gTitle

Abstract:

1. Challenges - Problem statement

need of data granulation

workspaces

2. Object Model

Настоящият параграф има за цел да представи модел на версионизиран обект. Моделът предоставя възможност за гъвкаво комбиниране между версионизирани обекти, като се изграждат композиции от обекти. Построяването и представянето на модел на версионизиран обект е реализирано като класически модел същност-отношение (Entity-Relationship Model). Той ни предоставя добър механизъм за гъвкаво и свободно реализиране на поставените задачи.

Lead author in version control domain [29, 85] determine the version object as two part entity - object states, and object versions' graph. Object version graph is a graph where the nodes represents object states, and arcs represents version transitions.

Main feature that an versioned object suppose to implement is to provide possibility to define data granulation in a free way. This feature suppose to be supported by the model of versioned object. This lease us to the need of defining of object compositions as part of the model. The formal definition of the composed object could be defined as follow:

1. Composed object is an object that is built from other object using composition entity. Съставен обект се нарича обект, който е съставен от други обекти (версии на обекти) посредством композиция.
2. As composition we will comprise the entity that defines the relation between super-object and sub-object. One composed object could be super-object for one or more composition instances, i.e. to be built by one or more sub-objects.

Adding of the composed objects and sub-objects in the domain leads to the need of redefinition of versioning process over versioned objects. On the following diagram is presented an example for change of version composition. Here arrows represent changes in objects’ content among different versions of end product.



Фиг. 26 An example for change in objects’ content

On versions binding of certain versioned object it is necessary to use relation of type foreign key. From this pursue that **versioned object entity** has to consist only of unique and immutable number. It is useful to regard that number as primary key for the entity.

Object’s versions could be разглеждат as its primitives (**versioned primitives**) with the following attributes:

* Versioned object id
* Version number – a serial number which specify in unique way the version within the versioned object.
* Object’s name. Setting the name to be on primitive level allows to the user to track among different versions even then the object is renamed. This model becomes more complete, excluding the weakness related with object (file) renaming in systems as CVS, SVN, Git, Mercury, and etc. [28, 41, 49, 61, 64, 75].
* Object content.

Versioned primitive is determined in unique way using the couple **versioned object id** and **version number**. In spite of the possibility to use that unique pair and compound key, good practice in ER model design [11] recommends that each entity to possess its own not compound key. In our case we will introduce an additional field as a primary key – global version number.

Versioning of composed object requires to define an additional entity – Versioned primitive composition (in short Composition). This entity is a relation entity that bind in a unique way version of a super-object with version of a sub-object. The following attributes of composition could be defined:

* Global number of super-object
* Global number of sub-object.

Despite of the fact that attribute combination is always unique, we will use additional attribute for primary key – composition id.

Due to accounting and change traceability needs, our model will be extended in order to support th feature of versioned graph. In ER models graph structures could be presented using two entities – entity for nodes and entity for arcs [86]. Looking to versioned object definition we could conclude that graph nodes correspond to versioned primitive entity. The missing part will be implemented as new entity which will correspond to graph arcs – arc of version graph. The new entity requires the following attributes:

* Arc id – primary key.
* Global number of source version.
* Global number of target version.
* User that made the change.
* Date and time of change.
* Additional change data.

On diagram ### is presented the ER model of versioned object.



Фиг. 27 ER model of versioned object

### Versioning of composed object

In current paragraph we will represent the peculiarities in versioning of composed objects with rank one. Based on that, we will specify the versioning process of composed objects with rank N. The following definition specifies the term rank of composed object.

1. Composed object with rank zero, i.e. simple object is an object which doesn’t have associated with itself sub-objects. Composed object with rank N is an object for which the largest rank of associated sub-object is equal to N-1.

Granularity degree of an object is its rank.

It is important to note that definition of composed object doesn't apply any restrictions on sub-objects. This leads to the following conclusion:

1. One sub-object itself can be part of composed object therefore we can build a composition of composed objects.

One of core tasks ahead of current article is to avoid unnecessary complication of the models. Having that as well as missing necessity, we can specify the following restriction rules during building of composed objects:

1. In given composition of composed objects, certain object can be at most once.
2. One object can be part of at most one object composition.



Фиг. 28 A tree of objects

In change of composition between two objects, we should regard the objects’ versions as different (29). Let examine a chair (super-object) with armrests (sub-objects). When we remove the armrests from a chair we get a new version of the chair - a chair without armrests. We have to underline that sub-objects don't change its version. Therefore we get only change in compositions of super-object. We have similar situation in building of super-object, i.e. when we have a simple object that is transformed to composed object. When we add armrests to a chair, we get new version of that chair, without changing the version of armrests.



Фиг. 29 Change in objects' composition through change of version

Another feature of composed objects is the case of sub-object change we get an indirect change of the composed object (30). Let look at the example: Let we have change of chair's upholstery from blue to red. In this case not only the version of the upholstery is changed but also the version of the chair. As a special cases of sub-object change could be regarded association of an object as a sub-object and removing of association with sub-object and its transformation to simple object.



Фиг. 30 Indirect change of version of composed object in case of its sub-object change

The opposite case - when we have a change in super-object - this doesn't means that version of its sub-objects is changed. So if you have a chair with three legs and red upholstery, the addition of the fourth leg of the chair doesn't change version of red upholstery's sub-object. (31)



Фиг. 31 Change in super-object doesn't affect the version of sub-object

The last two rules can be derived consequence:

1. Changing the version of a sub-object for a super-obejet doesn't affect the versions of its sibling sub-objects (32).

The composed object model and visibility principles (from previous section) lead us to the need to address the problem of composed object visibility.



Фиг. 32 Change in certain sub-object doesn't affect version of its siblings

1. Certain composed object version is visible in certain workspace only and solely when all its sub-objects are visible in that workspace.

3. Model of Hierarchical composed workspaces

First model that we are going to examine is the model of hierarchical composed workspaces. Within the model the following definitions will be used:

1. Product is called the subject of material or immaterial manufacturing, which since its creation can be reproduced and distributed to customers.
2. Product release is called a fixed version that has passed certain quantities controls and meets the criteria of quality, safety and security. Only product releases are distributed to customers. Versions that are not release, called the practice working versions.
3. Workspace is a location where certain activities are carried out on the development of a version of a product.
4. Main workspace called workspace, which made ​​the final preparation of equipment and product release.

Arrangement (composition) of the workspace is taken to provide opportunity for each participant in product development process and its releases to carry out its activities both individually and cooperating with other participants. Namely workspace provides the opportunity for independent work that does not affect or be affected by the work of other participants. On the other hand, composing workspaces in a hierarchy claim to be a mechanism to ensure cooperative work. Diagram (34) presents a diagram of hierarchical composition of workspaces.



Фиг. 33 Class diagram of Product-Release-Workspace model

### Versioned object visibility model in environment with hierarchical composition of workspaces

As in any hierarchical structure, and here will be considered a parent-child relations. We will focus on the versioned object visibility defined by the following, principles of visibility.



Фиг. 34 Example of hierarchical composition of workspaces

1. *Local version of the versioned object* is the version that is associated with a specific workspace.
2. *Visible version* *of versioned object* for given workspace means a version of that object with which the user can work.

Visibility principles:

1. *The local version of the versioned object for given* workspace is the visible version of that object in the same workspace, despite other local versions in the parent workspaces.
2. The local version of object in a workspace can be seen recursively in all subspaces, unless another local version defined therein.

From above principles we can deduce consequences:

1. In any workspace, in which objects don’t have a local version, they are presented with their version found in the nearest parent workspace
2. If for given workspace the object has no version in either parent workspace, it is not visible in the selected workspace.

On (35) we present how both principles of visibility influences on object version distributions for example hierarchical composition of workspaces.



Фиг. 35 Distribution of object versions according to the visibility principles

In order to achieve completeness and correctness of the model can be formulated as following constraint: an object can present only one version in a workspace.

## Transactions over versioned objects

We introduced transactions over versioned objects in [Pitesti]. Here we will make a short presentation and classification of them.

### Transaction within a single workspace

Creation is the first transaction for each versioned object. After completion of the transaction, the object has an initial (zero) version, in which it is "empty", i.e. contains no information.

The transaction of state marker creation which we create a new version of a versioned object can be regarded as the basis of a mechanism for creating safe-points.

As a reverse transaction to create a state may be classified for this waiver of marker status. Through it the model was released last state and current local version of the object is the version prior refusal.

The creation of long sequences of not branched versions, especially from the same user within the same workspace, leads us to the idea of the transaction of combining successive versions, in order to save memory and follow faster and more easier analysis of the work.

Updating versioned nonlocal object, i.e. an object that doesn’t have any local version in the current workspace can be defined as the most important of the current section. This transaction is not entirely limited only to a single workspace, since it is composed of the following steps:

* Retrieve the previous visible for workspace version of the site.
* Create a local version of the object in the current workspace.
* Създаване на релация на версиите (дъга в графа на версиите), в която предишната видима версия се явява версия-първоизточник за новата локална версия на обекта.

Deleting an object is possible by creating a transaction called “tag for deleted object”. This tag is intended to "hide" the object in its workspace and make it invisible in its sub-workspaces. It should be noted that all described in this section transactions over the object can no longer available, except for transaction rejection of tag status.

### Transaction among two workspaces

Transactions between two workspaces can be divided into two groups - the publication of object’s version and giving up a local version. Before examining, it is necessary to introduce the terms "derivative" and "parallel" version of an object.

1. Let's look at one versioned object and two versions of its X and Y. If there is a road the versioned graph of the object from version X to version Y, then Y version is called a derivative version of X, and version X is called previous version of Y.
2. Let's look at one versioned object and two versions of its X and Y. If there is no road the versioned graph of the object from version X to version Y, then both versions are called *parallel* versions or *not-derived* versions.

Publishing an object’s version means the transformation the local version of the object from the current workspace into a local version of its parent workspace.

The simple version publishing is in a situation where there is no local version of a published object in the parent workspace - 36.



Фиг. 36 Simple publishing

Following the transaction, which needs to be addressed is that of updating publication (Фиг. 37). Typical for it is that it is possible when simultaneously satisfy two conditions:

* In the parent workspace exists a local version of the object be published.
* The version of the object be published, is a derivative of the version in the parent workspace.

When updating publishing the merge of two versions is not needed, as a derivative version of evolution up to the previous version.



Фиг. 37 Updating publication

When the object version to be published in the parent workspace is a parallel to the version in paret workspace (Фиг. 38), then it should to merge both versions. As a result of the merger produce a new version of the object. We don’t aim to present a new method for merging versions of an object, so we can be used as handheld merge approach or algorithmic approach similar to the algorithm Вестфехтел [101].



Фиг. 38 Publication with merge

Transaction give up of local version is the reverse transaction by publishing a version. The only step of in transaction is the remove the local version of the object on the workspace. It is important to note that if any of the parent workspaces there is no version of the selected object, then the object it becomes unavailable for subsequent use. This situation should be taken into account when the transaction takes place in the main workspace of the product's release.

### Transactions over composed objects

… transaction from ….

Let have the following situation: a local version of the object **B** in the parent workspace and its visible version in current workspace. Let make in current workspace a sub-objects in **A** of object **B**. (Фиг. 39). After publication of version of **A** sub-object may not lead to a change in the version of object **B** in the parent workspace. However, in a subsequent publication version of the object **B** in conjunction with his compositions in the parent workspace will lead to automatic update of the compositional scheme of objects (on 39 with green dotted arrow). This is dictated by the fact that information regarding the organization of the composite object should be considered as an inseparable part of it.



Фиг. 39 Newly created sub-object to a super-object

Publication of the new version of the composite object B, v3 leads to the requirement that this be done in a set with the version of the newly created sub-object (Фиг. 39 – arrows with number 2).

Let us consider the situation where we have a local version of the object in the parent workspace that is visible in the current workspace (40). In current workspace we change the sub-object A, which leads to a change of object B, i. e. the creation of a new local version of the sub-objects leads to the automatic creation of a new local version of the entire composite object. We should notice that separate publishing the new version of sub-objects in the parent workspace should not be allowed. This restriction follows from the fact that a new version of sub-object assumes a new version of super-object (40 – red arrow with number 1), in addition we introduced the restriction that an object may be present only one version a workspace. As a conclusion to the situation we can define the following rule:

1. Publication of a version of local composite object should be made bundled with all local versions of its sub-objects that have a different version in the parent workspace  
   ( – the green and yellow arrows with number 2).

Local versions of the sub-objects in the parent workspace can be either derivatives as well as parallel. In these cases it is necessary to execute the transactions that are covered above.

Let us examine same situation when we have a local version of the object **B** in the parent workspace that is visible in the current workspace (41). From the composition of the composed object is excluded a sub-object. Reflecting this change in the parent workspace is achievable only by publishing the composed object. This publication only transfers the composition change in parent workspace without changing the version of sub-object. The former sub-object is no longer part of the new super-object version.



Фиг. 40 Indirect change inversion of super-object, caused by new version of sub-object



Фиг. 41 The absence of change in version of object **A**, i.e. there is no need of its publication

Let us have visible composed object **A** with sub-object **B**, object **A** and sub-object **B** are local versions in the parent space. We remove from the composition of object **A** its sub-object **B**, i.e. we create a new local version of object A (42). In this case a publication of any new version of object **B** would lead to the following conflict: version **v1** of **B** requires that in its workspace the object **A** to be with its version **v1** (visible or local version).

This fact could be regarded as prerequisite for the following rule:

1. Publication of object’s version, that have previous version
2. Публикуването на версия на обект, който притежава предишна версия, явяваща се подобект на съставен обект в родителското работно пространство на текущото работно пространство, следва да се извършва едновременно с публикуването на локалната версия на съответния съставен обект.



Фиг. 42 Публикуване на бивш подобект не е възможно, преди публикуване новата версия на бившия суперобект

Както бе отбелязано по-горе, обратната транзакция на публикуването се явява отказът от локална версия. При отказа от локална версия на съставен обект следва да се отчита фактът, че неговата версия до голяма степен зависи от версията на съставящите го обекти. Това води до следната формулировка на правило за отказ на съставен обект:

1. Отказът от локална версия на съставен обект следва да се извършва заедно с рекурсивен отказ от локална версия на всички негови подобекти.

4. Methodology framework

Методологията се определя като „система от [принципи](http://bg.wiktionary.org/wiki/%D0%BF%D1%80%D0%B8%D0%BD%D1%86%D0%B8%D0%BF) и [средства](http://bg.wiktionary.org/wiki/%D1%81%D1%80%D0%B5%D0%B4%D1%81%D1%82%D0%B2%D0%BE) за организиране и провеждане на дадена [дейност](http://bg.wiktionary.org/wiki/%D0%B4%D0%B5%D0%B9%D0%BD%D0%BE%D1%81%D1%82)” [1, стр.449]. В настоящия параграф е направен опит да се представи методологично решение при активното използване на моделите и средствата, разгледани в настоящата глава. Това предполага определянето на рамка на методологията.

В рамките на настоящата методологична рамка може да се разгледат следните моменти:

1. Подготовка на средата;
2. Създаване и определяне на задачите;
3. Изпълнение на задачите;
4. Публикуване на изпълнените задачи и сглобяване на крайния продукт.

Под подготовка на средата следва да се разбира процесът на определяне йерархичната архитектура от работни пространства, която съответства на избраната методология и подход на разработване. Тук се създава схемата, по която се организират работните пространства. На Фиг. 47 и Фиг. 48 са представени две примерни схеми на организация на работни пространства.

Тези диаграми демонстрират свободата на методологичната рамка, която тя предоставя при реализацията на проекти. За всеки отделен проект може да се организира самостоятелна схема на работни пространства, в зависимост от неговите особености и потребностите на потребителите.

Във всяка една методология за създаване на софтуерни продукти съществува етап на определяне на изискванията. В рамките на представената тук методология изискванията следва да се създадат под формата на версионизирани обекти. Това е породено от факта, че почти е невъзможно те да не се променят в рамките на целия жизнен цикъл на продукта. Изискванията, създадени във вид на версионизиран обект, позволяват да се проследи тяхното изменение, да се сравнят две техни версии, да се върнем към по-стара версия, както и да се намали рискът от изгубване на знания. Като последна стъпка от създаването на изискванията е необходимо те да се отбележат като работни задачи. Този последен момент представлява основното свързващо звено със следващите компоненти от методологията.



Фиг. 47 Модел на организация на работата в пространства по направления

Под изпълнение на задачи следва да се разбира същинският процес на създаване на софтуерния продукт. Резултатът от изпълнението на една задача може да представлява последваща задача, която разглежда първоначалната в по-големи детайли, с по-голяма прецизност. Така например създаването на архитектура на софтуерния продукт, както и на тестовите сценарии, може да се разглежда като задачи, продиктувани от изискванията, чиито краен резултат представлява задача съответно за разработването на продукта, така и за провеждането на тестовете, гарантиращи качеството на крайния продукт.

Ако се разгледа задачата по създаването архитектурата на един продукт, за нея е необходимо да се избере изискване (работна задача). След създаването на архитектурата като версионизиран обект, тук разглежданата методологгия изисква тя да се отбележи като работна задача. Забелязва се особеността, че компонентите 2 и 3 от методологията в този случай са в обърнат ред.



Фиг. 48 Модел на организация на работата в пространства по компоненти

Публикуването следва да се разглежда като средство за интегриране на отделните компоненти на продукта. От модела на видимост на обектите (представен по-горе в текущата глава) следва, че публикуването на обект в по-горно работно пространство води неговата видимост в сестринските работни пространства. Тук под сестрински работни пространства се разбира тези работни пространства, които се явяват дъщерни работни пространства на родителското работно пространство. Именно публикуването представлява механизъм за споделяне обектите, съответно и на сглобяване на крайната версия на продукта. Когато едно изискване се одобри, т.е. по него е достигнат консенсус между участниците в проекта, то може да се публикува в главното работно пространство. Така то става видимо за всички участници в проекта. Архитектите, инженерите по качество имат възможност да създадат своите артефакти, указвайки като причина за тяхното появяване новото изискване.

### Процес на създаване на нова функционалност

В настоящия пункт се демонстрира процесът на създаване на нова функционалност към нова или съществуваща софтуерна система. Тук се използват елементите от методологичната рамка, представени по-горе. За улеснение нека се приеме, че в проекта се използва опростена схема на организация на работните пространства по направления, която е представена на Фиг. 49.

Както е прието в практиката, процесът на създаване на нова функционалност се състои от следните стъпки:

1. Определяне на изискванията. В представената схема те следва да се създадат в работното пространство на Аналитика.
2. След тяхното съгласуване те се маркират като завършени и могат да се избират за работни задачи
3. Новите изисквания се публикуват до главното работно пространство, където стават видими за останалите участници в процеса.
4. Архитектът в проекта избира като работна задача за своето работно пространство новото изискване.
5. Той създава архитектура на системата, чиято реализация съответства на изискването.
6. След завършване на процеса по създаване на архитектурата, новият артефакт се отбелязва като работна задача.
7. Архитектурата на системата се публикува до главното работно пространство, където тя става видима за останалите участници в системата
8. Тестовият инженер отбелязва в своето работно пространство като работни задачи, над които ще се работи изискването (видимо от стъпка 3), и архитектурата на системата от предишната стъпка (зелената и червената пунктирани линии на ).



Фиг. Етапи по създаване на изсквания и архитектура

1. Той създава тестови план и тестови сценарии за проверка на качеството на бъдещата системна функция.
2. След приключване на работата по тестовия план и сценариите, те се отбелязват като работни задачи и се публикуват в главното работно пространство, където стават видими за останалите участници в процеса.
3. Разработчиците отбелязват като работна задача, над която ще се работи в своите пространства тези на изискването (от стъпка 3) и архитектурата (от стъпка 7).
4. Разработчиците реализират новата функционалност на системата.
5. След завършване фазата на разработване, артефактите на новата функционалност (изходен код, документация и др.) се публикуват до главното работно пространство, където те стават видими за всички участници в процеса.



Фиг. Етапи по създаване на изходен код и тестови сценарии

1. Тестовият инженер отбелязва в своето работно пространство като работна задача тестовите сценарии от стъпка 10 и започва да проверява качеството на продукта (жълтата и зелената пунктирани стрелки от ).
2. При откриване на дефект тестовият инженер създава нов обект за дефект, отбелязва го като работна задача и го публикува до главното работно пространство.
3. Всички участници в процеса проверяват постановката на тестовия сценарий и дефекта и подтвърждават неговата правилност. За улеснение в конкретния случай, нека се приеме, че дефектът описва несъответствие между реализираната функционалност и първоначалните изисквания.
4. Разработчикът отбелязва дефекта като работна задача, над която ще работи.
5. Той разработва поправка и я публикува до главното работно пространство.
6. Тестовият инженер извършва повторна проверка на функционалността и потвърждава, че дефектът е отстранен.



Фиг. 51 Стъпки по откриване и отстраняване на дефект

### Процес на промяна на съществуваща функционалност

В пункта се демонстрира процесът на промяна на съществуваща функционалност на дадена система чрез използване на елементи от представената методологична рамка.

1. Аналитикът определя съществуващите изисквания, които следва да се променят, и ги редактира. Следва да се отбележи, че изискванията притежават свойството „работна задача” от предишната итерация (тази на създаване или на промяна).
2. След тяхното съгласуване те се маркират като завършени и могат да се публикуват до главното работно пространство, където стават видими за останалите участници в процеса.



Фиг. 52 Стъпки 1 и 2 по определяне промяната в съществуващи изисквания

1. Архитектът в проекта извършва анализ за съответствие на версиите между обектите на архитектурата и изискванията. В справката за несъответствие на причинно-следствените връзки се отбелязва, че текущата версия на някои архитектурни обекти не съответства на текущата версия на изискванията. Архитектът променя архитектурните обекти, така че те да съответстват на променените изисквания.
2. Обстоятелството, че архитектурните обекти в своята предишна версия имат свойството на работна задача, се наследява и в тяхната нова версия. Ако на предишната стъпка са били създадени нови архитектурни елементи, те следва да се отбележат като „работна задача”. Променените архитектурни обекти се публикуват до главното работно пространство, където стават видими за останалите участници в процеса.



Фиг. 53 Стъпки 3 и 4 по промяна в съществуващата архитектура

1. Тестовият инженер, използвайки справката за несъответствие на причинно-следствените връзки, определя в кои тестови сценарии той следва да извърши корекции. Променените тестови сценарии наследяват свойството „работна задача” от предишната си версия. За новите тестови сценарии тестовият инженер добавя свойството „работна задача”.
2. Променените тестови сценарии се публикуват до главното работно пространство.
3. Разработчикът, използвайки справката за несъответствие на причинно-следствените връзки, определя в кои файлове от изходния код следва да извърши корекции.
4. Променените обекти на изходен код се публикуват до главното работно пространство.

Следва да се изтъкне фактът, че за представянето на стъпки от 5 до 6 може да се използва диаграмата от Фиг. 50, където в скоби са посочени номерата на стъпките от по-горе описания процес. Осигуряването на качествен контрол при промяна на съществуваща функционалност е идентично със стъпки 14 – 19 от предишния пункт (процес на създаване на нова функционалност).

5. Conclusion and future work

...