Rapid Development of ICT Business Services by Business Engineers Independent of Computer Scientists

Jürgen Spielberger[1], Markus Baertschi-Rusch[2], Marion Mürner[3], Gerald Perellano[4], Raimond Wüst[5]

[1] Zurich University of Applied Sciences (ZHAW), School of Engineering
[2] Swiss Federal Institute of Technology in Zurich (ETHZ), BWI Center for Industrial Management
[3] Posity AG, development division
[4] Zurich University of Applied Sciences (ZHAW), School of Engineering
[5] Zurich University of Applied Sciences (ZHAW), School of Engineering

Abstract
Current software development requires computer scientists to create and to adapt services to new or changing needs. In addition development and maintenance of software is time and cost intensive. Customizing of standard software is laborious. Software engineering research approaches as Domain Engineering, Model Driven Software Engineering and Product Line Engineering try to increase the abstraction level of the specification of the models to reduce the required time and money to build applications and services but they still demand the implementation by computer scientists.

In two projects supported by the CTI (Commission for Technology and Innovation of the Swiss Confederation) we analysed how to achieve a higher level of abstraction and how to specify database-centric business services in a manner business engineers are able to create and to adapt services completely by themselves. Besides the methodology to specify these services (data structure, business rules, etc.), methods and technologies to hide all technical aspects (infrastructure, software architecture, versioning etc.) entirely from the business engineer had to be developed.

In this paper an according graphical notation to specify services or complete applications is discussed. The methods and technologies to hide technical aspects are not part of this paper

The developed graphical notation consist of six different diagram types. They represent different aspects of the services (process map, business rules, data structure, organisation chart, user interface and data queries). To create executable services with these diagram types an IDE (Integrated Development Environment) was developed. This IDE is called Posity Design Studio (PDS). All diagrams drawn with the Posity Design Studio can directly be executed without any further coding. To increase the efficiency of creating new services a business wizard to specify uses-cases and to generate the corresponding diagrams was integrated into the Posity Design Studio.

Our main findings are:

› The graphical notation allows specifying executable services completely by diagrams. In that way the usage of common programming languages is no longer necessary nor desired.

› Graphical domain-specific languages empower business engineers to build business applications and services with little effort and without the help of computer scientists.

› The graphical notation is platform independent. Therefore it's possible to specify applications and services that can be executed on several platforms at the same time.

› Skilful structuring of the notation (design of the levels of abstraction) allows even novice users to make limited adjustments to the applications or services.

› Additional abstraction levels increase the productivity and adaptability of applications and services.

Keywords
Rapid Development, Business Service, Business Application, Model Driven Software Engineering, Domain Engineering, Domain-specific Language.

1 Introduction

Other industries have a natural transition from specification level to physical level. Specifying a building with plans and constructing the building are obviously two different types of activity. IT Industry has no such natural transition, there is no inherent modelling gap. Today usually the requirements are described by the business engineers and translated into program code by software engineers. The specification gets more detailed, but there is no obvious leap or modelling gap. Therefore three questions arise:

(1) Does a modelling language (specification language) exist that can be handled by non-computer scientists (without textual program code) and that can describe ICT business applications and services entirely. In this case the application or service can be build based on this specification (model) without any further program code, without software engineers and without modelling gap.

(2) Assuming this modelling language exists, is it possible to introduce different abstraction levels within the modelling language so that business engineers with different know-how can provide and change the specification without extensive knowledge of details (e.g. change a workflow / status flow).

(3) What are the possibilities to provide an assistance (hereinafter referred to as a wizard) on a higher abstraction level than the modelling language to generate specifications in the modelling language?

There are several attempts to build such a modelling language. A broadly known concept is the Model Driven Architecture® (MDA®) of the Object Management Group® (2014). These concepts separate the business model and the technology model in two parts. Therefore at least for the technological model computer scientists are needed. In addition the specification of the business model usually requires additional program code.

This paper shows a solution how a modelling language and the associated integrated development environment (IDE) can fulfil the requirement to enable business engineers to specify a business application or business service without the help of computer scientists. In addition this paper tries to give a deeper insight into how to solve, respectively how to answer the three questions mentioned above.

Within the next chapter we will introduce the six diagram types that build the modelling language, followed by a short introduction to the mechanisms of the IDE that prevent a technology model. The third chapter enlists the different levels of abstraction and the consequences of this structuring. The subsequent chapter shows methods to extend the modelling language with wizards. After the deeper introduction to modelling languages the main results are summarized, followed by a discussion of the findings and the conclusions.

2 Modelling Language

Modern programming languages have to fulfil various requirements such as efficiency, maintainability, portability, debugging support, etc. The considerations in this chapter apply only to the main problem mentioned in the first question: Does a modelling language exist that can be handled by non-computer scientists (without program code) and that is able to describe the IT solution entirely – hiding all technical details?

The analysis of existing approaches such as MDA (Model Driven Architecture), BPMN (Business Process Model and Notation) (Großkopf, A., Decker, G., Weske, M., 2009), ARIS (Architektur integrierter Informationssysteme) (Davis, R., Brabaender, E., 2007) and many more revealed several severe problems:

› The modelling language requires additional program code or the elements of the notation are representatives of program code – the notation is based on the code world (in most cases on an object-oriented approach).

› The modelling language generates program code (e.g. Java) but has no hundred per cent round-trip engineering functionality.

› The modelling language does not cover all necessary parts of the model for a complete specification or some parts of model are defined multiple times.

› The modelling language requires computer scientist to participate in one or more steps of the implementation process (e.g. setup of the infrastructure, deployment of code).

In order to avoid these problems a new modelling language was elaborated. This modelling language has the following characteristics:

› The modelling language is limited to specify all necessary parts of business applications and services. Languages with focused, limited scope are called domain-specific languages (DSL). Hereinafter we call the modelling language Posity-DSL (PDSL). Languages with a limited scope have an increased productivity, an improved quality and they have a better maintainability (Reinhartz-Berger, I., Sturm, T. C., Cohen, S., Bettin, J., 2013), but are only applicable in the predetermined domain.

› PDSL does not generate code or require any program code for specification.

› PDSL does not require computer scientist in any work process.

- PDSL consists of six diagram types. Each diagram type specifies a part of the complete model. The diagram types of PDSL are based on existing diagram types that are extended with necessary elements on one side and reduced by not required elements on the other. This simplifies the usage of these diagram types for business engineers familiar with diagrams used for the specification of models.

The following sub chapters explain the diagram types of PDSL. The process and the module diagram will be shown in more detail to give a better impression of the extensive potential of the diagram types. The remaining diagram types are shown in less detail. They will illustrate the functioning of PDSL as a whole.

2.1 Process Diagram

The process diagram (Fig. 1) is an adaption of the Business Process Model and Notation (BPMN) (Großkopf, A., Decker, G., Weske, M., 2009). The process diagram is mainly used to specify following information:

- Processes (blue arrows): There are different types of processes. Processes without business logic (system processes without module or manually executed processes) are used to organize and document the structure of the process model – processes can be nested in one another. Processes that are connected to modules (a hooked light blue square, containing the business logic) are executable processes. In order to control the access of the users to the processes it is possible to determine a role for each process.
- Workflow (blue lines): The workflow is used to define an automated processing order of the activities of a user. Different types of gateways (green diamonds, automatic or by questioning) optionally allow to determine the next process to execute.
- State flow (green lines) and status boxes (green rounded rectangle): While the workflow defines the process sequence to be executed, the state flow defines the sequence of states that data (optionally, defined per table) can traverse. Process diagrams offer the possibility to specify valid states of input data to a process as well as the states of resulting output data after processing has finished.
- Events (red bullets): Events provide asynchronous, timer based handling of the workflow or the state flow (the diagram in Fig. 1 shows an event used within a state flow).

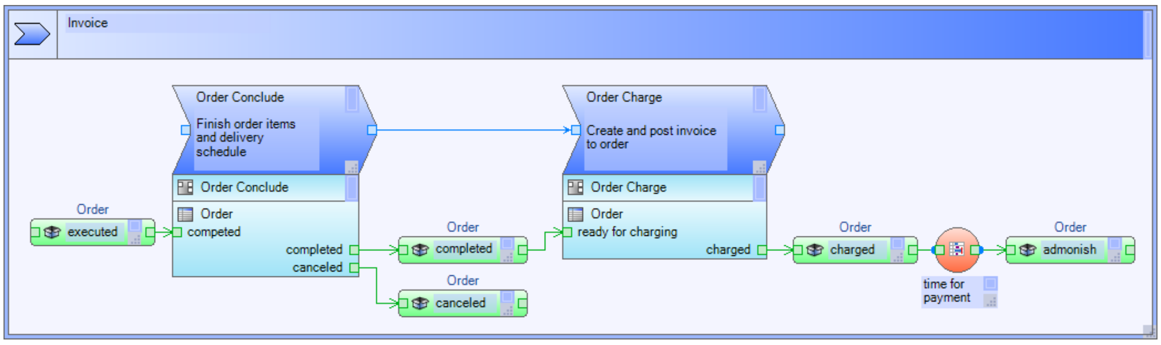


Fig. 1: Example of process diagram

This diagram *is* the specification of the application, there is no generated program code. Changing the diagram (e.g. changing the state flow) instantly changes the behaviour of the application.

2.2 Data Model Diagram

The data model diagram (Fig. 2) is related to the crow foot notation (Barker, R., 1990). In this extended version it holds all required information to completely define the data structure and has design elements (e.g. table references, connector joins, etc.) to work with very large data models.

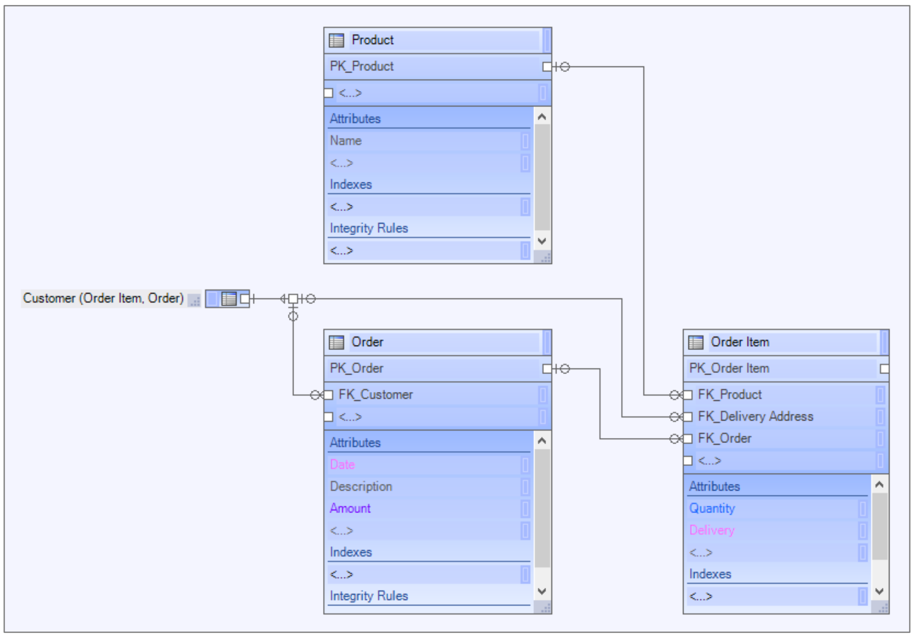


Fig. 2: Example of data model

2.3 Query Diagram

Reading and writing of data is specified in the query diagram (Fig. 3). The representation of the query diagram is derived from the representation method of the data model diagram.

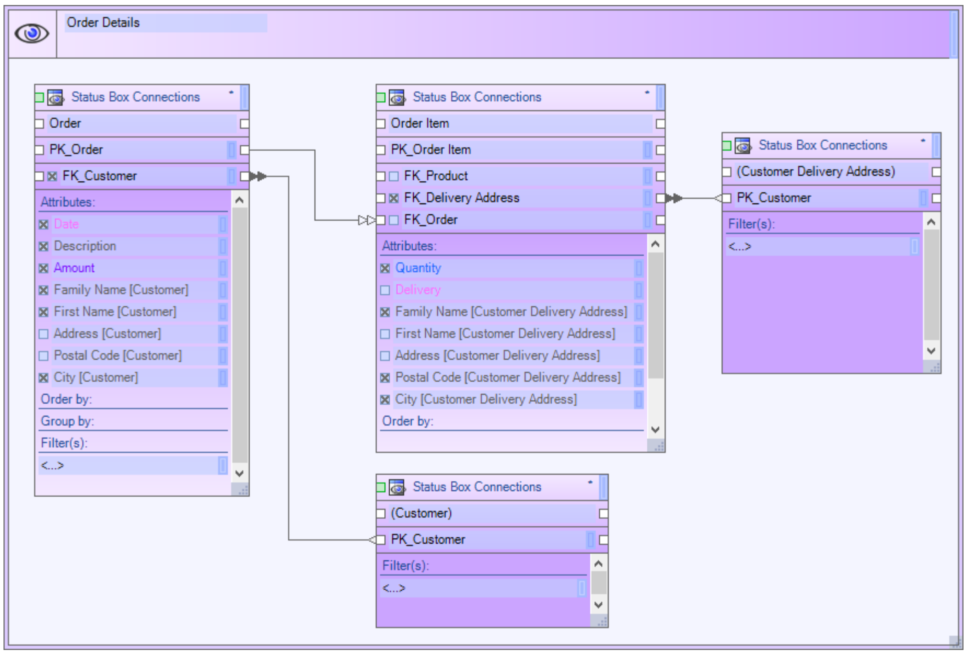


Fig. 3: Example of query diagram

2.4 Module Diagram

The module diagram (Fig. 4) is an extended data flow diagram (Yourdon, E., Constantine, L.L., 1975) and defines the business logic. It contains:

› Components (rectangles): Components (function blocks, e.g. calculating the square root of a number, show a message) have input ports (on the left) and output ports (on the right) receiving and sending data.

› Data flow (lines, color depends on data type of data flow): The data flow represents the flow of data through the components. Therefore the lines connecting the components define the execution sequence of the components.

› Control flow constructs (rectangular substructures, e.g. sequence, while loop, case): Control flow constructs similar to Nassi-Shneiderman boxes (Nassi, I., Shneidermann, B., 1973) extend the module diagram and allow to define the module logic consistent with the philosophy of structured programming (Fig. 4 shows a module with a case structure). Also recursive constructs, modules calling modules, are possible.

› Module events (frames of module): Each module diagram consists of one or more module events. In the diagram, a single module event is shown at once. Due to scroll or select the individual module events can be viewed. The name of the module event is below the name of the module (Fig. 4 shows a module event named ‘Save’). Module events are triggered by the start of the corresponding module, when a button is pressed on the user interface (e.g. a save button), by the module itself, etc.

According to domain-specific languages all diagrams, especially the module diagram, support business driven functionalities. Common problems such as time zone (including daylight-saving time), currency handling (including ledger currency), number ranges, multilingual support, etc. are an integral part of the diagram types.

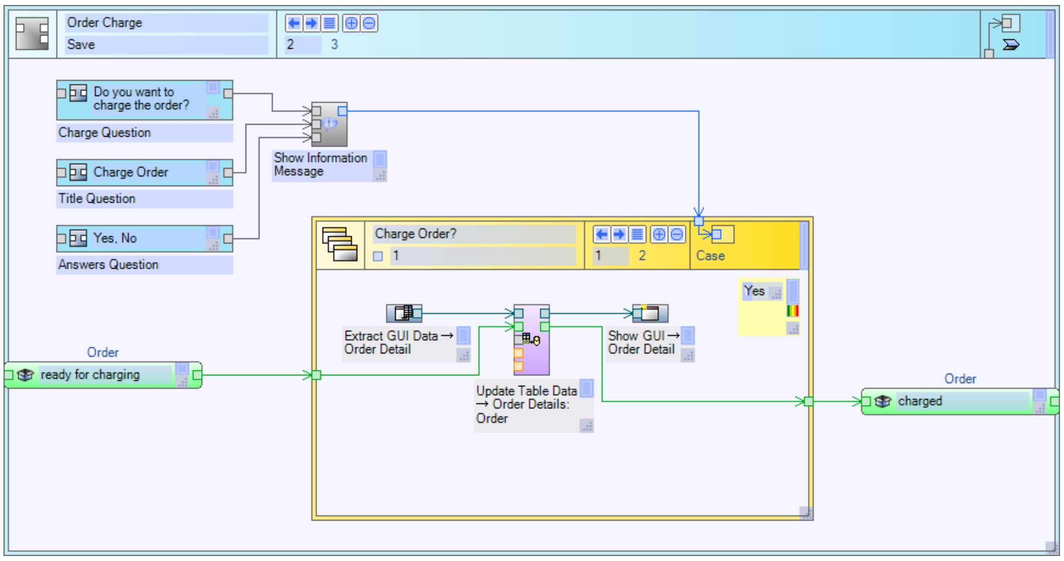
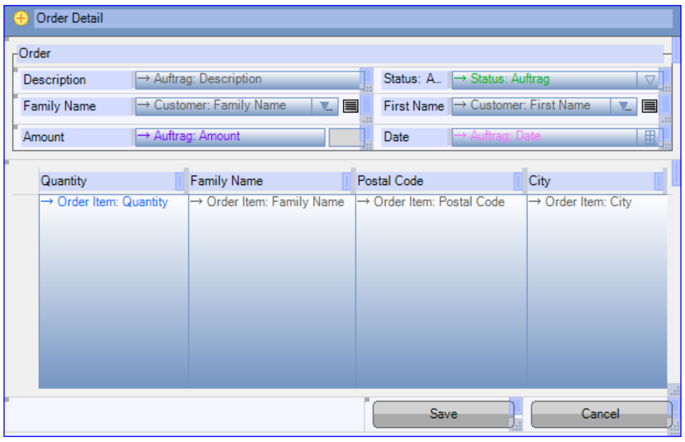


Fig. 4: Example of module diagram

2.5 User Interface Diagram

The user interface diagram (Fig. 5) defines the graphical presentation of data to the user. Depending on the users role the data can be modified. The user interface diagram is aligned to the query diagram.


Order Detail

Order

Description	→ Auftrag: Description	Status: A..	→ Status: Auftrag
Family Name	→ Customer: Family Name	First Name	→ Customer: First Name
Amount	→ Auftrag: Amount	Date	→ Auftrag: Date

Quantity	Family Name	Postal Code	City
→ Order Item: Quantity	→ Order Item: Family Name	→ Order Item: Postal Code	→ Order Item: City

Fig. 5: Example of graphical user interface

2.6 Organisational Diagram

The organisational diagram (Fig. 6) is used to define the structure of the company and the associated roles. This roles can be assigned to users and define the access rights the user has within the application.

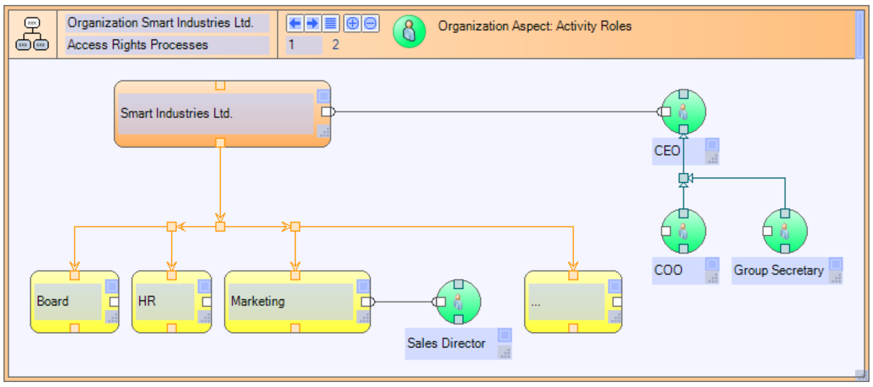


Fig. 6: Example of organisational diagram

2.7 Integrated Development Environment (IDE)

Using this six diagram types of PDSL an entire application containing any number of services can be constructed, no additional program code or specification in any form whatsoever is required. The integrated development environment used to draw these diagrams is the Posity Design Studio (PDS). Similar to other development environments PDS supports additional features such as debugging tools, regression testing, deployment, use of multiple environments (e.g. test environment, training environment), etc. to facilitate the development process.

To hide all technical aspects the infrastructure and architecture of the runtime environment is predetermined. These technical aspects are invisible at all time. Some details are:

› Typically all data is stored in SQL servers in the cloud. This applies to the data of the application as well as for the data of the diagrams (metadata).

› The infrastructure for rich client (deployed using ClickOnce mechanism) on MS Windows systems and web apps (implemented with an application browser) is automatically created.

› The general structure of the user interface (e.g. look-and-feel of user interface, general items, etc.) is predefined.

3 Levels of Abstraction

At first glance, the diagrams appear to have the same level of abstraction. A closer look reveals that this is not really true. The process diagram is based on the module diagram, extended with workflow and state flow. The user interface diagram (GUI) is based on the query diagram. The query diagram is based on the data model diagram and the module diagram on the user interface diagram and the query diagram. The organisational diagram is influencing several diagrams.

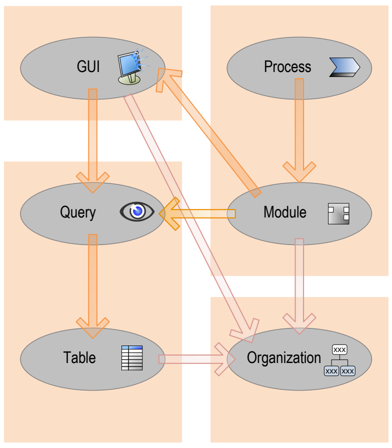


Fig. 7: Dependencies and levels of abstraction of the different diagram types

This hierarchy of information is not chosen or composed randomly. This levels of abstraction are chosen to give business engineers the maximum of flexibility with a minimum of necessary prior knowledge. Two examples will show the effects:

(1) In the current state flow (Fig. 1) an invoice is printed (second process step) after the order is completely processed (first process step), but now we want to print the invoice before the order is completely processed. A simple change of the state flow sequence in the process diagram will cause the system to be adapted to the new situation (Fig. 8). No business logic has to be changed; the logic of the state flow *is* represented in the process diagram. And of course one can also implement both state flows in parallel. This is possible due to the fact that the state flow is explicitly specified in the process diagram and not on the 'lower' abstraction level of the module diagram.

(2) During the live cycle of applications often additional information has to be stored in the system (e.g. if we want to store the middle name of a person in addition). In common systems the database and the programs have to be changed to implement this additional requirement. PDSL avoids as far as possible to handle details of the information within the module diagram (for example listing individual attributes). To add an additional information item (e.g. an attribute) just the data model diagram, the concerned query diagrams and user interface diagrams have to be extended (adding the attribute). With very little work the information can be modified and stored, no module diagram has to be changed.

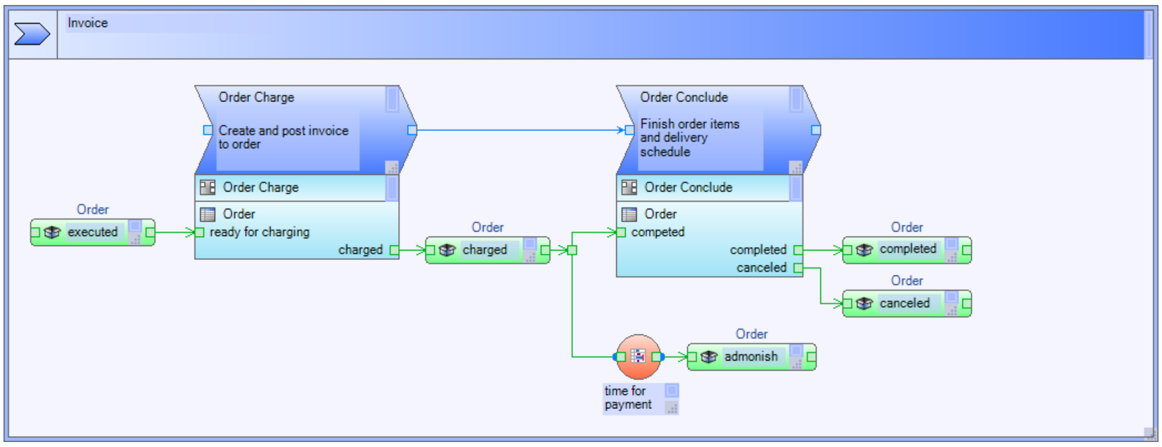


Fig. 8: Adapted example of the process diagram of Fig. 1

4 Expansion Levels of Abstraction

The previous paragraphs gave a brief insight into the effectiveness of different abstraction levels. This mechanism of abstraction can be used once more. In PDSL two areas for higher abstraction levels can be distinguished. The first one concerns the specification of complete use cases, the other one concerns the customization of an application for different customers.

4.1 Patterns for Use Cases

The PDS (Posity Design Studio) offers different patterns for use cases. A pattern for a use case represents a common way to work with information, e.g. editing a header record and its sub items in a list (e.g. editing a sales order and the order items). After selecting the pattern the basic work to do is to specify the selection of the affected data and how this data will be merged. Subsequently extending this information with further details (e.g. which events shall be generated, which data will be editable in the user interface, etc.) the PDS wizard will generate the according query, module, process and user interface diagrams. If necessary, the generated diagrams can be manipulated such that they fit the required use case exactly.

Like other wizards, the PDS wizard has no round-trip engineering for changes done in the diagrams. Changes of the diagram that do not fit into the pattern of the wizard cannot be represented in the pattern. To get some flexibility and extendibility the use case patterns themselves contain extension points to integrate functionality that is not part of the pattern.

The PDS wizard is very flexible and powerful. Therefore it's possible to create complex use cases and to cover a wide range of requirements.

4.2 Customizing Applications - Questionnaire

In practice applications are used more than once, implemented use cases are reused. But different companies have different requirements and the reused use cases have to be modified. Customizing use cases is again time and cost intensive.

In PDS it's possible to specify different variations within the diagrams and to combine these variations with questions (e.g. a question about the type of company and the activation of processes). The application gets customized by answering the questionnaire.

The questions of the questionnaire are not independent from each other. E.g. if a process gets deactivated by a previous question there is no need to ask another question that would deactivate this process again. The questionnaire takes these dependencies into account and only asks questions that still (after answering some questions) take influence on the application.

It's not necessary to answer all questions of the questionnaire at once. It's possible to answer only some questions (the questions can be prioritized), test the effect on the application, and then to return to the questionnaire. Alike it is possible to answer questions of a particular business department only (e.g. production), as the questions can be assigned to processes of the process diagram (not to confuse with a question about a process).

5 Main Results

› The proposed graphical notation with six different diagram types allows to specify executable ICT business applications or services completely by diagrams. The usage of common programming languages is no longer necessary nor desired.

› Graphical domain-specific languages empower business engineers to build business applications and services with little effort and without the help of computer scientists.

› The graphical notation is platform independent. Therefore it's possible to specify applications and services that can be executed on several platforms at the same time.

› Skilful structuring of the notation (design of the levels of abstraction) allows even novice users to make limited adjustments to the applications or services.

› Additional abstraction levels (wizard and questionnaire) increase the productivity and adaptability of applications and services.

6 Discussion of Findings

It’s not surprising that it’s possible to specify applications and services with a graphical domain-specific language, although only few solutions exist in practice (for the technical field, e.g. LabVIEW). Two facts of the diagram language are noteworthy: (1) The proposed language uses only six diagram types and (2) these diagram types are variations of well know diagram types, some of them are even well known in business management.

The usage of the graphical domain-specific language (PDSL) enables business engineers to implement applications all by their own. Therefore all technical details have to be hidden from the business engineer. This is only possible by predefining these technical details, for this reason the technical structure is more or less immutable.

Abstraction is a powerful instrument to achieve model languages that are simpler to use. A carefully defined language allows specifications on different levels of abstraction, in such a way also unexperienced business engineers can adapt some logic of the application. Limiting the operational area of the modelling language to a specific domain (database-centric business applications) made the language more efficient and more comprehensive, but of course the language is restricted to the domain.

Productivity has been increased considerably by introducing a wizard. Because the wizard generates diagrams, it’s possible to adapt and to extend the generated diagrams with details that cannot be specified within the wizard. However, this leads to gaps within the two abstraction levels, a complete round trip (from diagram to wizard) is no longer possible.

7 Conclusion

In the introduction three questions have been asked, they will be answered here:

(1) Yes it’s possible to create a domain specific-language without any textual program code. PDS was applied in several projects in daily practice and has shown the potential for efficient software development.

(2) Different abstraction levels simplify the use and the readability of the model. Even users with limited knowledge are able to make basic adjustments to the model.

(3) Tools with higher abstraction level accelerate the development of applications, but at a certain point new abstraction levels (wizard, questionnaire) lead to an information gap which disables round-trip engineering.

PDS shows that developing applications is possible without computer scientists and we are convinced that development is cheaper and faster. Unfortunately we could not make any surveys, which show whether and how much cheaper and faster the development of applications with PDS is.

The use cases still have a great potential to increase efficiency. For further enhancements, it would be important to investigate which additional use cases should be implemented, how these use cases exactly should be implemented and how they can be designed and extended to improve the effectiveness of round-trip engineering.

References

Barker, R.: (1990) CASE*Method: entity relationship modelling. University of Michigan: Addison-Wesley

Davis, R., Brabaender, E.: (2007) ARIS Design Platform: Getting Started with BPM. London: Springer

Großkopf, A., Decker, G., Weske, M.: (2009) The Process: Business Process Modelling using BPMN. Tampa: Meghan-Kiffer Press

Nassi, I., Shneidermann, B.: (1973) Flowchart techniques for structured programming. ACM SIGPLAN Notices 8 (8), 12 -26 (www.nassi.com/nassi-shneiderman diagrams.pdf)

Object Management Group (2014) The List of References Illustrated [online] available from http://www.omg.org/mda/presentations.htm [3 May 2014]

Reinhartz-Berger, I., Sturm, T. C., Cohen, S., Bettin, J. (2013) Domain Engineering: Product Lines, Languages, and Conceptual Models. Berlin Heidelberg: Springer

Yourdon, E., Constantine, L.L. (1975) Structural Design: Fundamentals of a Discipline of Computer Program and Systems Design. Upper Saddle River, NJ: Yourdon Press