Towards a Pattern-based Approach for Transforming Legacy COBOL Applications Into RESTful Web Services

Christoph Gaudl[1] and Philipp Brune[2,1]
[1]*SQ Solutions GmbH, Platz der Einheit 2, 60327 Frankfurt / Main, Germany*
[2]*Neu-Ulm University of Applied Sciences, Wileystraße 1, 89231 Neu-Ulm, Germany*

Keywords: Legacy Application Modernization, Microservices, Web Services, REST, Java, COBOL.

Abstract: Many aspects of modern life still depend on large-scale, monolithic legacy applications, e.g. in financial services, transport or public administration. Typically, these applications are written in ancient programming languages such as COBOL and use proprietary transaction processing monitors like CICS. While the modernization or replacement of these legacy application has been discusses in literature and practice for decades, still no universal solution exists. In many cases, an evolutionary modernization strategy has shown to be successful in practice, allowing to modernize the software architecture as well, not only the program code. Therefore, in this paper an analysis pattern is derived for transforming stateful, transactional COBOL programs into stateless RESTful web services. This pattern is evaluated by analyzing and transforming an example COBOL application to Java. While the approach shows to be useful in case of the example application, it needs to be further investigated in a broader range of real-world scenarios.

1 INTRODUCTION

In recent years, it became more an more obvious that not only legacy systems and COBOL applications still form the backbone of a large part of mission-critical enterprise IT, but also will continue to do so in the future, as their replacement or modernization is a complex and time-consuming task. E.g., in the United States, 43 percent of banking systems are built on COBOL, 80 percent of in-person transaction and 95 percent of ATM transactions rely on COBOL code (Reuters, 2022).

In example, in 2020, during to the pandemic, the case of the unemployment insurance in the US received worldwide media attention, when suddenly a large number of people tried to fill out their unemployment forms and the system went down. The systems managing these forms had been developed in COBOL and not updated for quiet some time (Makena, 2020). However, the main reason for this collapse was not COBOL as a programming language, but the fact that the systems where not updated and the COBOL code was not modernized for quite some time (Gregori, 2020). Also the neccessary investments for modernizing these appliactions had been considered too high for many years (Allyn, 2020).

This is one example of software aging (Huang et al., 1995): Not actively maintaining program code for a longer period of time leads to code degeneration and reduces its quality, until the code is considered legacy. This not only applies to ancient programm languages like COBOL or FORTRAN. but can also hold true for more modern ones like Java or C++ (Sneed, 2006; Srinivas et al., 2016; Knoche and Hasselbring, 2018).

Therefore, as most legacy systems are mission critical (Sneed and Erdos, 1996), their modernization remains an important and ongoing challenge in practice.In this paper, we discuss an evolutionary approach focusing on breaking up monolithic, transcational COBOL applications into web services (Knoche and Hasselbring, 2018), which then could be modernized and encapsulated. Today, various technological platforms exist for afterwards implementing and hosting such web services, be it on or off the mainframe platform (Sherill, 2015; Gohil et al., 2017; Brune, 2018).

While most early approaches for modernizing a legacy system nowadays could be considered legacy systems themselves (Canfora et al., 2008; Sneed, 2006), more recent works discussing a manual transformation into web services mainly focus on a lift-and-shift approach for non-transactional servics (Knoche and Hasselbring, 2018; De Lauretis, 2019),

Gaudl, C. and Brune, P.
Towards a Pattern-based Approach for Transforming Legacy COBOL Applications Into RESTful Web Services.
DOI: 10.5220/0011575300003318
In *Proceedings of the 18th International Conference on Web Information Systems and Technologies (WEBIST 2022)*, pages 159-165
ISBN: 978-989-758-613-2; ISSN: 2184-3252

moving the functionality off the mainframe completely.

Therefore, in this paper an analysis pattern for transforming transactional COBOL programs written for the IBM CICS transaction monitor into stateless web services is proposed and evaluated by means of a semi-realistic sample application. This approach is in principle agnostic with respect to the objectives of the modernization, be it a re-use of the existing COBOL code as part of a web service or a re-write in another language, and hosted on or off the legacy platform.

The rest of this paper is organized as follows: In section 2 the related work and previous modernization efforts are described in detail. Section 3 describes the proposed analysis pattern, which is evaluated by applying it to a semi-realistic example in section 4. Section 5 illustrates the actual re-implementation of the web services in Java. We conclude with a summary of our findings.

2 RELATED WORK

Legacy modernization has been discussed in the literature for a long time. Nevertheless, it remains an ongoing challenge for research and practice.

(Chiang and Bayrak, 2006) describe a process for extracting and reusing business rules from a given program. In this approach, program slicing is used. To find which components could be reused, the program code is divided into smaller parts. These parts are used to look into specific operation categories of the application. The first category is the user interface, the second category are database operations and the third is the business logic. The program point to slice must be the last statement for the respective category.

In (Comella-Dorda et al., 2000), the need for modernization is motivated by the goal of avoiding redundant code, and its adaptability to changing business requirements. System evolution can be achieved by maintenance, modernization and replacement of software components. Modernization is in between maintenance and replacement. It is used, when bigger changes to the software are needed, such as system restructuring, important functional enhancements or adding new software attributes. But in contrast to a replacement, by modernization major business values are preserved.

(Canfora et al., 2008) differentiate between three methods for modernization: redevelopment, wrapping and migration. Unlike (Comella-Dorda et al., 2000), redevelopment here is considered a form of modernization, defined as re-build of a software from scratch, on new hardware and with a modern architecture. Migration is defined as the process of moving the system to a new platform, while retaining the original data and functionalities. Wrapping refers to the encapsulation of existing data, programs, applications and interfaces with new interfaces.

For modernization on the mainframe platform itself, IBM earlier introduced e.g. the CICS Asynchronous API (Gohil et al., 2017). The basic principle of the asynchronous API is the parent-child model. A parent task can run multiple child tasks to execute logic asynchronously from the parent. Therefore, the logic executed by the children must be independent from the other children. Because all the parent and child tasks are CICS transaction themselves, the lack of transactional guarantees as mentioned in (Knoche and Hasselbring, 2018) is no problem here. The transactional guarantees are provided by CICS.

However, nowadays microservices are the state-of-the-art in application architecture. According to (Dragoni et al., 2017), a microservice is a cohesive and independent process, which is interacting via messages. In (Newman, 2021), a microservice is defined as a small and autonomous service, which works together with other microservices. Such services are the next step in softwae development, replacing the earlier after SOA paradigm. Therefore, in recent years predominantly the transformation of monolithic legacy applications into microservice architectures is discussed by many authors.

(Wolfart et al., 2021) give an overview by analysing multiple studies to derive a roadmap for modernizing legacy systems with microservices. The proposed roadmap consists of eight steps, grouped into four major phases: initiation, planing, execution and monitoring of the developed microservices.

Another approach was proposed by (De Lauretis, 2019). Here, the proposed strategy is based on five steps to go from the monolithic to the microservice architecture. The first step is the function analysis of the existing architecture. In the second step Business functionalities are identified and analysed in the third step. In the following fourth step the business functionalities are theoretically assigned to new microservices, and in the last step the microservices are implemented.

(Knoche and Hasselbring, 2018) also propose a five step modernization process: In the first step, an External Service Facade is defined, so microservices can have entry points to the system. In the second step, these External Service Facade are adapted to the existing application. In a third step, the clients is connected to the new Service Facade. In the fourth step, an Internal Service Facade is developed. This

step could also be completed parallel to the first three steps.

The authors chose not to do this, because of the lack of resources and the high risk to failure. In the last step, the old services can be replaced by microservices, which still are based on COBOL but are only accessed through well defined interfaces. These steps where all done manually and took almost four years to complete. With this method, so far mainly read-only operations where converted, because of the lack transactional guarantees microservices can provide.

The analysis of the related work revealed a major difference between older ($>$ 5 years) and newer research: While in the older work the focus was more on the development of an explicit solutions for modernization of a given application, in the newer papers the focus is on developing patterns for modernization in general.

Therefore, in this paper the question is addressed, how these approaches could be combined to modernize a given application with the use of microservices, using a generic pattern while still preserving transactional guarantees, such that read and write functionalities could be implemented.

3 A PATTERN FOR LEGACY COBOL CODE ANALYSIS

The first step in modernizing any legacy system is the analysis of the given application. This analysis is necessary to identify the business logic of the given code. After the business logic of the code is analysed, the code sections could be identified, which could be used for creating a microservice-like structure.

For CICS COBOL programs, these sections can be identified by using the embedded EXEC CICS statements. These statements define the different types of functions executed in the transaction. There are a variety of possible functions used in transactions. Depending on the functions different modernization options are possible. To get away from a monolithic application, in a first step user interface and data operations need to be separated. This is needed to implement a client-server structure. Here, the server should become stateless, so that the process state is only implemented in the client and the server only handles the data operations. The communication between server and client then works with web APIs.

To achieve this, the following sequence of three generic steps is proposed:

1. Analyze the user interface maps or mask definitions (UI screens), so that the user interface can be implemented in the client. If the application, like the one used in this paper, depends on user inputs, the fields in the map can be mapped to input fields in a web form.
2. Identify the so called interface commands. These are given by the EXEC CICS RECEIVE and EXEC CICS SEND functions (see figure 1 for an example). These functions are used to take data from the mask or send data to the mask. For the client server structure, these mark the communication points between server and client.
3. Analyze the other EXEC CICS statements, so the possible services can be identified. This step is highly individual for each given application. But in this work, a roadmap on how to analyse a given application is presented in the following.

4 EVALUATION

The evaluation was performed by transforming a semi-realistic sample CICS COBOL application provided by SimoTime Technologies for educational purposes (Simotime, 2022). As this is one of the few available non-trivial CICS example applications on the web, it is has been used also in other papers as well (Brune, 2018). It consist of three top-level programs, which were analyzed in this study.

The COBOL fragment in figure 1 as taken from the customer inquiry program, one of these three sample programs. In the following, this is analyzed as an example. A similar analysis was performed for all three programs.

The code in figure 1 shows the typical use of a UI map or mask, which allows the user to view any existing customer VSAM datasets by inserting the customer number. Like in many other COBOL and CICS implementations, the UI and the programming logic are not separated between host and client. So the first part of the PROCEDURE DIVISION is used to manage the masks and get back to a mask higher up in the hierarchy. This section of the code, which only contains UI logic, in a modern application would be implemented in the client, so the server would remain stateless.

The next section of the code gets the customer informations for a given customer number and arranges it so the information can be displayed by the mask. Afterwards, the program reads the information entered by the users into the mask. This can be identified by the EXEC CICS RECEIVE statement. This is the point where the user input is checked, if it applies to all the rules for a customer number. The fact that this

```
*****************************************************************
 END-OF-PROGRAM-12.
      EXEC CICS SEND
           FROM    (EOJ-MESSAGE)
           LENGTH (50)
           ERASE
           END-EXEC

      EXEC CICS SEND
           CONTROL FREEKB
           END-EXEC

      EXEC CICS RETURN
           END-EXEC
      exit.

*****************************************************************
 GET-CUSTOMER-INFORMATION.
      perform GET-CUSTOMER-NUMBER
      perform GET-CUSTOMER-RECORD
      perform POSSIBLE-SPECIAL-HANDLING
      perform SEND-INQUIRY-SCREEN
*     Should never get to here...
      exit.

*****************************************************************
 GET-CUSTOMER-NUMBER.
      EXEC CICS RECEIVE
           MAP     ('CU2INQ1')
           MAPSET ('CU2INQ1')
           RESP(WS-RESPONSE)
           RESP2(WS-REASON-CODE)
           END-EXEC.

      if  REQNUMBL > ZERO
          move REQNUMBI to Z-WORK-12
          perform Z-RIGHT-ADJUST-Z-WORK-12
          move Z-WORK-12 to REQNUMBI
          move REQNUMBI to LS-CUSTNO
      else
          move 'Please enter a 12 digit customer number...3'
            to MSG-LINES
          perform RECOVERY-ROUTINE
      end-if.
      exit.
```

Listing 1: COBOL code fragment from (Simotime, 2022) illustrating the use of EXEC CICS statements.

check appears similar in two programs makes this section a good part to be implemented as a microservice, as it is small in size and used multiple times.

The next routine which follows is GET-CUSTOMER-RECORD. In this routine the customer dataset is first read form the dataset of customers and then mapped to the fields on the mask of the UI. The first part if reading from the dataset is identified by the EXEC CICS read statement. In a microservice architecture this can be represented by sending a message with the customer number to a microservice, which searches a database or dataset and then returns the dataobject.

The second part of this routine assigns the received data to the equivalent positions on the mask. This part could be solved on the client site in a modernized approach, because this assignment only is used for display purposes. The keywords to identify such a functionality is the 'MOVE [..] TO' combination.

The following routine SEND-INQUIRY-SCREEN is also a routine to display the output onto the screen. This part of the code could also be handled on the client side of a modern architecture, e.g. with a JavaScript client.

The two routines Z-RIGHT-ADJUST-Z-WORK-

12 and Z-GET-DATE-AND-TIME also exists in all three of the programs analyzed in this work, so an approach to design them as a microservice is obvious. The first of these two routines takes the user input of the customer number and adjusts it so that the length is always 12 characters. The second routine is used to set the current time onto the screen for the user. The last routine Z-GET-DATE-AND-TIME-CICS uses the build in CICS method ASKTIME to update the time to the current time and not the initial time, when the task started.

After a deeper look into the code fragments, entry points for microservices were identified. The first pattern which should be looked for are the EXEC CICS SEND and RECEIVE statements. Between these parts of the code, normally operations on data are executed. This is a typical task for the server and not the client.

Another pattern to look for are the 'MOVE [...] TO' statements. These statements normally are used when data is assigned to different variables, either for operative purpose or displaying purpose. In a microservice architecture these are the communication points. Here the client sends the information via message to the server and the corresponding microservice, so the specific operation can be executed.

This analysis demonstrates, how complex even such small projects are, when implemented as a monolithic application. The disadvantages of the monolithic approach can also be seen in the doubling of the routine to adjust a customer number to 12 digits and the routine to get the current time.

In an approach using microservices, these routines would be only implemented once and then can be used by other applications that needs this functionality.

5 TRANSFORMATION OF THE APPLICATION

To demonstrate a possible microservice implementation, the identified services were implemented as RESTful services with Java and JAX-RS, using the OpenLiberty application server[1].

The OpenLiberty server was chosen, because this application server could also be run on a mainframe under z/OS. For the purpose of this work, the functionality only was tested on a local machine.

In the first step, an OpenLiberty application server was set up on a local machine using version 22.0.0.5 with the Eclipse IDE version 2022-03(4.23.0). For the client, a simple JavaScript web client was implemented. The underlying database is an ObjectDB database. ObjectDB is an open source Object-Oriented Database Management System implementing the Java Persistence API (JPA)[2].

The first Java class to be implemented is the customer record. This class was build after the template of the respective COBOL copybook. So all required values from the customer files in the CICS application could be accessed in this application and saved into a database. For JPA, this class is used as the JPA entity.

After the customer class, the services needed be implemented. To replicate the given CICS code, the services to find and create a customer must be implemented. For each of this services, an extra method was written. Figure 2 shows the respective Java code fragment for this service class, built using JAX-RS.

The first service is the createCustomer method. The service can be accessed from the client by sending a POST request over the given API. With this call all the needed customer informations are sent from the client to the server, by the use of the service API. Then in the function it is checked if the customer already exists. If not then the newly created customer is saved to the ObjectDB database using the EntityManger. After that the created customer is sent back to the client so it can be displayed for the user.

The second service is the findCustomer method. This service can be accessed by a GET request over the given API. The client sends the Id to the server over the API. The function then uses the EntityManager to find the wanted Customer using the Id. If the customer is found it is returned to the client so it can be displayed for the user.

The routine Z-RIGHT-ADJUST-Z-WORK-12 in the program in figure 1 can be replaced by the minlength and maxlength option on the input label in the HTML5 frontend. This options only allows inputs that are exactly 12 digits long. So the user cannot input an ID which not complies with the length of 12.

In this way, the application was re-implemented completely in Jakarta EE with the use of an OpenLiberty server. This makes the application more open for future changes to the application. For example, to add the functionality of updating customers, now only an extra service needs to be added into the existing application. Then the client needs to be adjusted so the new service can be accessed. After that the User can access the service over the client and not need to access an other sub program like in the CICS version.

[1] https://openliberty.io

[2] https://www.objectdb.com

```
@Stateless
@Path("server")
@Consumes(MediaType.APPLICATION_JSON)
public class ServerEJB {

        @PersistenceContext(unitName="CustomerDB")
        private EntityManager em;

        private static final Jsonb jsonb = JsonbBuilder.create();

        @POST
        @Path("create/{id}/{fn}/{mn}/{ln}/{a1}/{a2}/{c}/{s}/{pc}/{ph}/{pw}/{pce}")
        @Produces(MediaType.APPLICATION_JSON)
        public Customer createCustomer(
                @PathParam("id") Long cusnumber,
                @PathParam("fn") String firstname,
                @PathParam("mn") String midname,
                @PathParam("ln") String lastname,
                @PathParam("a1") String address_1,
                @PathParam("a2") String address_2,
                @PathParam("c") String city,
                @PathParam("s") String state,
                @PathParam("pc") String postalcode,
                @PathParam("ph") String phone_home,
                @PathParam("pw") String phone_work,
                @PathParam("pce") String phone_cell) {


                Customer customer = new Customer (cusnumber, firstname,
                  midname, lastname, address_1, address_2, city, state,
                  postalcode, phone_home, phone_work, phone_cell);
                Customer testcustomer;
                testcustomer = em.find(Customer.class, cusnumber);
                if (testcustomer == null) {
                        em.persist(customer);
                }

                return customer;
        }

        @GET
        @Path("find/{cusnumber}")
        @Produces(MediaType.APPLICATION_JSON)
        public Customer findCustomer(@PathParam("cusnumber") long cusnumber) {
                Customer c = em.find(Customer.class, cusnumber);
                return c;
        }

}
```

Listing 2: Java code fragment of the new implemenation of the customer service.

6 CONCLUSION

In conclusion, in this paper a pattern to transform monolithic transactional COBOL applications into stateless web services was proposed. It was evaluated by breaking down the given code of an existing semi-realistic sample application into pieces, which later where used to implement a modern application. This method could be used for any CICS application.

After the explanation on how to break down and find the modernization points in an application, a methodolgy to modernize the application was shown. In this case, a new Java application was developed, with the OpenLiberty server taking over the part of

the CICS transaction monitor. To retain the full functionality of the original application, also a web client was implemented, which uses the services provided by the server. This client took over the functionality of the maps in the old application.

With this implementation, a microservice-based client-server architecture was achieved. Because of the use of microservices, new functionalities only need to add an API so that the client can access them, and the data required by the client must be defined. Then the client can be adopted to the new services. So the development of the logic, which is handled by the client, and the operations on the data are separated, and the server becomes stateless.

The next step for this application should be an integration of the CICS with the Liberty Server itself, so it can be shown that a modernization can be possible with reusing the underlying COBOL and CICS code.

Another promising possibility could be the development of an automated pattern recognition algorithm, following the pattern presented in this work, which could make use of machine learning techniques to further automate legacy modernization.

REFERENCES

Allyn, B. (2020). 'cobol cowboys' aim to rescue sluggish state unemployment systems. https://www.npr.org/2020/04/22/841682627/cobol-cowboys-aim-to-rescue-sluggish-state-unemployment-systems. visited: 07/06/2022.

Brune, P. (2018). An open source approach for modernizing message-processing and transactional cobol applications by integration in java ee application servers. In *International Conference on Web Information Systems and Technologies*, pages 244–261. Springer.

Canfora, G., Fasolino, A. R., Frattolillo, G., and Tramontana, P. (2008). A wrapping approach for migrating legacy system interactive functionalities to service oriented architectures. *Journal of Systems and Software*, 81(4):463–480.

Chiang, C.-C. and Bayrak, C. (2006). Legacy software modernization. In *2006 IEEE international conference on systems, man and cybernetics*, volume 2, pages 1304–1309. IEEE.

Comella-Dorda, S., Wallnau, K. C., Seacord, R. C., and Robert, J. E. (2000). A survey of black-box modernization approaches for information systems. In *icsm*, pages 173–183.

De Lauretis, L. (2019). From monolithic architecture to microservices architecture. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 93–96. IEEE.

Dragoni, N., Giallorenzo, S., Lafuente, A. L., Mazzara, M., Montesi, F., Mustafin, R., and Safina, L. (2017). Microservices: yesterday, today, and tomorrow. *Present and ulterior software engineering*, pages 195–216.

Gohil, P., Horn, J., He, J., Papageorgiou, A., Poole, C., et al. (2017). *IBM CICS Asynchronous API: Concurrent Processing Made Simple*. IBM Redbooks.

Gregori, S. (2020). Cobol isn't the issue: A misinterpreted crisis. https://hackaday.com/2020/04/20/cobol-isnt-the-issue-a-misinterpreted-crisis/. visited: 07/06/2022.

Huang, Y., Kintala, C., Kolettis, N., and Fulton, N. D. (1995). Software rejuvenation: Analysis, module and applications. In *Twenty-fifth international symposium on fault-tolerant computing. Digest of papers*, pages 381–390. IEEE.

Knoche, H. and Hasselbring, W. (2018). Using microservices for legacy software modernization. *IEEE Software*, 35(3):44–49.

Makena, K. (2020). Unemployment checks are being held up by a coding language almost nobody knows. https://www.theverge.com/2020/4/14/21219561/coronavirus-pandemic-unemployment-systems-cobol-legacy-software-infrastructure. visited: 07/06/2022.

Newman, S. (2021). *Building microservices*. " O'Reilly Media, Inc.".

Reuters (2022). Cobol blues. http://fingfx.thomsonreuters.com/gfx/rngs/USA-BANKS-COBOL/010040KH18J/index.html. visited: 07/06/2022.

Sherill, S. (2015). Building an api bridge to your ibm z systems applications and data.

Simotime (2022). Cics cobol example application. http://www.simotime.com/sim4home.htm. visited: 16/05/2022.

Sneed, H. M. (2006). Integrating legacy software into a service oriented architecture. In *Conference on Software Maintenance and Reengineering (CSMR'06)*, pages 11–pp. IEEE.

Sneed, H. M. and Erdos, K. (1996). Extracting business rules from source code. In *WPC'96. 4th Workshop on Program Comprehension*, pages 240–247. IEEE.

Srinivas, M., Ramakrishna, G., Rao, K. R., and Babu, E. S. (2016). Analysis of legacy system in software application development: A comparative survey. *International Journal of Electrical & Computer Engineering (2088-8708)*, 6(1).

Wolfart, D., Assunção, W. K., da Silva, I. F., Domingos, D. C., Schmeing, E., Villaca, G. L. D., and Paza, D. d. N. (2021). Modernizing legacy systems with microservices: A roadmap. In *Evaluation and Assessment in Software Engineering*, pages 149–159.