2014 IEEE International Conference on Software Maintenance and Evolution

Autonomic Software Systems

Developing for Self-Managing Legacy Systems

James J. Mulcahy
Computer & Electrical Engineering
and Computer Science
Florida Atlantic University
Boca Raton, FL, United States
jmulcah1@fau.edu

Shihong Huang
Computer & Electrical Engineering
and Computer Science
Florida Atlantic University
Boca Raton, FL, United States
shihong@fau.edu

***Abstract*— Modern software systems have grown in complexity and expense, even while the cost for supporting hardware has decreased over time. Humans have a lot to do with why software is expensive, and they contribute to its cost in at least three significant areas: the maintenance and evolution of existing software, the run-time monitoring and configuration of executing software, and errors made during data entry and system configuration tasks. Software engineers seek to mitigate these costs by minimizing or removing expensive human participation in these areas where possible by adopting software and hardware approaches aimed at doing so. In this paper, we describe a commercial software engineering project where code reuse, service-oriented architecture, and self-* autonomic approaches were employed to extend the legacy enterprise system of a multi-channel vendor of musical equipment. In adopting these approaches, the developers were able to produce a highly-automated extension to an existing system that increased the number of orders places by customers, extending the business value of that system.**

Keywords—software engineering, software evolution, legacy software systems, autonomic software systems; self-adaptive software; service-oriented architecture; fault tolerance

I. INTRODUCTION

One of the many challenges in software engineering today is the evolution of existing legacy commercial software systems, particularly those of multi-channel retailers. Stakeholders seek to take advantage of approaches that minimize or remove the expensive human component in workflows in order to increase volume and reduce errors – all targeted at cutting costs and increasing profits. An even greater challenge is doing so while adding minimal complexity to the existing system, and without reducing performance or reliability. It is not uncommon to encounter legacy systems that are years or decades old, and are themselves the "bricks and mortar of many complex systems", increasing the potential that new faults and inefficiencies are added when the system is extended [2]. Engineers strive to adopt best practices and to design solutions that mitigate this challenge.

Software engineers have adopted a number of approaches to tackle the challenges of evolving legacy software systems, including the reuse of code and existing architecture when engineering new workflows that are similar to other existing workflows, the adoption of service-oriented architecture design patterns to provide fault-tolerant, loose coupling between systems, and the development of software that has self-managing autonomic attributes. Using "technology to manage technology" [3], the software is developed to take over roles normally held by humans in monitoring and reconfiguring systems in real-time to adapt to changing conditions related to data volume, connectivity, et cetera.

"Autonomic computing" is a term first coined by IBM just after the turn of the 21st century, and defines a system that is designed to be self-managing – that is, self-monitoring, self-configuring, and self-healing. Also referred to as a self-adaptive system, it can reduce the necessity for human monitoring and intervention, except for critical cases that cannot be solved automatically. A self-monitoring capability allows a system to continually check to confirm that it is running within expected parameters. A self-configuring capability allows the system to automatically alter its configuration at run-time to take advantage of or to mitigate current conditions when the system is no longer performing within those expected parameters. A self-healing system is able to recognize that an anomaly has occurred and configures itself to correct that anomaly [1][5][6].

In this paper, we show how basic principles of autonomic computing, along with software reuse and service-oriented architecture approaches were used in a real-world commercial project to automate the processing of orders originating from a third party website. The result was a low-complexity, high-value solution that, once implemented, required no direct human involvement to monitor and maintain the new revenue stream.

The rest of this paper is organized as follows: in Section II, we describe the motivation and background for the project, while in section III we detail the implementation. In Section IV we briefly discussed the results and offer our conclusions and suggestions for future work.

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.92

II. BACKGROUND

The stakeholder behind this project was a well-established multi-channel retailer of musical equipment that derived revenue from multiple sources. In multi-channel commerce, it is common for retailers and wholesalers to seek revenue through these several channels to capture as large a market as possible (Fig. 1). Prior to the appearance of the World Wide Web, customers physically visited brick-and-mortar locations, where they shopped and purchased products directly. Alternatively, some customers shopped from catalogs or direct mail advertisements, and placed their orders via conventional postal mail. A third revenue channel involved telephone communication, with customers either initiating calls to the retailer with the intention of placing orders, or sales staff for the retailer "cold calling" customers and suggesting products.

The arrival of the public internet and of the World Wide Web in the 1990s brought with it the potential for transactions initiated and handled entirely online. By 1991, restrictions were lifted upon the use of the internet for commercial purposes, and by 1995 the business model for electronic commerce – or "ecommerce" – had appeared. It became increasingly common for retailers and wholesalers to develop and host their own websites to leverage new revenue streams. Along with conventional companies, ecommerce-centric companies like Amazon.com and Ebay.com appeared with a business model that consisted of websites and warehouses, but no physical stores or catalogs.

Eventually, companies like Amazon began to expand their business model to include serving as a marketing partner for retailers by listing and selling their products on the Amazon.com website in exchange for a small commission. With this model, retailers and wholesalers could partner with companies like Amazon to take advantage of the brand name and marketing power of a well-established ecommerce company, while paying a small per-order charge in exchange for taking advantage of Amazon's own supporting infrastructure, like web servers and data stores, but also workflows that are related to the shopping for and placement of orders, electronic payment by the customer, and the initiation of product returns and exchange requests.

The stakeholder originally set up an account with Amazon to explore the possibility of increasing revenues, and posted a small subset of their products for sale on Amazon's website. The retailer's storefront on Amazon.com initially generated fewer than one hundred orders per day. At predefined intervals, a customer service employee of the retailer was tasked with using Amazon's browser-based interface to log onto the Amazon account, check for pending orders, and to manually enter each of them into the retailer's own enterprise system, using its own existing order entry software.

To encourage efficient order workflows between retailers and Amazon, the Amazon Marketplace Web Service (Amazon MWS) API was developed and exposed to help retailers automate the process of exchanging data with Amazon. It was this API that the stakeholder intended to leverage to increase web-based revenue streams. Automating the retrieval and processing and acknowledging order data from Amazon became the primary functional requirement for this project.

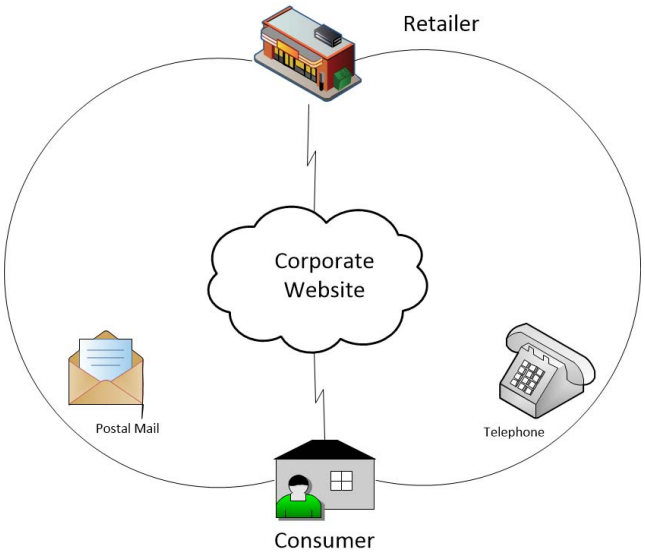


Fig. 1. Multi-Channel Commerce Model

The stakeholder established a non-functional requirement that the resulting system minimize the need for any humans in the loop between the online shopper and the personnel preparing the product(s) for delivery. In the "happy day scenario", the resulting system would collect and process incoming orders and reply back to Amazon with appropriate order acknowledgement information. Further, the system would be *self-monitoring* by detecting anomalous conditions during the workflow process and sending automated emails to customer service or technical staff for manual intervention. The system would be *self-configurable* by designing it to automatically adjust the frequency of the collection and delivery of order files and order acknowledgements according to certain thresholds. The system would be *self-healing* in a passive sense, with a workflow designed to be tolerant of unexpected faults, like a script or application crash, lack o connectivity to external systems, a server shutdown, et cetera [6].

III. IMPLEMENTING THE SOLUTION

The enterprise resource planning (ERP) software normally operated by the retailer was a versioned legacy system developed by a third party consisted of applications written in COBOL, running on an HP3000 platform using the MPE/iX operating system and the TurboIMAGE database management system. The retailer already owned the architecture it used to host its own retail website, a web server that the HP monitored

via a TCP/IP connection. It also owned a SOAP server that the HP had File Transfer Protocol (FTP) access to. Jobstream processes (scripts) written in job control language (JCL) were typically scheduled manually from the command line, or executed as part of a standard, scheduled "begin day" or "day close" process performed each day. Environment variables were used for inter-process communication, and were accessible from the command line or from any application. These type of variables have persistent values, and are accessible and modifiable in real-time from command line, jobstreams and individual applications.

For this project, the solution needed to provide a bridge between Amazon's MWS API and the retailer's ORDERPROC API. ORDRPROC was part of the existing legacy enterprise software and already used to import orders placed on the company's own website into the ERP database for processing. The existing SOAP server was repurposed to handle the exchange of data files between Amazon and the retailer (Fig. 2). Using service-oriented architecture approaches like SOAP allows for loose coupling between systems like Amazon and the retailer – either or both systems could crash or lose connectivity without directly affecting the other. Reducing the impact of the solution on the HP, using a SOAP server frees the HP from the expense involved in regularly polling Amazon's system in order to send or retrieve files. In this manner, the frequency of polling could be set to the roughly ten minute intervals at which Amazon produces batches of orders files, without requiring any process on the HP to adhere to that interval.

The implementation required the reuse of one existing COBOL program, the creation of two new programs to translate order-related data to and from Amazon's prescribed XML format, and three new jobstream processes (Fig. 3). For this project, ORDRPROC was reused to perform the same task with incoming Amazon orders.

The AMZGET jobstream was written to periodically poll the SOAP server for order files and translate the files from XML to the incoming data format expected by ORDRPROC, using a newly developed data transformation program named AMZORDR. After each order file was translated by AMZORDR, AMZGET rescheduled itself to execute again after some interval. An environment variable, GETFREQ, contained the value for this interval. The first value in the environment variable determined how many minutes in the future to schedule the next execution of AMZGET; the second value determined the threshold (in the number of order files detected) at which the jobstream would shorten or lengthen the interval of the first environment variable. This self-adaptive behavior was designed so that AMZGET autonomously adjusted the frequency at which files were collected from the SOAP server. If more files were found than the environment variable threshold value, the frequency

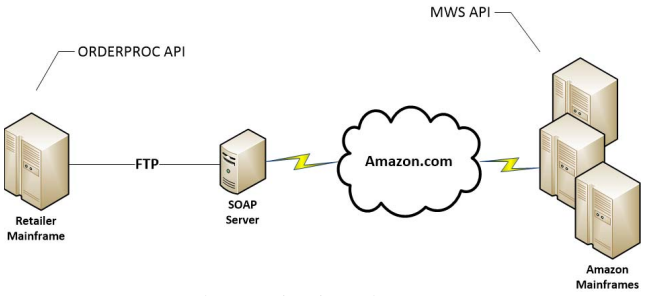


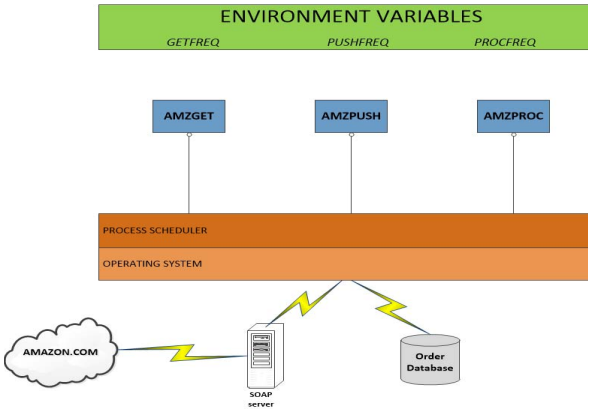
Fig. 2. Order Flow using SOAP

was increased, to reduce the chances of the SOAP server using too much disk space. If fewer files were found than the threshold value, the frequency was slightly decreased instead.

A second jobstream named AMZPROC was developed to periodically check a folder on the HP for translated order files from the SOAP server waiting to be imported into the ERP database. Order files were processed using the ORDRPROC application, resulting in the orders being either imported into the database or rejected. Rejected orders were reported to customer service via an automated email generated by the AMPROC jobstream. A new program, AMZACK, was executed as the next step in the process, to create an XML file for each accepted order in Amazon's prescribed format. The resulting order acknowledgement files contained a payload that included the retailer order number and shipping information, such as a tracking number. These files were later picked up by the third jobstream in the solution, AMZPUSH. Like AMZGET, AMZPROC accessed an environment variable named PROCFREQ that controlled the interval at which the jobstream executed, along with a file threshold for determining whether that interval should be altered.

The third jobstream, AMZPUSH, was designed to be the counterpart for AMZGET. Instead of checking the SOAP server for order files to copy to the HP, it checks the HP for order acknowledgement files to copy to the SOAP server, where they would be transmitted them to Amazon.

Fig. 3. System Architecture



Each of the three new jobstreams was designed to be fault-tolerant. That is, the first step of each one was to check its environment variable for the next interval to execute after, and rescheduled a copy of itself to execute after that interval. If any subsequent step of the currently running jobstream crashed, human intervention wouldn't be required to manually execute the process; its descendant would have already been scheduled for a future execution time.

Once implemented, the environment variables affecting the execution frequency of each jobstream were each initially set to a value of ten minutes, and executed for the first time each day at staggered times by the "begin day" process. Each stream adjusted its environment variable over time as necessary based upon changing volume of order files and order acknowledgements, centering about an initial "home" value of ten minutes. The dynamic nature of environment variables in MPE/iX means that technical staff would be able to modify the "home" values of any stream at run-time without disrupting the workflow. Subsequent jobstreams would simply continue to dynamically adapt, using the adjusted environment variable values as the new "home" value.

IV. RESULTS AND FUTURE WORK

After configuring the SOAP server and FTP software, and implementing the two newly engineered XML translators and three new jobstreams, and setting up the appropriate environment variables with default execution intervals (10 minutes) for each jobstream, the system was tested with mocked up data to simulate orders being transmitted to Amazon. Connections to the SOAP server were intermittently interrupted to observe how the jobstreams recovered. Eventually the system was released into production after technical staff adjusted the GETFREQ and PUSHFREQ intervals from ten minutes to six. In the weeks that followed the release of the implementation, the system experienced intermittent connectivity issues, high volume due to a holiday shopping season, and an unscheduled system crash. Although system administrators monitored the new processes over several weeks, no intervention was required. The solution was considered a success, proving its efficacy over a busy shopping season that saw the initial volume or orders from Amazon.com rise from under one hundred a day to over two thousand a day.

Future work includes implementing similar solutions to handle product returns, inventory reports and financial reconciliation workflows using the same Amazon MWS API. The solution's autonomic properties can be later enhanced by making the jobstreams self-optimizing, by adding machine-learning algorithms to anticipate future changes in order volume from historical order collection and processing data, and altering appropriate intervals accordingly in advance, or "just in time".

V. CONCLUSIONS

Software systems today are far more complex than their predecessors. In prior decades, software was written to augment the work performed mechanically by machines and by humans. Now we write software to replace machines where possible and to automate tasks normally performed by humans where practical. In replacing tasks once performed mechanically, we reduce the cost for hardware, along with the cost of human effort to maintain, repair and upgrade that hardware. In automating tasks once performed by humans, we reduce or eliminate the ever-increasing cost of skilled human labor. Removing humans from the loop also reduces costs associated with human error in data entry, system configuration, and system monitoring activities.

In this work we discussed the implementation of a commercial software engineering project that extended the business value of a legacy system by automating the processing of orders sold on a third party website. Using code reuse, service-oriented architecture, and autonomic computing approaches, a viable, low-complexity but high-value version was produced that completely eliminated humans between the online shopper and warehouse personnel for this commerce channel.

REFERENCES

[1] D. Garlan, "A 10-year Perspective and Software Engineering Self-Adaptive Systems," Keynote at the 8th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, May 2013.

[2] Y. Brun, G. Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M., and M. Shaw. "Engineering Self-Adaptive Systems through Feedback Loops," in Software Engineering for Self-Adaptive Systems. Springer Berlin Heidelberg, 2009. 48-70.

[3] IBM, "architectural blueprint for autonomic computing."; http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf 2005. (last accessed 1 July 2014).

[4] S. Neti, H. A. Müller, "Quality criteria and an analysis framework for self-healing systems," in International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'07). 2007.

[5] T. Haupt, "Towards mediation-based self-healing of data-driven business processes," in Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on. IEEE, 2012.

[6] J. O. Kephart, and D.M. Chess, "The vision of autonomic computing," Computer , vol. 36, no. 1, pp. 41-50, Jan 2003.