University of Groningen

From elemantary user wishes and domain models to SQL-specifications

de Brock, Bert

Published in:
Business modeling and software design

DOI:
10.1007/978-3-030-79976-2_6

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2021

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
de Brock, B. (2021). From elemantary user wishes and domain models to SQL-specifications. In B. Shishkov (Ed.), *Business modeling and software design: 11th International Symposium, BMSD 2021, Sofia, Bulgaria, July 5–7, 2021, Proceedings* (pp. 97-117). (Lecture notes in business information processing; Vol. 422). Springer. https://doi.org/10.1007/978-3-030-79976-2_6

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 04-06-2022

From Elementary User Wishes and Domain Models to SQL-Specifications

Bert de Brock(✉)

Faculty of Economics and Business, University of Groningen,
PO Box 800, 9700 AV Groningen, The Netherlands
E.O.de.Brock@rug.nl

Abstract. In the development of (software) systems, new user wishes must usually be implemented very quickly. This poses a real challenge for system development. This challenge led from *waterfall* to *incremental*, *agile,* and even *continuous* development. In this paper we treat the research question how to come from elementary user wishes and simple domain models all the way to concrete SQL-specifications in a quick, straightforward, and traceable way.

We will follow the classical distinction between the *static* part (i.e., the *data structures*) and the *dynamic* part (i.e., the *processes*) of the system under development. We also explain how these different aspects are coordinated. Moreover, we will distinguish between the *Problem Analysis* part and the *Software Design* part of system development. We introduce the notions of *elementary User Wish* and *textual System Sequence Description*, which help us to start in an early phase of development, to align our subsequent development steps, and to consider and treat a sequence of SQL-executions as one whole.

Keywords: Model driven engineering · Business model · Software development · Statics · Domain model · Conceptual data model · Database model · Dynamics · User wish · User story · Use case · System sequence description · MVC-pattern · (Stored) procedure

1 Introduction

Nowadays new user wishes must be implemented very quickly. Over the last decades, their 'time-to-market' had to become shorter and shorter. This 'need for speed' poses a real challenge and an increasing problem for system development. It led from *waterfall* to *incremental*, *agile,* and even *continuous* development. Moreover, in the beginning of a software project requirements are seldom clear, unambiguous, complete, etc. Therefore, we treat the challenging research question how to come from elementary user wishes and simple domain models all the way to concrete SQL-specifications in a quick, straightforward, and also traceable way [1, 2]. The essence of the answer to our question will be: By *stepwise clarification*, *stepwise refinement* and *stepwise specification.* To speed up development, the development steps should be carefully chosen and be well-aligned.

B. Shishkov (Ed.): BMSD 2021, LNBIP 422, pp. 97–117, 2021.
https://doi.org/10.1007/978-3-030-79976-2_6

We follow the classical distinction between the *static* part and the *dynamic* part of the system under development. The *static* part refers to the *data structures* and the *dynamic* part to the *processes*. They can be considered as the two sides of the same coin.

To answer the old question ‘*Should we be Data-oriented or Process-oriented?*’: You should do both, concurrently! The data structures and the processes must be (and stay) mutually consistent. But the data structures are usually more stable than the processes.

Furthermore, we will distinguish between the *Problem Analysis* part and the *Software Design* part of system development. The *Problem Analysis* part must be (and will be) implementation-independent. In this paper, the *Software Design* part is geared towards SQL. We will also explain how the different aspects are coordinated.

To search for other UC-based approaches, we studied the solid literature review [3] and many of its cited papers. [3] constitutes a systematic literature review concentrating on use case specifications research. It thoroughly examined almost 120 papers on use case specifications, including their strengths and weaknesses. In it, we could not find a similar comprehensive and in-depth approach towards use case specifications with a concrete design follow-up towards SQL, not even in the industry white paper [4] of the Oracle corporation. Based on the papers [5–8], Tiwari et al. conclude in [3] that ‘*unavailability of formal representation of some natural language may result in confusion, difficulties and varied opinions in understanding the user requirements*’. There exist many papers about automatic translation to SQL regarding individual **queries** that are *well-formulated* in natural language [9]. However, our current paper is NOT limited to queries, NOT limited to individual interactions, and NOT about already well-formulated interactions.

Regarding the applicability of our approach: The feasibility study [10] works out in all detail a substantial part of Larman’s large and known *Process Sale* example [11]. That technical report gives a good impression of the applicability and scalability of our approach in large, complex real-life situations. We also applied our approach to various other kinds of examples, such as *control systems*, where the emphasis is on the *processes* and less on the *data structures* [12]. And meanwhile we worked out (and taught our students) several ‘common development patterns’. We successfully taught this approach already to a few hundred students, who applied to it various cases. Moreover, the approach is based on more than 40 years of development experience of the author.

We will illustrate the steps in our development approach with a carefully designed running example. Along the way, the running example will unfold step by step. We work out everything in detail because, as you know, the devil is in the details. And this especially holds when developing software.

The rest of the paper runs as follows: In Sect. 2 we give a general overview of the development path. Section 3 treats the *Problem Analysis* regarding the *Statics/Data-structures* and is implementation-independent. Section 4 subsequently treats the *Software Design* regarding the *Statics/Data-structures* and is geared towards SQL. Section 5 treats the *Problem Analysis* regarding the *Dynamics/Processes* and is implementation-independent. Section 6 treats the *Software Design* regarding the *Dynamics/Processes* and is geared towards SQL. The paper ends with an overview of its contributions (Sect. 7).

2 Overview of Our Development Path

We start with a general overview of our development approach. As we recall, explain, and illustrate in Sect. 3, for the *static* part of the system under development we can start with a *simple* and *small* domain model. The domain model can start simple because it might initially only contain concepts and their associations, (later) to be extended with the properties of the concepts and the multiplicities of the associations. The domain model can start small because it might initially contain only very few concepts and their associations, and extended later with more concepts and associations, following an *incremental*, *agile*, or *continuous* development.

To reach a full-fledged conceptual data model, each many-to-many association must be transformed into a few many-to-one associations, references must be made explicit, uniqueness properties must be added, and it must be indicated per property whether a value is required or optional. Last but not least, the possible values per property must be determined and there might be some remaining (integrity) constraints to be added as well.

Once we have such a detailed conceptual data model, we can prepare to transform it to an SQL-database. First, each reference to a concept is replaced by a uniqueness property of that referenced concept. After that, the resulting data model leads in a straightforward way to a default SQL-specification: First of all, a *database* is created. Then each concept translates to a *table* and each property of a concept translates to an *attribute* in that table, followed by 'NOT NULL' if a value is required for that property, else followed by 'NULL'. Each uniqueness condition translates to a *primary key* constraint or a *unique* constraint, each reference condition translates to a *foreign key* constraint, and other remaining integrity constraints translate to *check* constraints. This is shown and explained in detail in Sect. 4.

For the 'dynamic' part of the system under development we have to implement (very) many user wishes. As we explain and illustrate in Sect. 5, each time we will take an elementary User Wish (eUW) as a starting point, for example *Register a Student* or *Process a Sale*. Such a User Wish will be further developed by *stepwise clarification*, *stepwise refinement*, and *stepwise specification*: When we add the actor role and the reason for the User Wish then we get the familiar notion of a User Story [13–15]. A User Story is often formulated as '**As a** <actor role>**, I want to** <user wish> [**so that** <reason>]' where the reason-part is optional. A User Story (US) can be worked out into a Use Case, which consists of a Main Success Scenario (MSS) and zero or more Alternative Scenarios [16, 17]. A use case (UC) roughly corresponds to an *elementary business process* in business process modelling [11, 18]. Up to this point in the development, this all can be expressed by - and discussed with - the users in their own (natural) language.

To integrate the different scenarios of a Use Case into one structure, we use a *System Sequence Description* (SSD). An SSD is a kind of stylised Use Case which schematically depicts the interactions between the primary actor (user), the system (as a black box), and other actors (if any), including the messages between them. An SSD is usually drawn as a (UML-)diagram, see [11], but we introduce and prefer a *textual* SSD (tSSD) instead.

In Sect. 6 we explain and illustrate how tSSDs can be transformed to SQL using (stored) procedures in case of a Database Management System based on SQL [19, 20].

Now we give a bird's-eye view of our development approach and the order of steps we just sketched. We also indicate which 'arrow' (transformation) is treated in which section:

Topic	**Problem Analysis** → **Software Design**
Statics / Data structures (System has to **know**)	Domain Models* → Conceptual Data Model → Database Model §3 §4
Dynamics / Processes (System has to **do**)	e UW → US → UC (= MSS + AS*) → tSSD → SQL-Procedures §5 §5 §5 §6

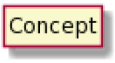
*: zero or more

3 From Simple Domain Models to Conceptual Data Model

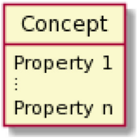
Regarding the question what the system under development must 'know' (i.e., which persistent data), an analyst often starts with developing a simple domain model. A Domain Model is a *visual* representation of the concepts, their properties, and their associations that might be relevant for the application to be developed (i.e., 'as we understood it until now'). The possible ingredients of a Domain Model are:

- concepts (a.k.a. conceptual classes),
- their relevant properties (a.k.a. their attributes), and
- their mutual associations (a.k.a. their relationships)

A Domain Model is usually drawn as a graph, consisting of nodes (for *concepts*, optionally with their *properties*) and lines (for *associations*). Although there are other popular ways to draw a domain model, e.g., using Entity-Relationship Diagrams [21], the ingredients could look as follows:



Concept only



Concept with properties

Association between concepts

where the phrase xxx (usually a verb phrase or preposition) indicates the association, the symbol ▶ indicates the reading direction, and m and n are multiplicities, usually '1', '0..1' (at most 1) or '*' (0 or more, a.k.a. 'many').

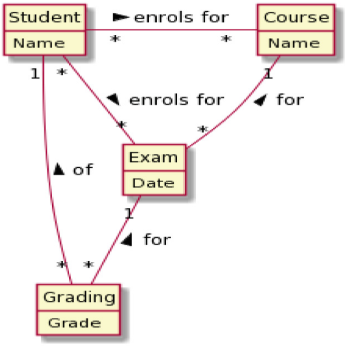
The ingredients of a Domain Model should be expressed in the terms as used in the application domain concerned. An early domain model represents a kind of minimum knowledge ('what we understood until now') and grows over time, sketching/making new versions. A series of simple, small domain models may help to structure the potentially unstructured information as provided by the users. The properties of the concepts and multiplicities of the associations need not be present in the Domain Model initially.

We will illustrate our development approach with a running example, which will be developed step by step.

Example 1: A simple domain model

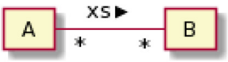
Our running example concerns a university and is about *courses*, *students*, their *exams*, and their *grades*. Courses can have exams. Students can enrol for courses and for exams. Students can get graded on an exam. We at least need to know the *name* of each student and of each course, the *date* of each exam, and the *grade* after each grading.

This leads to the simple domain model depicted on the right.

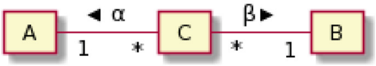


A many-to-many association (i.e., an association with a '*' on both sides) represents a 'hidden' concept, about which we need to know more. For instance, with respect to the m-to-m (many-to-many) association *Student enrols for Course* we must also know *which* students enrolled for *which* courses. We can transform any m-to-m association into two 'many-to-1' associations as follows, making the hidden concept explicit:

Transform



into



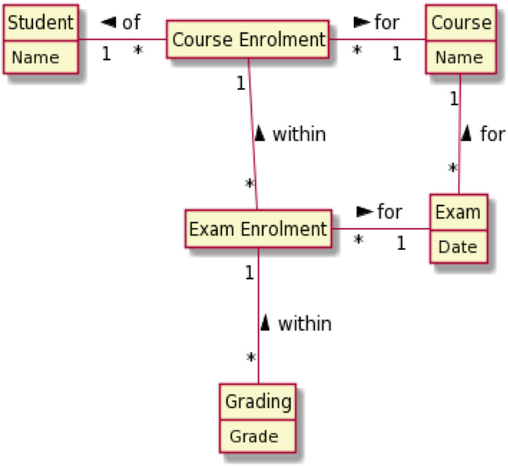
For instance, if A = 'Student', xs = 'enrols for', and B = 'Course'.
then we get C = 'Enrolment', α = 'of', and β = 'for'.

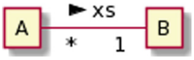
Example 2: The domain model with the hidden concepts made explicit

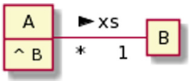
Example 1 has two many-to-many associations. After the two transformations we have two new concepts, *Course Enrolment* and *Exam Enrolment*.

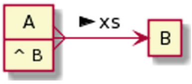
A student can only enrol for an exam if (s)he was *enrolled for* the corresponding course. And a student can only get a grade for an exam if (s)he was *enrolled for* that exam.

These transformations and extra ‘business constraints’ lead to the domain model on the right.



A many-to-one association  implicitly states that there is exactly one B related to each A.

Going to a Conceptual Data Model, that B must be indicated in A.  We will indicate that as follows:

To emphasize the functional relationship, we replace the *line* by a  many-to-one *arrow*. Then we can also leave out the multiplicities:

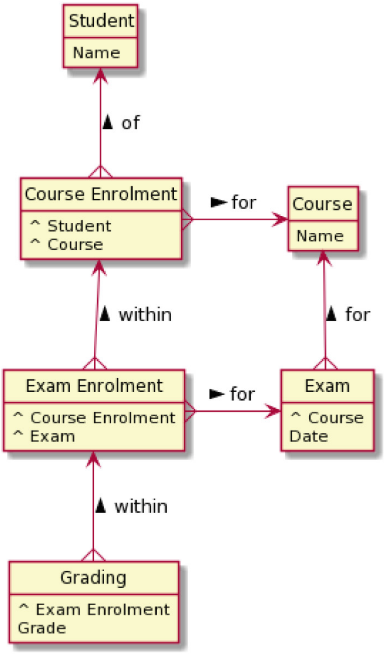
Next, per concept we must know and indicate by which (combinations of) properties each individual (a.k.a. ‘entry’) can be uniquely identified. We will indicate a uniqueness constraint by a ‘!’ in front of the properties involved; i.e., within each concept the value (combination) of the property(s) preceded by ‘!’ is unique. If there is another uniqueness constraint within the same concept, we will use ‘%’ in front of those properties involved.

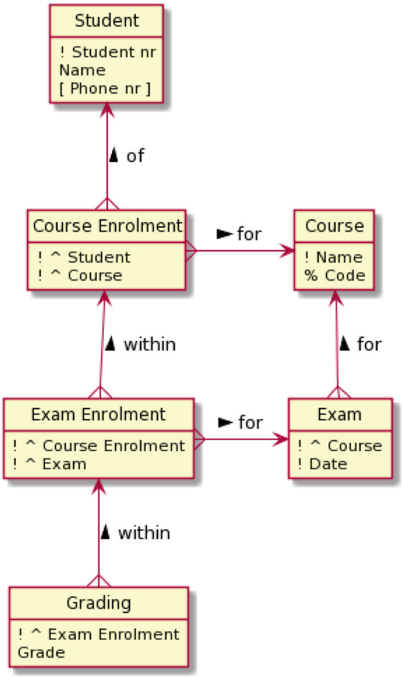
For each property we also have to know whether a value is required or optional. We will put properties for which a value is optional between the brackets ‘[’ and ‘]’.

Example 3: The references, uniqueness properties, and optionality made explicit

Example 2 has 6 many-to-one associations to be transformed. This leads to the next model, next page on the left.

After further requirements analysis for our running example: A *student* is uniquely identified by his/her student number, a *course* by its name but also by its course code, an *exam* by the combination of the course and the exam date, a *course enrolment* by the combination of the student and the course, an *exam enrolment* by the combination of the underlying course enrolment and the exam, and a *grading* by the underlying exam enrolment. Moreover, students might have a phone number. This all leads to the second model below, on the right.





Further analysis is needed to find out for each property what its possible values are. Finally, there might be some other constraints besides the ones already treated (i.e., uniqueness, references, optionality, and allowed values).

Example 4: The possible values per property and remaining constraints

Per concept in Example 3, the elicited details of the possible values for its properties are summed up below. The possible values for a property that refers to a concept implicitly follow from the concept it refers to. Note that the property lists below include all the info contained in the last graph in Example 3.

Student	/*
! Student nr	/* a natural number of 6 digits and divisible by 11 (for simple checks)
Name	/* a string in the Latin alphabet
[Phone nr]	/* a string of at most 20 characters (being a digit, ‘+’, ‘.’, or ‘ ’)
Course	/*
! Name	/* a string (in the Latin alphabet) of at most 50 characters
% Code	/* a combination of exactly 9 letters and digits
Exam	/*
! ^ Course	/* the Course the Exam is for
! Date	/* a date since the registration start (August 2010); maybe a future date
Course Enrolment	/* Enrolment of a Student for a Course
! ^ Student	/* the Student enrolled
! ^ Course	/* the Course enrolled for
Exam Enrolment	/* Enrolment for an Exam
! ^ Course Enrolment	/* the underlying Course Enrolment
! ^ Exam	/* the Exam enrolled for
Grading	/*
! ^ Exam Enrolment	/* the underlying Exam Enrolment
Grade	/* a natural number between 0 and 10, those two numbers included

There are no other constraints in this example. But if Course Enrolment (CE) and Exam Enrolment (EE) would have a date then we might have had the constraints that CE-date ≤ EE-date and EE-date < Exam date.

From Simple Domain Models to a Conceptual Data Model: Summary.
So, to come from a *domain model* to a full *conceptual data model*, we do as follows:

1. Replace the m-to-m associations in the domain model by many-to-1 associations
2. Extend the concepts with the references that follow from the associations in the (new) domain model
3. Add and indicate the properties following from the uniqueness discussions with the user organization
4. Indicate for which properties a value is optional, according to the user organization
5. Indicate the possible values for each property, after consulting the user organization
6. Add remaining constraints (if there are) after asking the user organization

The first two steps are more or less of a ‘mechanical’ nature. However, in the next steps (much) more requirements analysis is needed before you have a full conceptual data model, because a domain model is far from complete...

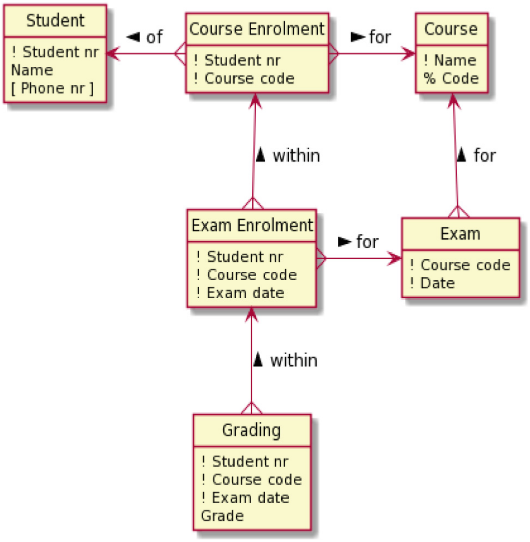
4 From Conceptual Data Model to SQL-Database

Once we have a detailed conceptual data model, it is pretty straightforward to transform it to an SQL-database. First of all, each reference to a concept is replaced by a uniqueness property of that referenced concept.

Example 5: References to a concept replaced by a suitable uniqueness property

In our running example, the concept *Course* has two uniqueness properties: *Name* is unique and *Code* too. We will use Code since it is more fundamental/stable.

From top to bottom, we replace '^Student' in Course Enrolment by 'Student nr', '^Course' in Course Enrolment by 'Course code', '^Course' in Exam by 'Course code', the combination '^ Course Enrolment' and '^ Exam' in Exam Enrolment by 'Student nr', 'Course code', and 'Exam date', and finally '^ Exam Enrolment' in Grading by 'Student nr', 'Course code', and 'Exam date'. This leads to the data model on the right.



When each reference is replaced by a uniqueness property of the referenced concept, the resulting data model leads in a natural way to a default SQL-specification:

- First, a declaration CREATE DATABASE <database name> is introduced
- Each concept translates to a *table*
- Each property of a concept translates to an *attribute* in that table with the corresponding *data type* followed by 'NOT NULL' if a value is required for that property, else followed by 'NULL';
- the precise syntax of these data types might be implementation-dependent
- Each uniqueness condition translates to a *primary key* or a *unique* constraint
- Each reference condition translates to a *foreign key* constraint
- Each extra constraint translates to a *check* constraint
- Each constraint also must get a name in SQL
- Each space in a concept or property name has been replaced by '_' to make it 1 word

We illustrate all this in Example 6. Often, a Database Management System (DBMS) automatically creates default indexes on some well-chosen table attributes in order to boost the performance of retrievals.

Example 6: The resulting data specification in SQL

Applying the rules, the model as specified until now leads quite naturally to the default SQL-code below. Constraint C1 expresses that Student_nr must consist of 6 digits, C2 that it must divisible by 11, C3 that Phone_nr must not contain a character which is not a digit, ‘+’, ‘.’, or ‘ ’, and C4 that Code must not contain a character which is not a letter or a digit.

```
CREATE DATABASE BMSD2021;

CREATE TABLE Student (
 Student_nr      INT             NOT NULL,          /* e.g. 123453           */
 Name            VARCHAR         NOT NULL,          /* e.g. John J. Smith    */
 Phone_nr        VARCHAR(20)     NULL,              /* e.g. +31.6.1234.5678  */
 CONSTRAINT C1 CHECK (100000 <= Student_nr AND Student_nr < 1000000),
 CONSTRAINT C2 CHECK (Student_nr  %  11 = 0),
 CONSTRAINT C3 CHECK (Phone_nr NOT LIKE '%[!0-9+. ]%'),
 CONSTRAINT K1 PRIMARY KEY (Student_nr)
);

CREATE TABLE Course (
 Name            VARCHAR(50)     NOT NULL,      /* e.g. Requirements Analysis  */
 Code            CHAR(9)         NOT NULL,      /* e.g. CS123BA02              */
 CONSTRAINT C4 CHECK (Code NOT LIKE '%[!a-z0-9]%'),
 CONSTRAINT K2 PRIMARY KEY (Code),
 CONSTRAINT K3 UNIQUE (Name)
);

CREATE TABLE Exam (
 Course_code     CHAR(9)         NOT NULL,              /* e.g. CS123BA02    */
 Date            DATE            NOT NULL,              /* e.g. 2020-10-10   */
 CONSTRAINT C5 CHECK ('2010-08-01' <= Date),
 CONSTRAINT K4 PRIMARY KEY (Course_code, Date),
 CONSTRAINT R1 FOREIGN KEY (Course_code) REFERENCES Course(Code)
);

CREATE TABLE Course_Enrolment (
 Student_nr      INT             NOT NULL,              /* e.g. 123453       */
 Course_code     CHAR(9)         NOT NULL,              /* e.g. CS123BA02    */
 CONSTRAINT K5 PRIMARY KEY (Student_nr, Course_code),
 CONSTRAINT R2 FOREIGN KEY (Student_nr) REFERENCES Student(Student_nr),
 CONSTRAINT R3 FOREIGN KEY (Course_code) REFERENCES Course(Code)
);

CREATE TABLE Exam_Enrolment (
 Student_nr      INT             NOT NULL,              /* e.g. 123453       */
 Course_code     CHAR(9)         NOT NULL,              /* e.g. CS123BA02    */
 Exam_date       DATE            NOT NULL,              /* e.g. 2020-10-10   */
 CONSTRAINT K6 PRIMARY KEY (Student_nr, Course_code, Exam_date),
 CONSTRAINT R4 FOREIGN KEY (Student_nr, Course_code)
   REFERENCES Course_Enrolment (Student_nr, Course_code),
 CONSTRAINT R5 FOREIGN KEY (Course_code, Exam_date)
   REFERENCES Exam(Course_code, Date)
);

CREATE TABLE Grading (
 Student_nr      INT             NOT NULL,              /* e.g. 123453       */
 Course_code     CHAR(9)         NOT NULL,              /* e.g. CS123BA02    */
 Exam_date       DATE            NOT NULL,              /* e.g. 2020-10-10   */
 Grade           TINYINT(3)      NOT NULL,              /* e.g. 7            */
 CONSTRAINT C6 CHECK (0 <= Grade AND Grade <= 10),

 CONSTRAINT K7 PRIMARY KEY (Student_nr, Course_code, Exam_date),
 CONSTRAINT R6 FOREIGN KEY (Student_nr, Course_code, Exam_date)
   REFERENCES Exam_Enrolment(Student_nr, Course_code, Exam_date)
);
```

Each of the two constraints C1 and C6 - each being a conjunction - could have been split into two constraints (which would lead to more refined error messaging).

5 From Elementary User Wish to SSD

Now we look at the ‘dynamic’ part of the system under development, i.e., the *processes* the system must support. Usually, (very) many user wishes have to be implemented. Informally, a User Wish (UW) is a ‘wish’, expressed in natural language, of a (future) user which the system should be able to fulfil. A UW often consists of an action verb and a noun (phrase). Examples of UWs in a university setting are *Register a Student*, *Enroll a Student for a Course*, *Update a Student Address*, *Enter a Grade*. Other examples are the following verb/noun-combinations:

> *Create/Retrieve/Update/Delete/Archive/Process/Handle* **a**
> *Customer/Product/Order/Sale/Supplier/Employee/…*

(Yes, indeed, the first 4 verbs are the well-known CRUD-operations.) We call such a UW without parameters an elementary user wish (eUW). Each time we will take an elementary User Wish as a starting point for development. Such a user wish will be developed by *stepwise clarification*, *stepwise refinement*, and *stepwise specification.* A parameterized user wish (pUW), another result of *stepwise refinement*, is an elementary user wish extended with its relevant parameters, e.g., the wish to ‘*Register a student with a given name, address, gender, and maybe phone number*’ (because you must specify *what* to register of a student). However, the proper set of parameters might only become clear (grow and change) during development.

When we add the actor role and the reason to a User Wish then we arrive at the familiar notion of a User Story (US), often expressed as ‘**As a** <actor role>**, I want to** <user wish> [**so that** <reason>]’ where the reason-part is optional [13]. A User Story can be worked out into a Use Case (UC), which consists of a Main Success Scenario (MSS) and zero or more Alternative Scenarios (AS); see [11, 16]. A Use Case roughly corresponds to an *elementary business process* in business process modelling [11].

We now summarize the refinement steps up to now: eUW → US → UC = MSS + AS*

Example 7: From User Wish via User Story to Use Case (= MSS + AS*)

We illustrate the refinement steps by working out the elementary User Wish ***Enter a grade*** into a User Story and then into a Use Case with a Main Success Scenario and four Alternative Scenarios in this case. Because data model and refinement steps should be in line with each other, we must keep the data model in mind. Note that those four ASs are in line with the Grading-part of the data model (see Example 4).

eUW1: **Enter a grade**

US1: As a lecturer, I want to Enter a grade so that the grade is officially registered

UC1: Enter a grade
Precondition: The user is authenticated as a lecturer and authorized for this UC.

MSS1:
1. The user asks the system to enter grade g for student s on exam e
2. The system tries to enter grade g for student s on exam e
3. The system informs the user about the result

Step 1 is the parameterized request, Step 2 the execution of the request, and Step 3 the result of the execution.

We have the following Alternative Scenarios:

AS1.1: At Step 1: As long as the grade is not (syntactically) correct - i.e., not a natural number between 0 and 10 - the user is asked to adapt it

AS1.2: At Step 2: If the student is unknown*
then the user is informed about that and 'nothing' happens

AS1.3: At Step 2: If the exam is unknown*
then the user is informed about that and 'nothing' happens

AS1.4: At Step 2: If student is known and exam is known but if the student is not enrolled for the exam then the user is informed about it and 'nothing' happens

*: By 'unknown' we mean unknown to the system (not represented in the system)

Note that up to now, this all can be expressed by - and discussed with - the user in its own (natural) language!

To integrate the different scenarios of a Use Case into 1 structure, we use a *System Sequence Description* (SSD). An SSD is a kind of stylised Use Case which schematically depicts the interactions between the primary actor (user), the system (as a black box), and other actors (if any), including the messages between them. An SSD is usually drawn as a (UML-)diagram, see [11]. However, we introduce *textual* SSDs (tSSDs) instead.

Our textual SSDs are meant as more formal representations of use cases, and used as a follow-up of use cases towards SW design. They integrate the different scenarios of a Use Case into one structure and have a formal syntax [22] and declarative semantics [23].

UML-diagrams can also be positioned between (*textual*) use cases and the final computer programs (which are also *textual*), but the UML-diagrams themselves are *graphical*. According to UML (https://www.omg.org/spec/UML), the semantics of UML defines how the UML concepts are to be realized by computers. Its sections on semantics are in fact explanations only. So, at best UML has some kind of *operational semantics* - see [24] for instance - but no formal, declarative semantics. Operational semantics is already looking forward to implementations, e.g., looking at execution models, intermediate states, parallelization, etc. However, this should not be in the analysis part.

It is important to note that [25] contains rules to translate *textual* SSDs systematically to *natural language* (English) as well as to *graphical* SSDs (more or less UML-diagrams). This can help to verify the integration result with the customer! Examples 9 and 10 will show such translation results.

In [22] a grammar for textual SSDs is proposed. We recall a part of that grammar below. The terminals are written in bold. The nonterminal A stands for 'atomic instruction' (step), P for 'actor' (or 'participant'), M for 'message', S for 'instruction' (or SSD), C for 'condition', N for 'instruction name', and D for 'definition':

A ::= P **➡** P**:** M /* *where* 'X ➡ Y: M' *means:* 'X sends M to Y'
P ::= **System** | **User** | …
S ::= A | S **;** S | **begin** S **end** | **if** C **then** S [**else** S] **end** | **while** C **do** S **end**
| **repeat** S **until** C | **do** N
D ::= **define** N **as** S **end**

The construct '**do** N' is known as an *Include* or a *Call*. We note that the values for the nonterminals P, M, and N are application dependent ('domain specific'), apart from the values **System** and **User** for P. The values for P, M, and N will appear naturally during the development of the specific application. The terminal **System** represents the system under consideration.

For atomic instructions we can distinguish the following situations:

1.	Actor **➡ System:** i	Elucidates the input messages the system can expect
2.	**System ➡ System:** y	Elucidates the transitions (or checks) the system should make
3.	**System ➡** Actor**:** o	Elucidates the output messages the system should produce
4.	Actor **➡** Actor2**:** x	A step outside the system (might be helpful in understanding)

where Actor ≠ **System** and Actor2 ≠ **System** (but Actor and Actor2 might be the same). We call step (a) an *input* step, (b) an *internal* step, (c) an *output* step, (d) an *external* step.

A quite common interaction pattern is: A *request*, followed by an *action*, followed by a *result* (message). In the above terminology: An *input step*, followed by an *internal step*, followed by an *output step.*

The different scenarios of a Use Case can now be integrated into 1 structure by using a textual SSD, as explained in [22] and illustrated in the next example. The refinement steps until now can be summarized as follows:

$$\mathrm{eUW} \rightarrow \mathrm{US} \rightarrow \mathrm{UC} = \mathrm{MSS} + \mathrm{AS}^{*} \rightarrow \mathrm{tSSD}$$

Example 8: The resulting tSSD after integration

The textual SSD for only the MSS looks as follows:

```
1.  User ➡ System: enter grade g for student s on exam e ;
2.  System ➡ System: EnterGrade(g, s, e) ;
3.  System ➡ User: Result
```

Now we must integrate the MSS and all the ASs for a complete Use Case.
We will do so by a textual SSD:

```
repeat
  User ➡ System: enter grade g for student s on exam e ;     /* Original user request
  System ➡ System: check whether g is correct ;                      /* ⎤
  if g is not correct then                                           /* |AS1.1
     System ➡ User: "The grade is not correct. Please adapt it" end  /* |
until g is correct ;                                                 /* ⎦
System ➡ System: check whether s is known ;                          /* ⎤AS1.2
if s is not known then System ➡ User: "Unknown student" end ;        /* ⎦

System ➡ System: check whether e is known ;                          /* ⎤AS1.3
if e is not known then System ➡ User: "Unknown exam" end ;           /* ⎦
if s is known and e is known                                         /* ⎤
then System ➡ System: check whether s is enrolled for e ;            /* |AS1.4
      if s is not enrolled for e                                     /* |
      then System ➡ User: "Student is not enrolled for the exam"     /* |
      end                                                            /* |
end ;                                                                /* ⎦
if everything was okay                          /* The system should keep track of that
then System ➡ System: EnterGrade(g, s, e) ;         /* The execution of the request
      System ➡ User: "Done"                         /* The execution result in this case
end                                                 /*
```

So, the complete Use Case starts with an input step followed by an internal check and maybe an output message. This will happen one or more times until the grade is (syntactically) correct. Then the system continues with several internal checks, each maybe followed by an output message. Finally, if everything was okay then the system does enter the grade and informs the user about it (via an output message).

We recall that [25] has rules to translate textual SSDs systematically to *natural language.*

Example 9: Translating the tSSD to natural language

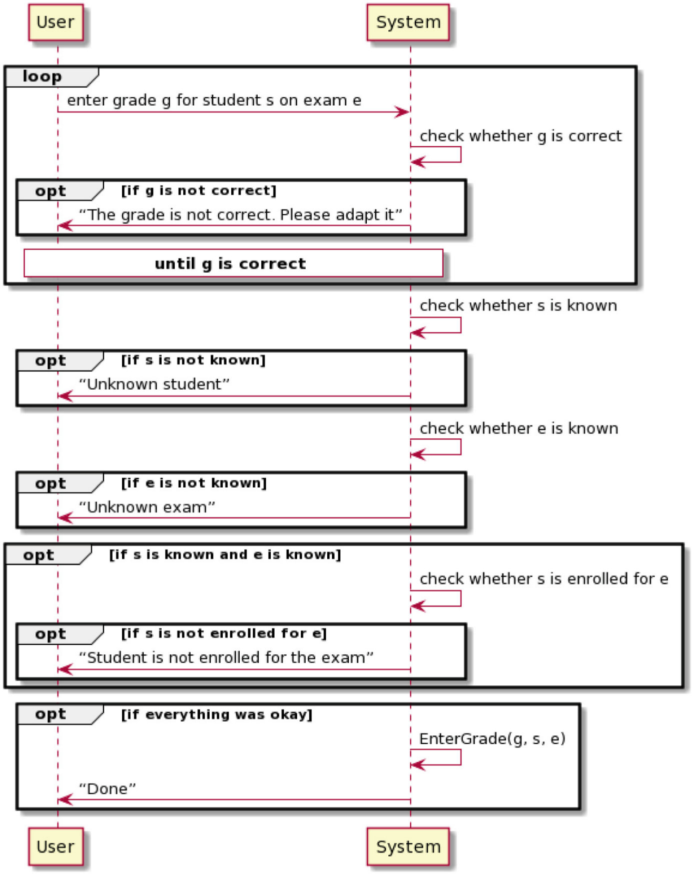
The rules from [25] to translate tSSDs to *natural language* (English) will result in:

```
Repeat
  the User asks the System to enter grade g for student s on exam e.
  The System does check whether g is correct.                                  ⎤1
  If g is not correct then                                                     ⎥
     the System sends "The grade is not correct. Please adapt it" to the User end ⎥
until g is correct.                                                            ⎦
The System does check whether s is known.                                      ⎤2
If s is not known then the System sends "Unknown student" to the User end.    ⎦
The System does check whether e is known.                                      ⎤3
If e is not known then the System sends "Unknown exam" to the User end.       ⎦
If s is known and e is known                                                   ⎤4
then the System does check whether s is enrolled for e.                        ⎥
      If s is not enrolled for e                                               ⎥
      then the System sends "Student is not enrolled for the exam" to the User ⎥
      end                                                                      ⎥
end.                                                                           ⎦
If everything was okay                       /* The system should keep track of that
then the System does EnterGrade(g, s, e).    /* The execution of the request
      The System sends "Done" to the User    /* The execution result in this case
end
```

We recall that we also have rules to translate *textual* SSDs systematically to *graphical* SSDs.

Example 10: Translating the textual SSD to a graphical SSD

The rules from [25] to translate *textual* SSDs to *graphical* SSDs will result in:



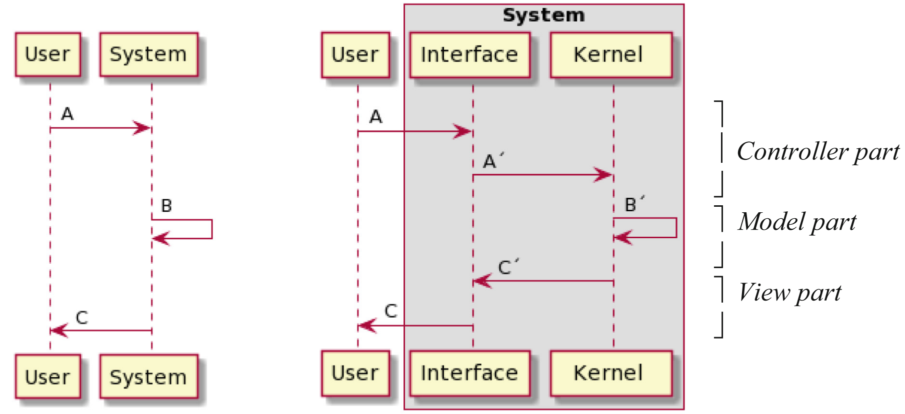
Summarizing tSSDs: A textual SSD schematically depicts the interactions between the primary actor (user), the system (as a black box), and other actors (if any), including the messages between them. A textual SSD integrates the different scenarios of a UC into one structure. A tSSD is written in a kind of 'structured natural language' and already exposes the final execution structure. Textual SSDs can be automatically translated back to *natural language* (such as English) as well as to *graphical* SSDs (more or less UML-diagrams), which is useful for verification purposes. Example 8 shows that a tSSD is already close to concrete programming, although it still is implementation-independent.

6 From Textual SSD to SQL-Procedures

To separate the *internal* representations in a system from the ways information is interchanged with an *external* actor, a system can (conceptually) be split into an 'interface' and a 'kernel'. The interface converts the input as received from an external actor into a proper call to the kernel (e.g., an OO-system or a relational DBMS) and it converts the output from the kernel into a proper message to the external actor. So, then the system is considered as a ‘grey box’ and no longer as a ‘black box’. This is related to the MVC-pattern (Model-View-Controller) a well-known software design pattern. We schematize it below. We indicate the Controller-, Model-, and View-part too:

Step	**Analysis**	**Design**	**MVC-part**
Input step	User ➡ System: A	User ➡ Interface: A Interface ➡ Kernel: A´	*Controller part*
Internal step	System ➡ System: B	Kernel ➡ Kernel: B´	*Model part*
Output step	System ➡ User: C	Kernel ➡ Interface: C´ Interface ➡ User: C	*View part*

We graphically illustrate these steps (in combination) by indicating how the analysis-SSD below, a common analysis interaction pattern, transforms into the design-SSD next to it.



Here A is an input message from the user, B expresses what the system must do, and C is an output message to the user. In the second diagram, A′ is a call to the kernel, B′ specifies the execution by the kernel, and C′ is the output from the kernel. So, the interface converts A to A′ (*Controller*) and C′ to C (*View*). The interface can be seen as a ‘front office’ and the kernel as a ‘back office’. The crux of the transformation is the specification of B′.

If the kernel is an SQL-DBMS then A′ is an SQL-call, B′ represents the SQL-execution, and C′ the SQL-output. Similarly if the kernel is an OO-system then B′ specifies an OO-execution (typically with *get*- and *set*-statements).

In order to make our SQL-design more resistant to all kinds of local SQL-dialects, we will use stored procedures in SQL. Then every SQL-call A′ can be a procedure call, i.e., the call of a (stored) procedure in SQL. An SQL-procedure might contain the typical SQL-statements SELECT, INSERT, UPDATE, and DELETE, but also control-of-flow language and calls to (other) procedures. A stored procedure will be compiled and gets an execution plan, which dramatically improves its performance.

In our next example we illustrate how a tSSD can be transformed into SQL.

Example 11: The resulting SQL-procedure needed for the textual SSD

The tSSD in Example 8 has only one input step, so we need only one procedure (though that procedure might be called repeatedly). The tSSD starts with an input step, followed by an internal check and maybe an output message. If the grade is not (syntactically) correct then the procedure is called again (until the grade is correct), and else the system continues with several internal checks, each maybe followed by an output message. Finally, if everything is okay then the system does enter the grade and informs the user about it via an output message.

Note that the resulting SQL-procedure below follows the structure of the tSSD. In the SQL-procedure, @output is declared as a return parameter. We recall that an exam is uniquely identified by the course and exam date.

```
CREATE PROCEDURE EnterGrade @g tinyint(3), @s int, @cc char(9),
                        @ed Date, @output varchar(50) OUTPUT AS
BEGIN        /* Invariant: @output = ‘’ ⇔ Everything is okay until now */
   SELECT @output = ‘’
   IF NOT (0 <= @g AND @g <= 10)
      THEN SELECT @output = ‘The grade is not correct.
                              Please adapt it. ’
      ELSE
   IF @s NOT IN (SELECT Student_nr FROM Student)
      THEN SELECT @output = ‘Unknown student. ’
   IF (@cc, @ed) NOT IN (SELECT Course_code, Date FROM Exam)
      THEN SELECT @output = @output + ‘Unknown exam. ’
   IF @output = ‘’
      THEN IF (@s, @cc, @ed) NOT IN (SELECT Student_nr,
                Course_code, Exam_date FROM Exam_Enrolment)
          THEN SELECT @output = ‘Student not enrolled for exam.’
   IF @output = ‘’                    /* i.e., if everything was okay */
      THEN BEGIN INSERT INTO Grading VALUES(@s, @cc, @ed, @g)
                 SELECT @output = ‘Done. ’
           END
END
```

On hindsight we overlooked the scenario that if ‘Everything was okay’ (i.e., known student was indeed enrolled for known exam), the grade could have been in the system already. But thanks to the uniqueness constraint K7 (see Example 6), the kernel would have raised an error message (see position C′ in the diagram on the previous page). Generally speaking, all the constraints specified in the declaration of the database will guard the system’s contents, even if some scenarios are overlooked in some use cases.

7 Contributions

First, the introduction of the notion of elementary User Wish allowed us to start development paths in an early phase of system development. The notion is concrete, simple to understand, and well-discussable with the user organization.

We recall that a Use Case consists of a Main Success Scenario plus zero or more Alternative Scenarios, all being texts. In the end, they must be integrated into one (computer) program, also being text. Then the question arises: What should come on the dots below to *integrate* all the scenarios and to have *aligned* development steps?

(UC =) MSS + AS* (texts) ⇨ ………….. ⇨ Program (text)

We put textual SSDs in between (instead of, e.g., graphical SSDs such as a UML-diagrams).

Then we get: (UC =) MSS + AS* (texts) ⇨ tSSD (text) ⇨ Program (text)

instead of:
(UC =) MSS + AS* (texts) Program (text)
⇩ ⇧
Several gSSDs (*diagrams*)

So, to solve the *integration problem* and the *alignment challenge*, we use the notion of *textual* SSDs. They play a crucial role to obtain integration and alignment. Textual SSDs are theoretically sound: They have a well-defined syntax [25] as well as a well-defined semantics [23], as opposed to many other 'formalisms' (such as UML-diagrams). Textual SSDs can be automatically translated to natural language (e.g., English) and also to well-formed graphical SSDs [25], for instance for verification purposes. So, in that case we get the following feedback loops for verification:

User: eUW → US → UC (= MSS + AS*) → tSSD
↰ ⇐ ⇐ *Text in Natural Language* ⇐ ⇐ ↲

and

User: eUW → US → UC (= MSS + AS*) → tSSD
↰ ⇐ ⇐ ⇐ *One graphical SSD* ⇐ ⇐ ⇐ ↲

Because the grammar for tSSDs aligns with those for imperative and declarative programming languages, tSSDs form a suitable basis for translations to (computer) programs. Although implementations often use *imperative* (object-oriented) languages, we considered translations to SQL, a *declarative* database language. Authors such as Jacobson [17] and Cockburn [16] don't go all the way to concrete code, as we do. We made use of *(stored) SQL-procedures*, which are quite performant. It allowed us to treat a sequence of executions as one whole, which is very helpful.

Our approach concurrently takes into account the *static* part (i.e., the *data structures*) and the *dynamic* part (i.e., the *processes*) of the system to be developed.

By *stepwise clarification*, *stepwise refinement*, and *stepwise specification*, an aligned straightforward development path for processes resulted:

User: eUW → US → UC = MSS + AS* (texts) → tSSD (text) → SQL-procedures (text)

As a consequence of the straightforward transformations and the alignment, our approach contributes to the (bi-directional) traceability of the generated artifacts as well [1, 2, 25]. The approach also brings semi-automatic software generation closer. Our contribution is not only in the individual steps, but also in their (new) *combination*, i.e., in the *choice/ selection* and the *alignment* of these steps.

References

1. Gotel, O.C.Z., Finkelstein, C.W.: An analysis of the requirements traceability problem. In: Requirements Engineering, pp. 94–101 (1994). http://discovery.ucl.ac.uk/749/1/2.2_rtprob.pdf
2. Cleland-Huang, J., et al.: Software and Systems Traceability. Springer, Heidelberg (2012). https://doi.org/10.1007/978-1-4471-2239-5
3. Tiwari, S., Gupta, A.: A systematic literature review of use case specifications research. Inf. Softw. Technol. **67**, 128–158 (2015). https://www.sciencedirect.com/science/article/abs/pii/S0950584915001081
4. Kettenis, J.: Getting started with use case modeling. White Paper, Oracle (2007). https://www.oracle.com/technetwork/developer-tools/jdev/gettingstartedwithusecasemodeling-133857.pdf
5. Fantechi, A., Gnesi, S., Lami, G., Maccari, A.: Applications of linguistic techniques for use case analysis. Require. Eng. **8**(3), 161–170 (2003)
6. Ilieva, M., Ormandjieva, O.: Automatic transition of natural language software requirements specification into formal presentation. In: [26], pp. 392–397 (2005). https://doi.org/10.1007/11428817_45
7. Savic, D., Antovic, I., Vlajic, S., Stanojevic, V., Milic, M.: Language for use case specification. In: Proceedings of the 34th IEEE Software Engineering Workshop, SEW, pp. 19–26 (2011)
8. Sinha, A., Paradkar, A., Kumanan, P., Boguraev, B.: A linguistic analysis engine for natural language use case description and its application to dependability analysis in industrial use cases. In: Proceedings of the IEEE/IFIP International Conference on Dependable Systems Networks, DSN 2009, pp. 327–336 (2009)
9. Kim, H., So, B.-H., Han, W.-S., Lee, H.: Natural language to SQL: where are we today? Proc. VLDB **13**(10), 1737–1750 (2020)
10. de Brock, E.O.: Converting a non-trivial Use Case into an SSD: an exercise. SOM Research Report 2018011, University of Groningen (2018)
11. Larman, C.: Applying UML and Patterns. Pearson Education, London (2005)
12. de Brock, E.O.: What to do when requirements are changing all the time? A control system example. In: Shishkov, B. (ed.): International Symposium on Business Modeling and Software Design (BMSD). LNBIP, pp. 317–329 (2021)
13. Lucassen, G., Dalpiaz, F., Werf, J.M.E.M.V.D., Brinkkemper, S.: The use and effectiveness of user stories in practice. In: Daneva, M., Pastor, O. (eds.) REFSQ 2016. LNCS, vol. 9619, pp. 205–222. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30282-9_14
14. Lucassen, G.G.: Understanding user stories. Ph.D. thesis, Utrecht University (2017). https://dspace.library.uu.nl/handle/1874/356784
15. Cohn, M.: User Stories Applied: For Agile Software Development. Addison (2004). https://www.pearson.com/us/higher-education/program/Cohn-User-Stories-Applied-For-Agile-Software-Development/PGM314163.html
16. Cockburn, A.: Writing Effective Use Cases. Addison Wesley (2001). https://www.infor.uva.es/~mlaguna/is1/materiales/BookDraft1.pdf

17. Jacobson, I., et al.: Use case 2.0: the guide to succeeding with use cases. Jacobson Int. (2011). https://www.ivarjacobson.com/publications/white-papers/use-case-ebook
18. Dumas, M., et al.: Fundamentals of Business Process Management. Springer, Heidelberg (2018). https://doi.org/10.1007/978-3-662-56509-4. https://www.springer.com/gp/book/9783662565087
19. Ullman, J.D., et al.: Database Systems: The Complete Book. Pearson, London (2009)
20. Elmasri R., Navathe S.B.: Fundamentals of Database Systems. Pearson (2016). https://www.pearson.com.au/products/D-G-Elmasri-Navathe/D-G-Elmasri-Ramez-Navathe-Shamkant-B/Fundamentals-of-Database-Systems-Global-Edition/9781292097619?R=9781292097619
21. Chen, P.: The entity-relationship model - toward a unified view of data. ACM Trans. Database Syst. **1**(1), 9–36 (1976)
22. de Brock, E.O.: From business modeling to software design. In: [28], pp. 103–122 (2020). https://doi.org/10.1007/978-3-030-52306-0_7
23. de Brock, E.O.: Declarative semantics of actions and instructions. In: [28], pp. 297–308 (2020). https://doi.org/10.1007/978-3-030-52306-0_20
24. Övergaard, G., Palmkvist, K.: A formal approach to use cases and their relationships. In: Bézivin, J., Muller, P.A. (eds.) The Unified Modeling Language. «UML»'98: Beyond the Notation. LNCS, vol. 1618, pp. 406–418. Springer, Berlin (1998).https://doi.org/10.1007/978-3-540-48480-6_31
25. de Brock, E.O.: On System Sequence Descriptions. In [27] (2020)
26. Montoyo, A., Munoz, R., Mtais, E. (eds.): Natural Language Processing and Information Systems. LNCS, vol. 3513, Springer, Heidelberg (2005)
27. Sabetzadeh, M., et al. (eds.): Joint Proceedings of REFSQ-2020 Workshops, Doctoral Symposium, Live Studies Track, and Poster Track. Pisa, Italy (2020)
28. Shishkov, B. (ed.): International Symposium on Business Modeling and Software Design (BMSD). Lecture Notes in Business Information Processing, vol. 391 (2020)