A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software

JAAP KABBEDIJK, SLINGER JANSEN and SJAAK BRINKKEMPER, Utrecht University

In order to maximize their customer base, business software vendors are trying to offer software products that support the business needs of as many customers as possible. The more standardized a software product is, the easier it will be to serve large numbers of uniform customers. However, if customers are not homogeneous, a trade-off must be made between flexibility and complexity. A case study is presented showing the implementation of the Command Query Responsibility Pattern (CQRS), a pattern dictating the strict separation between commands and queries. The study was performed at a large software product vendor currently designing a software product based on CQRS. Seven sub patterns related to CQRS are identified and discussed. The research results show the CQRS pattern is implemented and how its different sub patterns can result in a high level of variability within a software product and how the different sub patterns can interact to achieve this.

Categories and Subject Descriptors: D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*

General Terms: Software Patterns

Additional Key Words and Phrases: Case Study, CQRS, Software Pattern, Software Architecture, Variability

ACM Reference Format:

Kabbedijk, J., Jansen, S. and Brinkkemper S. 2013.A Case Study of the Variability Consequences of the CQRS Pattern in Online Business Software. jn 2, 3, Article 1 (May 2010), 10 pages.

1. INTRODUCTION

It is highly relevant in business software to offer a product to customers that fits their business processes, especially in ERP and related bookkeeping software. This can be problematic, since different customers have different business processes and because of this different, or even contradictory requirements to a software product. The architecture of a software product has to support the variability needed to offer a software product flexible enough to match all different customer requirements, while not introducing unwanted side effects, such as complexity, scalability or security challenges. In Software Product Lines (SPL), variability is known as the ability to change or customize a software product [11]. This definition is sufficient for software products that are manufactured in a software product line style and deployed on premises at customers, but does not hold true any more when it comes to online software products. Online software products have to be able to offer variable solutions *at the same time* from a single customizable instance, a concept known as runtime variability [16]. The principle of serving multiple customers from one online software product, giving each the idea they are the only customer using the product in terms of flexibility is known as multi-tenancy [12].

In order to create a software product, capable of offering a certain level of variability, most current software products separate logic into different layers. Each tier within this architectural principle is responsible for a different

This research is part of the 'Product as a Service' project (www.productasaservice.org).
Corresponding author: J. Kabbedijk (j.kabbedijk@uu.nl)

part of the architecture [14]. An often implemented solution to this multi-tier architecture is the three-tiered application in which there is a separate data, logic and presentation tier. Within this solution, the database in the data tier is often seen as one CRUD (Create, Read, Update and Delete data) data store in which all commands and queries are performed on the same database. This can lead to locking, performance and scalability problems, especially with larger commands or queries, since all things have to be taken care of sequentially. Distributing parts of the system in combination with selective locking of data provides a partial solution, but leads to a high probability of data inconsistency.

Since the CAP theorem [8] states that it is impossible for a distributed system to be have Consistency, Availability and Partition Tolerance at the same time, it is an option to split parts of the system that have an emphasis on consistency from parts that should have an emphasis on availability or partition tolerance. Following this line of thought, Greg Young and Udi Dahan came up with the CQRS (Command Query Responsibility Separation) pattern [23; 5] in which all logic of a software product is separated based on whether it changes the application state (commands) or only queries it (queries). This means executing commands is done by different components than the one responsibly for executing the querying tasks, all of which can be done distributed and in parallel.

Besides helping to solve the scalability problem of multi-tiered software products by enabling architects to distribute tasks of the system among an unlimited amount number of systems, CQRS also helps to implement a higher level of variability in online software products. The high level of variability is caused by the fact the main pattern keeps commands strictly separated from queries and has a large collection of sub patterns using the distributed nature of the pattern to enable, among others, all sort of different tenant dependant configurations, work flows, and business rules. This concept will be further explained in section 6.

This paper will first report on related research in section 2, after which the research approach will be discussed in section 3. After this, an example of the CQRS pattern will be shown we observed in the case company in section 4. Different sub patterns playing a role in CQRS will be dicussed in section 5, followed by the consequences CQRS has in section 6. The paper ends with a discussion and some future research in section 7, followed by a conclusion in section 8.

2. RELATED WORKS: CQRS AND VARIABILITY

The ground principle of CQRS, stating the strict separation of command and queries, is introduced by Betrand Meyer in his book Object-Oriented Software Construction [15]. He called it Command Query Separation (CQS), a pattern in which each method is either a command performing a certain action or a query returning data to the caller. Both commands and queries are performed independently from each other. In his own words, “asking a question should not change the answer”. This concept was picked up later on by Greg Young and Udi Dahan, who merged it with ideas out of Domain Driven Design (DDD) by Eric Evans [6] and combined this to create the CQRS pattern [23].

In a nutshell, the CQRS pattern is only about creating two subsystems, as can be seen in figure 1. From the user interfaces commands can be sent to the command manager or queries can be send to or received from the query manager. Commands are actions that will be performed on the data, while queries are requests for data to be shown. The CQRS pattern itself does not prescribe anything about communication between the command manager and the query manager, but there is a collection of patterns often used in combination with CQRS that take care of communication. An often applied pattern within CQRS for communication is communication through events, which will be elaborated on in section 5.1.

Currently there is an active community of developers, architects and enthusiasts working with the CQRS pattern, but it has not yet penetrated the broadly applied and widely known collection of software patterns described in the work of Gamma et al. [7] and the Pattern-Oriented Software Architecture books [4; 20; 13; 3; 2]. Several frameworks like NCQRS (`github.com/ncqrs`), Axon (`github.com/axonframework`) and Lokad (`github.com/Lokad/lokad-cqrs`) however, helping developers to implement the CQRS pattern in several languages are

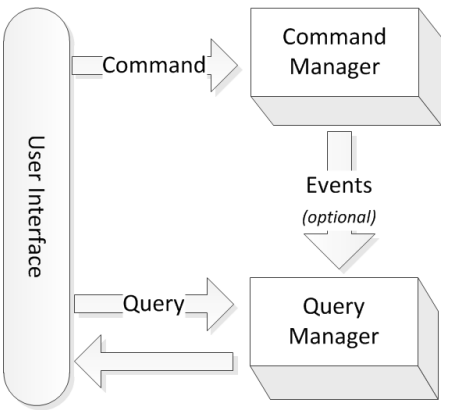


Fig. 1: CQRS core principle

released in the last two years, making the CQRS pattern increasingly popular. The documentation and community around these frameworks are also an important source of knowledge related to the CQRS pattern.

3. RESEARCH APPROACH

In order to gather all data related to this research, a case study was performed at large ERP software vendor from the Netherlands having approximately 10,000 small and medium enterprises using their current online bookkeeping software product on a day to day basis. From now on we will refer to the case company as ERPSoft. During this case study, **(1)** several CQRS information sessions for employees at ERPSoft in which the principles of CQRS and are explained, emphasising on the particular implementation within ERPSoft were attended. We **(2)** actively participated in the architecture team for a total of 40 hours in which we also **(2a)** conducted five interviews with architects working at ERPSoft on the implementation of CQRS within their software product and its consequences on scalability and variability. Within these interviews, architecture design artifacts were discussed and shared with the author. Finally, **(2b)** all results of the interviews were analysed within the architecture team to have a constant feedback loop for interpreting the architecture and consequences of implementing the CQRS pattern. This constant feedback of key figures within the research area is common practice within cooperative inquiry [18] and the design science cycle of Hevner [10]. Besides gathering data within ERPSoft, **(3)** all research findings are also discussed with an external expert panel consisting out of three leading CQRS experts from outside the case company. All experts are either authors of a CQRS framework or provide courses in applying the CQRS framework. The panel also actively participated in reviewing the research paper. The overview of the research approach used can be found in figure 2.

The case study is a single case case study according to the classification of Yin [22]. A case study database containing recordings of the interviews and all notes taken during the interviews was kept in order to improve the traceability and rigour of the case study research. The internal and construct validity of the research is ensured by using experts within the case company to check all artifacts and conclusions and by matching the results of the case study with expected results. A clear case study protocol was used as advised by Runesson and Höst [19] in which the planning and structure of the interviews was described.

3.1 Research Questions

The main research goal **"How can the CQRS pattern influence the variability of a software product?"** will be answered by answering three related research questions (RQs). These questions are:

(1) How is the CQRS pattern designed within the case company?

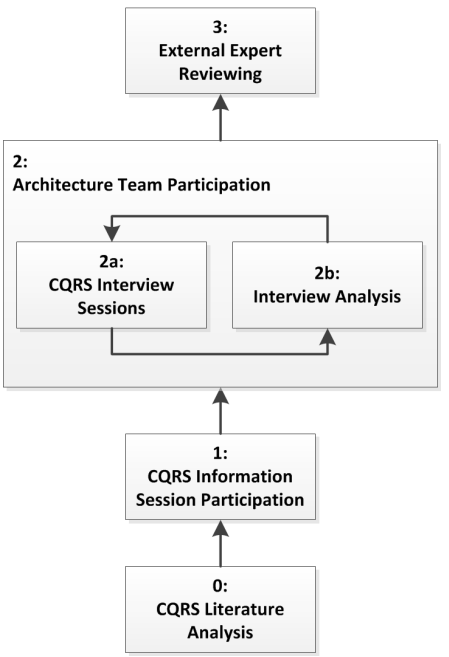


Fig. 2: Applied Research Method

(2) What sub patterns can be identified within the CQRS pattern?

(3) How do the different sub patterns influence the variability of a software product?

RQ1 is answered by a using interview data and architectural design artifacts gathered at the case company during CQRS information sessions, interviews and a constant feedback during architecture team participation. The answers to RQ2 are primarily answered by using design artifacts from the case company, combined with literature and expert reviews from an external expert panel consisting of three CQRS experts. RQ3 again is answered using the interview data resulting from interviews held with architects at the case company and a review on our conclusions by an external expert panel.

4. CQRS IMPLEMENTATION

This section will report on the design of the software architecture of the main product created by ERPSoft. This implementation report is aimed at giving an impression of the possibilities of the CQRS pattern and related sub patterns. Currently ERPSoft is redesigning their software product from scratch, keeping a strict separation between all queries and commands, as indicated by the CQRS pattern. This section reports on the new software architecture designed at ERPSoft, so no legacy code or systems are in place.

Figure 3 shows the CQRS-based design of the software architecture at ERPSoft. On the left side of the figure the *User Interface* is modelled from which *Commands* can be sent to the *Command Bus* (top of the figure) and *Queries* can be sent and received to and from the *Query Bus* (bottom of the figure). All arrows in the figure represent communication within the system. Whenever the communication is explicitly implemented as a command, query or event, this is indicated in the figure. All other arrows, including the double headed arrows only represent a certain form of communication, but nothing specific is specified. From the *Command Bus*, commands are sent to *Command Handlers*. The handlers perform the action indicated by the command, after which the action is stored in the /textitStream Store. From here all events are sent to the *Event Bus*, who can distribute the event among

different *Query Model Builders (QMBs)* or route events back to the *Command Bus*, through *Event Routers*. Add the query side, different *Query Model Builders* listen to the events broadcasted through the *Event Bus* and act on certain events. The events they reponse to depend on the goal of the specific builders. All built queries are stored in a *Query Store*, for easy access by both the *Query Model Builders* and *Query Handlers*. The *Query Handlers* can publish query results to the *Query Bus*, which can be used by he *User Interface* to display certain information.

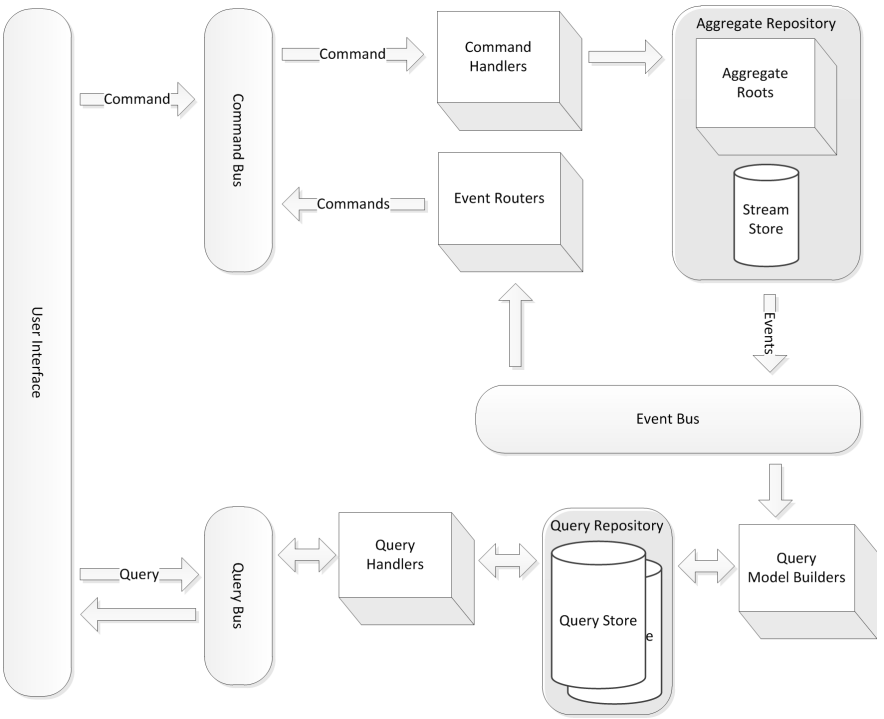


Fig. 3: CQRS implementation at ERPSoft

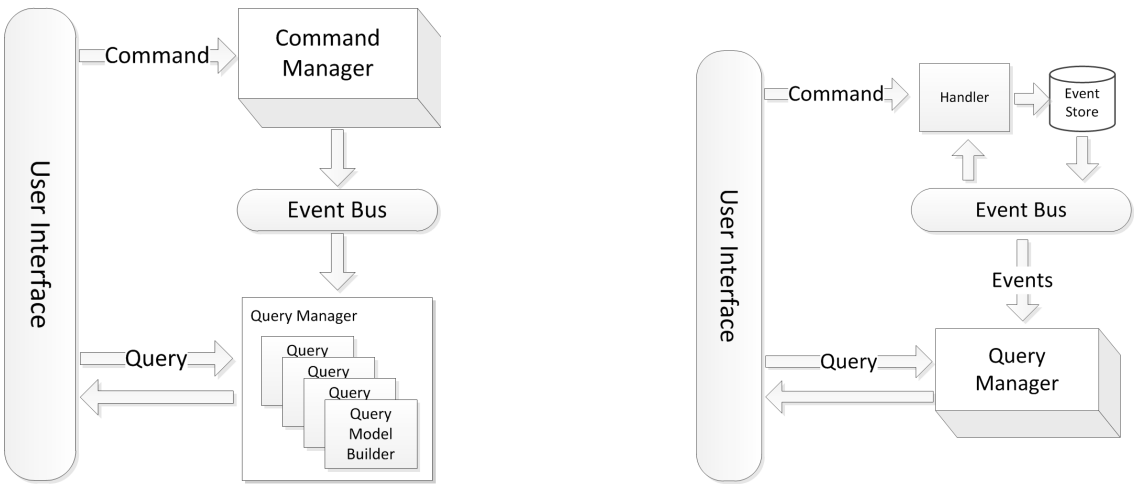
A further analysis of sub patterns that can be observed within the CQRS-based software architecture design can be found in section 5.

5. CQRS SUB PATTERNS

The CQRS pattern can be extended and complemented by applying several additional patterns. The number of sub patterns that can be applied within the CQRS pattern are numerous, but this research focusses on the selection of patterns we observed within the case company.

5.1 Event Sourcing

One of the possibilities within the CQRS pattern that can play a big role in terms of scalability is the sourcing of the events created by the command manager. These events can be sent to an event bus to which the query model builders in the query manager listen. A query model builder is a different sub pattern that is able of translating events to appropriate data views, which is discussed in more depth in section 5.5. The different query builders can be on the same system, but also on different physical or virtual machines. Query model builders can be on different geographical locations or even at clients. Because of this, the system becomes scalable and all sub



(a) Event Sourcing Sub Pattern

(b) Event Store Sub Pattern

Fig. 4: The Event Sourcing and Event Store CQRS sub patterns

parts are specially geared towards the task they have to do (ie. read or write) [23]. Please see figure 4a for a representation of the Event Sourcing pattern. The most important aspect of the event sourcing pattern is the fact different events are broadcasted by the command manager to be processed by different components.

5.2 Event Store

Events in the CQRS pattern do not necessarily need to be stored in any way. They could be sent to the query manager immediately after being processed and never be stored (as can be seen in figure 4a). The query manager then can do with the event whatever is necessary to get the data in an appropriate form. Because the storage by the query manager could be anything from stored in cache to stored at the client or in some database you can not rely on the availability and recovery of data if the system crashes. Because of this, an often implemented pattern within CQRS is the use of an event store. In this store all events can be stored sequentially, so the all data can be reconstructed based on the events in case of a system crash [17]. Figure 4b shows the representation of the event store pattern. From the *User Interface* commands are sent to a *handler* (see section 5.4 for more information on command or query handlers) who sends it to the event store as an event.

5.3 Aggregate Root

Because data in the CQRS pattern is created by different query model builders of which you do not necessarily know what or where they are, and because of the asynchronous way these listeners work you can not say anything about the correctness of data at the time of querying. As an example, think about a large web shop selling laptops. Whenever someone wants to order a laptop, the system needs to know whether the inventory is sufficient to approve the order. In other words, the system needs to be sure there is at least one laptop available before the order can be processed. In the core CQRS pattern, there is no way to know for sure the laptop is in stock, because all events are processed asynchronous. The only way to know for sure the laptop is in stock, is to store the number of laptops available together with with the laptop itself and also process this as one. If not, it is possible that the system checks whether a laptop is in stock, sees one laptop in stock, starts processing the order and ends up with an erroneous order since the laptop is sold just before through another process. The concept of storing and

processing all properties and entities that are dependent on each other together is know as aggregation. The main entity is called the entity root. An order, for example should always be processed together with its order lines, since the lines make no sense without the order. In the previously mentioned example, the order and order lines are an *aggregate* and the order is the *aggregate root*, since deleting the root would indicate deleting the other entities as well.

5.4 Command Handler

Commands coming in from the user interface have to be passed through to something that will perform the action dictated by the command. As discussed in section 5.3, these actions can be adequately performed by Aggregate roots, as observed within the design of ERPSoft. The command coming from the command bus has to be interpreted and translated somehow before it can be performed. A command handler is capable of catching one or more commands and passing it through to an object capable of performing the command [1].

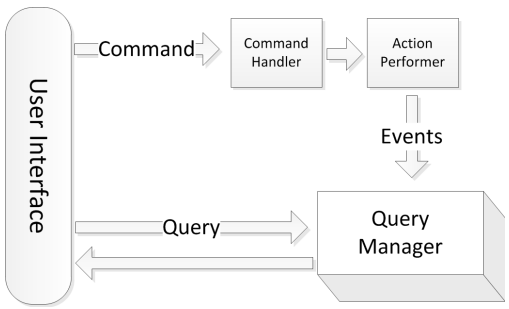


Fig. 5: Command Handler Sub Pattern

Figure 5 shows an overview of the command handler pattern. The action performer in the figure should some how make sure an action is actually performed. One way of doing this is using a two phase commit [9] in which a request is sent in two phases, but since this adds significant load to the system, other methods like delaying the sourcing of events until an aggregate root is totally finished are recommendable.

5.5 Query Model Builder

All events that are sent to the query manager can be caught by a query model builder, as discussed in section 4a. These query model builders can be everywhere from the clients cache to on all kind of different physical servers. The QMBs listen to events coming in through the event bus, and create a view of the data needed by the query manager. This view totally depends on the domain the QMB is in and the goal the data has. A QMB in a system responsible for generating inventories, for example, will build entirely different query models than a QMB in a system responsible a displaying the contact details of one person. Figure 6 shows a representation in combination with the query handler pattern discussed in section 5.6.

5.6 Query Handler

Queries sent by the user interface should be translated somehow in order to know what should be send back. The QMB only creates views of the data, but does not know how to relate this to the user interface. The use of a query handler can solve this problem by implementing a component able of receiving all queries and checking the query store for views created by the QMB [21].

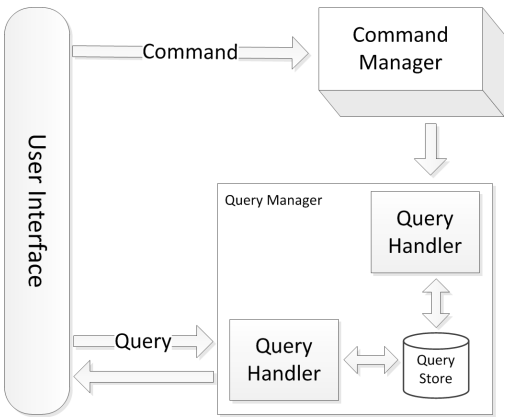


Fig. 6: QMB and Query Handler Sub Pattern

Figure 6 shows a combination between the QMB pattern (section 5.5) and the query handler pattern. The concept of a query store is introduced to store queries build by the QMB. This store is not obligatory, but can improve the response time of the system.

5.7 Snapshotting

It is common practice in the CQRS pattern to only store changes (events) and no states. This is because states can always be determined based on all the changes happened in the system so far. Rerunning al events will bring the system back in its last state after a possible system crash. States only occur in aggregate roots (see section 5.3), but recovering the state of an aggregate root after a system crash can be quite intensive, since aggregate roots often stay active in the system for a long time. A solution to this problem is the use of snapshotting. In the snapshotting pattern, the state of the aggregate root is stored together with the events every n^{th} event. The exact value of n depends on the processing load storing and monitoring the state of the aggregate root gives. When the system crashes, the latest stored state is recovered and only the events happened after this state storage have to be rerun. The snapshotting pattern is often used in combination with the memento pattern [7] that provides the ability to restore objects to their previous state.

6. VARIABILITY INFLUENCES

Applying the CQRS pattern in a software product does not immediately influence the level of variability of a software product. Applying CQRS in combination with sub patterns identified in this case study however, does have a positive effect on the variability level of a software product. On a functional level, it becomes possible to comply to specific customer groups or branches of industry having their own specific requirements.

Figure 7 shows how applying the *Event Sourcing* and *Event Store* patterns (section 5.1) can help in offering specific functionality to different industrial sectors. The example is an adapted version of a design observed at ERPSoft. The figure shows three different (A, B and C) sectors, but this can differ per implementation. In the system, one core system is created containing the functionality that is shared by all sub systems. For example, a CRM system, having specific sub systems for sectors like retail, furniture and bakeries. All domains would share names and addresses for customers, so commands related to this would be performed by *Aggregate Roots* in the core CRM system. All branches would listen to events broadcasted by the core CRM system and build query models based on events that are relevant for them. Operations on attributes or entities that only exist on one of the sub systems (for example a membership card number for retail) will only be processed within the specific sub system. The core system should only receive commands and does not have to be able to process queries, since

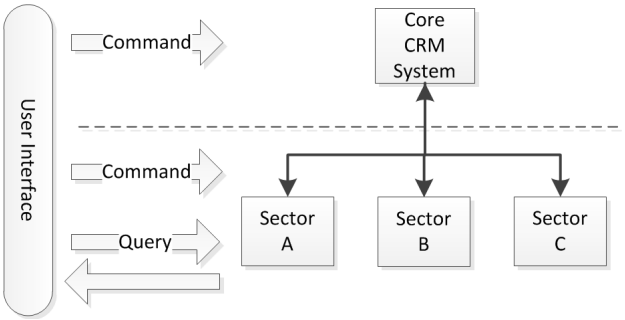


Fig. 7: Example of variability due to CQRS

the sector specific sub systems handle the representation of specific data. By identifying the different requirements of customers and grouping them in different sub systems, the level of variability possible within a software product will be high.

The possibility of running *Query Model Builders* (section 5.5) at the client side, also opens possibilities for customer specific requirements that are not shared with other customers. Custom *QMBs* and *Query Handlers* can be developed and implemented at customers, allowing them to perform the specific task needed for their business process. The customer specific listeners listen to events broadcasted and can react in a way specific for the wishes of one customer. The location of deployment does not play a role, making it possible to run QMBs at the customer, but also at third parties. Overall, as observed within ERPSoft, the CQRS pattern enables software vendors to create a software product better capable of complying to all kind of different customer requirements and by this achieve a high level of variability. This is primarily caused by the possibility to distribute events to specific event listeners and the ability to handle those events in a way that can be customer or customer type specific.

7. DISCUSSION AND FUTURE RESEARCH

Software products designed according to CQRS and sub patterns identified in this research can profit from an optimal configuration of data stores in such a way that it is geared towards a specific task (i.e. storing or reading data). By this, the CAP theorem can be less of a problem than it would have been if one data store had to do all tasks. The distributed asynchronous way in which events are handled, is primarily useful for business software product having a high concurrent load or a high need for variability. Business products, as analyzed within our case study, have both of the characteristics described above and will benefit from applying the CQRS pattern, including identified sub patterns, in terms of scalability, performance during load peaks and the level of variability.

The sub patterns reported on in the paper are all based on the patterns applied within the case company, cross checked with patterns currently described in CQRS related literature. The selection of patterns described is not a complete set of patterns related to CQRS. More and different patterns exist, but the patterns identified in this research are those used within ERPSoft. Further more extensive research at additional case companies can extend the collection of CQRS sub patterns and create an even more complete overview of CQRS related patterns.

Future research should also make clear when architects should choose certain patterns and how these different sub patterns work together to achieve some common goal. The characteristics of all patterns should be more extensively evaluated by domain expert to create a complete catalogue of all CQRS related patterns.

8. CONCLUSION

As the case study and variability example illustrates, the CQRS pattern can help in achieving a high level of variability in a software product. Different sub patterns are identified related to CQRS to solve specific problems within a CQRS based architecture design. This paper helps software architects by explaining the different sub

patterns and showing how they can influence the variability off a software product. We showed an implementation of the CQRS pattern, including seven sub patterns that are observed at ERPSoft. This example shows how implementing a CQRS based architecture instead of a multi-tier architecture can help in creating a software product capable of serving thousands of customers with variable product requirements. All identified sub patterns can be implemented together or individually to create, but none of them are obligatory for implementing the CQRS pattern. Some sub patterns, like the Event Sourcing pattern and the use of distributed query model builders, can contribute directly to the variability of a software product in a significant way. Other sub patterns however have a supporting role for the architecture, dealing with scalability, performance or consistency of the system. There is no perfect combination of sub patterns when it comes to CQRS since everything specific situation differs, but the pattern descriptions in this paper help in making a weighed decision for software architects.

Acknowledgement

Many thanks goes to our case company, their helpfulness and their lead software architect. We also want to thank Mark Nijhof, Udi Dahan and Greg Young for their input and Uwe van Heesch for shepherding our paper.

REFERENCES

R Abdullin. Theory of CQRS Command Handlers: Sagas, ARs and Event Subscriptions, 2010. http://abdullin.com/journal/2010/9/26/theory-of-cqrs-command-handlers-sagas-ars-and-event-subscrip.html.

F Buschmann. Pattern-Oriented Software Architecture, Volume 5: On patterns and Pattern Languages. 2007.

F. Buschmann, K. Henney, and D.C. Schmidt. *Pattern-Oriented Software Architecture, Volume 4: Pattern Language for Distributed Computing*. John Wiley & Sons Inc, 2007.

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, Chichester, 1996.

U Dahan. Clarified CQRS, 2010. http://www.udidahan.com/2009/12/09/clarified-cqrs/.

E. Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-wesley, Reading, MA, 1995.

S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, 2002.

J. Gray and L. Lamport. Consensus on transaction commit. *ACM Transactions on Database Systems (TODS)*, 31(1):133–160, 2006.

Alan R. Hevner, S.T. March, J. Park, and S. Ram. Design science in information systems research. *Mis Quarterly*, 28(1):75–105, 2004.

M. Jaring and J. Bosch. Representing variability in software product lines: A case study. *Software Product Lines*, pages 219–245, 2002.

Jaap Kabbedijk and Slinger Jansen. Variability in Multi-tenant Environments: Architectural Design Patterns from Industry. In *Lecture Notes in Computer Science 6999*, pages 151–160, 2011.

M. Kircher and P. Jain. Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management. 2004.

P.D. Manuel and J. AlGhamdi. A data-centric design for< i> n</i>-tier architecture. *Information Sciences*, 150(3):195–206, 2003.

B Meyer. *Object-oriented software construction*. Prentice Hall PTR, New York, New York, USA, 2 edition, 1988.

Ralph Mietzner, A. Metzger, Frank Leymann, and Klaus Pohl. Variability modeling to support customization and deployment of multi-tenant-aware Software as a Service applications. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, pages 18–25. IEEE Computer Society, 2009.

M Nijhof. Elegant Code > CQRS > Event Sourcing, 2010. http://elegantcode.com/2010/02/05/cqrs-event-sourcing/.

Peter Reason. *Three approaches to participative inquiry.* Thousand Oaks, CA, US: Sage Publications, Inc, 1994.

P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009.

D.C. Schmidt. *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. Wiley, 2000.

O Torkel. Queries & Aggregates & DDD, 2010. http://www.codinginstinct.com/2011/04/queries-aggregates-ddd.html.

R.K. Yin. *Case study research: Design and methods*. Sage Publications, Inc, 2009.

G Young. CQRS and Event Sourcing, 2010. http://codebetter.com/gregyoung/2010/02/13/cqrs-and-event-sourcing/.