Workshop Preprints

CSE 2017

2nd Workshop on Continuous Software Engineering

co-located with SE 2017
Hannover, February 20th , 2017

Editors:
Stephan Krusche. TU München
Horst Lichter, RWTH Aachen University
Dirk Riehle, FAU Nürnberg
Andreas Steffens, RWTH Aachen University

Preface to
2nd Workshop on Continuous Software Engineering

Stephan Krusche [1], Horst Lichter [2], Dirk Riehle [3]Andreas Steffens [4],

1 Introduction

In order to develop and deliver high-quality products to their customers, software companies have to adopt state-of-the-art software development processes. To face this challenge, companies are applying innovative methods, approaches and techniques like agile methods, DevOps, continuous delivery, test automation, infrastructure as code or container-based virtualization. These new approaches have a high impact on the specification, design, development, maintenance, operation and the evolution of software systems. Therefore, common software engineering activities, organizational forms and processes have to be questioned, adapted and extended to ensure continuous and unobstructed software development.

After the successful 1st Workshop on Continuous Software Engineering (CSE 2016) [SEW16], held in Vienna, the organizers of the 2nd workshop wanted to widen the scope of the workshop. Therefore, the call for papers and the list of workshop topics were adjusted to the following:

Processes and workflows

- Change management and handling user feedback
- Software development lifecycle for CSE
- Continuous delivery for requirements engineering
- Lean agile processes and practices

Technologies and tools

- Infrastructure as code
- Provisioning of software and infrastructure
- Application virtualization with container
- Engineering of deployment pipelines

[1] Technische Universität München, Chair for Applied Software Engineering, krusche@in.tum.de
[2] RWTH Aachen University, Research Group Software Construction, lichter@swc.rwth-aachen.de
[3] Friedrich-Alexander-University Erlangen-Nürnberg, Open Source Research Group, dirk.riehle@fau.de
[4] RWTH Aachen University, Research Group Software Construction, steffens@swc.rwth-aachen.de

Copyright © 2017 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

Architecture

- Design for scalability
- Software architecture for CSE
- Microservices
- Model driven architecture for CSE

Quality and testing

- Test automation and optimization
- Monitoring and performance
- Security and Metrics for DevOps

Culture and business

- Teaching CSE approaches
- Organizational issues for CSE
- Digital transformation and innovation

Overall, the workshop aimed at gathering together researchers and practitioners to present new ideas and discuss experiences in the application of state of the art approaches to Continuous Software Engineering.

2 Workshop Contributions

Altogether eleven papers were submitted. Finally, six papers were accepted by the program committee for presentation and publication covering very different topics. We grouped the papers into three sessions and added a final round-up session to discuss the major findings of our workshop. The following papers were presented:

Session A: Processes

- Jan Ole Johanssen, Anja Kleebaum, Bernd Brügge and Barbara Paech: *Towards a Systematic Approach to Integrate Usage and Decision Knowledge in Continuous Software Engineering*
- Martin Kleehaus, Ömer Uludag and Florian Matthes: *Towards a Multi-Layer IT Infrastructure Monitoring Approach based on Enterprise Architecture Information*

Session B: Techniques & Tools

- Konrad Schneid: *Versioning strategies for developing new features within the context of Continuous Delivery*
- Lukas Alperowitz, Andrea Marie Weintraud, Stefan Christoph Kofler and Bernd Brügge: *Continuous Prototyping: Unified Application Delivery from Early Design to Code*

Session C: Industrial Experience

- Masud Fazal-Baqaie, Baris Güldali and Simon Oberthür: *Towards DevOps in Multi-provider Projects*
- Thomas Kurpick and Sebastian Melchior: *Naming in deployment pipelines for SaaS*

3 Acknowledgements

Many people contributed to the success of this workshop. First of all, we want to give thanks to the authors and presenters of the accepted papers. Furthermore, we want to express our gratitude to the SE 2017 organizers for supporting our workshop. Finally, we are glad that these people served on the program committee, soliciting papers and writing peer reviews:

• Lukas Alperowitz	TU München	
• Jan Bosch	Chalmers University of Technology	
• Michael Goedicke	University of Duisburg-Essen	
• Willi Hasselbring	Universität Kiel	
• Martin Jung	develop group, Erlangen	
• Stephan Krusche	TU München	(Organizer)
• Horst Lichter	RWTH Aachen University	(Organizer)
• Christian Nester	Google Inc.	
• Dirk Riehle	FAU Nürnberg	(Organizer)
• Heinz-Josef Schlebusch	Kisters AG, Aachen	
• Andreas Steffens	RWTH Aachen University	(Organizer)
• Christian Uhl	codecentric AG, Düsseldorf	
• Andre von Horn	Universität Stuttgart	
• Stefan Wagner	Universität Stuttgart	
• Heinz Züllighoven	WPS und Universität Hamburg	

References

[SEW16] W. Zimmermann, L. Alperowitz, B. Brügge, J. Fahsel, A. Herrmann, A. Hoffmann, A. Krall, D. Landes, H. Lichter, D. Riehle, I. Schaefer, C. Scheuermann, A. Schlaefer, S. Schupp, A. Seitz, A. Steffens, A. Stollenwerk, R. Weißbach (eds) (2016): Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2016 (SE 2016), Wien, 23.-26. Februar, 2016. CEUR-WS.org., Vol-1559.

Towards a Systematic Approach to Integrate Usage and Decision Knowledge in Continuous Software Engineering

Jan Ole Johanssen[1] Anja Kleebaum[2] Bernd Bruegge[3] Barbara Paech[4]

Abstract: Continuous software engineering (CSE) employs activities such as continuous integration and continuous delivery to support software evolution. Another aspect of software evolution is knowledge management. There are two important knowledge types: usage knowledge derives from explicit and implicit user feedback and helps to understand how users utilize software. Decision knowledge encompasses decisions and their rationale on all aspects of the software lifecycle. Both knowledge types represent important information sources for developers to improve the CSE activities and the software product. We envision an integration of usage and decision knowledge in the CSE lifecycle. This extension consists of a monitoring and feedback component for user understanding as well as a knowledge repository and dashboard component for knowledge visualization and analysis. Usage and decision knowledge introduce challenges when integrating them in CSE. In this paper, we present our vision and detail the challenges.

Keywords: Continuous Software Engineering, Usage Knowledge, Decision Knowledge

1 Introduction

Continuous software engineering (CSE) has been described by Bosch [Bo14] as a holistic approach for software engineering that consists of several activities covering the complete software lifecycle including continuous integration and continuous delivery.

Fitzgerald and Stol indicate the importance of monitoring any behavior during the run-time of a system in CSE [FS15]. We refer to insights derived from users either through testing or normal use as *usage knowledge*. Decisions made during the CSE activities represent another type of knowledge: *decision knowledge*. Creating and managing decision knowledge has already been explored by several researchers such as Dutoit et al. [Du06], while they focused on design-time knowledge only. However, with its focus on evolution during run-time, CSE introduces new challenges on managing usage and decision knowledge.

We have already worked on the integration of both usage and decision knowledge in non-CSE environments: for instance, regarding the analysis of user behavior, we compared monitored user interaction to use cases [Ro13]. Furthermore, regarding decision management, we developed the decision documentation model as well as corresponding tool support and explored its application [HKR16].

[1] Technische Universität München, Munich, Germany, johansse@in.tum.de
[2] Universität Heidelberg, Heidelberg, Germany, anja.kleebaum@informatik.uni-heidelberg.de
[3] Technische Universität München, Munich, Germany, bruegge@in.tum.de
[4] Universität Heidelberg, Heidelberg, Germany, paech@informatik.uni-heidelberg.de
Copyright © 2017 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

Up to now, usage and decision knowledge have not been explored within CSE. In our research project CURES (”Continuous Usage- and Rationale-based Evolution Decision Support”), we plan to integrate both knowledge types in CSE. Beside their individual benefits, we expect to leverage synergies in their combination: for instance, decisions might rely on results derived from detected usage patterns. In addition, both face similar questions regarding a consistent way of defining and storing the right granularity of information.

In the following, we present our vision of integrating usage and decision knowledge in CSE. Thereafter, we outline integration challenges and describe our status and next steps.

2 Vision

As shown in Fig. 1, we envision an extension to CSE in which developers and users act as the main stakeholders. As for the CSE infrastructure, we plan to employ Rugby, an agile process model for continuous delivery which builds upon event-based releases [Kr14]. Developers utilize feature branches to add product increments in form of code commits. Feature branches are merged from and back to a master branch, which contains the final software product. Each feature branch on its own can be released to users, allowing the delivery of different proposals for one feature at the same time. Using a monitoring and feedback component, developers can examine usage information that is mapped to individual releases by applying approaches such as A/B testing for implicit usage information.

A knowledge repository continuously stores all information related to the development and monitoring process. Furthermore, it maintains decisions related to feature branches. Thereby, rationale can be accessed, visualized, and analyzed using a dashboard component. For instance, the claimed solution to an issue could be compared to an alternative solution based on the impact of a feature branch in the context of previous decisions. This enables the developer to interact and reflect on collected usage and decision knowledge.

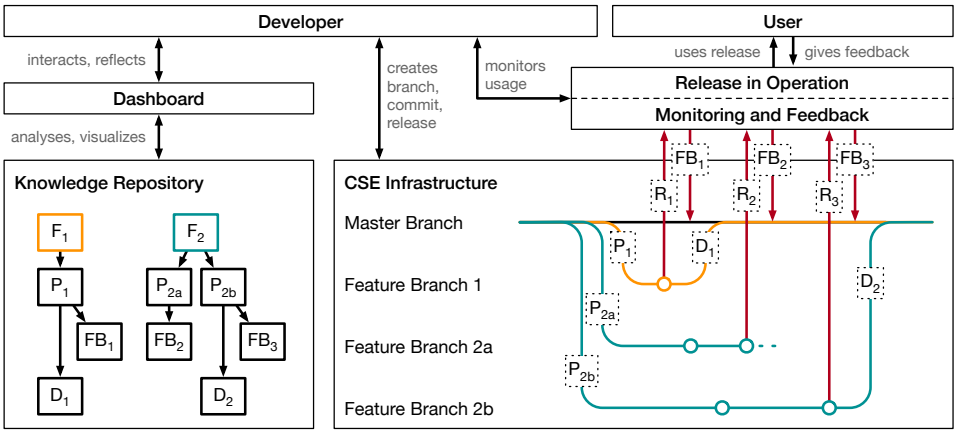

Fig. 1: Interaction between CSE Infrastructure, Knowledge Repository, and Dashboard: Feature *F* is based on a Proposal *P*, Feedback *FB* is collected from a Release *R* and leads to a Decision *D*.

For example, a feature F_1 is developed based on a proposal P_1. After providing the release R_1 to users and analyzing feedback FB_1, the developer makes decision D_1 and merges F_1 into the master branch. In the same manner, different proposals P_{2a} and P_{2b} for a feature F_2 are evaluated. Feedback FB_2 stops the work on P_{2a}, while FB_3 leads to decision D_2.

This CSE infrastructure extension forms a *continuous knowledge* activity and enables the exploitation and documentation of knowledge acquired during CSE. It supports the evolution of knowledge and handles it similar to the evolution of code in a CSE infrastructure.

3 Challenges

In this section we describe the challenges of integrating usage and decision knowledge in CSE and highlight references from the extension introduced in Fig. 1 in bold type.

There is a heterogeneous set of users. In CSE, the **user** role might be filled by the developers themselves, the customers who initiate the development of an application as part of a contract, or the actual end users of an application. All of them have different intentions, expectations, and approaches towards using an application which needs to be considered.

User **feedback** can either be explicit or implicit. Explicit feedback is actively provided by the user, such as an app store review, while implicit feedback relates to anything users reveal through their interaction, such as pressing a button. It has yet to be answered how both feedback types can be incorporated within the CSE infrastructure: different strategies for feedback elicitation need to be evaluated, such as in-application feedback requests for new features or automatic usage data collection for each feature branch.

Enriching users' explicit with implicit feedback in a **monitoring and feedback** component enables the creation of a comprehensive user understanding. A main challenge is to derive a user's motivation and construct a user behavior model. Additional context information, such as a user's location or current activity, might be useful for creating such a model.

Further challenges arise in the synchronization of releases, feedback, and their granularity: in a **CSE infrastructure** new increments are released frequently, while usage patterns develop only over time and the difference between releases can be as little as a commit.

The decision documentation model is our starting point to integrate decision knowledge in CSE since it allows for an intertwined documentation across agile development activities. We want to address the following challenges towards a continuous rationale management.

Due to CSE's focus on run-time, decision making is a high-frequency process and leads to large amounts of data. **Developers** utilize different components, such as issue tracking and version control systems. Knowledge is distributed across these components: for instance, requirements are stored in the issue tracking system, whereas source code is stored in the version control system. It is a challenge how distributed decision knowledge can be systematically managed and synchronized during CSE as part of a **knowledge repository**.

Regarding the decision knowledge repository further questions arise: (1) Which decisions are worthwhile to be captured and managed? Should they be captured within the feature branch or the master branch? (2) What is the right granularity of decisions and the related rationale? (3) How to enable a seamless switch between coarse and fine-grained decision knowledge? (4) What is the most suitable way to search for decision knowledge related to a problem during software evolution? How should the knowledge be presented in the **dashboard**? (5) How can a change impact analysis on decision knowledge be performed efficiently and effectively?

4 Status and Next Steps

Currently, we work on an empirical study to investigate on CSE needs in practice. The goal is to determine how developers deal with usage and decision knowledge to align the results with our next steps. We develop a decision documentation editor for JIRA and Git. This is our initial step to integrate the decision documentation model in the CSE infrastructure. We plan on applying machine learning techniques to classify implicit user feedback.

5 Conclusion

Usage and decision knowledge has not been explored in CSE up to now. We described our vision for a systematic approach to integrate both aspects in CSE. We identified challenges such as the heterogeneity of users and differences in explicit and implicit feedback. Another challenge is the alignment of decision making processes with respect to the fast pace of CSE activities such as continuous delivery, resulting in questions on decision capturing and granularity. Finally, we outlined our status and next steps for the integration of usage and decision knowledge in order to improve existing CSE infrastructures.

Acknowledgement

This work was supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

[Bo14] Bosch, Jan: Continuous Software Engineering: An Introduction. Springer, 2014.

[Du06] Dutoit, Allen H; McCall, Raymond; Mistrík, Ivan; Paech, Barbara: Rationale Management in Software Engineering: Concepts and Techniques. Springer, 2006.

[FS15] Fitzgerald, Brian; Stol, Klaas-Jan: Continuous software engineering: A roadmap and agenda. Journal of Systems and Software, 2015.

[HKR16] Hesse, Tom-Michael; Kuehlwein, Arthur; Roehm, Tobias: DecDoc: A Tool for Documenting Design Decisions Collaboratively and Incrementally. In: Proceedings of the 1st Int. Workshop on Decision Making in Software ARCHitecture. pp. 30–37, 2016.

[Kr14] Krusche, Stephan; Alperowitz, Lukas; Bruegge, Bernd; Wagner, Martin O.: Rugby: An Agile Process Model Based on Continuous Delivery. In: Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering. pp. 42–50, 2014.

[Ro13] Roehm, Tobias; Bruegge, Bernd; Hesse, Tom-Michael; Paech, Barbara: Towards Identification of Software Improvements and Specification Updates by Comparing Monitored and Specified End-User Behavior. In: Proceedings of the 29th IEEE International Conference on Software Maintenance. pp. 464–467, 2013.

Towards a Multi-Layer IT Infrastructure Monitoring Approach based on Enterprise Architecture Information

Martin Kleehaus, Ömer Uludağ and Florian Matthes[1]

Abstract: Enterprise Architecture Management (EAM) tools play an important supporting role in IT management of organizations to align their IT infrastructure to actual business needs. The continuously measurement and observation of each layer in an enterprise architecture is critical in order to achieve an holistic view about the EA operation. This paper describes a concept how to exploit and extend the metainformation of an IT architecture documented in an EA tool in order to support a multi-layer monitoring approach that provides traceability and correlation between monitoring data extracted from several abstraction layers.

Keywords: Business Process Monitoring, Application Performance Monitoring, Infrastructure Monitoring, Log Mining, Enterprise Architecture Management

1 Introduction

The measurement and control of IT and business services delivered by an Enterprise Architecture (EA) is based on a continuous process of monitoring the instances, reporting on failures, learning from their behavior and planing subsequent actions. These steps are fundamental as, although this monitoring process takes place during service operation, they provide important information about the EA which in turn can assist for improving the alignment of the IT and business strategy.

Many well known EA frameworks emerged in the last decades like TOGAF [Ha11], ArchiMate [Gr16], ITIL [Of11], etc., that deliver a standard how to model the EA based on abstraction layers: 1) the IT infrastructure encompasses all technological aspects, 2) the application layer defines the software running in the IT infrastructure and 3) the business processes that operate on top of the aforementioned layers. Although a plethora of monitoring solutions [Kl16] have been developed to account for these layered architectures, most of these solutions specialize on a specific layer or requirement, like Business Process Monitoring [ADO00], Application Performance Monitoring [Ra12], Infrastructure Monitoring [Jo07], or Log Mining [AGL98] solutions. This makes it challenging to obtain an integrated and holistic view on the behavior and status of the EA as most tools either support only a technical viewpoint, or a business oriented viewpoint [BGP11].

In the following sections, we present an conceptual implementation how to design an integrated real-time multi-layer EA monitoring solution that establishes a link between the

[1] Technische Universtität München, Software Engineering for Business Information Systems (sebis), Boltzmannstrasse 3, 85748 Garching bei München, {martin.kleehaus, oemer.uludag, matthes}@tum.de
Copyright © 2016 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

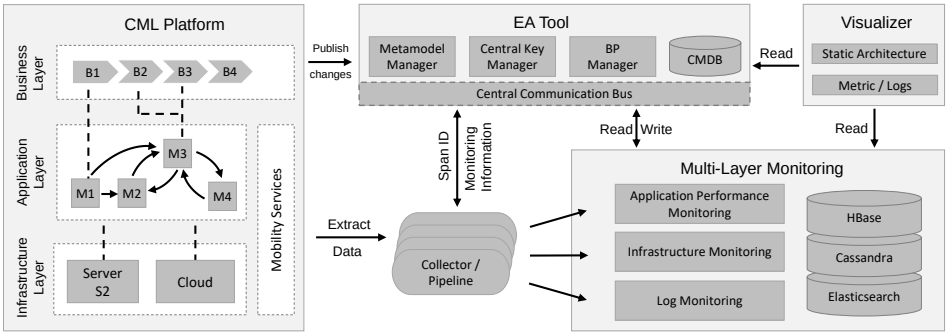

Fig. 1: Architecture of a multi-layer monitoring solution based on EA information

software and hardware components like the communication behavior, the dependencies between each components and the business processes that are supported by the services.

2 Connected Mobility Lab

The reference architecture that we use for evaluating our solution in future work will be the academic project *Connected Mobility Lab (CML)*[3]. The architecture is illustrated on the left side of figure 1. CML presents an open, digital mobility platform and is aligned on a microservice architecture. It consists of core services and mobility services. Core services undertake a supporting role by providing interfaces and required backend features. Mobility services provide business value to the end user and valuable data to other services that consume this data in order to enrich their own service.

The following use cases need to be addressed by the monitoring solution: 1) What is a root cause of an occurred error and who is accountable? 2) Which services suffer from bad performance? 3) Do the services comply with the specified Quality of Service (QoS) agreements? 4) What services fulfill a specific business process?

3 Multi-Layer Monitoring System

In the context of the proposed multi-layer monitoring solution we establish an *EA tool* enhanced with a *Configuration Management Database (CMDB)*, as it is illustrated in the top of figure 1. The tool integrates a *metamodel manager* that applies an EA model as the design-time model that defines visual representations of all monitored components like business processes, software systems, hardware elements and their relations. The metainformation of the components (id, name, type, etc.), their relations (id, communication type, etc.) and monitoring specific information like the path to log files are stored in the CMDB.

[3] http://tum-llcm.de/

Besides the EA tool which is the central control center of the monitoring solution we deploy a combination of several monitoring applications that collect data from each EA layer (right side of figure 1). However, as these applications perform their task rather self-sufficient and are connected to several database technologies (HBase, Cassandra, Elasticsearch as an example), we enhance the EA tool with a *central communication bus* that support the heterogeneous monitoring infrastructure. That means, each monitoring application has to communicate with the EA tool for extracting required information, like the component id in order to achieve a correlation between the monitored components.

Furthermore, the EA tool also includes an interface for receiving continuous deployments made on the CML application in order to keep the architecture information up to date. In particular, the introduction or the withdrawal of software components are monitored. The following sections describe the monitoring solutions in more detail.

3.1 Monitor the infrastructure layer

The infrastructure layer of the CML platform is monitored by deploying agents on the servers extracting status information about traffic, bandwith, CPU utilization, etc. Open source solutions like Nagios[4] or Sensu[5] fit for this purpose. However, these solutions do not provide information about what specific user transaction is accountable for a huge resource utilization or which application causes abnormal behavior.

One approach in order to address this challenge, we suggest to leverage the linked information in the EA tool in order to correlate hardware metrics to running transactions by using the written timestamps as a connection key. In addition, log events written by the infrastructure and the application layer can uncover further important information about the behavior of these systems. Fur this purpose, we deploy the ELK stack[6] for processing and indexing the log files. The information which IT component creates the log events and what user transaction is currently in process can be retrieved from the EA tool, passed to the Logstash pipeline and stored in Elasticsearch.

3.2 Monitor the application layer

The CML platform consist of microservices (illustrated as M_X in figure 1) which provide a bulk of different backend and mobility services. These applications are constructed from collection of software modules that were developed by different teams, in different programming languages. A huge challenge for monitoring this configuration is to identify how and to what extent the microservices are communicating with each other in order to understand system behavior and reasons about performance issues.

In order to achieve this goal we apply the application performance monitoring (APM) approach ”Dapper” described by Google [Si10] and adapted by several academic projects

[4] https://www.nagios.org/
[5] https://sensuapp.org/
[6] https://www.elastic.co/

like kieker [vHWH12] and pivottracing [MRF15], and open source projects like pinpoint[7], or zipkin[8]. Dapper provides a solution for analyzing the overall structure of a system and how components within them are interconnected by tracing transactions across microservices without changing the application code. Each transaction contains a collection of span identifier (span id) that refer to a specific Remote Procedure Call (RPC). However, as the span ids are generated from scratch by default, the APM solution has to be modified in the way that the keys describing the specific component of the platform architecture are issued by the *central key manager* from the EA tool. Furthermore, it has to be assured, that this modification has no significant performance impact on the services.

In addition, the span identifier need to be assigned to log events which are written from the particular service. This can be realised by altering the Logstash configuration file.

3.3 Monitor the business process layer

The goal of business process monitoring is to extract business events that refer to a well-defined step in the business process activity (marked as B_X in figure 1) from transaction logs. Hereby, it is challenging to know who performs the activity, what transaction events compose a whole activity and in particular when an activity has been started and finished [Do05].

To address this challenge, we propose to extend the EA tool with a *business process manager* that assists to define and manage business process events based on the trace information provided by the APM solution. Hereby, each relevant RPC refers to a business event and is mapped to one or more specific business activities that, in turn, compose a particular business case. However, in the first instance, the table for mapping RPC calls to predefined business process activities has to be done manually and kept always up to date.

4 Conclusion

In this short paper we presented an approach to enable real-time monitoring and visualizing of multi-layer Enterprise Architectures. We highlighted the concept of an EA tool with a central communication bus that supports the exchange of design-time and run-time information. Hereby, we were able to correlate monitoring data across the EA layers and assign them to each managed IT component and business process.

As this proposed concept is still in the design phase, our future work will be the elaboration of this solution. As the main research methodology, we will apply design science. First of all, we will investigate which monitoring application fits best for our needs. Afterwards we will model the details of our approach and develop a prototype accordingly. As the central EA tool we will use the academic project SocioCortex [MN11] and extend it with the above mentioned requirements. Finally, we will evaluate our prototype on CML.

[7] https://github.com/naver/pinpoint
[8] http://zipkin.io/

5 Acknowledgments

This work is part of TUM Living Lab Connected Mobility (TUM LLCM) project and has been funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie (StMWi). We also thank our reviewers for their valuable feedback and constructive reviews.

References

[ADO00] Aalst, Wil M. P. van der; Desel, Jörg; Oberweis, Andreas, eds. Business Process Management, Models, Techniques, and Empirical Studies, London, UK, UK, 2000. Springer-Verlag.

[AGL98] Agrawal, Rakesh; Gunopulos, Dimitrios; Leymann, Frank: Mining Process Models from Workflow Logs. In: Proceedings of the 6th International Conference on Extending Database Technology: Advances in Database Technology. EDBT '98, Springer-Verlag, London, UK, UK, pp. 469–483, 1998.

[BGP11] Brückmann, Tobias; Gruhn, Volker; Pfeiffer, Max: Towards Real-time Monitoring and Controlling of Enterprise Architectures Using Business Software Control Centers. In: Proceedings of the 5th European Conference on Software Architecture. ECSA'11, Springer-Verlag, Berlin, Heidelberg, pp. 287–294, 2011.

[Do05] van Dongen, B.; de Medeiros, A. K. A.; Verbeek, H. M. W.; Weijters, A. J. M. M.; van der Aalst, W. M. P.: The ProM Framework: A New Era in Process Mining Tool Support. In: Applications and Theory of Petri Nets 2005: 26th International Conference, ICATPN 2005, Miami, USA, June 20-25, 2005. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 444–454, 2005.

[Gr16] Group, The Open: ArchiMate 3.0 Specification. Van Haren Publishing, 2016.

[Ha11] Haren, Van: TOGAF Version 9.1. Van Haren Publishing, 10th edition, 2011.

[Jo07] Josephsen, David: Building a Monitoring Infrastructure with Nagios. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[Kl16] Kleehaus, Martin; Landthaler, Jörg; Huth, Dominik; Matthes, Florian: Multi-Layer Monitoring and Visualization. In: Digital Mobility Platforms and Ecosystems. pp. 90–110, 2016.

[MN11] Matthes, Florian; Neubert, Christian: Wiki4EAM: Using Hybrid Wikis for Enterprise Architecture Management. In: Proceedings of the 7th International Symposium on Wikis and Open Collaboration. WikiSym '11, ACM, New York, NY, USA, pp. 226–226, 2011.

[MRF15] Mace, Jonathan; Roelke, Ryan; Fonseca, Rodrigo: Pivot tracing: dynamic causal monitoring for distributed systems. In: Proceedings of the 25th Symposium on Operating Systems Principles. ACM, pp. 378–393, 2015.

[Of11] Office, Cabinet: ITIL Service Operation 2011 Edition. The Stationery Office, Norwich, 2011.

[Ra12] Rabl, Tilmann; Gómez-Villamor, Sergio; Sadoghi, Mohammad; Muntés-Mulero, Victor; Jacobsen, Hans-Arno; Mankovskii, Serge: Solving Big Data Challenges for Enterprise Application Performance Management. Proc. VLDB Endow., 5(12):1724–1735, August 2012.

[Si10] Sigelman, Benjamin H.; Barroso, Luiz Andr; Burrows, Mike; Stephenson, Pat; Plakal, Manoj; Beaver, Donald; Jaspan, Saul; Shanbhag, Chandan: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical report, Google, Inc., 2010.

[vHWH12] van Hoorn, André; Waller, Jan; Hasselbring, Wilhelm: Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In: Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering. ICPE '12, ACM, New York, NY, USA, pp. 247–248, 2012.

Branching strategies for developing new features within the context of Continuous Delivery

Konrad Schneid

Abstract: This paper evaluates based on current literature, whether the versioning strategies "branch by feature" and "develop on mainline" can be used for developing new software features in connection with Continuous Delivery. The strategies will be introduced and possible applications for Continuous Delivery will be demonstrated and rated. A solution recommendation is finally given. It becomes evident that develop on mainline is the more recommendable method in form of "features toggles" or in case of bigger changes in form of "branch by abstraction" within the context of Continuous Delivery.

Keywords: Continuous Delivery, Branch by Feature, Develop on Mainline, Feature Toggles, Branch by Abstraction

1 Problem formulation

Different attempts exist around developing new software features. In practice, several developers are involved in programming software. They work in parallel on developing the software. In order to perform the parallel development of different software versions, version control systems (VCS) like Git are used [CS14, p.27]. VCS offer various versioning strategies for developing new features. The development of different features should be separated to provide these independently to the production system. It must be guaranteed that features which are still in development do not disturb the productive operation, but can still be developed. Two appropriate strategies are branch by feature and develop on mainline [HF10, Chap.14]. This paper researches whether and how these strategies can be used within the context of Continuous Delivery (CD).

One important aspect of CD is Continuous Integration (CI) [HF10, p.24]. The goal of CI is to improve the software quality by integrating every change of code. This means that software changes have to be continually integrated, tested and built. The central question which will be answered in the following, is how this aspect can be guaranteed during developing new features. Both strategies, branch by feature and develop on mainline, will be seen from this angle in the following chapters. According to RODRÍGUEZ ET AL. both methods are used in practice. But there is no recommendation which strategy is more suitable. This question also shall be answered in the context of this paper. CHEN also sees a need for further research in the area of software development within the CD process. [Ch15, p. 54; Ro16].

2 Branch by Feature

This chapter explains the strategy branch by feature and evaluates the possible application for CD.

In VCS there is one mainline, also known as trunk or master. Branching means to deviate from the mainline and to create a new branch. Branch by features means, that every new feature has to be developed in an own branch. As soon as the implementation of the feature is finished, it will be integrated into the mainline (see figure 1). The mainline can be held in a release status. Thus branches allow an isolated development of new features. The productive environment is not interrupted because only the mainline is live. [HF10, p.410; PS13, p.135]

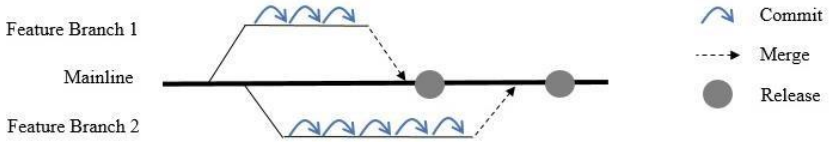

Fig. 1: Schematic diagram of branch by feature

2.1 Impact for Continuous Delivery

As described in chapter 1, CI is one important aspect of CD. Feature branches allow an isolated development. This is a problem if this strategy is used for CD. CI cannot be ensured because there is no integration of the individual branches. For this reason, this strategy is the wrong approach according to HUMBLE & FARLEY: "[…] branching by feature is really the antithesis of continuous integration […]" [HF10, p.412]. [Wo15]

If the definition of CI is not interpreted as strictly as defined by Fowler, the strategy for CD could be possible. Let's assume that a weekly integration is sufficient for a project. This is the case if a company which has used the strategy branch by feature till now would like to implement CD. Thus the developers are not completely derailed from their usual workflow and a higher acceptance can be created for CI. CI only works if the developers let themselves in for it and accept it. [HF10, p.57; Fo06].

In order to use the advantages of CD, the strategy must be supplemented with some rules. Feature branches should be short-lived and exist for at most a few days. As a consequence, the so-called "merging hell" is reduced. This arises if many long-lived feature branches exist and are integrated only after weeks or even months into the mainline. That means merging gets extremely complex and risky because the source code of the branches and the mainline strongly diverge. A further aspect to reducing the "merging hell" is to transfer every change of the mainline promptly to each branch. The number of merging conflicts can be minimized. For this reason, refactorings should also be transferred immediately to the mainline. Thus the number of feature branches should not exceed the number of the features to be developed and a new feature branch should

only be created if a feature was implemented successfully into the mainline. A merge from a feature branch into the mainline is only allowed if the feature branch was tested successfully before. [HF10, S.410]

This procedure can be used as the first step if CD should be introduced and the strategy branch by feature should be maintained although it is not fully suitable for CD. The following table shows the pros and cons of the strategy branch by feature.

Pros	Cons
Features in development do not cause problems during running time	Short-lived feature branches often not possible
Traceability of feature development	Merging effort
Each feature is independently in own branches	Integration delay

Table 1: Advantages and disadvantages of branch by feature

3 Develop on Mainline

In the following the strategy develop on mainline will be introduced and evaluated in the context of CD.

With this strategy the features are implemented directly on the mainline (see figure 2). Therefore, all changes are immediately available for all developers. Branches do not exist during development. Consequently, there are no merging problems. Since all developers work in parallel on the mainline, additional strategies are needed to develop the feature programs on the mainline in parallel. It must be guaranteed that the features, which are in development, still do not disrupt the productive operations. To develop directly on the mainline there are two approaches feature Toggles and branch by abstraction. These are described in the two following subchapters. [HF10, p. 405]

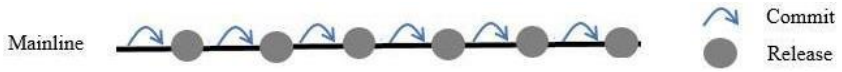

Fig. 2: Schematic diagram of develop on mainline

3.1 Feature Toggles

Feature toggle (synonymous feature flags, flipper, feature switches) is a programming technique that a feature in development can be turned on and off during the running time of the software. Toggles can only be in the state “on” or “off”. The software uses toggles during the running time for the evaluation and for the application and activates the feature or deactivates it depending on their condition. In case a new feature causes any problems, it can be switched off directly during running time. The simplest implementation variant of feature toggle is the use of if-then-else-statements. HODGSON differs toggles in four categories which are subdivided by the factors

lifetime and dynamics of a toggle. [Ho16; Eb15]

Release toggles allow to implement unfinished and untested source code directly on the mainline. These toggles must not be switched on during running time. The state of this toggle is typically static. According to HODGSON's recommendation release toggles should not exist longer than two weeks. HODGSON suggested to remove release toggles after the final acceptance of the new feature in the code to ensure manageability and prevent deactivating the feature inadvertently. [Ho16; Eb15]

Another application example of toggles is not just to hide unfinished code but also to use it depending on user groups or other environmental factors. This variant is known under the name experiment toggles. New features can be tested only with a small user group and released after a successful test stage for all users according to the "Dark Testing method" or using "A/B-Testing". The toggle condition is set dynamically and according to HODGSON has a lifetime of a few hours up to several weeks depending on the frequency of the use. Popular services using this method are Facebook and Flickr according to own information. [Ha09; Ho16; Ta15]

Permissioning toggles are very similar. With these toggles features are provided only for specific user groups. Selected beta or premium users can use features which are disabled for standard users. In contrast to experiment toggles the features are not randomly provided to users. They are explicitly turned on for specific users. The lifetime of premissioning toggles are appropriately long-lived. HODGSON himself has estimated the lifetime for several years. [Ho16]

EKART recommends a static configuration for features toggles. Thus toggles can be treated according to the "Configuration-as-Code" principle, which allows versioning and provides transparency. For extended configurations EKART recommends distributed "Key-Value Stores". [Ek16]

Feature toggles do not increase the complexity of testing according to NEELY & STOLT, although new and old features are in the same mainline. The number of test cases to be written is just as high as in case of using feature branches because of the combination difficulty of different features is the same. NEELY & STOLT go even further and argue that testing is easier with feature toggles than with feature branches: "[…] the combinatorial criticism would be the same problem with feature branches, and with feature toggles it is simpler as you toggle on and off in test code" [NS13, p.124]. The only additional effort is to specify which toggles can be turned on simultaneously. [NS13, p.124]

On the contrary, BIRD expresses the following criticism on feature toggles: "Feature flags make the code more fragile and brittle, harder to test, harder to understand and maintain, harder to support, and less secure." [Bi14]. According to BIRD it is dangerous to put untested code with uncertain effects in production. As an example he mentions an unintentional change in the business process of a financial institution. [Bi14]

TANG ET AL describe the successful application of feature toggles at the social

network Facebook. Through an administration interface (Gatekeeper) features can be adaptably released and selectively to certain user groups ("cohorts"). First, a new feature can be released with the Gatekeeper for one percent of the internal employees. The percentage is continuously increased with trouble-free use. Only after a successful internal test the feature is released for about five percent of the users of a specific region. In case this phase is also successful, the new feature is eventually released in further steps continuously worldwide. [Ta15]

3.2 Branch by Abstraction

Another possibility to run develop on mainline is the use of the pattern branch by abstraction defined by HAMMAT. In comparison to feature toggles this pattern is used for bigger changes, when it is not possible to implement these in small incremental steps. [Ha07]

With this pattern an abstraction layer is put on the part of the software, that should be changed. The abstraction layer points at the old implementation and allows a parallel development of the new feature. It can be set during the deployment or running time to which code the abstraction layer refers. As soon as the new development is finished, the abstraction layer refers to the new code and will be removed together with the old code if no problems appear. The CI principle is guaranteed. Only if the user has to choose the implementation, the abstraction layer will not be removed. According to HUMBLE & FARLEY the pattern is also used to transfer a monolith code base into modular built up software. In that case the old implementation is running parallel to the new, modular built code with the same functionality. [HF10, p.351]

To test the functionality of the new implementation, the pattern "verify branch by abstraction" can be used. In that case a toggle follows the abstraction layer and can refer to the new implementation (see figure 3). Both implementations are used with the same input data. The test fails if the result is not the same. This prevents that the business behavior of an application changes with the new implementation. This strategy is only possible, if there is no change in the functionality. [Sm13; HF10, p.351]

Fig. 3 Schematic diagram of verify branch by abstraction

According to HUMBLE & FARLEY a difficulty of the abstraction layer is to find an entry point in the source code which has to be isolated. If no entry point can be found, the code base has to be refactored. Nevertheless, HUMBLE & FARLEY consider it easier to handle this problem than that of branch by feature. [HF10, p.351]

HUMBLE sums up the advantages of branch by abstraction that way: "your code is working at all times throughout the re-structuring, enabling continuous delivery" [Hu11].

3.3 Impact for Continuous Delivery

Since the whole development is done on the mainline, CI is ensured. To develop new features in parallel, both methods, feature toggles and branch by abstraction, can be used. In this way new features can be transported to production without disturbing it. Correspondingly the strategy develop on mainline is very suitable for the development of new features in the context CD. This strategy represents a necessary condition for CI to HUMBLE & FARLEY: “In fact, it is an extremely effective way of developing, and the only one which enables you to perform continuous integration” [HF10, p.405].

Pros	Cons
No merging problems and integration delay Testing during running time Separation of deployment and release “Big Bang release” is avoided	Maintainability of software is complicated when features are cross-cutting

Table 2: Advantages and disadvantages of develop on mainline

4 Hybrid solutions

As described in the previous chapters, develop on mainline suits better than branch by feature in the context of CD. Nevertheless, branch by feature should not be excluded generally. It can make sense in hybrid scenarios. But this should only be practiced for small critical hotfixes according to HUMBLE & FARLEY. Instead of rollbacks the authors recommend rollforwards. This results from the fact, that the deltas are small between two releases. [HF10, p.351]

Another use case to HUMBLE & FARLEY, which allows hybrid scenarios or makes them even necessary, concerns software programmed after the "Big Ball of Mud" pattern. It can be difficult to put an abstraction layer over an entry point. If such an entry point (typically in form of an interface) is not found, the code has to be refactored. In that case, branches can be used. [HF10, p.351]

In order to increase the acceptance of develop on mainline, hybrid scenarios are conceivable in the transitional phase.

5 Recommendation

As a conclusion of this paper, a solution is recommended and further practical examples are introduced.

KRUSCHE & ALPEROWITZ describe an experiment to lead students to CD by using the strategy branch by feature. Over 50% of the student did not have any experience with

CD. As a result of the experiment students actually want to use this strategy for further projects. That paper does not mention that this method leads to “merging hell” and no CI is possible. This could be because the projects were very small and there have been just a few merging conflicts. In the paper there is no information about the size of the group. [KA14, p.337]

It becomes evident that a first acceptance for CD can be created by using the strategy branch by feature. In small project teams (up to 3 people) this strategy can be sufficient if the rules from chapter 2.1 are followed. The most important point is that only short-lived feature branches are used. But in general this strategy is not suitable for CD.

One disadvantage of branch by feature is that short-lived branches are not always possible. Thus, CI cannot be guaranteed. Even short-lived feature branches result in an integration delay because features are integrated after they were completed. These substantial disadvantages lead to the recommendation that develop on mainline within the context of CD should be used in combination with feature toggles. If changes in the software architecture are intended, branch by abstraction can be used. NEELY & STOLT and MEYER explain the successful use of the strategy develop on mainline with the help of feature toggles. Both describe their positive experience with the strategy using only one mainline. According to the authors the use of feature toggles is mandatory in order to develop new features. NEELY & STOLT used to work with branch by feature before. They switched to feature toggles because the effort of merging became too complex. As described in chapter 3.1., the authors TANG ET AL also report positively in about the use of feature toggle at Facebook.

Another crucial advantage of feature toggles in contrast to feature branches is the possibility to test new features in production and to turn them on and off. This allows for example A/B testing. According to HUMBLE the use of branch by abstraction is recommended especially for bigger changes in context of CD. In order to test the changes, the combination verify branch by abstraction with feature toggles and branch by abstraction should be used. [HF10, p.351; Sm13]

Hybrid scenarios as described in chapter 4 are conceivable but should be used just in special cases. According to HUMBLE & FARLEY this is for example necessary for software that was programmed after the „Big Ball of Mud” pattern. Furthermore, small hotfixes can be implemented as feature branches. [HF10, p.351]

The strategy develop on mainline combined with the techniques feature toggles and branch by abstraction is the most suitable solution for CD. Only in exceptions hybrid scenarios should be used.

References

[Bi14] Bird, J.: Feature Toggles are one of the Worst kinds of Technical Debt, 2014, https://dzone.com/articles/feature-toggles-are-one-worst (retrieved 11.02.2016).

[Ch15] Chen, L.: Continuous Delivery: Huge Benefits, but Challenges Too. IEEE Software, Band 32, pp. 50-54, 2015.

[CS14] Chacon, S.; Straub, B.: Pro Git – Everything you need to know about Git. 2. Auflage, Apress, Berkeley CA, 2014.

[Eb15] Ebbers, H..: Jede Änderung ein Feature, 2015, https://jaxenter.de/jede-aenderung-einfeature-18354 (retrieved 11.02.2016).

[Ek16] Ekart, G..: Feature Toggles Revisited, 2016, http://www.infoq.com/news/2016 /02/featuretoggles (retrieved 11.02.2016).

[Fo06] Fowler, M.: Continuous Integration, 2006, http://www.martinfowler.com/articles /continuousIntegration.html (retrieved 11.02.2016).

[Ha07] Hammat, P..: Introducing Branch By Abstraction, 2007, http://paulhammant.com/blog/branch_by_abstraction.html (retrieved 11.02.2016).

[Ha09] Harmes, R..: Flipping Out, 2009, http://code.flickr.net/2009/12/02/flipping-out/.

[HF10] Humble, J.; Farley, D.: Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation. Addison Wesley, Boston, 2010.

[Ho16] Hodgson, P..: Feature Toggles, 2016, http://www.martinfowler.com/articles /continuousIntegration.html (retrieved 11.02.2016).

[Hu11] Humble, J..: Make Large Scale Changes Incrementally with Branch By Abstraction. 2011, http://continuousdelivery.com/2011/05/make-large-scale-changes-incrementally-with-branch-by-abstraction/ (retrieved 11.02.2016).

[KA14] Krusche, S.; Alperowitz, L.: Introduction of Continuous Delivery in Multi-customer Project Courses. In: Companion Proceedings of the 36th International Conference on Software Engineering, ACM, pp. 335-343, 2014.

[Me14] Meyer, M.: Continuous Integration and Its Tools. IEEE Software, Band 31, Heft 3, pp.14-16, 2014.

[NS13] Neely, S.; Stolt, S.: Continuous delivery? Easy! Just Change Everything (well, maybe it is not that easy). In: Agile Conference, IEEE, pp. 121–128, 2013.

[PS13] Preißel, R.; Stachmann, B.: Git: Dezentrale Versionsverwaltung im Team - Grundlagen und Workflows. 2. Auflage, dpunkt.verlag, Heidelberg, 2013.

[Ro16] Rodríguez, P. et al.: Continuous deployment of software intensive products and services: A systematic mapping study. The Journal of Systems and Software, 2016.

[Sm13] Smith, S.: Application Pattern: Verify Branch By Abstraction. 2013 https://dzone.com/articles/application-pattern-verify (retrieved 11.02.2016).

[Ta15] Tang, C. et al.: Holistic configuration management at Facebook. ACM, New York, 2015.

[Wo15] Wolff, E.: Continuous Integration widerspricht Feature Branches. 2015 https://heise.de/-2736487 (retrieved 11.02.2016).

Continuous Prototyping: Unified Application Delivery from Early Design to Code

Lukas Alperowitz[1], Andrea Marie Weintraud[2], Stefan Christoph Kofler[3], and Bernd Bruegge[4]

Abstract: Developing for devices like smartphones, tablets or smartwatches is more than just "shipping code". Especially in mobile development there is a strong focus on user interface design and user experience. In order to explore the design space, development teams and designers need early feedback from users testing the designs.

Continuous Delivery (CD) is a well-established technique for the delivery of software. In this paper we describe Continuous Prototyping which extends CD to cover the delivery of early artifacts like user interface mockups that usually do not benefit from an automated delivery process. Continuous Prototyping enables stakeholders to receive all artifacts through a unified delivery channel in fast cycles, from the first mockup to the finished product.

We developed PROTOTYPER as a tool to demonstrate the technical feasibility of Continuous Prototyping. PROTOTYPER allows developers and designers to deliver mockups, mobile applications as well as a mixture of both using the same deployment pipeline.

1 Introduction

Developing for devices like smartphones or smartwatches is more than just "shipping code". Especially in mobile application development user expectations regarding the user interface design and user experience are high. Therefore, development teams do not only consist of software developers. User interface and experience designers play an essential role in all phases of the evolution of a mobile application. In early project phases they provide interactive mockups to help the development team to define and refine the requirements. By providing design elements or mockups for a certain functionality, they contribute to new features in later phases of a project [Ru09]. These mockups help the software team to define the overall interaction of the user with the application [Be09]. Current mobile applications are leveraging multi-modal interaction techniques such as touch or speech input. They also often incorporate heterogeneous IoT devices like sensors and wearables. A mockup for such an application can be complex already in an early project stage.

[1] Technical University of Munich, Germany, alperowi@in.tum.de
[2] Technical University of Munich, Germany, weintraa@in.tum.de
[3] Technical University of Munich, Germany, koflers@in.tum.de
[4] Technical University of Munich, Germany, bruegge@in.tum.de
Copyright © 2017 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

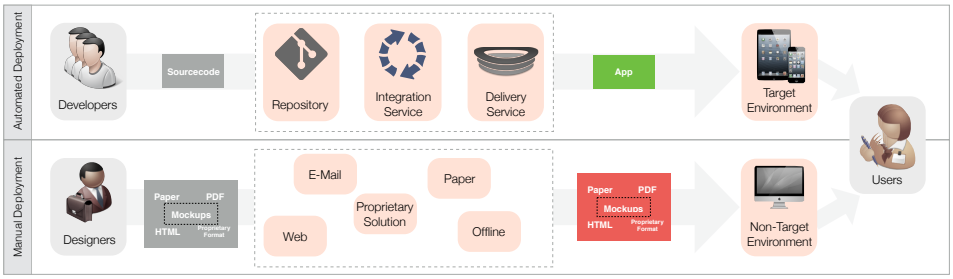

Figure 1: Current Practice: Software is delivered using a deployment pipeline, mockups manually and using various channels.

Although mockups are an important part of mobile application development, their delivery is given little attention. While Continuous Delivery (CD) is a well-established technique for the delivery of software, mockups are delivered manually using various channels ranging from e-mail, offline media to web-based solutions and come in various file-formats. Figure 1 illustrates this discrepancy.

We think the delivery of mockups should be as easy and automated as the delivery of software. We believe there is a need for a unified delivery concept for application development which covers all phases from the creation of mockups to actual software.

This paper introduces the concept of *Continuous Prototyping* to bridge the gap between the delivery of mockups and software. Continuous Prototyping allows designers and developers to deliver mockups as well as software using the same, unified deployment pipeline.

The remainder of this paper is structured as follows. In Section 2 we revisit the current research in the field of user interface prototyping and CD. We then define the term Continuous Prototyping in Section 3. In Section 4 we describe how we evaluated Continuous Prototyping using a design-demonstration called PROTOTYPER. We finally provide an outlook on our next steps in Section 5.

2 Background

2.1 Prototyping

There are several definitions of prototypes in software engineering and human-computer interaction. Bruegge and Dutoit refer to prototypes as simplified versions of a system, similarly to Guida et al. who state that ”[a] prototype is a dynamic model of the software system under construction” [BD10]. Prototypes are demonstrating parts of the final system in a simplified manner. They are often used for the exploration and elicitation of requirements.

Several purposes and benefits of prototypes are mentioned in literature: a prototype tests and demonstrates the feasibility of the functionality of a system while assessing possible risks; it provides a common basis for discussion between developers, users and other stakeholders; and it is useful to explore a project's requirements which are often not completely known upfront [Ru09][Ur92][Bu92][Be09][GLZ99].

In the domain of mobile applications, the term *mockup* is frequently used in place of or in relation to prototype. Within this paper the term 'mockup' shall refer to prototypes of applications that are not written in code.

Mockups can be analyzed along different dimensions: representation, precision, interactivity and evolution [BLM12]. The representation depends on the format, which might be on paper, created with a design tool or with an online service. According to the representation, the mockup can then vary in detail and can be interactive or non-interactive, evolutionary or revolutionary. Furthermore, mockups can be executable. We define a mockup to be executable if it can be executed on the target environment of a system to be developed.

There is a variety of tools to create mockups of different form, precision, interactivity and executability. Several design programs and online tools, such as *Illustrator*[1], *Sketch*[2], *Balsamiq*[3] and *Marvelapp*[4] are capable of creating interactive mockups. Some services offer the possibility to download and execute a mockup on a mobile phone using proprietary solutions. However, as shown in Figure 1, none of these services comprehends the delivery process as a whole of both mockups and software being executable in the target environment. The term target environment refers to the hardware and software environment the product in development will be deployed to when released to its end users.

2.2 Continuous Delivery

CD is a software delivery concept which fits, because of its iterative nature, well into agile software projects [HF10a]. It implies that the software being developed is always in a state where it can easily be delivered to the customer or end user – but also that software is in fact frequently and regularly released. With the practices Humble and Farley describe in their book "Continuous Delivery", they claim that "[s]oftware releases can – and should – be a low-risk, frequent, cheap, rapid, and predictable process" [HF10a]. In order to achieve this, the process of building, deploying, testing and releasing software should be largely automated and tracked [Na15][HF10a]. CD is based on appropriate tool support for building a deployment pipeline. A typical deployment pipeline consists of a version control system which keeps artifacts like source code or media items, an integration service which automates the build and test process and a delivery service which covers aspects related to deployment.

[1] `http://www.adobe.com/de/products/illustrator.html`
[2] `https://www.sketchapp.com`
[3] `https://www.balsamiq.com`
[4] `https://www.marvelapp.com`

2.3 Our Contribution

Although there are several connections in literature between prototyping on the one hand and agile methods on the other, there does not seem to be an explicit link between CD and prototyping. While CD is well established for the delivery of software built from code, the automated delivery of mockups is not covered yet. Indeed, during our literature research we could not find a source explicitly concerned with the automated delivery of mockups. We also found no source that analyzed the role of the delivery process during the transition from mockups to actual software. In the next section we discuss how Continuous Prototyping solves this by allowing development teams to use the same delivery process for mockups as they use for software.

3 Continuous Prototyping

Continuous Prototyping incorporates iterative prototyping as an approach to software development. Additionally, it adopts CD for the delivery of these prototypes, such as mockups. Continuous Prototyping provides an automated and repeatable delivery process for all kinds of product increments from mockups to applications. We believe that the benefits of

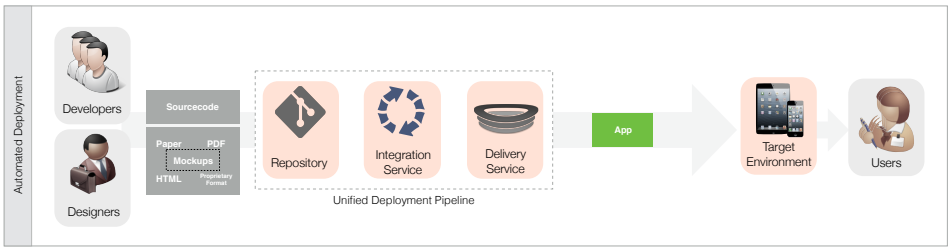

Figure 2: Solution: Continuous Prototyping - Deployment of software as well as interactive mockups using a unified deployment pipeline.

The foundation of Continuous Prototyping is a unified deployment pipeline, shown in Figure 2. Using this pipeline, artifacts like interactive mockups and actual software can be delivered to the target environment and accessed by users through the same channel. We believe that developers, designers and users gain the following benefits if they adopt Continuous Prototyping in their software projects right from the beginning:

Developers apply a certain set of workflows to structure their development activities like using a branching model in the version control system or a structured build promotion process for the delivery to different groups of users. Adopting Continuous Prototyping allows them to apply the same workflows for mockups as they are already using for apps. For example they could include the same user feedback system or usage analytics framework, starting from the delivery of the first mockup.

Designers can deliver executable mockups using the unified deployment pipeline. In collaboration with the development team they can prepare combined deliveries which include mockups and already implemented parts. The unified delivery pipeline allows *Designers* to apply concepts such as A/B testing to both mockups and mobile apps.

Testers and Users can access each new iteration, regardless of whether it is an early mockup or already implemented software, using the same delivery service such as an internal app-store. Starting from the first mockup, they can provide feedback using the same workflow for each release.

In the next section we present PROTOTYPER, a tool we developed to demonstrate the technical feasability of Continuous Prototyping.

4 Design Demonstration

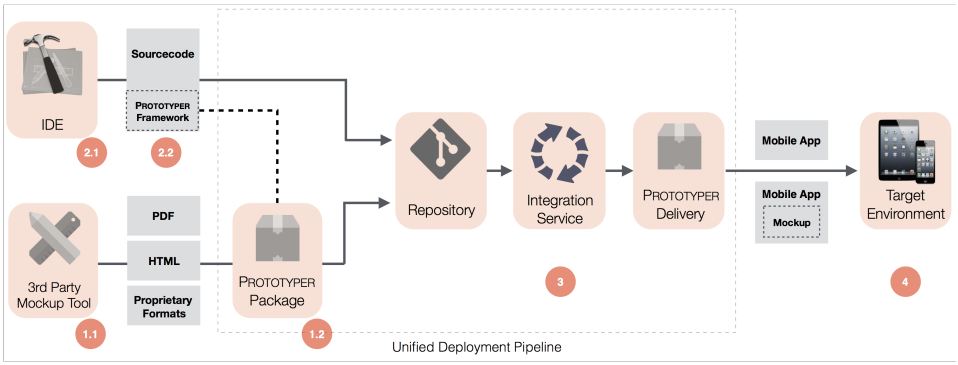

Figure 3: PROTOTYPER'S deployment pipeline architecture

PROTOTYPER is a software solution we developed in order to allow developers and designers to automate the delivery of both mockups and software. In this section we describe the architecture of the PROTOTYPER solution.

Figure 3 shows PROTOTYPER's overall architecture. PROTOTYPER allows software teams to deliver a user-interface mockup – created with a 3rd party mockup tool of their choice – as an executable mobile application. In order to achieve this, PROTOTYPER is integrated into a deployment pipeline, which in our example consists of a version control service, continuous integration service and a delivery service.

Within this pipeline, PROTOTYPER can operate in three modes:

PROTOTYPER PACKAGER *Designers* can use PROTOTYPER PACKAGER to transform mockups created using mockup tools into mobile apps. Figure 3 shows the components involved in this workflow. A developer or designer creates a mockup using a 3rd party tool (1.1). He uses PROTOTYPER PACKAGER to create an executable app out of the mockup

(1.2): PROTOTYPER PACKAGER first downloads e.g. a web-based representation of the mockup from the 3rd party tool. In a next step PROTOTYPER PACKAGER adds functionality for in-app user feedback and analytics to the downloaded mockup. Finally it creates a native mobile app matching the target environment of the project, such as an iOS app for the Apple iOS ecosystem. A developer or designer can then use the same deployment pipeline (3) that the software team uses for actual mobile apps to deploy the packaged mockup (4).

PROTOTYPER FRAMEWORK comes into place if a software team has already written parts of the application in code and creates a mockup for a new or redesigned feature that needs to be evaluated. Using PROTOTYPER FRAMEWORK, developers can create a package consisting of the mockup and a framework matching the target environment of the project. A developer can easily integrate this package into the existing software project using the IDE of the target environment (2.1). Using PROTOTYPER FRAMEWORK, he can now add a new part to the existing application which shows the new feature demonstrated in the mockup (2.2). After this step both parts, i.e. the parts of the application which are written in code and the mockup, are delivered using the unified deployment pipeline (3) and can be tested by a group of users (4).

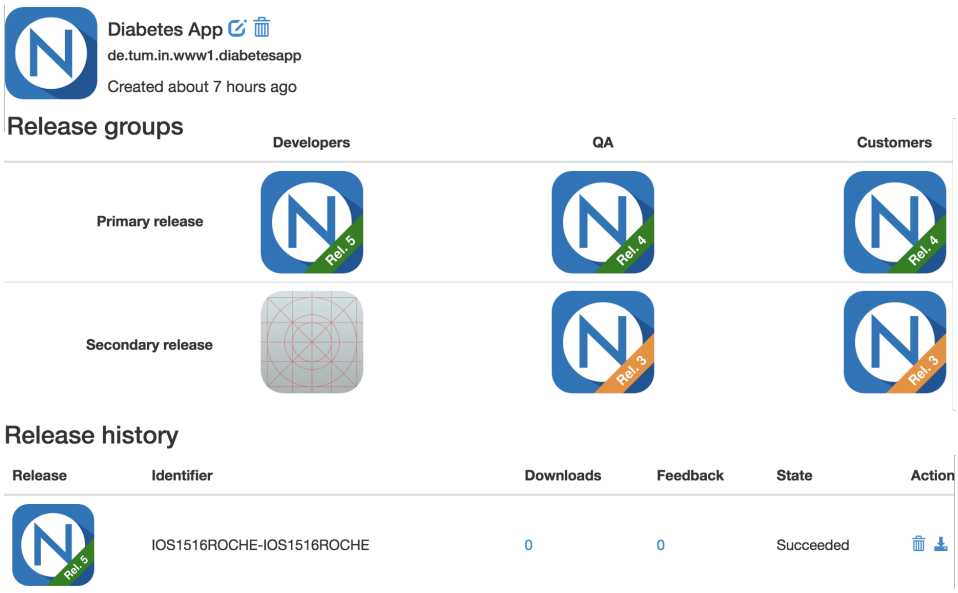

Figure 4: PROTOTYPER'S user interface for release group management (Excerpt). Mockup releases are orange, native apps green.

PROTOTYPER DELIVERY is the third part of our PROTOTYPER solution and the web-based app delivery solution developed. PROTOTYPER DELIVERY allows the delivery of all kinds of applications: Pure mockups, applications mixed with mocked parts and applications developed only in code. An excerpt of PROTOTYPER DELIVERY'S user interface is presented in Figure 4 and presents shows functionality: Developers and Designers can

define release groups, e.g. to target a Release to the QA-department or to a group of beta testers. Sometimes a Designer wants to deliver two different variants of a mockup to a group of testers in order to get feedback which one to choose. Using PROTOTYPER PACKAGER he can deliver two Releases at the same time to a release group. A tester can then install both simultaneously on his mobile device and compare them in the target environment.

5 Conclusion

In this paper we introduced the concept of Continuous Prototyping. Continuous Prototyping allows the automated delivery of user interface mockups as well as actual applications in fast cycles and using a unified deployment pipeline. We described the concept and showed PROTOTYPER, a design demonstration implementing the core principles of Continuous Prototyping in the domain of mobile application development.

The PROTOTYPER solution consists of three components: PROTOTYPER PACKAGER allows designers and developers to package mockups into executable mobile applications. PROTOTYPER FRAMEWORK allows teams to combine mockups and applications developed in code. With PROTOTYPER DELIVERY we implemented a uniform delivery pipeline for both, mockups as well as mobile apps.

As a next step we want to explore the impact of Continuous Prototyping on the communication between designers, developers and users. For instance, we will investigate how the more frequent and rapid delivery of mockups influences the quality of the developed software product. With regard to the PROTOTYPER solution, we will add an analytics component to improve the area of user feedback by e.g. automatically collecting contextual data or recording user interaction steps for a more complete picture of the usage context. We also plan to support additional target platforms like web-based environments. Finally we will evaluate PROTOTYPER in an industrial setting.

We believe that applying the concept of Continuous Prototyping will have a lasting benefit on the collaboration between designers, developers and users.

References

[BD10] Brügge, Bernd; Dutoit, Allen H.: Object Oriented Software Engineering Using UML, Patterns, and Java. Prentice Hall, 2010.

[Be09] Berenbach, B.; Paulish, D.; Kazmeier, J.; Rudorfer, A.: Software & Systems Requirements Engineering: In Practice. McGraw-Hill Education, 2009.

[BLM12] Beaudouin-Lafon, Michel; Mackay, Wendy E.: The Human-Computer Interaction Handbook. CRC Press, chapter Prototyping Tools and Techniques, pp. 1081–1104, 2012.

[Bu92] Budde, Reinhard; Kautz, Karlheinz; Kuhlenkamp, Karin; Züllighoven, Heinz: Prototyping. Springer-Verlag Berlin Heidelberg, 1992.

[GLZ99] Guida, Giovanni; Lamperti, Gianfranco; Zanella, Marina: Software Prototyping in Data and Knowledge Engineering. Springer Netherlands, chapter The Prototyping Approach to Software Development, pp. 1–32, 1999.

[HF10a] Humble, Jez; Farley, David: Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Pearson Education, 2010.

[HF10b] Humble, Jez; Farley, David: Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.

[Na15] Narayan, S.: Agile IT Organization Design: For Digital Transformation and Continuous Delivery. Pearson Education, 2015.

[Ru09] Rupp, Chris: Requirements-Engineering und -Management. Carl Hanser Verlag München, 2009.

[Ur92] Urban, Joseph E.: Software Prototyping and Requirements Engineering. Technical report, Arizona State University, 1992.

Towards DevOps in Multi-provider Projects

Masud Fazal-Baqaie[1], Baris Güldali[2], Simon Oberthür[3]

Abstract: DevOps addresses the continuity of development and operations activities in the software development lifecycle in order to achieve a better software experience via shorter development and release cycles with improved quality. A challenge in enterprise context is to achieve DevOps in multi-provider projects by synchronizing and coordinating various teams. We report on our experience in implementing DevOps principles in such a multi-provider environment and present good practices as well as open challenges.

Keywords: Multi-provider projects, DevOps, Continuous Software Engineering, Continuous Quality Management

1 Motivation

Big, enterprise-scale software systems are nowadays typically developed in a multi-provider environment with several onshore and offshore vendors. Adopting agile practices allows for frequent deliveries by the vendors, thus enabling frequent integration and early feedback by customers and reducing the project risks. In order to handle the customer requirements and the production incidents as soon as possible, companies are thriving to implement DevOps principles [Da16].

DevOps is based on and extends agile principles by fostering communication and collaboration. It advocates to overcome an "us-and-them" mentality, especially between people involved in software development (Dev) and people involved in software operations (Ops). Ideally, software teams have end-to-end responsibility for a software artefact throughout its whole lifecycle: from the product planning and the implementation, over its delivery and rollout to its operation. This is especially challenging in multi-provider environments that induce an additional layer of complexity for delivery management and quality management.

In this paper, we report on our experience based on a project from the financial domain. The client company is introducing a new client-centered, digital sales channel. Technically, this requires to open and extend existing backend systems for access via web-based interfaces for business partners and customers. Business processes need to be redeveloped and aligned, involving multiple business departments at once. The project consists of multiple sub-projects, where teams from the client company and teams from

[1] S&N CQM GmbH, Klingenderstraße 5, 33100 Paderborn, masud.fazal-baqaie@sn-cqm.de
[2] S&N CQM GmbH, Klingenderstraße 5, 33100 Paderborn, baris.gueldali@sn-cqm.de
[3] Mobile & Cloud Systems, Software Innovation Campus Paderborn, Zukunftsmeile 1, 33102 Paderborn, oberthuer@sicp.de

Copyright © 2017 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

several providers work together, involving also an offshore provider. The project was set up to follow an agile methodology. After realizing several improvements by better aligning with agile practices [FR15], it now adopts more and more DevOps principles for further improvements.

2 Experiences in Dev and Ops in Multi-provider Projects

Applying DevOps in a multi-provider project poses additional challenges. The first factor is heterogeneity. On the one hand, multi-provider projects can have cultural heterogeneity on different levels: Based on responsibilities, e.g., development vs. operation, based on the region people are working in, e.g., when offshoring, or based on differing company philosophies, e.g., regarding formality and hierarchies. On the other hand, multi-provider projects can have legal differences based on regulations and contracts, e.g., who is allowed to work on what and how. The second factors is scale: multi-provider projects are facing additional complexity with respect to the coordination of all the activities due to the size of the project that is typically big.

In the following, we want to describe challenges and good practices based on our experience. We do this based on the software lifecycle from planning to operations. Adopting a DevOps software lifecycle requires continuous quality management. We describe also the integration of quality assurance activities throughout the lifecycle.

In **Product Planning,** the project is facing mainly two challenges. First, overall business processes need to be realized by the interplay of various IT systems, thus they need to be cut down to system-specific requirements and distributed among the teams. Here, the heterogeneity described in the introduction needs to be accounted for, e.g., differing delivery dates. Second, new requirements have to be prioritized together with defects and improvement changes reported from production. In our project, we have created a macro plan (Project backlog) specifying the features of components (Product backlogs) and delivery schedules. Visual workflow modelling using BPMN helps for better understanding of the workflows, the component interfaces and the SLAs. Dev teams of providers derive their requirements (Sprint Backlogs) based on the Product backlog and the SLAs. All backlogs and production incidents are transparent to the project management all the time. Continuous reporting enables monitoring of risks and synchronizing the sprint planning of provider teams (see also using the agile release train pattern [Le11]). As quality guards, we use agile metrics indicating the execution status of test cases and criticalities of defects. The defects are classified and prioritized in direct communication between business departments, project managements and vendor teams [FGS15].

For the **Implementation**, beside architectural topics, a main concern is to achieve transparency about the quality delivered by various teams. This varied heavily among the different teams and was an obstacle for judging about the overall quality of the product. We have developed a maturity model to get transparency about the quality levels of subprojects and to motivate them to reach the next maturity level. The maturity

levels define minimal requirements on quality aspects (e.g. unit testing, UI testing) and standards (e.g. minimum test coverage, code quality measures) [FGG16]. Defining standardized Git (source code management) workflows and continuous integration pipelines helps to onboard new teams quickly. Teams use common artifact repositories and common Docker base images, which enable quick reaction to security vulnerabilities and to performance issues. We defined a microservice-based architecture [Ne15], which improves the flexibility of applications and reduces the dependencies between functional components.

The **Delivery Management** coordinates the deliveries of various teams on the basis of macro planning and product backlogs. After we have experienced some heavy delays of critical components, which resulted in delay of the whole project, we have worked on backward-compatibility of components. Teams have to implement database changes and interface changes in a backward compatible way. Thus, we can deploy components of various vendors as soon as they are delivered. Automated test scripts help to quickly validate that no regressions are injected between component interfaces due to unsynchronized deliverables.

Rollout Management: Deployments of many components by different vendors can be complex because of dependencies and thus need to be carefully planned. Typically such deployments contain applications, microsservices and databases. If offshore providers are involved, time zone differences and holidays may be a real problem for rollout management. Also the conditions of the hosting providers must be considered in rollout planning. The more components and teams a deployment involve, the longer is the installation time and the higher is the risks that something goes wrong. We made use of container technologies, e.g. Docker, for efficiently preparing installation packages and push them to various environments in very short times. Automated sanity tests validate, whether the productive system behaves as expected in production environment. Both the container technology and the backward compatible deliveries help us to reduce the risks of rollbacks in case of production problems.

Operations: Using an agile delivery model allows to continuously improve the product based on the feedback from the operations. Especially for enterprise-scale software in a multi-provider environment, it is important to maintain an overview and to associate incidents with responsible components/providers. In order to ensure the reliability and high availability of our systems, we have implemented clustering and failover mechanisms. Monitoring techniques detect performance issues and if components are not available they restart them. Meanwhile emergency teams are informed to resolve incidents, when automated start/stop scripts do not manage to resolve the production problems. We have used central logging for collecting runtime data from all components and created a log dashboard for various teams. We had to fulfill some special legal requirements for logging in order to supply offshore teams with logs for debugging.

3 Outlook

While we are making progress in our project with adopting DevOps principles, we are at no means at the end. One interesting aspect we are seeing is that in the past, effort was put in test mainly to increase the quality. Today with DevOps, automated software test is an enabler for shortening the release cycles while keeping or even increasing the quality. Thus, quality assurance can lead not only to increased quality but also to reduction of cost. Our vision is to push our concept of quality guards further, such that they become a self-contained part of the software lifecycle. Quality guards at different stages of the complete development and operation pipeline (from unit tests, over integration tests, to monitoring in production) assure that the requirements are fulfilled and, if they are not met, appropriate measures can be applied. The tool and framework landscape, which support such processes is today manifold, but needs proper selection and integration. A continuous quality management is therefore required. In this management, the quality guards have to be defined. Also important is to implement proper reactions for when a guard is not met. If, for example, an integration test fails, the team which added a new version of a component must be informed. If a guard fails in production, e.g. the monitoring detects the failure of components, unavailability of a service or unusual memory usage, resilient mechanisms must take place. Embracing the failure, e.g., like Netflix [Tse13] is doing by injecting failure to the productive system (Simian Army) is a good way to force every involved party to build systems to be able to heal itself. Independent delivery of chunks of functionality/parts of an application can help to keep speed for new/changing functionality. Building on an architecture like microservices [Ne15] and having end-to-end ownership (from dev over deploy to run) allows teams to deploy on their own speed independently.

Literaturverzeichnis

[Da16] Daly, D. et.al.: Enterprise DevOps – Building a Service Oriented Organization. Atos, 2016. http://ascent.atos.net/?wpdmdl=12439

[FGG16] Fazal-Baqaie, M.; Güldali, B.; Grieger, M.: Ganzheitliches Qualitätsmanagement in agilen Groß- Projekten. In (Engstler, M. et.al. Hrsg.): Proc of Projektmanagement und Vorgehensmodelle 2016. Köllen Druck+Verlag GmbH, Bonn, 2016, S. 109-120

[FGS15] Fazal-Baqaie, M.; Grieger, M.; Sauer, S.: Tickets without Fine - Artifact-based Synchronization of Globally Distributed Software Development in Practice. In (Abrahamsson, P.; Corral, L.; Oivo, M.; Russo, B. Hrsg.): 16th Int. Conf. of Product Focused Software Development and Process Improvement. Springer, LNCS, vol. 9459, 2015, S. 167-181

[FR15] Fazal-Baqaie, M.; Raninen, A.: Successfully Initiating a Global Software Project. In: Industrial Proc of the 22nd European Systems Software & Service Process Improvement & Innovation Conference (EuroSPI²2015). WHITEBOX, Denmark, 2015.

[Le11] Leffingwell, Dean: Agile Software Requirements. Addison-Wesley, 2011.

[Ne15] Newman, S.: Building Microservices: Designing Fine-Grained Systems. O'Reilly, 2015.

[Ts13] Tseitlin, A.: The antifragile organization. In: Commun. ACM 56, 8 (August 2013), S. 40-44. DOI=http://dx.doi.org/10.1145/2492007.2492022

Naming in deployment pipelines for SaaS

Thomas Kurpick[1], Sebastian Melchior[2]

Abstract: In the race to minimize operational costs, Software as a Service (SaaS) platforms has become increasingly popular. The development of SaaS, however, introduces several aspects that must be considered. In this paper, we present the continuous deployment pipeline of our eTrusted Enterprise SaaS Platform. Thereby, we focus on lessons learned during the evolution of our Continuous Deployment Pipeline with regards to naming and build step order.

Keywords: Engineering of Deployment Pipelines, Provisioning of Software & Infrastructure, Test Automation

1 Introduction

Trusted Shops is a supplier of feedback solutions in the ecommerce sector and offers several services for offline [Lo15] and online shops, such as customer reviews, product reviews and insurance of purchases. As digitalization requires enterprises to put the customer in the center of their organization, Trusted Shops founded the enterprise unit in 2016 offering a transactional feedback platform as a solution, not only for ecommerce but also for a wider variety of business areas, like insurance, mobility and banking.

Experience in scalable and robust service design for over a decade resulted in a microservice architecture that decouples the different aspects that are necessary for a transactional feedback platform. To minimize operational effort for operation and maintenance we choose a continuous deployment approach which allows changes to become visible as fast as possible [Ha10]. Furthermore, to maximize efficiency of our workforce and reduce human error we have minimized manual validation tasks.

Continuous deployment approach [Fi09] automatically releases every change that passes all the stages of the pipeline. We choose this approach to minimize the actual risk of a single change, and to add value to our platform as efficiently as possible.

Developing a SaaS platform together with a continuous deployment pipeline approach introduced the following aspects that needed further attention [HF10]: The general architecture of the platform that allows us to use independent deployment of systems and subsystems, source code management of these components and the actual deployment steps necessary to build, test and deploy the system and subsystems.

[1] Trusted Shops, Trusted Enterprise, Subbelrather Str. 15c, 50823 Köln, thomas.kurpick@etrusted.com
[2] Trusted Shops, Trusted Enterprise, Subbelrather Str. 15c, 50823 Köln, sebastian.melchior@etrusted.com

Copyright © 2017 for the individual papers by the papers' authors. Copying permitted for private and academic purposes. This volume is published and copyrighted by its editors.

2 Enterprise Continuous Deployment Pipeline

The platform was developed as a greenfield approach. The team decided to structure the new platform as a microservice architecture. Each microservice is developed independently from each other. No compiled sources are shared between the microservices. Runtime dependencies resulting from requests made to other services must be robust in case of failures. Deployments of a new version are always done with a rolling deployment.

2.1 Revision Source Control with git

For every microservice, we create a standalone git repository that holds all source code and build scripts that are necessary to build the final deployable artifact. For the git branching model we decided not to choose the popular GitFlow [Dr10] branching model, instead we used a simplified version in favor of a quality strategy that uses the pipeline as a quality gate and not the merges of the GitFlow branching model.

Development of new features, bug fixes and changes are developed in their own feature branch. Each commit is checked by our continuous integration server with unit testing and static code analysis. The results are added as commit comments. Once the changes are completed, the developer submits a merge request via our git repository server. The team member, who is assigned to the merge request, does the code review and can comment on critical parts of the changes. When the merge request is accepted, the merge is pushed automatically to the master branch.

This branching model allows us to minimize merges; guarantees that only the master branch is the source of the following pipeline. On the other hand, it is possible to reject changes before they are merged to the master branch. It is easier to reject a commit that has a typo or small errors, than to create an additional bug report or change request just for small changes.

The additional branches that are used by GitFlow are not necessary for SaaS, because we do not deliver or support multiple service versions, as suggested in [Ha07]. We only support the current deployed version x and the new version x+1.

2.2 Pipeline Steps with Jenkins

For each microservice we have two Jenkins jobs [Je11], a simple check job and the continuous deployment job. The check job is triggered for each non-master branch commit. It executes the unit and integration test. Additionally, a static code analysis with Sonarqube is triggered [So07]. The result of the code analysis is not recorded in the global Sonarqube instance, but only posted as comments to the responsible commit in Gitlab. This way potential technical debt is visible to everyone before the merge to master is approved. It can be discussed whether this should be fixed before the merge is approved, or if the issues found by Sonarqube can be accepted.

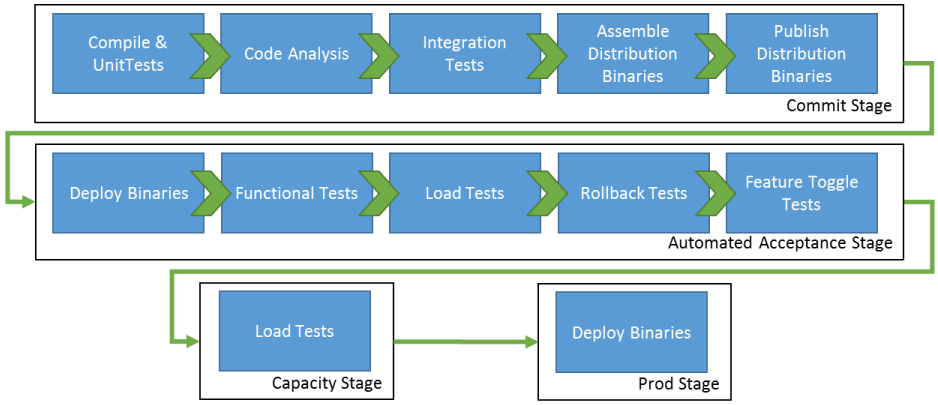

Figure 1 Overview of the deployment pipeline steps used for the eTrusted Enterprise Platform

The continuous deployment pipeline job is triggered with each commit to master. The master branch is checked out and built. After the build the unit tests are executed with the code coverage tool JaCoCo. After the test, the code is analyzed with Sonarqube. The results are transmitted to the Sonarqube server. After the analysis, the integration tests start. The difference between the unit tests and the integration tests is the intention of the tests. The unit tests are purely white box tests for functional tests of internal components without communication with other external components (database, other services, third party services). Whereas the integration tests focus on consumer driven tests that are determined from other teams.

After the tests, the microservice is assembled into a Docker container and published into the container repository. We use AWS as our cloud provider with the ECR service [Ec15].

After the publication of the new container, the pipeline triggers the deployment to the Automated Acceptance stage. The deployment is completed via helm on our Kubernetes Cluster. The deployment is configured as a rolling deployment, with no downtime of the services. After a successful deployment, automated functional end-to-end tests are initiated. These tests focus on user functionality that is visible to users of the platform. Load test for various parts of the platform are started after the automated end-to-end tests are finished.

Additionally, we execute service roll back tests. This test deploys the new version of the service, executes all functional tests, rolls the service back to the previous version and executes the tests again. This procedure guarantees that we are able to roll back to the previous version of a service, if there are problems during or after the deployment of the service. Another test step is used to validate our feature toggles for our platform. We use the feature toggle concept to release changes only for specific tenants or users.

After the Automated Acceptance Test stage, we deploy the service to the capacity stage. Here we stress test our service in a production like environment with load that simulates the production load. Test results are collected and stored in our monitoring system.

After successful performance tests, we deploy the service to the production environment. Besides the automated smoke test, there are no automated tests on this stage but monitoring of system properties. The production environment is monitored by Prometheus.

Errors that where not caught through the automated tests can break our system. But through the automated rollback test, we have the possibility to revert the change or fix forward. We have established an incident review meeting for these errors in order to improve our deployment pipeline. After the actual error is fixed, we have a meeting with the involved team to understand which automated tests were missing in our pipeline.

3 Lessons learned in naming in deployment pipelines

Existing naming guidelines for components focus on descriptive names [Jc99] or [Vs03]. At the beginning, we always used long descriptive names for our components like FeedbackRequestTriggerService, as suggested by the common naming guidelines.

As we developed our pipelines and development setup for our microservice architecture, we encountered different problems with names used to identify services and single page applications. In each system we added to the development and deployment setup, we required names, e.g.: git repository name, Jenkins job name, Kubernetes Helm chart name, etc... Each system has its constraints for the concepts we used, e.g.: Kubernetes [Ku14] in combination with Helm [He16] has a length constraint for its container names of 12 characters; otherwise the name will be trimmed. This resulted in naming our components with two names, a long descriptive name and also a short name with max 8 characters.

In addition, Kubernetes' service names are limited to only 63 characters, because the service names are used by Kubernetes as DNS name segments. This constraint is derived from RFC1035. Jenkins has a best practice of naming its job without the space character. Although it is possible, the job name is used as directory name in the underlying filesystem. Depending on the OS and filesystem format, directory names with spaces can lead to errors in build scripts. At the moment there are no other constraints known in our deployment pipeline. In figure 2 you can see the different systems and how they are interconnected with the names on the logical component.

With each new system, we integrate in our deployment pipeline we must check if new constraints are introduced to our component names, which makes it difficult to choose tools. Violated constraints are usually noticed during integration of these systems. This causes additional work to reassess the integration type and/or tool choice. In listing 1 is a selection of our current naming guidelines, that connect the different systems with each other.

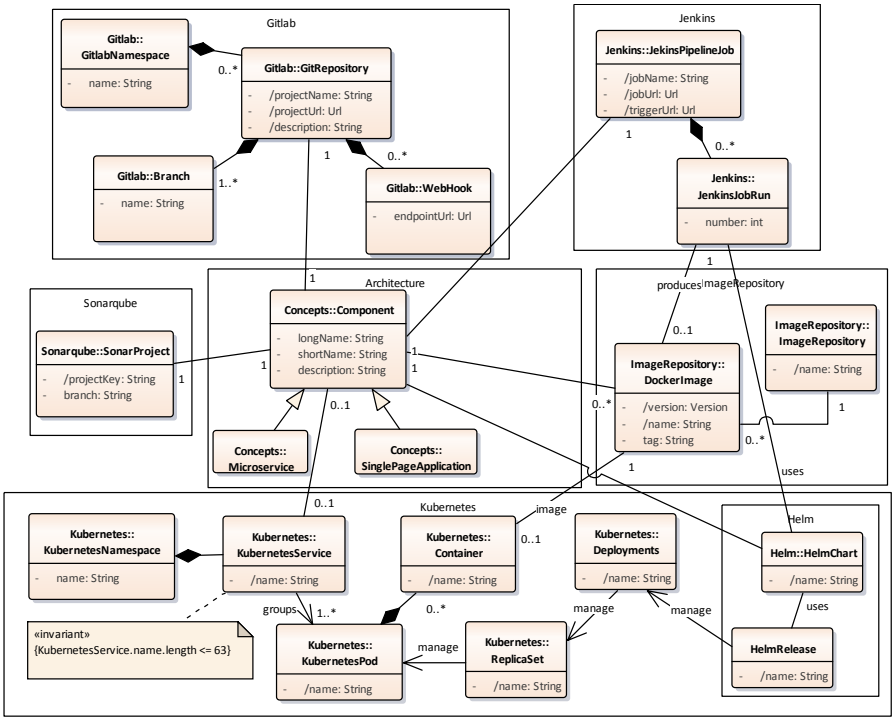

Figure 2 Connected naming concepts of the deployment pipeline systems

Because we integrate new systems in an incremental way, the risk that new systems are not well integrated exists. It could lead to the renaming of all other objects in existing systems, because we change the guidelines of an existing name. Or we introduce an additional name for our components, and then we have to name our existing components according to the new name. Both possibilities are not favorable.

```
KubernetesService.name = component.shortName
ReplicationSet.name = component.shortName + “-“ + <randomeNr>
KubernetesPod.name = ReplicationSet.name + “-“ + <randomeString>
HelmChart.name = component.shortName
HelmRelease.name = component.shortName
HelmFullname = HelmChart.name + “-“ + HelmRelease.name
Depoyments.name = HelmFullname
JenkinsPipelineJob.jobName = component.longName + “-pipeline-master”
GitRepository.projectName = component.longName
GitRepository.projectUrl = GitlabNamespace.name + “/” +
GitRepository.projectName
SonarProject.projectKey = OrganizationName + “:” + component.longName +
“:” componenent.GitRepository.Branch.name
```

Listing 1 Selection of naming guidelines and mappings used in our deployment setup

Our current naming guidelines for our components are:

1. **Long descriptive name**, use only lowercase words separated with “-“
2. **Short abbreviation name**, use only lowercase letters without any special characters. The maximum length is 8 characters.

3. **Additional description**, describe the business function of the component for a new or external person.
4. **Use the long descriptive name whenever it is possible in the deployment pipeline!**

At the moment the naming guidelines are manually enforced by our review process for new components. The other naming mappings of derived names are enforced by our deployment pipeline with scripts. We differ from the previous mentioned guidelines [Vs03, Jc99], because the names are part of URLs in our systems. This imposes constraints on the names that we have to address.

4 Summary & Conclusion

The current setup of our continuous deployment pipeline is working. We are confident that errors are detected in the early stages of our pipeline. Though our agile approach of adding new systems to our pipeline, we run into naming problems that forced us to reassess our naming guidelines and patterns more than once. In the future, we will automate the process for new components, so that we can create and deploy new components in a consistent way.

References

[An10] Anderson, David J: Kanban. Blue Hole Press, 2010.

[Dr10] Driessen, Vincent: A successful Git branching model, http://nvie.com/posts/a-successful-git-branching-model, retrieved: 8.1.2017

[Ec15] Amazon EC2 Container Registry, https://aws.amazon.com/ecr, retrieved: 8.1.2017

[Fi09] Fitz, Timothy: Continuous Deployment, http://timothyfitz.com/2009/02/08/continuous-deployment, retrieved: 8.1.2017

[Ha07] Hamilton, James R: On Designing and Deploying Internet-Scale Services. In: LISA'07 Proceedings of the 21st conference on Large Installation System Administration Conference. p. 1-18, 2007.

[He16] Helm: The Kubernetes Package Manager, https://helm.sh, retrieved: 8.1.2017

[HF10] Humble, Jez; Farley, David: Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education, 2010.

[Jc99] Java Naming Conventions, http://www.oracle.com/technetwork/java/codeconventions-135099.html, retrieved: 8.1.2017

[Je11] Jenkins, https://jenkins.io, retrieved: 8.1.2017

[Ku14] Kubernetes: A Container Cluster Manager, http://kubernetes.io, retrieved: 8.1.2017

[Lo15] Locatrust, Entwicklung von Vor-Ort-Trusted-Shop-Zertifikaten für Handel und Kleinunternehmen, Pressemitteilung Trusted Shops, Köln

[So07] Sonarqube: Continuous Code Quality, https://www.sonarqube.org, retrieved: 8.1.2017

[Vs03] Component Naming Recommendations, https://msdn.microsoft.com/en-us/library/4867dawt(v=vs.71).aspx, retrieved: 8.1.2017