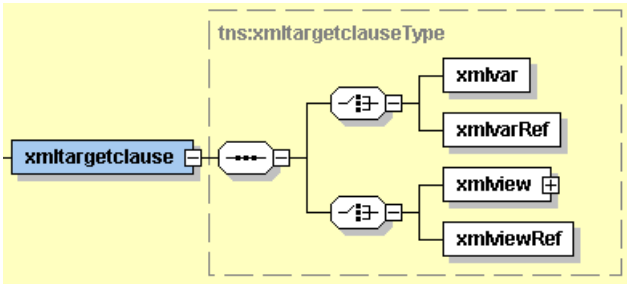
Позволява извикването на **xmldeleteview**.

Всяко едно **xmldeleteviewCall** се състои от различни тагове. Някои от таговете като **xmldeleteviewRef**, **xmltargetclause** и **templatebinding** са задължителни, докато други от тях като **parameters** са допълнителни.

- **xmldeleteviewRef**

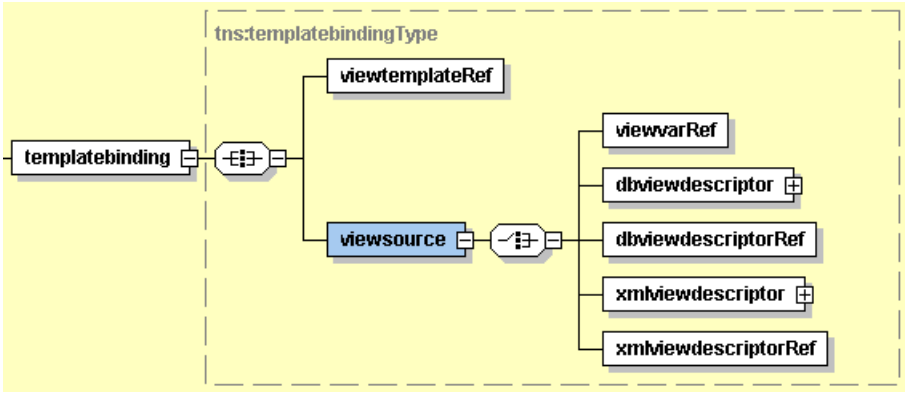
Целта на този таг е да реферира декларирано вече **xmldeleteview**.

- **xmltargetclause**



Целта на тази клауза е да дефинира променливата, от която ще бъдат изтривани данните посредством **xmlvar** или **xmlvarRef**, както и изгледа към тази променлива посредством използването на **xmlview** или **xmlviewRef**.

- **templatebinding**

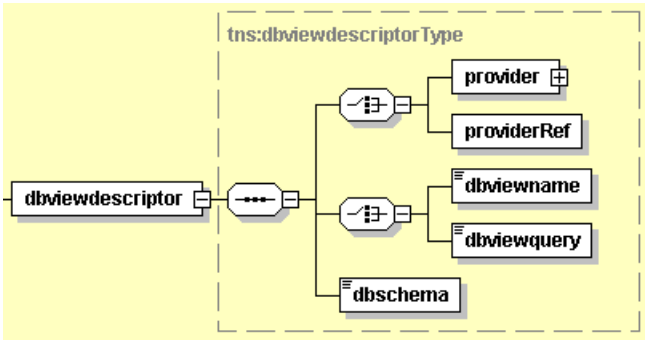


Целта на този таг е да свърже конкретен **viewtemplate**, които ще бъде използван при обработката на данните за **xmlinsertview** - то. Целта на всеки един **viewtemplate** е да бъде използван като контейнер, от който да бъдат извличани данни, които да участват в необходимите изчисления. Реалните данни, които ще участват в изчисленията се асоциират използвайки някоя от възможните опции – **viewvarRef**, **dbviewdescriptor**, **dbviewdescrioptorRef**, **xmlviewdescriptor** или **xmlviewdescriptorRef**.

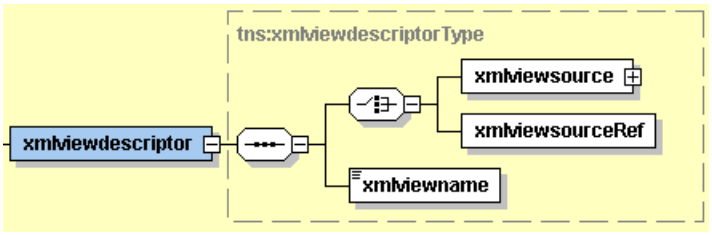
Име на тага	Предназначение
viewvarRef	Реферира създадена вече **viewvar**, която да се използва като източник на данни.
dbviewdescriptor	Указва таблица или изглед в база данни, който да бъде използван за източник на данните.

dbviewdescrioptorRef	Реферира деклариран вече **dbviewdescriptor**, който да се използва като източник на данни.
xmlviewdescriptor	Указва xml документ, част от данните на който да бъдат представени под формата на таблица, която да бъде използвана за източник на данните.
xmlviewdescriptorRef	Реферира деклариран вече **xmlviewdescriptor**, които да се използва като източник на данни.

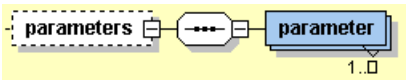
Структурата на **dbviewdescriptor** има следния вид:



Структурата на **xmlviewdescriptor** има следния вид:



- **parameters**



Целта на този таг е да събере всички **parameter**, които се подават при извикването на **xmldeleteviewCall**.

Реализацията на **xmldeleteviewCall** компонента се явява изключително комплексна задача, която включва в себе си много други подзадачи. Детайлно обяснение за това как точно е имплементиран компонента ще бъде дадено при обясняване на реализацията.

4.2 Изисквания към компонентите използвани в системата.

Винаги при изграждането на някаква голяма система трябва да се вземат в предвид характерните особености на средата, в която тя обикновено ще функционира. Тези особености условно могат да бъдат разделени на два вида : хардуерни и софтуерни.

Когато става въпрос за хардуер, възможностите ни са ограничени поради много пазарно - икономически причини и за това като хардуерна платформа е избрана традиционната такава за PC – компютри, т.е като изискване пред нас стои задачата тази система да може да работи на обикновен домашен компютър.

Когато става въпрос за софтуер ситуацията вече не е толкова проста. Причините за това са много на брой и комплексни и ще бъдат разгледани последователно. Главното нещо, за което трябва да се помисли в случая е операционната система. Тъй като за програмен език е избран Java, то следва да се отбележи, че за платформите на които ще може да се инсталира системата трябва да има имплементирана Java Virtual Machine. За щастие днес Java Virtual Machine е имплементирана за почти всички платформи, което прави изборът й лесен, логичен и желан. Друго много важно нещо, което трябва да се вземе в предвид е, че софтуерните изискванията на системата представляват съвкупност от софтуерните изисквания на отделните компоненти, които изграждат системата. Следователно за да си изградим пълна представа за софтуерните изисквания на системата ще трябва да разгледаме отделните компоненти на системата.

4.2.1 *Изисквания към конектора използван в системата.*

Пример за компонент със специфични изисквания се явява конекторът към СУБД. Поради своята същност и специфика, на конектора му се налага да осъществява достъп до различни СУБД - та. Поради факта, че на пазара съществува голямо разнообразие от СУБД – та, всяко от които има своите особености следва, че системата трябва да се ограничи само до едно разумно количество от тях, което да поддържа. При това все пак трябва да се гарантира възможността на системата да запази своята **разширяемост** и в случай на необходимост да се осигури свързаност към други СУБД - та. За да може конекторът да осъществява нормално своите функции на него му се налага да използва допълни компоненти. Тези допълнителни компоненти се наричат драйвери и всяка една база данни притежава свой собствен драйвер. Драйверът за всяко една СУБД може да бъде написан на различни езици в зависимост от това за кои от тях е решено да могат да осъществяват връзка към СУБД. За да може даден драйвер да работи със системата той трябва да е бил написан за Java. Другото важно нещо е този драйвер да може да работи със същата версия на Java, на която е написана да работи самата система. За щастие днес по – голямата част от производителите на бази данни предлагат драйвери написани на Java за техните СУБД - та така, че за системата остава възможността да избере просто драйверите, с които иска да работи, а оттам и СУБД –тата, до които иска да има достъп. Друго важно нещо което трябва да се знае е, че драйверът за дадена СУБД може да има различни версии. Версията на

драйвера обикновено съвпада с версията на СУБД към която той се свързва, но това не е задължително. Някои от драйверите може да поддържат обратна съвместимост, докато други да не поддържат такава. Под обратна съвместимост се разбира възможността драйверът да работи с по – стара версия на СУБД от тази, за която е бил написан.

4.2.2 *Изисквания към контейнера използван в системата.*

За да може системата нормално да извършва своите функции свързани с обработката на данни и генерирането на справки, на нея трябва да и се предостави възможност да съхранява данните, които в последствие ще обработва. За тази цел се използва контейнер, чиято цел е да съхранява данните. Във връзка с това, че данните ще се наложи да бъдат обработвани следва да се вземе в предвид начинът, по който те ще бъдат съхранявани, както и времето за достъп до тях. Нека приемем условно, че видовете обработка са само два – интензивен и неинтензивен. Под интезивен начин на обработка ще разбираме такава, при която се налага често четене и записване на данни в контейнера. Под неинтензивна обработка ще разбираме такава, при която не се налага многократно четене или записване на данните. Независимо каква ще бъде обработката на данни тя трябва да може да се извърши за разумно време т.е. на лице вече е факторът **производителност** на системата. Ясно е, че ако обработката на данните изисква прекалено много време, то тя едва ли е високо ефективна, което от своя страна води до ниска производителност. Преди да пристъпим към обяснение на останалите изисвания към контейнера ще се наложи да разясним от какво зависи високо - производителната обработка на данните както и какво е мястото на контейнера в нея.

Известно е, че за да може да се извършва каквато и да било обработка са необходими ресурси. Ресурсите свързани с всяка обработка обикновено са процесорно време, оперативна памет и дисково пространство. Процесорното време обикновенно е свързано пряко с операциите които са необходими за извършване на самата обработка, а самата обработка пък зависи от алгоритъма, който е избран. Ако предположим, че алгоритъмът на обработка зависи конкретно от самата цел на обработката става ясно, че това е ресурс който не може да има нищо общо с контейнера. И така остават ни двата други ресурса – оперативна памет и дисково пространство, които бихме могли да използваме за постигане на по – добра производителност. Когато се говори за данни е важно да се вземе в предвид техния обем. В зависимост от обема на данните може да се търси ефективен начин за тяхното съхранение, а от там и ефективен начин за достъп до тях. Както се знае данните могат да се съхраняват в оперативната памет, с което се постига минимално време за достъп до тях и се увеличава производителноста на обработката или пък данните могат да се съхраняват върху дисковото пространство, при което достъпът до тях е бавен и се увеличава времето да достъп до тях. И двата начина на съхранение си имат своите предимства и недостатъци, които са изброени в съответната таблица :

Оперативна памет	Дисково пространство
Предимство : Бърз достъп до данните.	Предимство : Голям съхраняван обем от данни.
Недостатък : Малък съхраняван обем от данни.	Недостатък : Бавен достъп до данните.

Както се вижда от таблицата и двата варианта не са подходящи за директно използване в реална система. Причината за това е, че системата не може първоначално да знае колко данни ще бъдат поместени в контейнера, за да знае къде да разположи контейнера (оперативна памет или дисково пространство). Другият проблем, който трябва да решим тук е търсенето на баланс между **производителност** и **скалируемост** : ако имаме голям обем от данни, то той не може да бъде поместен в оперативната памет, което води до ниска скалируемост на системата и невъзможност тя да участва в обработката на голями обеми от данни.

Изводът, който следва да бъде направен е, че за да постигнем баланс между **производителност** и **скалируемост** трябва една част от данните да бъдат разположени в оперативната памет, а друга част от данните да бъдат разположени върху дисковото пространство т.е. налага се контейнерът да може да работи както с опрепативната памет така и с дисковото пространство. Възможността какво точно да е съотношението на между използвана опретивна памет и използвана дискова памет трябва да може да се задава директно от потребителя. Само по този начин контейнера би могъл да участва в една високо производителна обработка.

4.2.3 *Изисквания към контекста използван в системата.*

Всяка една голяма система съдържа обикновено някакъв контекст. Целта на всеки един контекст е да съхранява групите от данни, за които той е проектиран и предназначен, както и да осигурява достъп до тях в случай, че това е необходимо.

Нуждата от специфичен контекст при нас е продиктувана от необходимостта да се осигури възможност на системата да съхранява информацията, която се получава по време на изпълнението на workflow - а. Тази информация обикновено има нужда да бъде съхранявана някъде с цел да бъдат използвани в по - късен етап от изпълнението на workflow - а. От тази гледна точка контекстът може да се представи като компонент, чиято основна задача е да съхранява информацията необходима за нормалното изпълнение на workflow – а.

За да може да бъде използван ефективно всеки един контекст, то той трябва да е проектиран да осигурява възможност за надежден и бърз достъп до неговите данните. Данните от контекста обикновено трябва да бъдат извличани бързо и за тази цел се налага той да е поместен в оперативната памет.

Като заключение може да се каже, че от производителността на контекста зависи цялостната производителност на системата.

4.2.4 *Изисквания към всички компоненти използвани в системата.*

Освен специфичните изисквания на отделните компоненти свързани със системата, съществуват и отделни изисквания, които са общовалидни за всички компоненти. Пример за такова изискване е възможността всеки един компонент да работи както с главният контекст, така и с помощния контекст. Само по този начин би могло да се гарантира, че дори при неуспех на транзакцията тя ще успее да възстанови напълно първоначалните данни в главният контекст.

5. Реализация на системата.

Едва след като бъде проектирана системата се преминава към следващата стъпка от цялостното й изграждане, която се явява нейната реализация. Тази реализация има за цел да осигури нормалното протичане на цялостната имплементация на системата.

Друго характерно за този етап е, че при него се определя окончателният завършен облик на архитектурата на системата, както и това каква част от дефинираните по дизайн характеристики ще бъдат имплементирани.

Изборът на начина за изграждане на структурата на системата и използваните от нея библиотеките също заема важно място в етапа на реализация.

5.1. Стъпки свързани с реализацията на системата.

Реализацията на системата може да се представи като комплексна последователност от стъпки през които трябва да се премине за да може да се имплементира системата.

- *Начало на реализацията* – тази стъпка има за цел да верифицира дали областта и обхвата на системата са реално изпълними с дефинираните за целта технически средства, както и да провери дали наличните ресурси са достатъчни за реализирането на системата преди да се премине към имплементирането й. Целта и областта на проекта също може да се наложи да бъдат проверени внимателно за да се прецени дали може да се използва опита от подобни разработвати системи. След като се установи, че може да се премине към разработването на проекта следва да се разяснят характерните за него дефиниции, акроними и абривиатури, както и да се укажат референции към други източници, даващи по – детайлна информация необходима при разработката и имплементирането.

- *Цялостен преглед на реализацията* – тази стъпка има за цел да разгледа подробно всяка една от перспективите свързани с разработването на системата, както и да определи окончателно обхвата на функциите, които последната ще изпълнява. Друг основен детайл, който не бива да се забравя е да се провери до колко системата удовлетворява потребителски нужди за които е предназначена, както и ограниченията, предпоставките и зависимостите, пред които последната е поставена.

- *Специфични изисквания за реализацията* – тази стъпка има за цел да верифицира и осъществи имплементацията на някакви специфични изисквания, които могат да имат отношение към потребителския, хардуерния, софтуерния или комуникация интерфейс на системата. Допълнително се налага и пълното репродуциране на специфични

информационни потоци и процеси, които могат да протичат в системата. За системите използващи бази данни се налага цялостна проверка на вътрешната организация на базата данни и доколко последната покрива реалните нужди на системата.

- *Имплементация на системата* – на тази стъпка се имплементира архитектурата описана при дизайна на системата. За тази цел се налага декомпозицията на системата до съставни й модули и определяне на предназначението и обхвата на всеки модул. При обособяването на отделните модули е желателно да се определи вътрешното ниво на зависимост между тях с цел да се осигури безпроблемната им интеграция в системата. Обикновено тук също така се определят и точните *desing patterns*, които ще бъдат използвани във всеки модул с цел да се постигне по – голяма структурираност и производителност на системата. За да бъде имплементирана една система напълно е необходимо да бъдат имплементирани отделните й съставни компоненти.

5.2. Реализация на базовите компоненти на системата.

Помощните компоненти могат да се използват от основните компоненти на системата и представляват удобен начин да се организира кода в систематизиран и прозрачен за използване вид. Тяхната роля е особено важна за опростяването на логиката на основните компоненти и е основна причини те да съществуват.

5.2.1 Реализация на IBinaryContext.

IBinaryContext служи за съхраняване на основната информация получена в резултат на изпълнението на workflow – а. Информацията, която може да се съхранява е или дефинирана в workflow – а посредством използването на някой основен компонент или получена в резултат на използването на други помощни или основни компоненти на системата.

Ако трябва да бъдат обобщени накратко функциите, които се налага да извършва този компонент, то те са свързани най – основно с добавянето, изтриването, проверката и замяната на данни в контекста на системата по време на изпълнението на workflow – а. За да може главният контекст да поддържа получаването на информация от други контексти или възнановяването на информация използвайки помощния контекст се налага наличието на съответните методи в интерфейса.

Изискванията на системата са тясно свързани с това информацията от контекста да може да бъде копирана в случай на необходимост и за това се налага използването на методи, които да предоставят такава функционалност.

След като беше описано накратко какво прави съответния компонент, следва да бъде представен и неговия интерфейс.

```java
/**
 * IBinaryContext.java - IMD - 15.04.2006
 */
package workflow.context.interfaces;

import java.util.List;

import org.w3c.dom.Element;

/**
 * IBanryContext is used to hold all the information that is result of
 * executing the workflow.
 * <P>
 * The information that is holded depends on those
 * class who set it.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface IBinaryContext
{
	/**
	 * Gets namespaces.
	 *
	 * @return List If no namespaces exist it returns empty list.
	 */
	public List getNamespaces();

	/**
	 * Gets group names for the namespace given as parameter.
	 *
	 * @param namespace Namespace that group names are needed.
	 * @return List If no group names exist it returns empty list.
	 */
	public List getGroupNames(String namespace);

	/**
	 * Gets identifiers for the namespace and group name given as
	 * parameters.
	 *
	 * @param namespace Namespace. Not null.
	 * @param groupName Group name. Not null.
	 * @return List If no identifiers exist it returns empty list.
	 * */
	public List getIdentifiers(String namespace, String groupName);

	/**
	 * Dereferences the element and provides it binary presentation.
	 *
	 * @param referenceElement Element that is to be resolved.
	 * @return Object (entry) that corresponds to Element.
	 */
	public Object resolveReference(Element referenceElement);
```

```
/**
 * Resolves entry in context and provides it binary presentation.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @return Object possessing above characteristics. May be null.
 */
public Object resolveEntry(String entryNS,
                           String entryGN,
                           String entryID);

/**
 * Appends entry to the binary context.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @param entry   Entry to append presented as binary object.
 */
public void appendEntry(String entryNS,
                        String entryGN,
                        String entryID,
                        Object entry);

/**
 * Removes entry from the binary context.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @return Object possessing above characteristics.
 */
public Object removeEntry(String entryNS,
                          String entryGN,
                          String entryID);

/**
 * Checks if entry is in binary context.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @return True if binary context contains the entry.
 */
public boolean containsEntry(String entryNS,
                             String entryGN,
                             String entryID);
```

```
        /**
         * Replaces entry in the binary context with another one.
         *
         * @param entryNS Entry namespace.
         * @param entryGN Entry group name.
         * @param entryID Entry identifier.
         * @param entry   Entry to append presented as binary object.
         * @return Object that was replaced. May be null.
         */
        public Object replaceEntry(String entryNS,
                                   String entryGN,
                                   String entryID,
                                   Object entry);

        /**
         * Clears context.
         * */
        public void clear();

        /**
         * Merges the binary context given as parameter with the current
         * one.
         *
         * @param otherBinaryContext Other binary context.
         */
        public void merge(IBinaryContext otherBinaryContext);

        /**
         * Restores the original values of some of the variables in the
         * binary context using the transaction context given as
         * parameter. Only variables that are in the transaction context
         * will be restored.
         *
         * @param transactionContext Transaction context with original
         *                           values.
         */
        public void restore(ITransactionContext transactionContext);
}
```

5.2.2 Реализация на IDomContext.

IDomContext служи за съхраняване на помощната информация получена в резултат на изпълнението на workflow – а. Информацията, която може да се съхранява представлява най – често xml тагове, които са били обработени, но се налага да бъдат ползвани и за в бъдеще. Тези тагове, представени под формата на DOM елементи е необходимо да се съхраняват някъде и именно за това служи помощния контекст.

Ако трябва да бъдат обобщени накратко функциите, които се налага да извършва този компонент, то те са свързани най – основно с добавянето, изтриването, проверката и замяната на данни в помощния контекст на системата по време на изпълнението на workflow – а.

Изискванията на системата са тясно свързани с това информацията от помощния контекст да може да бъде копирана в случай на необходимост и за това се налага използването на методи, които да предоставят такава функционалност.

След като беше описано накратко какво прави съответния компонент, следва да бъде представен и неговия интерфейс.

```
/**
 * IDomContext.java - IMD - 15.04.2006
 */
package workflow.context.interfaces;


import java.util.List;

import org.w3c.dom.Element;

import workflow.exception.ContextException;

/**
 * IDomContext is used to hold all the information that is result of
 * executing the workflow.
 * <P>
 * The information that is holded are Element-s from the workflow that
 * were processed.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface IDomContext
{
	/**
	 * Gets namespaces.
	 *
	 * @return List If no namespaces exist it returns empty list.
	 */
	public List getNamespaces();

	/**
	 * Gets group names for the namespace given as parameter.
	 *
	 * @param namespace Namespace that group names are needed.
	 * @return List If no group names exist it returns empty list.
	 */
	public List getGroupNames(String namespace);

	/**
	 * Gets identifiers for the namespace and group name given as
	 * parameters.
	 *
	 * @param namespace Namespace. Not null.
	 * @param groupName Group name. Not null.
	 * @return List If no identifiers exist it returns empty list.
	 * */
	public List getIdentifiers(String namespace, String groupName);
```

```
/**
 * Resolves element in the context and provides it DOM
 * representation.
 *
 * @param elementRef Element reference.
 * @return Element Entry pointed by the element reference.
 * @throws ContextException If entry can not be resolved.
 */
public Element resolveEntryByRef(Element elementRef)
      throws ContextException;


/**
 * Resolves entry in the context and provides it DOM
 * representation.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @return Element possessing above characteristics. May be null.
 */
public Element resolveEntry(String entryNS,
                            String entryGN,
                            String entryID);

/**
 * Appends entry to the DOM context.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @param entry   Entry to append presented as binary object.
 * @return
 */
public void appendEntry(String  entryNS,
                        String  entryGN,
                        String  entryID,
                        Element entry);



/**
 * Removes entry from the DOM context.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @return Element possessing above characteristics. May be null.
 */
public Element removeEntry(String entryNS,
                           String entryGN,
                           String entryID);
```

```
        /**
         * Checks if entry is in DOM context.
         *
         * @param entryNS Entry namespace.
         * @param entryGN Entry group name.
         * @param entryID Entry identifier.
         * @return True if DOM context contains the entry.
         */
        public boolean containsEntry(String entryNS,
                                     String entryGN,
                                     String entryID);


        /**
         * Replaces entry in the DOM context with another one.
         *
         * @param entryNS Entry namespace.
         * @param entryGN Entry group name.
         * @param entryID Entry identifier.
         * @param entry   Entry to append presented as binary object.
         * @return Element that was replaced. May be null.
         */
        public Element replaceEntry(String entryNS,
                                    String entryGN,
                                    String entryID,
                                    Element entry);
}
```

5.2.3 Реализация на ITransactionContext.

ITransactionContext служи за съхраняване на допълнителна информация имаща връзка към транзакциите изпълнявани във workflow – а. Информацията може да бъде от най- различен тип и има отношение към данните, които ще бъдат модифицирани по време на транзакцията.

Ако трябва да бъдат обобщени накратко функциите, които се налага да извършва този компонент, то те са свързани най – основно с добавянето, изтриването, проверката и замяната на данни в допълнителния контекст на системата по време на изпълнението на workflow – а.

Изискванията на системата са тясно свързани с това информацията от допълнителния контекст да може да бъде да може да бъде копирана в случай на необходимост и за това се налага използването на методи, които да предоставят такава функционалност.

След като беше описано накратко какво прави съответния компонент, следва да бъде представен и неговия интерфейс.

```java
/**
 * ITransactionContext.java - IMD - 15.04.2006
 */
package workflow.context.interfaces;

import java.util.List;


/**
 * ITransactionContext is used to hold all the information that is
 * result of executing the workflow.
 * <P>
 * The information that is holded depends on those class who set it.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface ITransactionContext
{
	/**
	 * Resolves entry in context and provides it representation.
	 *
	 * @param entryNS Entry namespace.
	 * @param entryGN Entry group name.
	 * @param entryID Entry identifier.
	 * @return Object possessing above characteristics. May be null.
	 */
	public Object resolveEntry(String entryNS,
	                           String entryGN,
	                           String entryID);

	/**
	 * Appends entry to the transaction context.
	 *
	 * @param entryNS Entry namespace.
	 * @param entryGN Entry group name.
	 * @param entryID Entry identifier.
	 * @param entry   Entry to append presented as object.
	 * @return
	 */
	public void appendEntry(String entryNS,
	                        String entryGN,
	                        String entryID,
	                        Object entry);

	/**
	 * Removes entry from the transaction context.
	 *
	 * @param entryNS Entry namespace.
	 * @param entryGN Entry group name.
	 * @param entryID Entry identifier.
	 * @return Object possessing above characteristics. May be null.
	 */
	public Object removeEntry(String entryNS,
	                          String entryGN,
	                          String entryID);
```

```
/**
 * Checks if entry is in transaction context.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @return True if transaction context contains the entry.
 */
public boolean containsEntry(String entryNS,
                             String entryGN,
                             String entryID);


/**
 * Replaces entry in the transaction context with another one.
 *
 * @param entryNS Entry namespace.
 * @param entryGN Entry group name.
 * @param entryID Entry identifier.
 * @param entry   Entry to append presented as object.
 * @return Object that was replaced. May be null.
 */
public Object replaceEntry(String entryNS,
                           String entryGN,
                           String entryID,
                           Object entry);

/**
 * Clears context.
 * */
public void clear();

/**
 * Gets namespaces.
 *
 * @return List If no namespaces exist it returns empty list.
 */
public List getNamespaces();

/**
 * Gets group names for the namespace given as parameter.
 *
 * @param namespace Namespace that group names are needed.
 * @return List If no group names exist it returns empty list.
 */
public List getGroupNames(String namespace);
```

```
        /**
         * Gets identifiers for the namespace and group name given as
         * parameters.
         *
         * @param namespace Namespace.
         * @param groupName Group name.
         * @return List If no identifiers exist it returns empty list.
         * */
        public List getIdentifiers(String namespace, String groupName);

        /**
         * Merges the transaction context given as parameter with the
         * current one.
         * <P>
         * Merging usually means coping the content of the transaction
         * context given as parameter to the current one.
         *
         * @param otherTransactionContext Other transaction context.
         */
        public void merge(ITransactionContext otherTransactionContext);
}
```

5.2.4 Реализация на IConnectionContext.

IConnectionContext служи за съхраняване информация имаща връзка с конекциите осъществанявани към различни СУБД – тата. Тази информация може да се използва както за успешното приключване на дадена транзакция, така и за нейното отхвърляне.

Ако трябва да бъдат обобщени накратко функциите, които се налага да извършва този компонент, то те са свързани с извличането на конекциите, приключването на конекциите и отхвърлянето на конекциите.

Изискванията на системата са тясно свързани с това информацията от този контекст да може да бъде копирана в случай на необходимост и за това се налага използването на методи, които да предоставят такава функционалност.

След като беше описано накратко какво прави съответния компонент, следва да бъде представен и неговия интерфейс.

```
/**
 * IConnectionContext.java - IMD - 15.04.2006
 */
package workflow.context.interfaces;


import java.util.List;

import workflow.exception.ConnectionContextException;
```

```java
/**
 * IConnectionContext is used to hold all the information
 * about the connections.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface IConnectionContext
{
	/**
	 * Gets list with connections.
	 */
	public List getConnectionList();

	/**
	 *  Commits all connection.
	 *
	 *  @throws ConnectionContextException If some of the
	 *          connection(s) can not be commited.
	 */
	public void commitConnections()
		throws ConnectionContextException;

	/**
	 *  Rollback all connections.
	 */
	public void rollbackConnections();

	/**
	 * Merges the connection context given as parameter with the
	 * current one.
	 * <P>
	 * Merging usually means coping the connections from the
	 * transaction context given as parameter to the current one.
	 */
	public void merge(IConnectionContext otherConnectionContext);
}
```

5.2.5 Реализация на IDataSet.

IDataSet служи за съхраняване на данни, които ще бъдат обработвани от системата. Изискванията към начина за съхранение могат да бъдат доста строги по отношение на време за достъп и капацитет.

Ако трябва да бъдат обобщени накратко функциите, които се налага да извършва този компонент, то те са свързани с активирането или деактивирането на обекта, прочитането, актуализирането, добавянето или изтриването на запис от фрагмент на обекта и оптимизирането на вътрешните структури на фрагментите. Споменатите фрагменти представляват обособена част от цялата съвкупност от данни, където се съхранява полезната информация.

След като беше описано накратко какво прави съответния компонент, следва да бъде представен и неговия интерфейс.

```
/**
 * IDataSet.java - IMD - 06.01.2007
 */
package workflow.data.interfaces;


import java.util.Map;

import workflow.exception.DataSetException;
import workflow.data.metadata.interfaces.IDataSetMetaData;
import workflow.data.record.interfaces.IRecordSet;

/**
 * IDataSet provides interface to manipulate with data.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface IDataSet
{
	/**
	 * Activates fragment.
	 *
	 * @throws DataSetException If fragment can not be activated.
	 */
	public void activate()
		throws DataSetException;

	/**
	 * Passivates fragment.
	 *
	 * @throws DataSetException If fragment can not be passivated.
	 */
	public void passivate()
		throws DataSetException;

	/**
	 * Optimizes data set.
	 *
	 * @throws DataSetException If optimization can not be performed.
	 */
	public void optimize()
		throws DataSetException;

	/**
	 * Clears fragment.
	 *
	 * @throws DataSetException If fragment can not be cleared.
	 */
	public void clear()
		throws DataSetException;
```

```
/**
 * Gets meta data.
 *
 * @throws DataSetException If fragment set can not load the
 *                          record.
 */
public IDataSetMetaData getMetaData()
      throws DataSetException;

/**
 * Selects record.
 *
 * @param recordPosition The record position.
 * @throws DataSetException If fragment can not select
 *                          the record.
 */
public IRecordSet selectRecord(int recordPosition)
      throws DataSetException;

/**
 * Inserts record.
 *
 * @param record The record to insert.
 * @throws DataSetException If fragment can not insert
 *                          the record.
 */
public void insertRecord(IRecordSet record)
      throws DataSetException;

/**
 * Updates data.
 *
 * @param recordPosition The record position.
 * @param updates The record updates.
 *
 * @throws DataSetException If data set can not update
 *                          the record.
 */
public void updateRecord(int recordPosition, Map updates)
      throws DataSetException;

/**
 * Deletes record.
 *
 * @param recordPosition The record position.
 * @throws DataSetException If data set can not delete
 *                          the record.
 */
public void deleteRecord(int recordPosition)
      throws DataSetException;
```

```
	/**
	 * Provides fragment size.
	 *
	 * @throws DataSetException If data set size can not be
	 *                          calculated.
	 */
	public int size()
		throws DataSetException;
}
```

За да се изясни по – добре логиката на **IDataSet** ще бъде представен помощния компонент, който той използва - **IDataSetFragment**.

```
/**
 * IDataSetFragment.java - IMD - 06.01.2007
 */
package workflow.data.interfaces;


import java.util.Map;

import workflow.data.record.interfaces.IRecordSet;
import workflow.exception.DataSetFragmentException;

/**
 * IDataSetFragment provides interface to manipulate the data set
 * fragment.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface IDataSetFragment
{
	/**
	 * Activates fragment.
	 *
	 * @throws DataSetFragmentException If fragment can not be
	 *                                  activated.
	 */
	public void activate()
		throws DataSetFragmentException;

	/**
	 * Passivates fragment.
	 *
	 * @throws DataSetFragmentException If fragment can not be
	 *                                  assivated.
	 */
	public void passivate()
		throws DataSetFragmentException;
```

```
/**
 * Clears fragment. Fragment remains usable for future usage.
 *
 * @throws DataSetFragmentException If fragment can not be
 *                                  cleared.
 */
public void clear()
      throws DataSetFragmentException;

/**
 * Loads data.
 *
 * @param recordPosition The record position.
 * @throws DataSetFragmentException If fragment set can not load
 *                                  the record.
 */
public IRecordSet loadRecord(int recordPosition)
      throws DataSetFragmentException;

/**
 * Modifies data.
 *
 * @param recordPosition The record position.
 * @param updates The record updates.
 *
 * @throws DataSetFragmentException If fragment can not modify
 *                                  the record.
 */
public void modifyRecord(int recordPosition, Map updates)
      throws DataSetFragmentException;

/**
 * Stores record.
 *
 * @param record The record to store.
 * @throws DataSetFragmentException If fragment can not store
 *                                  the record.
 */
public void storeRecord(IRecordSet record)
      throws DataSetFragmentException;

/**
 * Removes record.
 *
 * @param recordPosition The record position.
 * @throws DataSetException If fragment can not remove the record.
 */
public void removeRecord(int recordPosition)
      throws DataSetFragmentException;
```

```
	/**
	 * Provides fragment size.
	 *
	 * @throws DataSetException If fragment's size can not be
	 *                          calculated.
	 */
	public int size()
		throws DataSetFragmentException;
}
```

5.2.6 Реализация на IEvaluator.

IEvaluator служи за изчисление на стойността на определени изрази, които следва да бъдат пресметнати в хода на изпълнението на workflow – а . За пресмятането на тези изрази може да се наложи да бъдат използвани контексти от където да бъде изтеглена нужната информация.

Ако трябва да бъдат обобщени накратко функциите, които се налага да извършва този компонент, то те са свързани единствено с пресмятането на даден израз като за целта може да се наложи да се използват различни класове.

След като беше описано накратко какво прави съответния компонент, следва да бъде представен и неговия интерфейс.

```
/**
 * IEvaluator.java - IMD - 31.12.2005
 */
package workflow.component.evaluator.interfaces;


import java.util.Map;

import workflow.exception.EvaluatorException;
import workflow.component.evaluator.cache.interfaces.IEvaluatorCache;

/**
 * IOperator provides interface for the operators.
 *
 * @author Iliyan Mihaylov Dimov
 * @version 1.0
 */
public interface IEvaluator
{
	/**
	 * Gets parameter map.
	 */
	public Map getParameterMap();
```

```
        /**
         * Sets parameter map.
         *
         * @param binaryContext The binary context.
         */
        public void setParameterMap(Map parameterMap);

        /**
         * Executes operator over the input list. Input list contains all
         * the items that should be tranformed to produce an Operand.
         * After all transformations are done the operator can start it's
         * evaluation.
         *
         * @param evaluatorCache The evaluator cache.
         * @return Object Contains the result value.
         * @throws EvaluatorException If operator can not evaluate
         *                            the result.
         */
        public Object evaluateResult(IEvaluatorCache evaluatorCache)
              throws EvaluatorException;
}
```

6. Тестване, оценка и усъвършенстване на системата.

Тестването представлява доста важен етап свързан с изграждането на една система. След като бъде проектирана и имплементирана системата, следващата стъпка в цялостното й изграждане се явява нейното тестване т.е. на този етап се проверява доколко изискванията заложени в дизайна на системата са спазени при тяхното имплементиране.

Целта на тестването се състои в това не само да се провери дали даден компонент функционира правилно сам за себе си, но и дали той може да си взаимодейства правилно и да функционира заедно с другите компоненти на системата.

Съществуват многобройни методологии, които могат да се използват за цялостното тестването на дадена система. Изборът на определени методологии, които да бъдат използвани за тестване зависят от естеството на самата система, нейното предназначение и нейната структура.

За да може да се планират дейностите свързани със самото тестване на системата е необходимо да бъдат зададени първоначално следните въпроси :

- какво да се тества ?
- какви предпоставки трябва да са изпълнени преди тестовете ?
- как да се тества ?
- кога да се тества ?
- как да се описват резултатите ?
- как да се обработват резултатите ?
- кой отговаря за тестовете ?

За да може да бъде получен отговор на всичките тези въпроси трябва да се вземат в предвид следните фактори :

- тип на решението
- архитектура на решението
- средата, в която ще работи решението
- среда за разработка
- предназначението на проекта
- опитът и качествата на разработчика / тестера
- функционалните и нефункционални характеристики на решението

6.1 Тестов план на системата.

Най - важният документ свързан с тестването на системата представлява тестовия план. Той може да се състои от няколко етапа на всеки от който да се тества определена функционалност на системата. Някоя от частите на системата може да се наложи да бъде тествана в повече етапи, като целта в този случай е да се проверят различните аспекти на нейното функциониране.

6.1.1 *Тестване на отделните компоненти изграждащи системата.*

Този тип на тестване е по - известен като *Unit Testing* и има за цел да тества отделните компоненти изграждащи системата. Крайната цел на тестването се състои в това да покаже, че отделните компоненти функционират изрядно. Тестването обикновено се извършва от самите програмисти и изисква детайлно познаване на дизайна на системата и нейния код. Най - често при него се използват дебъгери и инструменти за анализ, като целта е да се провери имплементацията на дизайна, да се отстранят грешки, а също така да се изследва поведението на дадена част от кода при различни входни комбинации. Съществото при това тестване е да се предвидят сценариите свързани с изследване на интерфейсите, както и манипулациите с данни. Добра практика е да се документират тестовите данни и сценариите, които се изпълняват, а също така да се посочат и очакваните резултати. Предимствата на *Unit Testing* са :

- *улеснява тестването* – позволява на програмистите да правят рефакторинг на кода и все още да са убедени, че той функционира правилно. Изисква писането на тестов случай за всяка една функция или метод така, че винаги когато дадена повторна модификация на метод води до грешка, тя може лесно да бъде идентифицирана, локализирана и отстранена. Наличните вече тестове позволяват на програмиста да провери дали дадена част от кода работи правилно и да насочи вниманието се към друга представляваща потенциален източник на проблема. Добре проектираните тестови случаи дават възможност да се тестват всички разклонения на метода за да се гарантира, че той работи правилно за всички случаи.

- *улеснява интеграцията* – позволява да се избегне несигурността в самите компоненти, което от своя страна може да бъде използвано от **bottom – up** стилът на тестване. Като се тестват предварително отделните компоненти на системата, а след това сумата на тези работещи компоненти заедно, интеграционното тестване като цяло става доста по – лесно и ефективно.

- *осигурява документираност* – позволява да се осигури някаква форма на документираност на системата. Потребителят и останалите разработчици, имащи нужда да разберат начинът на работа на даден компонент, могат да погледнат тестовите случаи, за да определят до каква степен този компонент отговаря на техните нужди. По този начин може да се осигури и по - ясна представа за самото API на системата. Тестовите случаи въплъщават в себе

си характеристики, които са критични за успеха на самия компонент. Тези характеристики могат да изискват тестване свързано както правилното, така и с неправилното използване на компонента. Тестовия случай може да документира в себе си, както и извън себе си тези критични характеристики въпреки, че много работни среди не разчитат изцяло на кода за документиране на продукта, който се разработва.

Недостатъците на *Unit Testing* са свързани с това, че той няма възможност да улови всички проблеми, които могат да възникнат в системата. Това се дължи на факта, че по предназначение неговата цел е да тества само и единствено функционалността на отделните компоненти от системата. Оттук следва, че много от проблемите свързани с интеграцията на компонентите и производителността на системата няма да бъдат забелязани и отстранени. Като извод може да се каже, че *Unit Testing* може да бъде ефективен само в случаите, когато се използва съвместно с другите методи на тестване.

Нереално е също така да се смята, че може да се тества цялото многообразие от функционалност на едно нетривиално парче код. Като всички останали форми на софтуерно тестване, *Unit Testing* може единствено да докаже наличието на грешки т.е. той не може да гарантира отсъствието на такива. За да се постигнат желаните резултати от *Unit Testing* се налага спазването на известна доза дисциплина в процеса на разработка. Важно е да се знаят не само тестовете, които са били направени върху даден компонент от системата, но и да се записват всички негови промени, както и промените на другите компоненти в системата. Използването на система за контрол на версиите (СКВ) е съществена – ако съществуваща версия на компонента не може да премине определени тестови случаи, които преди това е удовлетворявала, то можем да използваме СКВ и да свалим актуалната версия на компонента, която вече се налага да удовлетворява всички тестови случаи. След това тестването на компонента може да продължи по обичайният за това начин.

6.1.2 *Тестване на степента на интеграция на компонентите изграждащи системата.*

Този етап на тестване е по - известен като *Integration Testing* и има за цел да тества взаимодействието между компонентите, които ще бъдат интегрирани в системата. Извършва се след като е приключил *Unit Testing* етапа и преди да започне *System Testing* етапа.

Интеграционното тестване приема за вход отделни компоненти, които са преминали *Unit Testing*, групира ги в по – големи модули, прилага върху тях тестовете дефинирани в интеграционния план и връща като резултат интегрирана система, готова да бъде подложена на *System testing*.

Целта на интеграционното тестване е да верифицира изискванията за функционалност, производителност и надежност върху една по – голяма група

"компоненти на дизайна" наречена модул. След като бъдат сформирани модулите, те могат да бъдат тествани посредством интерфейса, който предлагат използвайки *black box testing* и набор от тестови случаи, симулиращи коректни входни параметри и данни. Целите на симулацията са да се използват както споделени области от данни така и вътрешно – процесни комуникации, за да бъдат проверени отделните подсистеми посредством техния входящ интерфейс. Тестовите случаи имат за цел да тестват дали всички компоненти в рамките на модула работят изправно, като за тази цел може да се викат определени процедури и активират опрделени процеси.

Цялостната идея на интеграционното тестване е сформирането на някаква основа, която състои от верифицирани модули и се използва при тестване на останалите модули. Съществуват два похода за извършване на интеграционното тестване : *big bang* и *bottom-up*.

Big Bang : При този подход всички или по – голяма част от разработваните модули се групират заедно, за да сформират завършена система или съществена част от системата и след това се използват за основа на интеграционното тестване. *Big Bang* методът е много ефективен за спестяване на време при процесът на интеграционно тестване. Разбира се, ако тестовите случаи и техните резултати не се използват и оценени правилно, то целият интеграционен процес ще бъде усложнен и може да попречи на тестващия екип да постигне целите на интеграционното тестване.

Bottom Up : При този подход всички независими или от ниско ниво модули, процедури и функции се интегрират и след това се тестват. След успешното интеграционното тестване на модулите от ниско ниво, се взимат модулите от следващото ниво и интеграционното тестване се извършва отново. Интеграционното тестване се повтаря докато се достигнат модулите от най – високото йерархично ниво. Този подход е успешен само когато всички или по – голяма част от модулите от същото ниво на разработка са готови едновременно. Този подход също така помага да се определят нивата на софуерно разработване и прави по – лесно описанието на тестовия процес.
След като бяха разгледани различните подходи интеграционно тестване следва да бъде обяснено как те могат да бъдат приложени към различните компоненти на системата. Така например *Bottom Up* следва да се извърши по следния начин.

Модул на системата	**Компоненти изграждащи модула**
Модул I	*transaction, statement*
Модул II	*var, setvar, binaryvar, setbinaryvar, array , setarrayelement, addarrayelement, removearrayelement, map, addmapentry, removemapentry*
Модул III	*if, foreach, while, procedure, procedureCall*
Модул IV	*viewvar, view, viewCall, compositeview, compositeviewCall, insertview, insertviewCall, updateview, updateviewCall, deleteview, deleteviewCall*

Модул V	*dbinsertview, dbinsertviewCall, dbupdateview, dbupdateviewCall, dbdeleteview, dbdeleteviewCall*
Модул VI	*xmlvar, xmlvarload, xmlvarstore, xmlinsertview, xmlinsertviewCall, xmlupdateview, xmlupdateviewCall, xmldeleteview, xmldeleteviewCall*

6.1.3 *Цялостно тестване на системата.*

Този тип на тестване е по - известен като *System Testing* и има за основна цел да тества цялостното поведение на системата. Освен да открива несъответствия между системата и специфичните изисквания предявявани към нея, той има за цел да открива дефекти във вътрешните взаимовръзки на системата.

Съществуват много различни типове системно тестване, по – важните от които са :

- *Usability testing* – дава възможност да се тества каква част от компонентите на системата могат на практика да бъдат използвани от потребителя т.е. до колко даден потребител разбира правилно предназначението на всеки един компонент и случаите в които маже да го използва. На практика всеки един компонент на системата, който се ползва пряко от потребителите трябва да бъде тестван.

- *Performance testing* – дава възможност да се тества производителността на всеки един компонент от системата. С този вид тестване се дава възможност да се покрие широк спектър от случаи, които имат за цел да докажат практически способността на даден компонент да изпълнява ефективно свойте функции. За измерването на производителността обикновено се използват определени показатели, които трябва да са в съответните норми за да бъде тестът успешен.

- *Compatibility testing* – дава възможност да се тества до колко системата е съвместима със средата в която работи. Средата може да съдържа някои или всичките изброени елементи : изчислителна мощ на хардуерната платформа, честотна лента на мрежовите устройства, периферни устройства, операционни системи, системи за управление на бази данни, обратна съвместимост и други.

- *Error handling testing* – дава възможност да се тества до колко системата е способна да се справи с грешките, които могат да възникнат по време на нейната нормална работа и не могат да бъдат избегнати от самото начало. При това тестване целта е да се провери дали системата може да се справи по предвидения за това начин и да обработи изключителната ситуация.

- *Load testing* – дава възможност да се тества до колко системата е способна да издържа на натоварвания. В рамките на системата натоварванията могат да бъдат свързани или с голям обем от данни, които се налага да бъдат

обработени или с голям брой брой потребители, които се налага да бъдат обслужени по едно и също време. Извършването на този вид тествате позволява да се определи пределното натоварване на което може да издържи системата, а от там да се открие и обичайното натоварване за което е предназначена да работи.

- *Security testing* – дава възможност да се тества до колко системата е способна да защитава данните и да предоставя желаната функционалност. Базовите концепции свързани със сигурността са : конфиденциалност, интегритет, аутентикация, ауторизация, достъпност и неоспоримост. Под конфиденциалност се разбира предоставяне на информацията само на желан кръг от потребители. При интегритет се гарантира еквивалентността на предадената и получената информация. При аутентикация се гарантира валидността на потребителя изпращаш информацията. При ауторизация се гарантира правото на потребителя да изпраща информация. Под достъпност се разбира възможността информацията да бъде достъпна до имащите право потребители. Под неоспоримост се разбира невъзможността да се оспори източника на информация.

- *Regression testing* – дава възможност да се тества каква част от компонентите на системата продължават да функционират изрядно след като са били модифицирани. Модификацията на даден компонент може да се дължи на добавянето на нова или модифицирането на съществуваща функционалност. На практика този вид тестване трябва да се приложи над всеки един компонент, който е бил модифициран по някакъм начин с цел да се провери дали той не оказва някакво влияние върху останалата част на системата.

- *Reliability testing* – дава възможност да се определи до колко системата е надеждна и способна да изпълнява своите функции. Между надеждността на системата и надеждността на отделните компоненти съществува правопропорционална статистическа зависимост.

След като бяха разгледани различните типове системно тестване следва да бъде обяснено как те могат да бъдат приложени към различните компоненти на системата. За някои от компонентите може да се изисква повече от един тип системно тестване.

Компонент за тестване	**Приложими типове системно тестване**					
	Usability testing	*Performance testing*	*Compatibility testing*	*Error handling testing*	*Load testing*	*Reliability testing*
transaction	✓	✓	×	✓	✓	✓
statement	✓	✓	×	×	×	✓
if	✓	×	×	×	×	✓
var	✓	×	×	×	×	✓
setvar	✓	×	×	×	×	✓
viewvar	✓	×	×	×	✓	✓
binaryvar	✓	×	×	×	✓	✓
setbinaryvar	✓	×	×	×	×	✓
array	✓	×	×	×	✓	✓
setarrayelement	✓	✓	×	×	×	✓
addarrayelement	✓	✓	×	×	✓	✓
removearrayelement	✓	✓	×	×	×	✓
map	✓	×	×	×	✓	✓
addmapentry	✓	✓	×	×	✓	✓
removemapentry	✓	✓	×	×	×	✓
foreach	✓	✓	×	✓	×	✓
while	✓	✓	×	✓	×	✓
procedure	✓	✓	×	✓	×	✓
procedureCall	✓	✓	×	✓	×	✓
provider	✓	×	✓	×	×	×
view	✓	✓	×	✓	✓	✓
viewCall	✓	✓	×	✓	✓	✓
compositeview	✓	×	×	×	✓	✓
compositeviewCall	✓	✓	×	✓	✓	✓
insertview	✓	✓	×	✓	✓	✓
insertviewCall	✓	✓	×	✓	✓	✓
updateview	✓	✓	×	✓	✓	✓
updateviewCall	✓	✓	×	✓	✓	✓
deleteview	✓	✓	×	✓	✓	✓
deleteviewCall	✓	✓	×	✓	✓	✓
dbinsertview	✓	✓	✓	✓	✓	✓
dbinsertviewCall	✓	✓	✓	✓	✓	✓
dbupdateview	✓	✓	✓	✓	✓	✓
dbupdateviewCall	✓	✓	✓	✓	✓	✓

dbdeleteview	✓	✓	✓	✓	✓	✓
dbdeleteviewCall	✓	✓	✓	✓	✓	✓
xmlvar	✓	✓	×	✓	✓	✓
xmlvarload	✓	✓	×	✓	✓	✓
xmlvarstore	✓	✓	×	✓	✓	✓
xmlinsertview	✓	✓	✓	✓	✓	✓
xmlinsertviewCall	✓	✓	✓	✓	✓	✓
xmlupdateview	✓	✓	✓	✓	✓	✓
xmlupdateviewCall	✓	✓	✓	✓	✓	✓
xmldeleteview	✓	✓	✓	✓	✓	✓
xmldeleteviewCall	✓	✓	✓	✓	✓	✓

6.2 Оценка на системата.

Цялостна оценка на системата може да бъде направена след като бъдат извършени всички тестове и отчетени съответните им показатели. След анализ на получените показатели може да се определи до каква степен системата отговаря на поставените изисквания и има ли нужда системата да бъде доработвана. Анализът на всеки един показател се извършва като се сравни дали неговата стойност получена в резултат на тестването принадлежи на интервала от допустими стойнсти, които той може да заема. В случай, че даден показател не може да бъде измерен количествено, а качествено, то тогава за да се определи дали той отговаря на определени критерии се изисква субективна преценка.

Оценката на различните показатели на системата може да се извършва от различни хора в зависимост от това колко комплексна може да е тя. Това важи особено много за качествените показатели пример за какъвто може да бъде *степен на обратна съвместимост*. Хората, които се допускат до оценката на такива показатели са екипи от разработчици или самият потребител на системата. За оценката на някои от показателите може да се използват предварително съгласувани общоприети или фирмени стандарти.

Освен оценка на системата като цяло, може да се направи оценка и на отделните модули или етапи свързани с разработването на системата. В резултат на тази оценка може да се наложи даден модул да бъде променен или цялата система да бъде преработена.

Предпоставките свързани с позитивното оценяване на дадена система са :

- строго дефинирани изисквания – ясни, напълно описващи обхвата на системата, в нужната детайлност, които могат да се проверят с тестове, които са приети и от потребителя и от разработчика.
- реалистични планове и графици – за целта се провеждат анализи, данните от тези анализи служат за изграждане на знания за оценка и планиране в следващите разработки.
- адекватно планиране на дейностите за контрол и тестване – във времето и като задачи.
- регламентирана комуникация с потребителя (срещи, разговори, периодични отчети – седмични, месечни, на ниво фази; чрез използване на различни средства - софтуерни системи за управление на промените, за управление на регистрираните грешки, за проследяване статуса на системата).

Предпоставките свързани с негативното оценяване на дадена система са :

- изисквания, които са много общи, неясни или непълни, с неточно дефинирани функционални и нефункционални критерии
- нереалистично планиране на системата
- неадекватно планиране на тестването

- недооценени изисквания за промени в хода на разработка или приемане на системата
- недобра комуникация между хората от екипа, между екипа и ръководството на фирма, между фирмата и клиента или между екипа и клиента

Изискванията, които трябва да бъдат взети в предвид при оценката на дадена система са :

- коректност по отношение на функционалните и нефункционални изисквания към системата
- коректност по отношение на приетите за системата стандарти
- надеждност
- ефективност и бързодействие
- удобство за експлоатация и поддръжка
- гъвкавост
- изисквания за конфигуриране
- устойчивост при невалидни входни данни

Допълнителните индиректни условия на които трябва да отговаря всяка една положително оценена система са :

- да приключи в рамките на договореното време
- да приключи в рамките на договорените ресурси при оптимално съотношение на показателя (пълнота на решението / качество / цена)
- да покрива договорените изисквания и качество
- да удовлетворява договорените очаквания на потребителя

6.3 Бъдещо развитие и усъвършенстване.

Всяка една система може да бъде развита и усъвършенствана, като за тази цел се налага да бъдат направени известни подобрения по нея. Те могат да бъдат от най - различно естество, което зависи строго от същността на целите, които преследват.

Някои от добрите перспективите за бъдещо развитие на системата "**Генератор на справки и SQL процесор**" са :

- изграждане на интегрирана среда за разработка
- добавяне на нови компоненти за работа с уеб услуги
- добавяне на нови компоненти за работа с база данни
- добавяне на нови компоненти за повишаване на сигурността
- добавяне на поддръжка за нови СУБД
- усъвършенстване на наличните компоненти

Всяко едно от тези възможни подобрения ще бъде описано в самостоятелна секция имаща за цел да поясни предназначението му.

6.3.1 *Изграждане на интегрирана среда на разработка.*

Всяка една система може да се състои от много на брой компоненти. Всеки от тези компоненти може да е строго специфичен сам по себе си и да съдържа много допълнителни детайли. В резултат на това се получава, че колкото повече нараства броят на компонентите, толкова по - сложно става те да бъдат използвани пълноценно и ефективно управлявани. С цел да се улеснят потребителите на системата е желателно да се създаде интегрирана среда за разработка, която да позволява на потребителите да избират компонентите, които желаят да използват и да позволява те да бъдат конфигурирани посредством потребителски интерфейс.

Както вече споменахме всеки един от компонентите представлява xml таг, който може от своя страна да съдържа други подтагове или атрибути. За да не се налага на потребителя да помни цялостната структура на тага и тази на подтаговете му, е необходимо те да бъдат създавани посредством използването на някакъв потребителски интерфейс – форма или съветник. Потребителският интерфейс би позволил на потребителя да се съсредоточи върху семантиката на тага т.е. какво би искал да свърши с този таг, вместо да се старае да запомни неговата структура.

Друго предимство на потребителския интерфейс е, че през него биха могли да бъдат тествани някои по – малки и независими компоненти като например конекторите. По този начин се улеснява цялостното тестване на workflow – а.

6.3.2 *Добавяне на нови компоненти за работа с уеб услуги.*

Уеб услугите са много популярни в днешно време поради факта, че позволяват на потребителите да получават относително лесно и бързо актуална информация. Във връзка с това броят на потребителите, който ги използва също се увеличава, което от своя страна води до необходимостта да се създадат подходящи средства във workflow-а за работа с тях.

Освен стандартните за една услуга характеристики, уеб услугата може да предоставя и защита на данните. За тази цел трябва да се използва цифрово подписване и криптиране на данните, което от своя страна гарантира тяхното надеждно и сигурно предаване.

Във връзка с нарастването на потенциалния брой потребители ползващи уеб услуги е желателно следващите версии на системата да съдържат необходимият набор от компоненти и средства за работа с тях.

6.3.3 *Добавяне на нови компоненти за работа с база данни.*

Характерна особеност на наличните до момента средства в системата за работа със СУБД – та е тяхната директна същност свързана с обработката на данните. Тази същност се състои в извличането на данните, модифицирането на данните, добавянето на данните или изтриването на данните. Освен обработка на самите данни, може да се изисква и модифицирането на схемата в която те се съхраняват. Пример за модифицирането на схема е добавянето на нова таблица, изтриването на съществуваща таблица и така нататък. От казаното до тук може да се направи извода, че са необходими компоненти, които да имплементират функционалността предоставяна от Data Definition Language, явяващ се част SQL езика.

6.3.4 *Добавяне на нови компоненти за повишаване на сигурността.*

Сигурността на данните понякога може да бъде от изключителна важност. Поради тази причина реалната работна среда в която функционира системата може да изисква от последната данните получени в резултат на някаква обработката да бъдат трансферирани надежно и сигурно до дадено място. За тази цел е необходимо да се разработят нужните компоненти от системата, които да осигуряват надежден, защитен и сигурен транспорт изискван за трансферирането на данните.

6.3.5 *Добавяне на поддръжка за нови СУБД.*

Системата може да се разшири да поддържа връзка с други СУБД – та, които потребителят прецени, че има нужда да ползва.

6.3.6 *Усъвършенстване на наличните компоненти.*

В следствие на хода на разработка и тестване може да се установи, че дизайнът или реализацията на някои от компонентите и модулите на системата може да бъде подобрен. В този случай е удачно преди да се премине към усъвършенстване на системата да се отстранят известните и добре познати проблеми, стига това да не изисква кардинални промени в цялостната архитектура.

7. Заключение.

В днешния съвременен и динамичет свят, където данните нарастват с експоненциална скорост възниква все повече и повече необходимостта от това те да бъдат обработвани и систематизирани под някаква форма. Идеален вариант за това се явява използването на справки, които извличат само дадена част от данните представляваща интерес за нас и ни позволяват да се концентрираме върху нея. Още по – полезна се явява възможността да извлечем сами тези данни от различни източници и да укажем как точно те да бъдат обработени и систематизирани така, че резултатът да има смисъл за нас и да удовлетвори напълно нашите потребности.

Компонентният подход използван при разработката на системата позволява нейното лесно разширение чрез добавянето на нови компоненти. По този начин, за нас остава единствено задачата да надградим системата така, че тя да позволява викането на уеб услуги, да осигурява определено ниво на сигурност и да запази своята разширяемост и за в бъдеще.

Използването на съвременни методологии за тестване позволява да се осигури и потвърди голямата гъвкавост на системата, нейната надежност и работоспособност при големи натоварния и не на последно място забележителната й производителност.

8. Използвани материали.

- ***Mastering SQL*** (*Michael Gruber*)
- ***SQL : Practical guide for developers*** (*Michael Donahoo*)
- ***Algorithms in Java*** (*Robert Sedgewick*)
- ***Professional XML Databases*** (*Michael Brundage*)
- ***Processing XML with Java : A Guide to SAX, DOM, JDOM, JAXP, and TrAX*** (*Elliotte Rusty Harold*)
- ***Xml Lectures*** (*Boyan Bonchev*)
- ***Quality Assurance Lectures*** (*Maria Ilieva*)
- ***http://en.wikipedia.org/wiki/Main_Page***