Towards pattern-driven requirements engineering

de Brock, Bert

Published in:
Model-Driven Engineering Workshop (MoDRE)

DOI:
10.1109/MoDRE.2018.00016

Publication date:
2018

Towards pattern-driven requirements engineering: Development patterns for functional requirements

Bert de Brock
University of Groningen
Groningen, The Netherlands
E.O.de.Brock@rug.nl

***Abstract*—A recent paper answered the question how to come from initial user wishes up to a running system in a straightforward, transparent, modular, traceable, feasible, and agile way. That paper sketched a complete development path for functional requirements, starting from *user stories* via *use cases* and their *system sequence diagrams* to a so-called *information machine* and then to a realization, an *information system*.**

To support that promising approach and increase its effectiveness, we now introduce *development patterns* for such development paths (focusing on functional requirements). We present the basic idea, several generally applicable development patterns (including patterns for the important and well-known class of CRUD functions), and various examples. This leads us into the direction of Pattern-Driven Requirements Engineering (PaDRE).

To reach our goal we had to cross the boundaries of several (sub)disciplines such as requirements engineering, machine theory, and (database) systems development. Although we used (variants of) many existing ingredients, the strength of our approach also lies in the *combination* of the ingredients chosen (and the ones ignored).

***Index Terms*—Requirements engineering, user story, use case, system sequence diagram, information machine, information system, development path, development pattern, CRUD.**

I. Introduction

There are several ways (when we also consider various possible intermediate steps) to come from initial user wishes to a running system, even when we limit ourselves to the functional requirements. In [1] a straightforward development path for functional requirements is sketched. As pointed out there that development path is complete, transparent, allows an agile and traceable approach, and leads to apparently modular systems.

To support that approach and increase its effectiveness, the current paper introduces *development patterns* for such development paths (focusing on functional requirements). As generally known, patterns can enhance productivity.

Section II recalls and illustrates the necessary basic notions from [1]. Section III presents the (incremental) *development path* proposed in [1].

Section IV presents several *development patterns* and forms the core of this paper. It starts with the introduction and explanation of a very general development pattern (in subsection A). This general development pattern will be illustrated under B.

Under C ('*Me, Myself, and I*') we present a general development pattern for the often-occurring situation that the user him/herself represents a 'hidden' parameter (value), a special case of the very general development pattern of subsection A.

In subsection D we recall the four important basic functions generally applicable to data in a system, known in the literature as CRUD (Create, Read, Update and Delete). They are representative for many use cases. In subsection E we elaborate on general development patterns for the CRUD functions. In principle, those CRUD patterns are special cases of the general development patterns sketched in subsections A and C.

II. Basic Notions

Following [1] we sketch the basic notions for our development patterns for functional requirements, starting from *user stories* via *use cases* and their *system sequence diagrams* to a so-called *information machine* and then to a realization, an *information system*.

Informally, a **user story** (US) is a 'wish' of a (future) user which the system should be able to fulfil [2], e.g., the wish of a university employee to '*Remove a student*'. Initially written USs might need to be improved (refined/detailed/ completed) to clarify what the system should do. E.g., the wish above could be sharper: '*Remove a student with a given student number*'. So, the user should also indicate *which* student to remove. According to [3] user stories are popular as a method for representing requirements, especially in agile development environments, e.g., using a simple (but popular) template like

'*As a <role>, I want to <wish> [so that <benefit>]*'

originating from [4]. For our example this would be: '*As an administrator, I want to Remove a student with a given student number*', leaving out the optional *benefit*-part.

A **use case** (UC) is a text in natural language that describes the consecutive steps in a typical usage of the system [5, 6]. A UC roughly corresponds to an *elementary business process* in business process engineering [7]. For structuring purposes larger UCs are often refined into smaller parts (sub-functions or 'sub use cases'), usually factoring out duplicate sub-steps shared by other UCs [7]. Further elaborating our US '*Remove a student with a given student number*' could lead to the UC

1. *An administrator (the 'user') asks the system to Remove a student with a given student number*
2. *The system removes the student info (only if the student was known to the system)*
3. *The system informs the administrator that it did it (or that the student number was unknown)*

978-1-5386-8406-1/18/$31.00 ©2018 IEEE
DOI 10.1109/MoDRE.2018.00016

A **system sequence diagram** (SSD) of a UC is a 'diagram' that depicts the interaction between the primary actor (user), the system, and other actors (if any), including the messages (with their parameters) between them; e.g., see [7]. An SSD is a kind of stylised UC that makes the prospective inputs, state changes, and outputs more explicit. Although SSDs can be drawn in much fancier ways (e.g., see [7, 8]) we only draw their bare essence (because that eases their analysis). For example, an SSD for (a refined version of) the UC just given:

- *User → System: RemoveStudent(<number>);*
- ***if** the student number is known to the system*
  ***then** System → System: remove the student info;*
  System → User: "Done"
  ***else** System → User: "Unknown student number"*

USs, UCs and their corresponding SSDs can be expressed in the natural language of the user (say English or Dutch).

An **information machine** is a 5-tuple (I, O, S, G, T) consisting of:

1. a set I (of *inputs*)
2. a set O (of *outputs*)
3. a set S (of *states*)
4. a function G: S x I → O (the *output function*), mapping state-input pairs to the corresponding output
5. a function T: S x I → S (the *transition function*), mapping state-input pairs to the corresponding next state

The notation f: X → Y indicates that f is a function with X as its domain and its range being a subset of Y. (An information machine (IM) is equivalent to a – not necessarily finite – Mealy machine without a special start state; see [9, 10].)

A state often consists of several sets, e.g., a set representing students, a set representing lecturers, a set for courses, etc. Therefore, we might model a state as a function over a collection of labels that assigns such sets to the labels. E.g., s(STUD) might represent the set of students in state s. An element of such a set, e.g., t ∈ s(STUD) representing an individual student, often is modelled as a function as well, a function assigning to each relevant property the value for that element, e.g., t(NR) representing the student number.

We can distinguish three types of basic SSD steps – each represented in our earlier SSD example – but now with 'User' generalized to 'Actor', where an actor can be any other system or person with which the system communicates. Those three basic SSD steps and their relationship with an IM are:

Actor → System:	Elucidates part of the set I of inputs
System → Actor:	Elucidates part of the set O of outputs and of the output function G of the IM
System → System:	Elucidates part of the set S of states and of the transition function T of the IM

For instance, our earlier SSD example '*Remove student with a given student number*' leads to inputs of the form *RemoveStudent(<number>)* and to the outputs "Unknown student number" or "Done". Suppose in this case that each state s ∈ S contains STUD as a label for the set of students and NR as a property label for the elements of s(STUD). Now we can easily read from the SSD that if the student number is known to the system, that is, if $n \in \{ t(NR) \mid t \in s(STUD) \}$ and i = RemoveStudent(n) then output G(s, i) will be "Done" and the student table in the new state T(s, i) will be $\{ t \in s(STUD) \mid t(NR) \neq n \}$. Otherwise, output G(s, i) will be "Unknown student number" and the new state T(s, i) will be s; i.e., the state stays the same.

An IM is a kind of *blueprint*. It is a useful intermediate mathematical model, both as a clear, unambiguous capture of the user wishes regarding the functional requirements as well as a formal model of the system to be built.

An IM can have quite different realizations. For instance, an IM can be realized by a human servant (say a clerk), by an 'SQL servant' (i.e., a computer with SQL software), or by a 'Java servant' (i.e., a computer with Java software). We often illustrate realizations using SQL specifications.

We call the realization of an IM an **information system** (IS), i.e., 'an organized system for the collection, organization, storage and communication of information' [11, 12]. Output G(s,i) and state T(s,i) can be seen as the *postconditions* for input i. The aforementioned *sets* and their *elements* in a state of an IM translate to *classes* and *objects* in an OO system and to *tables* and *tuples* in a relational database system.

For a realization of user stories, we might use *methods* in an OO system or (stored) *procedures* in a relational system. We should look at the earlier SSD and the details in the IM. E.g., translating our user story '*Remove a student with a given student number*' to, say, SQL leads naturally to the procedure below. (Parameter names are preceded by an "@" in SQL.)

```
CREATE PROCEDURE RemoveStudent @n INTEGER,
                 @output VARCHAR OUTPUT   AS
IF @n IN (SELECT NR FROM STUD)
THEN DELETE FROM STUD t WHERE t.NR = @n;
     SELECT @output = "Done"
ELSE SELECT @output = "Unknown student number"
```

III. Development Path

Starting from a set of USs via their UCs and their corresponding SSDs we can define an IM, which is a blueprint for a realization, an IS. Interaction modelling mainly takes place in UCs and SSDs. In an incremental/agile development environment, developers will start with a simple, small version and extending/adapting it in several small steps into larger, more sophisticated versions. [1] presents the following general scheme (where the arrows indicate what is input for what):

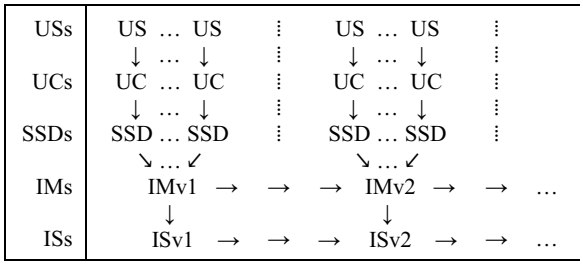

Fig. 1. Proposed development path (incrementally)

In other words, via one or more USs, UCs, and their corresponding SSDs (and the previous version of the IM) we

can define an initial (resp. next) version of the IM and, based on the IM (and the previous IS-version), we can define an initial (resp. next) version of the IS. Altogether, this forms a straightforward and complete development path.

So, we can ignore all kinds of separate intermediate (logical) languages for instance. Essentially, this is because we translate from the natural language of the user (say English or Dutch) into the language of the system (say Java or SQL), with a precise mathematical model as an intermediate stage.

As indicated in [1], this approach is applicable in an agile development of functional requirements too, when a simple 'core' scenario (or 'Main Success Scenario') of a - maybe yet unclear - 'full' use case might be delivered first, followed by 'fuller'/improved/refined versions in subsequent cycles (see [7]). Hence, previously developed US/UC/SSD-triples might be incrementally extended in later development rounds.

IV. Development Patterns

After having worked out many cases along these lines we could recognize certain 'development *patterns*'. We will treat some of them here.

A. A General Development Pattern

We start on a very general level. First, we consider the popular general template for a **user story** taken from [4]:

As a <role>, I want to <wish> [so that <benefit>]

According to [3], 70% of the practitioners use this template. For the development of functional requirements, we primarily look at the *wish*-part. The *role*-part becomes relevant in case of a 'personal wish' (see subsection C) and (later) when *privileges* have to be granted to roles (that can be assigned to concrete users afterwards). We will ignore the optional *benefit*-part.

Starting from a user story we consider a use case as the next step in the development of functional requirements, with the *<role>*-part of a US corresponding to the *primary actor* of a UC and the *<wish>*-part corresponding to the *title* of the UC !

For the general US-template given above the main steps of a corresponding **use case** scenario could be as follows:

1. *A <role> (the 'user') asks the system to <wish>*
2. *The system tries to <wish>*
3. *The system informs the user about the result*

We emphasize that the system *tries to* execute the wish (see Step 2) because that is not always possible, e.g., due to violations of constraints or presuppositions (e.g., try to update the address of a student who is not known to the system).

For the above UC-template the **SSD** could look as follows:

1. *User → System: <name of wish>(<parameters>)*
2. *System → System: try to <wish>*
3. *System → User: result*

The name of the wish introduced in Step 1 (*RemoveStudent* in our earlier example), usually derived from the original user story, can be used for input of the IM and subsequently as the name of the procedure or method that implements the wish in the IS, as we did in our example. This will keep the development path (from US, via UC and its SSD to the IM and then to the IS, the realization) traceable and leads to transparent and apparently modular systems.

Step 1 mentions parameters because usually a US and a UC contain parameters as well (e.g., a student number), even if it is only implicit in the natural language expression (see for instance subsection C). As we can see in our running example, these parameters will come back as part of the input of the IM and subsequently as parameters in the procedure or method that implements the wish in the IS.

Within Step 2 the system might need to interact with another system, e.g., to ask for some information. That other system (a.k.a. a *supporting actor*) has to send the answer back. In that case, Step 2 has to be replaced by multiple steps, including 2 steps of the form:

- *System → System2: the question*
- *System2 → System: the answer*

The wish in Step 2 might include the wish to send a message to somebody (or something) else. In that case, Step 2 can be replaced by multiple steps including one of the form:

- *System → <other 'actor'>: the message*

In Step 3 the result could consist of:

(a) the information asked for, in case of retrieval,
(b) "Done" (plus any system generated values), in case of a successful state change, or
(c) the reason(s) why the wish could not be executed.

In several cases, Step 1 in the UC-template and SSD-template above might be replaced by a few (sub)steps, where the user first indicates which US he/she wants to execute (e.g., remove a student), without mentioning the parameter values. After that the system can present the potential values for one of the parameters, after which the user can choose the intended value for that parameter (e.g., *which* student to remove). This can be repeated when there is more than one parameter with a limited set of potential values. Maybe this set of potential values can even become more restricted based on the chosen value for the earlier parameter(s). All in all, the first step in the SSD-pattern could then be replaced by:

1. *User → System: <name of wish>*
2. *System → User: potential values for 1st parameter*
3. *User → System: intended value for 1st parameter*
4. *System → User: remaining potential values for 2nd param.*
5. *User → System: intended value for 2nd parameter*
6. Etc. (if there are more parameters)

Something similar will hold for the preceding UC-template. This of course influences the set of possible inputs of the IM and the interaction pattern between the user and the final system. For instance, the system should now also be able to retrieve the potential values for the 1st parameter, the remaining potential values for the 2nd parameter, etc.

B. An Illustrative Example

The next, subtler example illustrates all the foregoing:

User Story: *As a lecturer, I want to*
Enter the grade of a given student for a given exam of a given course

Probably the set of potential courses is the most limiting option, especially when restricted to the courses that lecturer is responsible for. The set of potential exams of the chosen course (i.e., the exams in near future) is probably very small and the set of students enrolled for that exam is also limited. This brings us to the following elaboration:

Use Case
1. *A lecturer asks the system to Enter a grade*
2. *The system presents the potential courses (of that lecturer)*
3. *The lecturer indicates the intended course*
4. *The system presents the potential exams for that course*
5. *The lecturer indicates the intended exam*
6. *The system presents the students enrolled for that exam*
7. *The lecturer indicates the intended student*
8. *The lecturer adds the grade*
9. *The system tries to*
   Enter that grade of that student for that exam of that course
10. *The system informs the lecturer about the result*

This brings us systematically to the following SSD:

System Sequence Diagram
1. *User → System: EnterGrade*
2. *System → User: all courses of that user/lecturer*
3. *User → System: course C1*
4. *System → User: all (future) exams of C1*
5. *User → System: exam E1*
6. *System → User: all students enrolled for E1*
7. *User → System: student S1*
8. *User → System: grade G1*
9. *System → System: EnterGrading(G1, S1, E1)*
10. *System → User: "Done" or reason(s) why not*

Hence, the system should also be able to retrieve the following sets (where L1 represents the lecturer executing the UC):

For Step 2: *{ c | c is a course for which lecturer L1 is responsible }*
For Step 4: *{ e | e is a (future) exam of course C1 }*
For Step 6: *{ s | s is a student enrolled for exam E1 }*

It already follows from the foregoing that exam E1 and student S1 are known to the system and that S1 is enrolled for E1.

Translating all this to an IM: Suppose that each state y of the IM contains as a component a set y(Gradings). If input i is of the form *EnterGrading(g, s, e)* and [*g*; *s*; *e*] represents the grading with grade *g* for student *s* on exam *e*, then the output G(y, i) will be "Done" and the Gradings table in the new state T(s, i) will be s(Gradings) ∪ { [*g*; *s*; *e*] } when the grading could be added successfully. Otherwise, the new state T(y, i) will be y, i.e. the state stays the same, and the output G(y, i) will contain the reason(s) why the grading couldn't be added.

In a realization in, say, SQL we assume at least the next tables, attributes, keys (bold), and foreign keys (underlined):

Students:	***SID***	
Lecturers:	***LID***	
Courses:	***CID***, *LID*	*refers to the responsible lecturer*
Exams:	***EID***, *CID*	*refers to the course*
Enrolments:	***EID, SID***	*refers to the exam and to the student*
Gradings:	***EID, SID***, *Grade*	*refers to the enrolment*

So, {EID, SID} is a key for Enrolments and for Gradings too.

In case of SQL we might introduce a (stored) procedure `EnterGrading`, with three input parameters: @g, @s, and @e, representing the grade, the student ID, and the exam ID. The main statement in the SQL-procedure will then be

```
INSERT INTO Gradings(Grade, SID, EID)
VALUES(@g, @s, @e)
```

and the three auxiliary sets for steps 2, 4 and 6 can be retrieved by means of (simple) SQL-statements of the form

`SELECT * FROM <table> WHERE <condition>`, namely:

```
2: SELECT * FROM Courses c  WHERE c.LID = L1.LID
4: SELECT * FROM Exams e    WHERE e.CID = C1.LID
6: SELECT * FROM Students s WHERE
       s.SID IN (SELECT x.SID FROM Enrolments x
                   WHERE x.EID = E1.EID)
```

Note that this is a straightforward translation of the three auxiliary sets specified earlier.

C. Me, Myself, and I

Increasingly people are allowed to (or forced to) maintain their own data in a system ('Do-it-yourself'). What does that mean for the USs, UCs, etc.? For example:

User Story: *As a student, I want to Remove myself*

The basic idea behind this is that the user was already authenticated (so 'known') by the system and that the system is able to reconstruct the 'object' that user represents. In this example the user represents a student, in the case in Section B a lecturer.

We can now change the general US-template into

As a <role>, I want to <my wish>

The UC for our sample US could be

1. *A student (the 'user') asks the system: Remove me*
2. *The system identifies the user as a certain student*
3. *The system tries to Remove that student*
4. *The system informs the user about the result*

Our general UC-template can therefore change into

1. *A <role> (the 'user') asks the system: <my wish>*
2. *The system identifies the user as a certain <role>*
3. *The system tries to <wish of that <role>>*
4. *The system informs the user about the result*

Here we consider <my wish> as a special case of <wish>, where the user represents a concrete <role> object in the general <wish>. The SSD for our sample UC could then be

1. *User → System: RemoveMe*ASa*Student()*
2. *System → System: Me := student represented by the user*
3. *System → System: try to RemoveStudent(Me.NR)*
4. *System → User: result*

where *Me* in Step 2 is introduced as a variable. The general SSD-template can therefore change into

1. *User → System: <name of my wish>(<parameters>)*
2. *System → System: Me := <role> represented by the user*
3. *System → System: try to <wish of Me>*
4. *System → User: result*

We note that <my wish> ('personal wish') usually has less parameters than the corresponding more general <wish> because the (known) user represents a 'hidden' parameter.

We also note that the <role> might provide a relevant hint of what should happen. E.g., the next US should lead to the removal of the same person but now as, say, an employee:

User Story: *As a student assistant, I want to Remove myself*

The 'personal' <my wish> is (much) more limited than the corresponding <wish>. E.g., a student can only remove him/herself but a lecturer can remove 'any' student in our examples.

D. CRUD Functionality

We now have a look at the 4 general basic functions applicable to data in a system, known in the literature as CRUD (Create, Read, Update and Delete); e.g., see [13, 14]. One can add data to the system (Create), only 'look' at data in the system (Read), change data in the system (Update), or remove data from the system (Delete). Figure 2 presents some alternative verbs as well as the corresponding operations in SQL. Verbs in (the *wish*-part of) USs, UCs and SSDs should preferably already indicate this, i.e., indicate what should be done with the data.

Name	Alternatively used verbs	SQL
Create	Register, Add, Enter	INSERT
Read	Retrieve, View, See, Search	SELECT
Update	Change, Modify, Edit, Alter, Adapt, Replace	UPDATE
Delete	Remove, Destroy	DELETE

Fig. 2. CRUD, alternative verbs, and the SQL-counterparts

The main example in subsection B is clearly a Create while the auxiliary functions for steps 2, 4, and 6 are Read-examples. The running example '*Remove student*' in Section II obviously is a Delete. An example of an Update would be the user story

As a lecturer, I want to do a grade correction, i.e.,
Adapt the grade of a given student for a given exam of a given course

This example will be like the one in subsection B, with only a few changes:

(1) The texts 'Enter' must be replaced by 'Adapt'
(2) UC-step 8 becomes: *The lecturer gives the new grade*
(3) When the grading could be changed successfully and *og* represents the old grade, the gradings table in the new state T(s, i) will be (s(Gradings) – { [*og*; *s*; *e*] }) ∪ { [*g*; *s*; *e*] }, i.e., the old gradings table minus the old grade entry plus the new grade entry
(4) The new main statement in the SQL-procedure becomes

```
UPDATE Gradings SET Grade = @g
WHERE SID = @s AND EID = @e
```

E. CRUD Patterns

In principle, CRUD patterns are special cases of the general development patterns sketched in subsections A and C. We will now zoom in on each of them.

1) Create

If the <wish> in the general US-template in subsection A starts with a verb like *Enter*, *Register*, *Create*, or *Add*, then it probably concerns a **Create**. In that case the <wish> is often of the form '*Enter/Register/Create/Add <specific object>*', e.g., a specific student, lecturer, or grading. When applying that US, the values for the relevant properties of that object should be given as well, e.g., the student's name, address, city, etc.

Translating this to an IM: Suppose that state $s \in S$ contains a label E for the set of objects where some <*specific object*> *t* has to be added. Then the set of E-objects in the new state T(s,i) will be $s(E) \cup \{ t \}$, i.e., the old set of E-objects plus the new object. The output could be "Done". In some cases, the system must also generate one or more additional values, e.g., a student number or an order number. Those system generated values should also be part of the output given back to the user.

In an OO-based IS the main method will be a *set*-method. In case of a realization in an SQL-based IS, the main statement in the (stored) procedure implementing the Create will be an `INSERT` statement, as already indicated in Fig. 2. In particular, the statement will have the form

```
INSERT INTO E(a1, …, an) VALUES(v1, …, vn)
```

where E is the table, a_1, …, a_n are the attributes, and v_1, …, v_n are their respective values (whether or not system generated).

2) Read

If the <wish> in the general US-template in subsection A starts with a verb like *View*, *Retrieve*, *Read*, or *See*, then it probably concerns a **Read** (so, retrieval).

Translating this to an IM: In case of a Read, the state stays the same, i.e., the new state T(s,i) in the IM will be s, and output G(s,i) will either contain the information asked for or the reason(s) why the information couldn't be given.

In case of a realization in an SQL-based IS, the main statements in the (stored) procedure implementing the Read will be (one or more) `SELECT` statements. In an OO-based IS the main methods will be *get*-methods.

Retrievals can be simple like the three auxiliary retrievals in subsection B. But retrievals can also be very complicated, especially regarding the specification of what exactly should be in the result. An example is the following user story:

As a manager, I want to
See an overview of my department over the last month

First of all, here are two 'hidden' parameters, namely, the department ('*my department*') and the overview period ('*last month*'). The actual value for each of these parameters can be reconstructed via the authenticated user (see subsection C) and the 'clock' in the system. The general, parameterized wish in the SSD will be something like *DepOverview(d, m)* with department and month as parameters, or *DepOverview(d, bd, ed)* with department, begin date, and end date of the relevant period as parameters (which is more general).

But the question remains what kind of information should be in that overview. And in which detail? When a US contains open terms like 'overview', 'report', etc., (quite) some extra (requirements engineering) work has yet to be done. For example, summarized information might require complex and subtle underlying computations (summations, averages, etc.).

3) Update

If the <wish> in the general US-template in subsection A starts with a verb like *Change*, *Increase*, *Decrease*, *Modify*, *Edit*, *Alter*, *Replace*, *Update*, or *Adapt*, then it probably concerns an **Update**. In that case, the <wish> might be of the form

'*Change/Increase/… <specific components in a specific way for each object with a given property>*'. For example:

Increase the number of expected students by 10% and the work load by 5% for each master course for academic year 18/19

As a special case, often only 1 component of 1 object needs to be changed, like in the example in subsection D, where only 1 grade of 1 student needs to be changed.

Translating this to an IM: Suppose that each state $s \in S$ contains E as a label for the set of objects under consideration (say *Courses* in our example) and P(t) indicates that $t \in s(E)$ has the intended property P (in our example: *at master level and for academic year 18/19*). Then the set of E-objects in the new state T(s,i) will be

$$\{ t \mid t \in s(E) \text{ and not } P(t) \} \cup \{ t' \mid t \in s(E) \text{ and } P(t) \}$$

where t' represents the update result for t. The output could be "Done" (and, as a service, might contain the number of updated objects too).

Translating this to, e.g., an SQL-based IS would lead to

```
UPDATE E SET a1 = e1, …, ak = ek WHERE P'
```

where $a_1, \ldots, a_k$ are the attributes to be updated, $e_1, \ldots, e_k$ are the expressions (in terms of the old values) to compute their new values, and P' is the SQL-representation of property P. For our 'master course' example it becomes something like

```
UPDATE Courses
SET    NES = NES * 1.1,
       WL  = WL * 1.05
WHERE  Level = 'Master' AND Year = '18/19'
```

4) Delete

If the <wish> in the general US-template in subsection A starts with a verb like *Remove*, *Delete*, or *Destroy*, then it probably concerns a **Delete**. In that case, the <wish> is often of the form '*Remove/Delete/… <each object with a given property>*', e.g., *Remove each course of last year*. As a special case, sometimes only 1 object should be removed: '*Remove <the object with a given ID value>*'. For example, the student with a given student number or the department with a given name.

Translating this to an IM: If each state $s \in S$ contains E as a label for the set of objects under consideration (say *Courses* in our example) and P(t) indicates that $t \in s(E)$ has the intended property P (*of last year* in our example), then the set of E-objects in the new state T(s,i) will be $\{ t \in s(E) \mid \text{not } P(t) \}$. The output could be "Done" (and, as a service, might contain the number of deleted objects too).

Translating this to, e.g., an SQL-based IS would lead to

```
DELETE FROM E WHERE P'
```

where P' is the SQL-representation of property P.

V. In conclusion

A straightforward and complete development path how to come from initial user wishes to a running system was sketched in [1]. Building on that, the current paper introduced the notion of *development patterns* for such development paths (focusing on functional requirements). Several generally applicable development patterns were introduced (e.g., patterns for the important and well-known class of CRUD functions). We also presented various examples of such development patterns. This resulted in generally applicable development patterns that are transparent and enhance traceability. This leads us into the direction of Pattern-Driven Requirements Engineering (PaDRE). The development patterns add to the knowledge and skills of those involved in the development process from initial user wishes up to a running system. So, a significant contribution.

Although we mainly used (variants of) existing ingredients, the strength of our overarching approach also lies in the *combination* of the ingredients we chose (and the ones we ignored).

To reach our goal, we had to cross the boundaries of several (sub)disciplines such as requirements engineering, machine theory, and (database) systems development.

VI. Future Work

We are extending our theory with additional development patterns, e.g., for more complicated USs, UCs, and SSDs, more complex CRUD functionality (e.g., cascading deletes, compound transactions, and rollbacks), for batches (sequences of inputs and their outputs), and for interacting systems. We are also extending our theory with a 'grammar' for SSDs and, e.g., the semantics and correctness of (complex) transactions.

Acknowledgments

The author wants to thank Herman Balsters and Rein Smedinga for our fruitful discussions.

References

[1] E.O. de Brock, Towards a Theory about Continuous Requirements Engineering for Information Systems, CRE Workshop, REFSQ, 2018

[2] G.G. Lucassen, Understanding User Stories, PhD thesis, Utrecht University, 2017.

[3] G. Lucassen, F. Dalpiaz, J. M. E. M. van der Werf, and S. Brinkkemper, The Use and Effectiveness of User Stories in Practice, Proceedings of the International Working Conference on Requirements Engineering: Foundation for Software Quality (REFSQ), pp. 205–222, 2016.

[4] M. Cohn, User Stories Applied: For Agile Software Development, Addison Wesley Professional, 2004.

[5] I. Jacobson et al, Use Case 2.0: The Guide to Succeeding with Use Cases, Ivar Jacobson International, 2011.

[6] https://en.wikipedia.org/wiki/Use_case

[7] C. Larman, Applying UML and patterns, Addison Wesley, 2004.

[8] https://en.wikipedia.org/wiki/System_sequence_diagram

[9] G.H. Mealy, A Method for Synthesizing Sequential Circuits, Bell System Technical Journal, pp. 1045–1079, 1955.

[10] https://en.wikipedia.org/wiki/Mealy_machine

[11] L. Jessup, J. Valacich, Information systems today, Pearson, 2008

[12] https://en.wikipedia.org/wiki/Information_system

[13] J. Martin, Managing the Data-base Environment, Prentice Hall, 1983.

[14] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

All links were last accessed on 2018/07/17