

CCL: A Lightweight ORM Embedding in Clean

Bas Lijnse^{1,2}, Patrick van Bommel¹, and Rinus Plasmeijer¹

¹ Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, The Netherlands

² Faculty of Military Sciences,
Netherlands Defense Academy, Den Helder, The Netherlands
`{b.lijnse,pvb,rinus}@cs.ru.nl`

Abstract. Agile software development advocates a rapid iterative process where working systems are delivered at each iteration. For information systems, this drive to produce something working soon, makes it tempting to skip conceptual domain modeling. The long term benefits of developing an explicit conceptual model are traded for the short term benefit of reduced overhead. A possible way to reconcile conceptual modeling with a code-centric agile process is by embedding it in a programming language. We investigate this approach with CCL, a compact textual notation for embedding Object-Role Models in the functional language Clean. CCL enables specification of Clean types as derivatives of conceptual types. Together with its compact notation, this means that defining data types with CCL as intermediary requires no more programming effort than defining data types directly. Moreover, because embedded ORM is still ORM, mappings to other ORM representations remain possible at any time.

1 Introduction

The foundation of a successful information system is a solid understanding of the domain it represents. A way of capturing such understanding on a conceptual level is through the use of Object-Role Models (ORM [4]). They allow modelers to express the conceptual relationships in a domain without committing to the level of detail that implementation data structures require. A common approach to ORM is to make models with dedicated tools as analysis and design activity, and use them to bootstrap development of information systems by generating relational schemas and template code. An implicit assumption of this approach is that the development process has a requirements analysis or design phase prior to a construction or programming phase. With the increasing popularity of Agile development [2] this assumption does no longer always hold. Agile development advocates an iterative process with short cycles in which working systems are delivered at each iteration and stakeholders are actively involved in the development. In such a process, with a focus on delivering working systems in a short amount of time, it is tempting to skip conceptual modeling, because the overhead of using dedicated modeling tools while already programming an information system may outweigh the short-term benefits. Unfortunately this

means that the long-term benefit of an explicit conceptual model that can be used in communication with stakeholders is also lost.

A possible way to reconcile conceptual modeling with a code-centric Agile development process is by embedding ORM in a programming language. At first glance, this appears to be a compromise. For ORM modelers, it means that conceptual models have to be specified textually instead of graphically in order to let it be embeddable in the structured text of a programming language. For programmers this means that they cannot define data types directly, but that they need to specify a conceptual model of the domain first. Yet, we expect that embedding ORM in a programming language can have several benefits: First, conceptual relations between different data structures used in the code of an information system are often implicit. By making them explicit, data types can be specified more compactly as derivatives of shared conceptual types, giving programmers an immediate short-term benefit. It also gives compilers additional opportunities to check code and detect programming mistakes.

Second, because models are integrated in the source code of information systems there is a direct link between the model and the working information system. This means there cannot be discrepancies between the ORM models and the behavior of the information system caused by misinterpretation of the models during implementation. Moreover, embedded ORM models are still ORM models. This makes it possible to maintain bidirectional mappings to representations used by other ORM notations and tools.

In this paper we investigate the embedded ORM approach with CCL (Concepts in CLean), a compact notation for embedding ORM models in the pure functional programming language Clean [10]. Clean is a statically typed language where data types play an important role. Not only are they used to detect programming errors, but they are also for type-driven generic programming, a technique in which data types are used to parameterize algorithms. Contemporary Clean frameworks like iTasks [6] aim to improve agility by enabling large parts of information systems to be generated from abstract patterns that depend on types. This makes specification of data types a key element of agile Clean programming.

The main contributions of this paper are:

- We provide a new compact textual notation of a core subset of ORM.
- We demonstrate how this notation is used to embed ORM in a programming language.
- We extend the Clean language with the possibility to specify conceptual types underlying multiple data types.

The remainder of this paper is organized as follows: We start with an example of CCL in Section 2. We then continue with an explanation of the notation of ORM constructs for defining conceptual models in CCL in Section 3 and for defining derivative data types in Section 4. We reflect on the benefits as well as the scope and limitations of our work in Section 5 and put it the context of related work in Section 6. In Section 7, we end with concluding remarks and an outlook on future work.

2 A CCL Example

Before going into technical details of the CCL notation, we present a simple, yet nontrivial example. We model a personal audio catalogue of digital audio files like for example a collection of ripped CD's or an iTunes library. The central concept is an album, which can be a music album or an audiobook. Additionally songs, artists and authors of audiobooks are modeled.

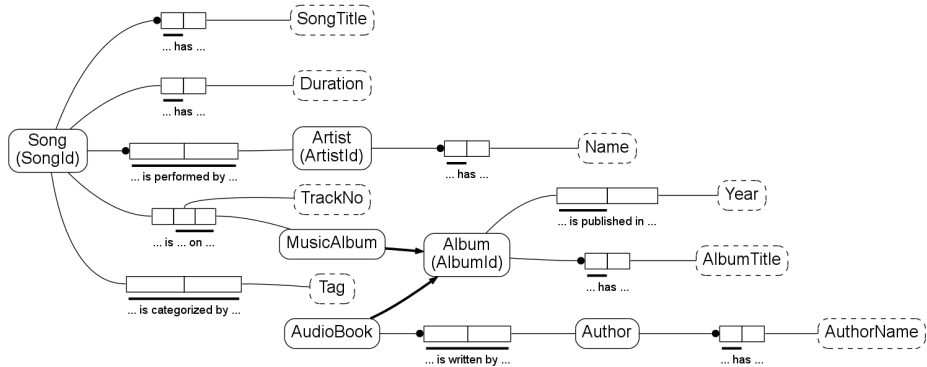


Fig. 1. ORM diagram of a personal audio collection

Figure 1 shows the ORM2 diagram of such an audio collection which is defined by the CCL code in Figure 2. This definition consists of three parts. A series of object type definitions consisting of entity type definitions and value type definitions, a series of fact type definitions and a series of fact container type definitions that define Clean data types in terms of the conceptual definitions. In the next section we explain this definition in more detail.

3 Defining Conceptual Models with CCL


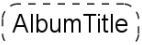
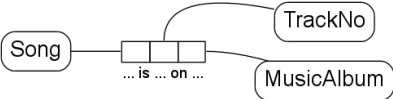
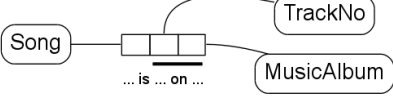
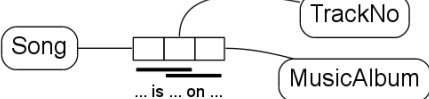

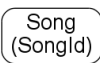
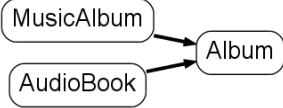
Conceptual models are defined as structured text in CCL source modules, text files with `.ccl` as extension. Each CCL module starts with a module header defining its module name.

concept module AudioCollection

By giving each module a name we make it compatible with the module system of Clean such that we can refer to its content from within other modules.

The module header is followed by a series of declarations. These can be *object type* declarations, *fact type* or declarations, that define the model, or *fact container type* declarations that link the conceptual level to concrete first-order data structures. In Table 1 an overview of the CCL declarations is listed together with examples and their corresponding ORM diagrams.

Table 1. CCL Language Constructs

Entity Types	
\$\$ Song	
Value Types	
\$\$ AlbumTitle = String	
Fact Types	
## Song is TrackNo on MusicAlbum or ## songs = Song is TrackNo on MusicAlbum	
Uniqueness Constraints	
## Song is << TrackNo on MusicAlbum >>	
## <1< Song is <2< TrackNo >1> on MusicAlbum >2>	
Total Roles	
## !Song has SongTitle	
Primary Roles	
\$\$ SongId = Int ## << !!Song >> has << SongId >>	
Subtypes	
\$\$ Album \$\$ MusicAlbum [Album] \$\$ AudioBook [Album]	

```

//Module header
concept module AudioCollection
// Entity types
$$ Album
$$ AudioBook [Album]
$$ Author
$$ MusicAlbum [Album]
$$ Artist
$$ Song
// Value types
$$ Name      = String
$$ SongId    = Int
$$ SongTitle = String
$$ AlbumId   = Int
$$ AlbumTitle = String
$$ ArtistId  = Int
$$ Year      = Int
$$ Duration  = Time
$$ TrackNo   = Int
$$ Tag       = String
$$ AuthorName = String
// Fact type definitions
## album_id =
  << !!Album >> has << AlbumId >>
## album_title =
  << !Album >> has AlbumTitle
1  ## album_year =
2  << Album >> is published in Year
3  ## song_id =
4  << !!Song >> has << SongId >>
5  ## title =
6  << !Song >> has SongTitle
7  ## duration =
8  << Song >> has Duration
9  ## songs =
10 Song is << TrackNo on MusicAlbum >>
11 ## performed_by =
12 << !Song is performed by Artist >>
13 ## tags =
14 << Song is categorized by Tag >>
15 ## artist_id =
16 << !!Artist >> has << ArtistId >>
17 ## artist_name =
18 << !Artist >> has Name
19 ## author_name =
20 << !Author >> has AuthorName
21 ## author =
22 << !AudioBook is written by Author >>
// Fact container types
23 #: Album    = Album {..}
24 #: Artist   = Artist {..}
25 #: Song     = Song {..}

```

Fig. 2. CCL Definition of Personal Audio Collection

3.1 Entity Types

Entity types are defined by declaring a name for a concept. Their declarations consist of an *object type marker*, two dollar signs, followed by an *object type name*. The object type marker is a symbol (\$\$) that indicates we are defining an object type declaration. The object type name must consist of alphanumeric characters only and must start with a capital letter.

Optionally a list of super types can be specified after the object type name to declare subtyping constraints. This list starts with an open bracket, is followed by entity type names separated by commas and ends with a closing bracket.

3.2 Value Types

Value types are defined by assigning a name to a first-order Clean type. They are declared the same way as object types, but with the addition of an equals sign, followed by the name of a first-order Clean type. Specification of super types is not allowed for value types.

3.3 Fact Types

Fact type declarations are defined by a *fact type marker* (##) followed by a sentence consisting of capitalized words and all lowercase words. The capitalized words are interpreted as names of object types.

It is allowed to reference object types that are not explicitly declared by an object type declaration. Undefined references are interpreted as implicit entity type declarations. It is recommended to explicitly declare entity types, but by allowing implicit use it is possible to construct a model by supplying fact types only. The CCL compiler can detect implicit references and issue warnings.

Optionally one can assign names to fact types by adding a *fact type name* followed by an equals sign between the fact type marker and the sentence. Assigning a name to a fact type makes it possible to reference the fact type in the declaration of *fact container types*.

3.4 Uniqueness Constraints

Uniqueness constraints are expressed by annotations in fact type declarations. Unique sets of roles can be marked by enclosing parts of the sentence with double angle brackets (<< >>). When multiple uniqueness constraints within the same fact type overlap making their definition becomes ambiguous. This can be resolved by labeling the constraints with a unique number. This number is placed between the angle brackets. Non-adjacent uniqueness can be specified by using the same label on multiple annotations.

3.5 Mandatory and Primary Roles

Mandatory role constraints are specified by prefixing object type references in a fact type declaration with an exclamation mark (!).

To indicate that a mandatory role is not only mandatory, but that the associated value type(s) may be used as reference for an entity type, we use an annotation to mark a role as the *primary* role. Because it is essentially a stronger version of a mandatory constraint, we use a double exclamation mark as annotation (!!).

This annotation does not necessarily have to be used together with uniqueness constraints, but if it is used in a binary fact type between an entity type and a value type with two unique roles, the value type is depicted with the shorthand notation for standard names in the derived diagram. Each entity type can have only one primary role annotation.

4 Defining Clean Types with CCL

To manipulate facts in Clean programs, we need to represent them in compound data structures. Because Clean is a strong statically typed language that uses Burstall-style algebraic data types to type compound structures, we need to

define (Clean) types of such data structures representing facts. To achieve this, CCL offers notation to define Clean types in terms of CCL fact types, which are automatically expanded. This way, the specification of CCL fact types reduces the specification effort of Clean types, because a single fact type can contribute to the definition of multiple Clean types if it relates to multiple entity types. Even for the simple model in Section 2 the CCL definition is more concise than the expanded Clean data types derived from it.

4.1 Fact Container Types

To define Clean types that can contain composite structures of facts, CCL offers so called *fact container types*. Their notation is similar to the notation for *record types* in Clean. For example:

```
#: SongSummary = Song {song_id,title,songs}
```

Which expands to the following Clean record type:

```
:: SongSummary =
  { song_id :: SongId
    , title :: SongTitle
    , songs :: [(TrackNo,AlbumId)] }
```

A fact container type definition starts with a *container type marker* (`#:`), followed by a type name and an equals sign. The righthand side of the equals sign consists of a *focus entity* and a selection of facts that get mapped to field names. The focus entity is the name of a conceptual entity type, that we collect facts about. It is implicit in all the facts that the type contains. The selection of facts is a comma separated list of named facts in which the focus entity has a role. Each fact maps to a field in the expanded record type. The types of the fields (in Clean denoted with the double colon) do not have to be specified because they can be inferred from the conceptual model. For value types this is simply their Clean type. For entity types, the type of the fact type with a primary role annotation is used. This inference makes the type specifications more concise.

4.2 Complete Container Types

To make the definition of data types even more concise, a shorthand notation is available for defining fact container types that contain *all* facts about an entity. To define a type that expands to include all named facts a focus entity has role in, one can use the following notation:

```
#: Album = Album {..}
```

4.3 Explicit Field Type Specifications

If more control over the data types of fields in fact container types is desired, selected facts may be annotated with type information. To not just reference related entities, but to include facts about them one could for example specify:

```
#: SongSummary = Song {song_id,title,songs :: (TrackNo,Album)}
```

The name of a fact in the selection is annotated by a double colon followed by a comma separated list of Clean types enclosed in parenthesis for all roles except the role of the focus entity. For binary facts the parenthesis may be omitted because only one role remains.

5 Discussion

5.1 Conceptual Modeling

CCL is a lightweight embedding of ORM that provides notation for a core set of ORM constructs only. The reason for this is partially intentional and partially practical. Covering all ORM constraints fully would make the CCL language more complex while the additional value is uncertain. Because we are embedding ORM in a general purpose programming language, we don't need to be complete. Many constraints can also be expressed alternatively in Clean and addition would only duplicate existing functionality. The supported ORM subset adds the structural conceptual level that could not be expressed explicitly in Clean. Exploration of the effects of additional constraints remains a topic for future research.

Although unnecessary for the generation of Clean code, the CCL notation uses complete sentences to define fact types. For constraints however, it uses annotations instead of a more verbose verbal form. This approach aims to provide a notation that is concise but still contains all information necessary for verbalization. The notation is therefore not as close to natural language as for example SBVR, but still starts from stating facts about a universe of discourse.

5.2 Effects on Agility

The use of CCL as intermediate step in the definition of collections of data types, may improve agility of a Clean programmer in two ways. First, it reduces the amount of code that has to be written, because CCL makes the definition of types more compact. Secondly, it automatically keeps the data types that represent entities that share fact types consistent. This makes it easier to incrementally extend the system under development, because changes that involve multiple conceptual entities can be made in one place.

A possible negative effect may be that too much of a domain is modeled. If in CCL more types are defined than are used in the current iteration of an information system, time is spent on something that does not contribute to the working system. Luckily this can easily be detected by the compiler.

5.3 Implementation

To be able to test and investigate the use of CCL in information systems developed with Clean, we have implemented a basic compiler. This is a proof-of-concept compiler that serves as a preprocessor to the Clean compiler. The

primary purpose of our compiler is to compile CCL to Clean. Although CCL's syntax is designed to avoid conflict with Clean's syntax such that CCL can be mixed with Clean in the same module, we currently only support separate CCL modules that can transparently be compiled to Clean using the Clean IDE's preprocessing support.

The secondary purpose is to generate representations of the conceptual models for communication with domain experts. Currently we support the generation of ORM2 diagrams, but one could also think of verbalizations, or a combination of both in hyperlinked documents. Because, CCL does not allow the specification of diagram layout, we use the popular open-source Grapviz tool, to visualize CCL. The CCL compiler generates a graph structure in the DOT language which is rendered by Graphviz. All diagrams shown in Section 2 and Section 3 have been generated from CCL in this way.

Both the CCL notation, and the CCL compiler have many opportunities for improvement. Obvious additional targets are the generation of relational schema's for storage, and access functions for conversion between flat relational structures and CCL fact container types. Another area in which the CCL compiler could be improved, is interoperability with formats from tools such as Norma, or languages as CQL or SBVR.

6 Related Work

It is obvious that CCL as presented in this paper is not intended to replace any of the current ORM notations and tools, but rather to introduce ORM in a new context where explicit conceptual models have added value. Therefore it may be less obvious where to position CCL in the ORM literature. Unlike most concrete ORM notations, like the earlier NIAM [9], FCO-IM [1] or more recent ORM2 [4], CCL is a textual language with a formal concrete syntax, instead of a graphical language. In its textual approach it is closer to SBVR vocabularies [3] or CQL [5], but less natural language oriented. CCL is a pure data definition language. Where languages as RIDL [8] and LISA-D [11], as well as CQL incorporate the querying and manipulation of information, CCL defines only the conceptual structure. Manipulation of data is already provided by the host language Clean. In its aim to accommodate an agile process and use of a text oriented approach, CCL has some overlap with CQL. However, because CCL is an embedded language it has a different focus. CCL's syntax emphasizes concise notation, to align with the host language, whereas CQL chooses a linguistic approach for better alignment with domain experts. CCL is less expressive as CQL, because it is just a lightweight embedding, not a standalone language. With regard to information system development in Clean, CCL can be related to earlier work on automated mapping between relational tables and Clean data types [7]. That approach relied on the encoding of conceptual relations in Clean types because no explicit ORM model could be expressed. CCL could be used to improve this mapping.

7 Conclusions

In this paper we have presented CCL, a lightweight embedding of ORM in the functional programming language Clean. We have shown that it is possible to define conceptual models from within a programming language without additional overhead. Because derivation of data types from conceptual types enables a more concise specification of collections of data types that model a domain, the additional effort of specifying the conceptual types pays off immediately. Moreover, because a tightly integrated conceptual model is developed from within the programming language, ORM diagrams or verbalizations can be extracted from an information system's source code at any time.

References

1. Bakema, G., Zwart, J., van der Lek, H.: Fully Communication Oriented Information Modelling. FCO-IM Consultancy (2002)
2. Fowler, M., Highsmith, J.: The agile manifesto. *Software Development* 9, 28–35 (2001)
3. O.M. Group. The semantics of business vocabulary and business rules (2009), <http://www.omg.org/spec/SBVR/1.0/>
4. Halpin, T.: ORM 2. In: Meersman, R., Tari, Z., Herrero, P. (eds.) OTM-WS 2005. LNCS, vol. 3762, pp. 676–687. Springer, Heidelberg (2005)
5. Heath, C.: The Constellation Query Language. In: Meersman, R., Herrero, P., Dillon, T. (eds.) OTM 2009 Workshops. LNCS, vol. 5872, pp. 682–691. Springer, Heidelberg (2009)
6. Jansen, J.M., Plasmeijer, R., Koopman, P., Achten, P.: Embedding a web-based workflow management system in a functional language. In: Brabrand, C., Moreau, P.-E. (eds.) Proceedings 10th Workshop on Language Descriptions, Tools and Applications, LDTA 2010, Paphos, Cyprus, March 27–28, pp. 79–93 (2010)
7. Lijnse, B., Plasmeijer, R.: Between Types and Tables - Using Generic Programming for Automated Mapping Between Data Types and Relational Databases. In: Scholz, S.-B., Chitil, O. (eds.) IFL 2008. LNCS, vol. 5836, pp. 272–290. Springer, Heidelberg (2011)
8. Meersman, R.: The RIDL conceptual language. Technical report, International Centre for Information Analysis Services, Control Data Belgium Inc. (1982)
9. Nijssen, G.M., Halpin, T.A.: Conceptual schema and relational database design: A fact oriented approach. Prentice-Hall, New York (1989)
10. Plasmeijer, R., van Eekelen, M.: Clean language report, version 2.1 (2002), <http://clean.cs.ru.nl>
11. ter Hofstede, A., Proper, H., van der Weide, T.: Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems* 18(7), 489–523 (1993)