panSL: A schema-language encompassing user interface and security aspects

By Bjørn Erling Fløtten

Updated April 2012: Added description of Formulas as new section 2.1.

Updated March 2012: Added syntax highlighting for most of the panSL-code and hyperlinks to panSL reference pages and sample pages. Some minor editing done in order to clarify meaning.

Updated February 2012: Added two new sections, 6.1 and 6.2, detailing use of SubNames enabling for instance automatic generation of ER diagrams and user friendly descriptions. Use of underscore , _ , as line-continuating character specified in section 17.

First preliminary draft January 2012

Summary

panSL is a schema-language that simplifies the development process of database-centric applications by simultaneously expressing concepts within the areas of databases, entity relations, object models, user interfaces and security.

We show how panSL reduces the number of cognitive areas where one has to specify aspects of a database-centric application. For simple projects the whole specification can be encompassed in a single text file.

We also argue that panSL is suitable in the preliminary phase of complex software projects by making it easier to involve ordinary users in the specification and prototyping phase. This is ensured primarily by a simple intuitive syntax and by automatic generation of a prototype user interface.

Note: An HTML-version of this document is available at **panSL.org**, with hyperlinks for every code-example using the AgoRapide.com Implementation of panSL. Every code-example may thus be tested immediately, showing how database schemas, object oriented programming code and a graphical user-interface may be generated from the code-examples.

Note: Reference information is available at **panSL.org/Reference**. More samples are available at **panSL.org/Samples**.

Contents

1 Background, supporting different areas of software development within one language

Software development is much about repetition of the same principles and tasks over and over again.

There is a continued drive for improvement of this situation by the relentless introduction of new paradigms, "best practices", frameworks, programming languages, compilers and other tools.

panSL is a pragmatic attempt to remove some of the repetition by encompassing more aspects within a single language. It started as a personal need of the author related to a business idea for peer-to-peer insurance. A complex datamodel was required but it was not desired to build an expensive web-prototype just for discussing the idea with others. Instead the author started experimenting with a general solution for expressing complex datamodels while simultaneously enabling automatic creation of corresponding user interfaces.

Some tradeoffs are accepted, panSL is not meant to implement every possible feature. panSL is a pragmatic solution especially suited for simple problems and preliminary prototyping.

The author aknowledges the existence of many other excellent approaches for this problem area and does not claim to possess a "silver bullet" of any kind (yet).

panSL supports concepts central to different areas within software development:

1) Relational databases: Tables, fields and relations. ER-diagrams.

2) Object oriented programming (OOP): Inheritance, complex types and enumerations (enums).

3) Security: Access rights, "log in" methods, logging.

4) User interface: Input validation, localization, reporting.

2 General concepts, property types and the "schema-tree"

Central concepts of panSL are PropertyTypes and relations between PropertyTypes.

A property type may be something like "Person" or "First name". A relation (not in the database relational sense) between these two may specify that "First name" is an obligatory element of "Person". Indentation is used to specify such relations. Example:

```
Person
   First_name
   Last_name
```

The number of indentation steps is not important as long as it is consistent. Underscore is used for space (an Implementation should change this back to space in the user interface). This first very simple example also hints about the tree structure of panSL where entities (also called root property types) are specified without indentation and underlying properties are organized hierarchically by deeper and deeper levels of indentation.

Any Implementation of panSL should be able to infer that "First name" is of DataType ShortText and that the Cardinality for the relation is Obligatory. A complete definition might look like this:

```
Person              Heading
   First_name       ShortText Obligatory
   Last_name        ShortText Obligatory
```

This shows how panSL uses keywords for concepts like DataType and Cardinality (explanation of the other concepts with corresponding keywords will follow later in this document). But often these keywords constitue an unneccessary complication. For complex models this will just result in a lot of redundant text making the model harder for the human eye to read. Unneccessary keywords are therefore eliminated or shorthanded as much as possible.

For transfer of schemas between different Implementations however, it might actually be desired to use the complete definition with all the keywords in clear text, in order to reduce the risk of different systems using different methods of simplification.

If we wanted to add a new property type called "Description" but which is not Obligatory we might use something like:

```
Person
   First_name
   Last_name
   Description      ShortText Optional
```

If more than one instance is possible Cardinality Many may be used:

```
Person
   First_name
   Last_name
   Phone_number     Many
```

Note that Many is actually a simplification of DataType ShortText and the actual Cardinality ZeroToMany. ShortText OneToMany may also be used, meaning that at least one phone-number is obligatory and additional numbers may be added optionally.

Examples of other DataTypes are Integer, Decimal and Date, for instance:

```
Person
```

```
   First_name
   Last_name
   Date_of_birth    Date
   Age              Integer
```

More specialized datatypes are SMS and EMail. These could be used to enable features like sending one-time passwords as SMS or EMail. The idea here is to support common datatypes occuring frequently in real-life and at the same time giving hints through panSL for what the datatype might be used for.

The more information that can be given in this manner through panSL the more functionality may be included when source-code for an actual ApplicationImplementation is generated.

Examples of even more advanced datatypes are "DateInterval" for specifying "From date" and "To date" as one single property type. This would enable the user interface of an Implementation to enforce better validations of data entered by the user.

2.1 Formulas, deriving a PropertyType from other PropertyTypes.

A PropertyType may be calculated from other PropertyTypes through the use of Operators and Functions.

```
Person
   First_name
   Last_name
   Full_name = Concat(FirstName, " ", LastName)
```

An Implementation should support all Common spreadsheet Functions and Operators.

3 Enumerations (enums)

The elements of an Enumeration (enum) are separated with commas like this:

```
Person
   First_name
   Last_name
   Gender
      Male, Female, Unknown
```

The representation above is a Simplification of repeated DataType Existence and Cardinality ChooseOne like this:

```
Person
   First_name
   Last_name
   Gender
      Male          Existence ChooseOne
      Female        Existence ChooseOne
      Unknown       Existence ChooseOne
```

The non-simplified example is given for completeness only, it does not give any meaningful additional information.

4 Relations and the easy change between one-to-many and many-to-many relations

We have already seen relationships like this:

```
Person
   First_name
   Last_name
   Phone_number     Many
```

In a relational database management system this would be expressed as two tables, one called Person and one called Phone_number, with a one-to-many relationsship between Person and Phone_number. panSL does not uses the keyword "relation" for such simple cases. The single keyword Many is sufficient, making it dramatically easier for an ordinary user to understand the schema.

When entities relate to other entities panSL uses named relationsship. The simplest form of named relationship is when an entity is related to itself. For instance a person with friends:

```
Person
   First_name
   Last_name
   Friends          RelationMany
```

RelationMany is a Simplification of DataType Relation and Cardinality ZeroToMany.

Relations between different entities are specified like this:

```
Person
   First_name
   Last_name
```

```
   Car_ownership    RelationMany
Car
   Mark
   Model
   Car_ownership    RelationOne
```

One person may own many cars but a car may only be owned by a single person. Note that relationships are named. It is the name, "Car ownership" in this case, that binds the two entities together.

Linguistically, the following version is better suited for generating a good user interface:

```
Person
   First_name
   Last_name
   Cars | Car_ownership    RelationMany
Car
   Mark
   Model
   Owner | Car_ownership   RelationOne
```

This feature is called SubNames and is described in more detail in a separate section later in this document. The feature again shows how panSL is suitable for specifying both database aspects and user interface aspects.

Many-to-many relations are specified like this:

```
Person
   First_name
   Last_name
   Car_ownership    RelationMany
Car
   Mark
   Model
   Car_ownership    RelationMany
```

In this example we imagine that a car may be owned by more than one person.

We just showed how extremely simple it is to change from a one-to-many to a many-to-many relation with panSL. This is much more complicated to implement either in a traditional relational database schema or in an object oriented programming model.

In the database case for instance we would need to introduce a whole new third table in order to keep track of the relationships. By using panSL for preliminary prototyping of systems it is possible to easily play around with the datamodel until it feels "right". An Implementation using panSL, like AgoRapide.com for instance, is then able to quickly autogenerate the necessary databases

schema (and other aspects of the project), drawing a small demand on resources for getting started with a project, and almost eliminating the cost of early mistakes.

5 Inheritance, mismatch between object oriented world and relational database world

```
Unit
   Person                Type
      First_name
      Last_name
      Date_of_birth      Date
   Organisation          Type
      Name
      Registration_number
   Group                 Type
      Name
   EMail_address
   Postal_address
```

Type is a Simplification of DataType Heading and Cardinality ChooseOne. ChooseOne means that one specific property type has to be chosen among all PropertyTypes with cardinality ChooseOne.

The schema specifies, in object oriented terms, an abstract superclass called "Unit" with three inherited subclasses, "Person", "Organisation" and "Group". Common property types between all three subclasses are "Email address" and "Postal address", these are placed within the superclass "Unit". All other property types are placed within their respective subclassses.

The representation is quite natural from an object oriented view. For storage in a relational database an ApplicationImplementation must choose an "impedance mismatch resolution" approach to use, either

a) four tables, "Unit", "Person", "Organisation", "Group" which would reflect the object classes (called E/R approach or Table per type TPT), or

b) three tables "Person", "Organisation", "Group" which would reflect the object instances (called OO-approach or Table per concrete class TPC), or

c) one table "Unit" which is a pragmatic approach, using null-values for unused fields (called null-value approach / Table per hierarchy TPH).

An Implementation for rapid software development, AgoRapide.com, gives the user a choice of desired approach. panSL in itself is agnostic regarding which solution is optimal but the problem is expressed in a manner easily facilitating different approaches. If for instance a database-schema in SQL was used instead of panSL, it would be difficult to infer the inheritance situation when making an object-model representation.

6 Inheritance and relations, experimenting with relationships

The following simple representation gives powerful possibilities for experimenting with relationships. Employees of an organisation should be a physical person, "Person" in this case. On the other hand, members of a group could be anything, other groups, persons or organisations. We can specify this like:

```
Unit
   Person                    Type
      First_name
      Last_name
      Date_of_birth          Date
      Employment             RelationOne
   Organisation              Type
      Name
      Registration_number
      Employment             RelationMany
   Group                     Type
      Name
      Group_membership       RelationMany
   EMail_address
   Postal_address
   Group_membership          RelationMany
```

Small changes in the desired structure will lead to great changes in the underlying database schema, object oriented programming code and user interface.

panSL facilitates easy generation of a live user interface. By playing around with "live" data the schema-model becomes much easier to understand and analyze. This makes it very easy for ordinary users to catch design mistakes like allowing groups to be employees of a company.

By using panSL in the preliminary development phase through an Implementation like AgoRapide.com costly mistakes can therefore be avoided.

Another example: If it is desired to allow a person to be employed by many organisations then just one change has to be made, changing the text "Employment RelationOne" to "Employment RelationMany".

The path from schema to live user interface is short and simple, making it possible to experiment with many different strategies and then just discard the unsuccessful ones.

6.1 SubNames. UI, Model, ER diagram and user-friendly description of relations, distinction between singular and plural

For better generation of ER-diagrams, object models and UI (User interfaces) it is possible to specify the relationsship in more detail with SubNames as follows:

[UIName] | [ModelName] | [ERDiagramRole] | [UserfriendlyName] / IdentificatorName

The right-most part, after the last | that is [UserfriendlyName] / IdentificatorName, must be identical at both places where the relation occurs in the schema (note the use of / (slash) as deliminator, not |). The names to the left may differ.

If a particular name is not given the name of its closest right-hand neighbour is used instead. As a minimum the IdentificatorName must be specified.

The schema just given could in this manner be written like this:

```
Unit/Units
   Person/Persons                                    Type
      First_name
      Last_name
      Date_of_birth                                  Date
      Employer|Is_employed_by|Employment/Person_org  RelationOne
   Organisation/Organisations                        Type
      Name
      Registration_number
      Employees|Employs|Employment/Person_org        RelationMany
   Group/Groups                                      Type
      Name
      Members|Has|Group_membership/Group_unit        RelationMany
   EMail_address
   Postal_address
   Groups|Is_member_of|Group_membership/Group_unit   RelationMany
```

Note that it is still possible to add [UIName] to the left of [ModelName] but in the example given the names are considered identical and [UIName] has therefore been left out.

The rest of this document will mainly, for the sake of simplicity, not use SubNames.

6.2 Tree structures

For tree structures, that is when a PropertyType refers to itself through a relation, SubNames must be used in order to distinguish the roles as follows:

```
Person
   First_name
   Last_name
   Supervisor|Is_supervised_by|Supervision   RelationOne
   Subordinates|Supervises|Supervision       RelationMany
```

Note: If the role is identical, for instance "Friend", then the relation need to be written only once, as already shown earlier in this document:

```
Person
   First_name
   Last_name
   Friends RelationMany
```

7 Complex types, reuse of property types

The tree-like structure of panSL may be extended as deep as desired, for instance:

```
Company
   Name
   Address
      Street
      Postal_code
      City
```

Address in this case is regarded as a complex type. For a relational database an ApplicationImplementation might choose to use a separate table. In the object oriented world it will typically constitute a separate class or struct. In the user interface it will typically constitute a heading.

An Implementation should be aware of practical limitiations, like maximum allowed identifier length when generating database schemas, programming code and code files. If a Person object in the example just given is stored in a single database table, the fields would typically be called "Address_Street", "Address_Postal_code" and "Address_City". Deeper levels of nesting would generate corresponding longer identifiers.

Once a complex type has been defined in the schema-tree it does not have to be repeated further down. Only the first line is needed and allowed. Example:

```
Company
   Name
   Office_address|Address
      Street
```

```
      Postal_code
      City
   Delivery_address|Address
```

"Address" has only to be defined once. Note the use of SubNames in order to distinguish an office address from a delivery address.

Datatype has to be identical when a property type is repeated like this Other parameters like Cardinality and IdentificationUsefulness (see later in this document) may differ. By this it also follows that the names of different property types have to be unique even if they are placed at different locations in the schema tree.

8 Reporting and the "Identification Usefulness" concept

Any good reporting system should be able to link entities together. A web-based system for instance, showing a table of cars should have hyperlinks linking to the respective owners. When searching for persons a quick list of possible results must include the necessary information making it possible for the user to make a choice.

panSL includes a concept called IdentificationUsefulness which is used to specify what identification information should be given in such links.

```
Person
   First_name       Essential
   Last_name        Essential
   Date_of_birth    Date
```

In this case it is specified that any hyperlink to a person, or any quick search-result or similar, should include the persons first name and last name but not his or her date of birth.

Other IdentificationUsefulness keywords, in addition to Essential, are Useful and Additional.

When using inheritance it is often natural to specify Essential together with Type, so any shortlisting or hyperlink clearly shows what kind of entity (which subclass) is specified.

9 Default ordering of entities and types (subclasses)

The keyword Default may be used to indicate what entity-class or type (subclass) we would like the user interface to present first. Example:

```
Entity
```

```
   Corporation       Type Default2
      Name
   Person            Type Default1
      First_name     Essential
      Last_name      Essential
```

This indicates that the user should be shown a list of persons first or given the opportunity to create a new person first. "Corporation" should be the second priority.

Note: The keyword-name "Default" is not considered optimal and might be changed in the future.

10 Logging of access and changes to the database

A common functionality in many applications is logging of changes to the data. panSL supports this through a DataType called History with corresponding AccessType specifying which of Create, Read, Update and Delete should be logged.

Since logging of Create, Update and Delete is quite common, the abbrevation Log may be used:

```
Person
   First_name
   Last_name
   Changes           Log
```

This is equivalent to the full specification:

```
Person
   First_name
   Last_name
   Changes           History ZeroToManyReverseAdd Create Update Delete
```

11 Access rights and roles, fine grained access to underlying properties

For every PropertyType access rights may be specified by AccessRole. AccessRole consists of AccessType (Create, Read, Update or Delete), immediately followed by Role (Nobody, Administrator, Owner, Everyone, Anonymous) in a single keyword.

The common combination of Create, Update and Delete is supported as Change:

```
Person              ReadEveryone ChangeOwner
   First_name
   Last_name
```

This means that everyone (meaning all recognized entities) are allowed read access but only the entity considered as owner is allowed to do changes.

If the same role is required for all AccessType only the Role has to be given:

```
Person              Owner
   First_name
   Last_name
```

In this example Owner is a Simplification of CreateOwner, ReadOwner, UpdateOwner and DeleteOwner.

AccessRole may be specified multiple times at any place in the schema-tree. For instance

```
Person                        ReadEveryone ChangeOwner
   First_name
   Last_name
   Sensitive_health_info      Owner
```

Everyone may read general information about a person but sensitive health information should only be accessible by the owner.

12 Access to specific entities (in production / online access management)

Access may also be given to specified entities, that is entities identified by their database primary keys in a specific ApplicationImplementation:

```
Money_transfer      Create(42) ReadEveryone UpdateNobody DeleteNobody
   Transfer_date    Date
   Amount           Decimal
   From_account     RelationOne
   To_account       RelationOne
```

This means that only the entity with primary-key 42 is allowed to create a money-transfer, everyone may read it and it cannot be changed after creation. It is implicit that this only gives

meaning after an ApplicationImplementation is up and running and an entity with primary-key 42 has been defined.

The entity with primary-key 42 would in a specific ApplicationImplementation typically consist of a group to which can be added persons or other groups.

Access to specific entities is an example of how panSL may be used in quite diverse areas of application management. The feature is used by MyLittleDatabase.com, eliminating the need for a separate access-management application.

An ApplicationImplementation, for instance with the concept of users, might also choose to let its users give access to other user. A feature like this is probably unnecessary to specify within panSL, but it might be considered.

13 Log in process, identificators, username and password

By "log in process" is meant the method of which a user of an ApplicationImplementation is recognized, in order to give the user the necessary access rights for perfoming his or her tasks. We use the term "entity being impersonated" to mean the current logged in user for a given session.

panSL supports two concepts for "logging in" to a specific ApplicationImplementation:

a) Traditional, by using DataTypes Username and Password:

```
Person
   First_name
   Last_name
   Username          Username
   Password          Password
```

The Implementation should take this as a hint to generate log in functionality by asking for Username and Password in the traditional manner. The entity recognized (in this case a "Person") is then regarded as the entity currently being impersonated.

b) By using a LoginIdentificator like SMS or EMail

```
Person
   First_name
   Last_name
   Mobil_phone       SMS
   Email_address     Email
```

The user is recognized by something that he has, either a mobile-phone (SIM-card) able to receive an SMS message or access to an e-mail account. The user enters either his mobil-phone number or

his e-mail address and a one-time password is sent to this address. If more than one match is found the user may be given a choice of which entity to impersonate.

An ApplicationImplementation may of course offer a combination of these two concepts, that is requiring both Username / Password and a LoginIdentificator. This would correspond to the concept of identifying a user both by something the user knows and something the user has.

14 Schema access and administrator roles, boot-strapping a new implementation

For any specific ApplicationImplementation, administrator rights should be required in order to change a schema. The GivingRole-keyword GivingAdministrator can be used to grant administrator rights to the entity being impersonated:

```
Administrator_group
   Name
   Administrator_person  RelationMany GivingAdministrator
   Changes               Log
Person
   First_name
   Last_name
   Administrator_person  RelationOne
   Username              Username
   Password              Password
   Mobil_phone           SMS
```

GivingAdministrator means that a relation to an entity of type "Administrator group" gives the related entity Administrator-rights. In this case "Persons" related to an entity of type "Administrator group" become themselves administrators.

An Implementation can use this information to give administrator privileges to any "Person" related to an Administrator group. It should be implicit that Administrator-role is needed for any access to a GivingAdministrator-entity. Read access could however be specified, for instance ReadEveryone, enabling users of an ApplicationImplementation to see which persons have administrator rights and may be contacted for assistance.

Note that in this case the "Person"-entity is specified with both Username / Password properties and a log in identificator in the form of a mobile phone number which can receive SMS-messages. In addition all changes to any GivingAdministrator-entity is logged.

For a freshly created database without any entities defined, there must be a process of boot-strapping the environment. One possibility, used by a specific Implementation MyLittleDatabase.com, is outlined below:

MyLittleDatabase.com gives Anonymous access to all entities for a freshly created database. This access is revoked after the first entity with log in credentials is created (in this case after a "Person"-entity is created).

Therefore, a new GivingAdministrator-entity (Administrator group) and a new Person-entity may be created anonymously.

Every entity with log in credentials is considered an administrator as long as no relations are added to any GivingAdministrator-entity (Administrator group). The first "Person"-entity is therefore considered an administrator and may therefore enter itself into the Administrator group, giving itself permanent Administrator privileges.

After that only "Persons" in the Administrator group may add or delete new administrators.

Note that it could be possible to place log in information directly under a GivingAdministrator entity like this, much more simpler (but not supported in panSL), example:

```
Administrator_group      GivingAdministrator
   First_name
   Last_name
   Mobil_phone           SMS
```

The property type "Mobil phone" with datatype SMS makes it possible to log in as an Administrator-entity. In this manner logging in as an administrator is totally separated from logging in as a normal user. This solution might have been simpler to implement and understand but would probably feel slightly unnatural for the users. It is therefore not supported by current Implementations of panSL.

Note: As of February 2012 this boot-strapping mechanism has not been tested thoroughly in a production environment. Some changes to the methods and syntax given may therefore be done in the formal version of panSL.

15 Granting access rights through relationships

For simple solutions is might not be desired or practical for users to grant access rights to specific entities, like outlined earlier in this document. Such access-granting requires changes in the schema, and may be too coarse-grained.

Often the relationship in itself should be sufficient to indicate desired access, especially when entities are close together in the natural world like a family and its members. This can be done with the same GivingRole mechanism outlined earlier in this document but by using GivingOwner instead of GivingAdministrator.

```
Family                   ReadEveryone ChangeOwner
```

```
   Name
   Family_membersship    RelationMany GivingOwner
Person                   ReadEveryone ChangeOwner
   First_name
   Last_name
   Family_membersship    RelationOne
   Mobil_phone           SMS
```

Both "Family" and "Person" demand Owner-role in order to do any changes.

The "Person"-entity has defined a LoginIdentficator "Mobile phone" making it possible to impersonate a "Person", which in turn implicitly should result in Role Owner. This means that management of a "Person"-entity is possible in a natural manner.

But the management of a "Family"-entity would normally be limited to its Owner, that is the impersonated entity creating the "Family"-entity originally. A more natural solution would be if every member of a family could manage it. The GivingOwner-keyword in the "Family membersship" relation does just this, by giving Owner-role to any "Person" related to the "Family".

In practical terms: Person A creates a Family entity (becoming its owner). He then adds person B to the family. Person B is then automatically granted the Owner-role throught the keyword GivingOwner making also him or her able to manage the Family-entity.

Note that GivingOwner is restricted to the single related entity, but the GivingAdministrator-keyword outlined earlier in this document gives Administrator-role valid for all entities (for the whole database).

It would be natural to consider, in addition to GivingAdministrator and GivingOwner already outlined, the keyword "GivingEveryone", but this would probably not confer any meaningful information. In order to be given Everyone-role an entity must already have been recognized, that is it would already have the role Everyone.

Note: As of March 2012 this access granting mechanism has not been tested thoroughly in a production environment. Some changes to the methods and syntax given may therefore be done in the final version of panSL.

16 Meta tags, future versions of panSL

Meta tags are proposed as starting with a name followed by a colon. The colon would signify that the meta tag is not part of the schema definition per se but describes something about the schema.

The meta tags "SchemaName" and "panSLVersion" can be used like this:

```
panSLVersion:  0.9
SchemaName:    Test
```

```
Person
   First_name
   Last_name
```

"SchemaName" would be used by an Implementation to name a database, code-project and application.

"panSLVersion" would be useful for future versions of the language. In absence of a panSLVersion-declaration version 1.0 should be assumed.

Note that this document is preliminary and does not describe any final or specific version of panSL.

Other proposed meta tags are

"Status": With specifiers like "Development", "Test", "ProductionTest", "Production". This would greatly simplify the process of setting up testing environments and making it easy for an Implementation to indicate clearly to the user what kind of environment he or she is working in (test or production).

"Version": For official versioning of a schema / database. This would be useful for complicated schemas where it would be important to keep track of different versions (note that SchemaName could also be appended with versioning information allowed concurrent testing of different versions at the same time).

17 Formal specification of panSL

This section is an attempt to define panSL. It is not complete, nor formally 100 percent correct. See also **panSL.org/Reference** and **panSL.org/Samples**.

Any ambiguities should be resolved by looking at how leading Implementations of panSL like AgoRapide.com parse and use the language.

The author would welcome initiatives for starting a formal standardisation process of panSL.

Blank lines are ignored.

Indentation either by SPACE (ASCII value 32) or TAB (ASCII value 9) is used to indicate hierarchical order.

Zero indentation indicates a root property-type or entity.

The number of indentation steps are not important as long as it is consistent. A parser accepting inconsistent indentation (varying number of indentation steps) should always output a consistent

version after parsing clearly showing its interpretation.

By newline is meant either a single character with ASCII value 10 or two characters with ASCII values 10 and 13.

An underscore character, "_", at the end of a line means that the following line should be concatenated to the end of the line containing the underscore. In other words the following newline-character shall be removed when parsing, together with the underscore. This corresponds to the "Visual Basic style" of line continuation for long statements.

A newline character marks the beginning of a new property type definition. However, newline characters within Java-style block markers { and } are to be treated like whitespace (SPACE or TAB).

Java-style comments, either single-line commments // or multiline (block comments) /* ... */ may be used and should be interpreted according to the specification for version 1.0 of the Java programming language.

A schema may optionally start with any of the following elements:

```
[PanSLVersion:      decimalNumber]
[SchemaName:        name]
```

A property type is defined by its name followed, optionally, by one or more specifier keywords separated by whitespace (SPACE or TAB).

```
PropertyName [DataType] [Cardinality] [IdentificationUsefulness]
[AccessRole] [Default] [GivingRole]
```

The PropertyName itself may not contain any white space (expect when using — and / to specify SubNames). Underscore should be used for space and the corresponding replacement done at UI-level by the actual Implementation.

PropertyName may consist of the following SubNames, either

```
[SingularName] / [PluralName]
```

or

```
[UIName] | [ModelName] | [ERDiagramRole] | [UserfriendlyName] / IdentificatorName
```

The vertical bar | and slash / are used as separators within the PropertyName. Space is permitted before and after the separators.

If a particular SubName within the PropertyName is not given the name of its closest right-hand neighbour is used instead. As a minimum the identificator must be specified.

A PropertyType may be derived (calculated) from other PropertyTypes by using the equal-sign, that is writing PropertyName as "Name = [Formula]". DataType Formula is then implicit and Cardinality becomes unneccesary to specify.

The order of the specifier keywords is not significant.

Any group of specifier keywords may be enclosed in Java-style block-markers, { and }. Within such a block newline characters may be used in addition to SPACE or TAB for separating keyword-specifiers, and Java-style comments may be used withing these lines again.

(A parser could remove all comments within a Java-style block first, then replace newline with SPACE and then remove the block-markers { and } before parsing the keyword specifiers).

DataType may be one of:

```
Heading, ShortText, LongText, Integer,
Decimal, Percent, Date, URL, Relation,
History, Username, Password, SMS, EMail
```

DataType Heading is only valid for property types with children.

DataTypes other than Heading is only valid for property types without children.

DataType History must be accompanied with one or more HistoryType-specifiers like

```
Create, Read, Update, Delete
```

Cardinality is not legal at entity-level (not legal for root property types). For all other property types Cardinality may be one of:

```
Obligatory, Optional, ChooseOne, ZeroToMany, OneToMany
```

IdentificationUsefulness may be one of the following

```
Essential, Useful, Additional
```

AccessRole may be one of any combination of

```
Create, Read, Update, Delete
```

plus either

```
Administrator, Owner, Everyone, Anonymous
```

or a entity primary key enclosed in parentheses. Examples of such combinations:

```
CreateEveryone, ReadAnonymous, Update(42), Delete(42)
```

Default is followed directly with an integer from 1 upwards. Example:

```
Default1
```

GivingRole is only relevant when the DataType is Relation. It may be one of the following:

```
GivingAdministrator
GivingOwner
```

18 Simplifications

The following Simplifications are possible:

At entity-level (root property type), DataType is simplified as:

```
Heading                  ->   [Nothing]
```

When the property type has children, DataType + Cardinality is simplified as:

```
Heading Obligatory       ->   [Nothing]
Heading ZeroToMany       ->   Many
```

When the property type does not have children, DataType + Cardinality is simplified as:

```
ShortText Obligatory     ->   [Nothing]
ShortText ZeroToMany     ->   Many
```

Other cases, DataType plus Cardinality is simplified as:

```
Heading ChooseOne        ->   Type
Relation Obligatory      ->   RelationOne
Relation ZeroToMany      ->   RelationMany
LongText Obligatory      ->   LongText
Date Obligatory          ->   Date
Decimal Obligatory       ->   Decimal
Integer Obligatory       ->   Integer
SMS Obligatory           ->   SMS
EMail Obligatory         ->   EMail
```

Other cases:

```
History ZeroToManyReverseAdd Create Update Delete          ->   Log
```

For a PropertyType whose children are all of DataType Existence and Cardinality ChooseOne (in other words, when the PropertyType is an Enumeration) the children are put onto the same line separated with commas. Example:

```
Gender
   Male             Existence ChooseOne
   Female           Existence ChooseOne
   Unknown          Existence ChooseOne
```

is shortened to

```
Gender
   Male, Female, Unknown
```

AccessRoles with the same Role for Create, Read, Update and Delete is shortened to just the Role. Example:

```
CreateOwner ReadOwner UpdateOwner DeleteOwner     ->   Owner
```

AccessType with the same Role for Create, Update and Delete is shortened to Change plus the Role. Example:

```
CreateOwner UpdateOwner DeleteOwner               ->   ChangeOwner
```

AccessType with the same entity primary-key for Create, Read, Update and Delete is shortened to Access plus the entity primary-key. Example:

```
Create(42) Read(42) Update(42) Delete(42)          ->   Access(42)
```

Access with the same entity primary-key for Create, Update and Delete is shortened to Change plus the entity primary-key. Example:

```
Create(42) Update(42) Delete(42)                   ->   Change(42)
```

(A parser could expand all Simplifications before attempting to parse the property type)

19 Use of panSL today, MyLittleDatabase.com and AgoRapide.com

panSL is currently a central aspect of two web based services

1) MyLittleDatabase.com: panSL is used as the central element to build and host small web accessible ApplicationImplementations complete with security management. Ordinary users may create solutions themselves, or they may ask for assistance by freelance developers. If a solution with MyLittleDatabase grows very big and more complex features are needed, then AgoRapide (see next paragraph) may be used to export autogenerated code and data to a traditional setup for continued traditional development.

2) AgoRapide.com: panSL is used as a preliminary specification mechanism making it easier to try out different data structures and access patterns. After the preliminary phase the system is used to autogenerate code for continued development with traditional tools. The developer chooses his or her own favourite paradigms and tools. AgoRapide strives to support all the popular combinations.

In addition to being suitable for developers wanting a quick way to prototype with their customers, AgoRapide might also be useful in an educational setting. AgoRapide can show practical implications of different strategies, for instance different ways of solving the "impedance mismatch" problem between the object oriented programming world and the relational database world.

20 Conclusion, how panSL can contribute to cheaper and better software development

By basis in this document and the examples given we claim that:

1) panSL gives ordinary users an easier entrance to the world of software development. panSL encompasses several different aspects of software development in one language, lowering the level of knowledge needed to getting started.

2) panSL may also reduce the number of applications and tools a developer has to install in order to prototype ApplicationImplementations. A web-based Implementation like AgoRapide.com for instance offers instant prototyping without any local tool installation at all.

3) panSL makes it easier and quicker to prototype database-centric applications. Big changes in the database-schema can quickly be tried out and just discarded if not found to be useful.

4) panSL is suitable as a starting point for automatic creation of database schema, object oriented code and user interface. This is demonstrated by the Implementation AgoRapide.com (in alpha-version at the moment).

5) panSL is suitable for building simple but complete database-centric applications. A web-based Implementation MyLittleDatabase.com demonstrates this by offering development and hosting of applications using panSL as a single-point of specification.

Note: Reference information is available at **panSL.org/Reference**. More samples are available at **panSL.org/Samples**.

21 About the author

Bjørn Erling Fløtten currently lives in the city of Trondheim in Norway. He has worked within software development for about 20 years. He has a background as an Engineer within Computers and Electronics. Bjørn has been repeatedly involved in the creation of new companies within diverse areas of business. The last year and a half he has studied mathematics at the Norwegian University of Science and Technology while simultaneously working on new ideas for software development. He is currently working on the commercialization of two new services, MyLittleDatabase.com and AgoRapide.com, both of which uses panSL as a central aspect. He may be contacted at e-mail address bef@bef.no.