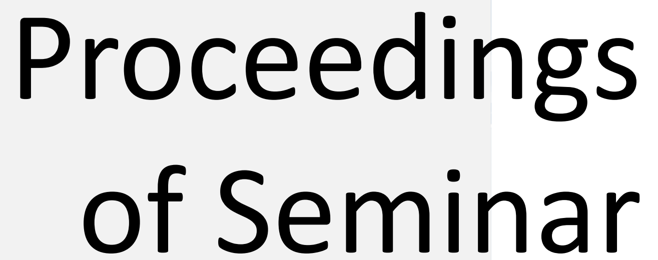


Proceedings of Seminar

Full –Scale Software Engineering

2014

Editors: Horst Lichter
Ana Nicolaescu
Andreas Steffens
Andrej Dyck
Firdaús Harun

Software Construction

RWTH AACHEN UNIVERSITY

The current state of 'Infrastructure as Code' and how it changes the software development process

Joel Hermanns
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
joel.hermanns@rwth-aachen.de

Andreas Steffens
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
andreas.steffens@swc.rwth-aachen.de

ABSTRACT

In recent years the DevOps movement gained much popularity. A key aspect of DevOps is automation, and one part of the automation is the so called "Infrastructure as Code" principle. This paper will describe goals and use cases of "Infrastructure as Code" and show benefits when integrating this principle in the software development process. Therefore we will look at different levels of infrastructure, from single machine level to multi-cloud systems, to see different use cases and point out the advantages and problems that may appear in these cases. Furthermore we will compare existing ideas and concepts to handle the infrastructure as code and analyze tools that implement these concepts. Finally we will look at ways to integrate these principles into the software development process, especially in terms of testing the infrastructure code.

Keywords

DevOps, Infrastructure as Code, Cluster, Provisioning

1. INTRODUCTION

Since the first DevOpsDays in 2009, the DevOps movement generated lots of interest and became very popular[18]. Similar ideas were already present before 2009, but not known as DevOps. For example, describes Debois the need for infrastructure agility 2008[13]. The concept of combining the roles of developers and operations was fastly adopted by various companies. The goal of this concept is to avoid common problems when operations and developers are separated teams. Developers want to iterate fast and deploy often, whereas operations are more interested in creating a stable and reliable infrastructure and want to avoid changes. Since agile processes became more popular recently the need for DevOps practices increased a lot, especially in fast moving environment at smaller companies and startups[18].

Similar to the processes, the datacenter and infrastructure needs have evolved in the recent years. As Hashimoto describes, the datacenter has changed from a single server structure to more complex scenarios combining virtualized hardware with several on-demand services [19]. Whereas software formerly ran on a single server, today's software may be deployed to fleets of virtualized servers, making use of various XaaS providers. These complex scenarios can no longer be handled manually by multiple administrators. Thus new ways are needed to manage and administrate the infrastructure. This is where DevOps ideas, especially "Infrastructure as Code", get important. Specialized software is used wherever repetitive tasks can be replaced and automated. The community and industry has produced a large amount of tools and techniques for this purpose in the recent years. This ranges form various provisioning and configuration management tools, which are meant to replace error-prone shell scripts, making it easier to manage the state of multiple machines, to software to manage clusters of servers. Frameworks like Mesos let applications view a cluster as a set of resources that can be allocated [20]. A similar idea is behind CoreOS, a minimial linux distribution, that works like an operating system for a cluster by providing a distributed init system to run applications in isolated containers [2].

These techniques allow better scalability and flexibility. For example, Spotify was able to scale to about 300 servers per operations engineer, which would have never been possible without a vast amount of infrastructure automation [24]. At Salesforce DevOps principles also reduced cost, in terms of time and money, for developing software, since each engineer is able to run production similar environments on the development machine without the hassle of setting up complex environments by themselves [23].

We can see that infrastructure automation is a hot topic and already widely adopted. In this paper, we will look at the variety of tools and concepts to maximize automation and see how it changes the software development process.

The rest of this paper is structured as follows. The second section will introduce the concept of provisioning and configuration management. Additionally, we will give two examples of tools. Section 3 will describe tools to setup machine enviroments and share them. We will look at Vagrant, a tool to setup development environments, and Docker, a tool to create Linux Containers. In section 4, we will look at multi-cloud systems, i.e. techniques to combine and manage various cloud related services. Section 5 will discuss changes in the software development process and will especially look at changes in testing and deploying software as well as testing infrastructure code. Finally, in the last section we will discuss similarities and differences between the tools and concepts we have looked at.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

2. PROVISIONING AND CONFIGURATION MANAGEMENT

2.1 Basic Concepts

The focus of provisioning and configuration management tools is to replace error-prone shell scripts that are used to manage the state of machines. For example, installing a certain set of packages and applying defined configurations to running services. Shell scripts may get hard to maintain and it is not easy to produce modular, reusable code. The idea is to provide or use another language to define these properties without the limitations of shell scripts. These languages are designed to allow great modularity to maximize reusability. The concept of the language itself may vary from a domain specific language (DSL) implemented in a common programming language (e.g. Puppet[6]), over tools using an existing language and acting in terms of a framework (e.g. chef[1]), to tools inventing a new language with specific, desired properties (e.g. Nix [15]). These languages are often declarative and describe the desired state of the system, instead of a way to achieve it.

Of course there are various other configuration tools designed for different purposes (e.g. IBM Tivoli System Automation for Multiplatforms or HP Server Automation System) that may follow other principles, but a more detailed look at these is out of scope for this paper (for a better overview see [14]).

In the following, we will look at two different tools: Puppet, based on Ruby, as a popular representant for open source configuration tools, and NixOS, coming from academica, based on the Nix language.

2.2 Puppet

In Puppet, everything is treated as a so-called resource. A resource can be a file, a package, a user or a group [26]. A Puppet installation consists of a server and multiple clients, where the server provides the configuration for each client. The various configurations are grouped into so-called manifests and stored on the server. The configuration on the server describes the desired state for each of the clients, by matching the clients hostname. The clients connect to the server and fetch and apply their configuration. The configurations itself can be organized in manifests and classes.

The following is a very basic example to show how the Puppet language looks like. If this manifest is applied, it will ensure that a package called "apache2" is installed, a user called "admin" is present and a file "/tmp/foo" with the content "I'm the foo file" and permissions 0640 is present in the system:

```
1 package { "apache2"
2   ensure  => installed
3 }
4 user { "admin"
5   ensure  => present
6 }
7 file {'foo':
8   path    => '/tmp/foo',
9   ensure  => present,
10  mode    => 0640,
11  content => "I'm the foo file.",
12 }
```

2.3 NixOS

NixOS is a purely functional linux distribution, build on top of the package manager Nix that works on UNIX based systems [15]. Purely functional means that each package is treated as kind of an immutable data structure. All packages are installed in a so-called Nix store on the filesystem. The file path consists of a cryptographic hash that is computed using the dependencies of the corresponding package. To describe how the package is built and installed, Nix uses a functional language called Nix. Since each package is installed in its own unique path and symlinked to all other packages that depend on it, Nix allows to have multiple different versions of one package installed without dependency problems. This way, Nix is able to allow (nearly) atomic upgrades and rollback of your installed packages.

NixOS extends the approach of package management to configuration management [16]. Basically every static part of the system (i.e. configuration files, bootup scripts, all other parts that do not change during runtime) is treated the way Nix works with packages. This allows to build the whole operating system based on configuration files written in Nix, including partitions, filesystems, packages, services and their configuration files. Everything is based on the functional language Nix, without the need to know different configuration formats or languages. When the configuration file has changed, the user can build it with optional activation at next boot. Furthermore, the user can also create a virtual machine with these changes, without affecting the host system. Because of Nix purely functional manner, the same configuration file guarantees to produce the same result, either if you install from scratch or upgrade an existing system (except globally mutual data such as database contents or personal data).

The following shows an example of a NixOS configuration. Each configuration is basically a function. The first line defines the head of the function, which takes at least two arguments: "config" and "pkgs". The function returns a set of option definitions, which is denoted by the block between { and }. In this case it sets two options. The first one enables a service called "httpd" and the second sets the attribute "documentRoot" of this service to "/webroot".

```
1 { config, pkgs, ... }:
2
3 { services.httpd.enable = true;
4   services.httpd.documentRoot = "/webroot";
5 }
```

Summary.

As we have seen configuration management and provisioning tools try to add better flexibility for the installation process of single machines. The most common concept to achieve this flexibility is the use of a DSL.

3. MACHINE ENVIRONMENTS

This section is about environments applications are running in. A common problem with software that is deployed in production, is the enviroment. Software needs to be developed and tested in production like environments. Virtualization and recently containerization are concepts to make such environments portable, but without any additional tools this may get hard as well. In this section, we will focus on tools

that aim to improve this situation. We will introduce Vagrant, a framework to set up more complex development environments, which is used by various projects. Secondly, we will look at Docker, because it is currently a very popular tool for creating and sharing containers and its ecosystem is moving fast.

3.1 Vagrant

Vagrant is basically a Ruby abstraction layer on top of the creation of development environments [9]. It comes with a command line application that reads a so-called Vagrantfile. The Vagrantfile contains information about the desired system in form of Ruby code. The developer describes the base image of the resulting machine, some system settings like hostname, network settings and provisioning methods. Additionally, one can add shared folders, i.e. folders on the host system that can be accessed from within the machine. With just a few commands Vagrant will create and provision the machine and connect via SSH. This way, the development can take place on the developers machine, also if complex production like settings are needed, whereas the Vagrantfile and any needed scripts can easily be shared via version control systems. A sample Vagrantfile could like the following:

```
1 Vagrant.configure("2") do |config|
2   config.vm.box = "hashicorp/precise64"
3   config.vm.synced_folder "src/", "/srv/
      website"
4   config.vm.provision "shell",
5     inline: "echo Hello, World"
6 end
```

In this example a base box called "hashicorp/precise64" is used. If the box is not present locally, it will be downloaded from a publicly available catalog. Line 3 defines a synced folder. This syncs the local subfolder "src/" with the folder "/srv/website". Additionally, we define a shell provisioner in line 4 and 5. It will run "echo Hello, World" when the machine has booted successfully the first time. Besides shell, Vagrant supports various other provisioning techniques, such as Puppet or Chef.

Vagrant is able to create machines via different providers, such as VirtualBox and VMware.

3.2 Docker

Docker is a tool that lets you create environments called images, share them and run isolated applications inside. The concept of containerization is similiar to virtualization. Whereas a virtual machine runs on a set of virtualized resources, a container shares the resources with its host. Docker uses different techniques offered by the Linux kernel to isolate and abstract the container form the host system. The Linux kernel allows to group processes into different namespaces, where different groups cannot see the processes of each other. This namespace isolation allows Docker to run each container in its own sandbox. Using these techniques, Docker is able to run applications in an environment similar to virtualization, but way more lighweight. Because of this very little overhead, a container can be started in just a few seconds instead of minutes. Another important aspect of Docker is the way to share and distribute images. A Docker image is always derived from a base image, e.g. ubuntu or fedora. On top of this basic environment one can perform different actions, such as adding an environment variable,

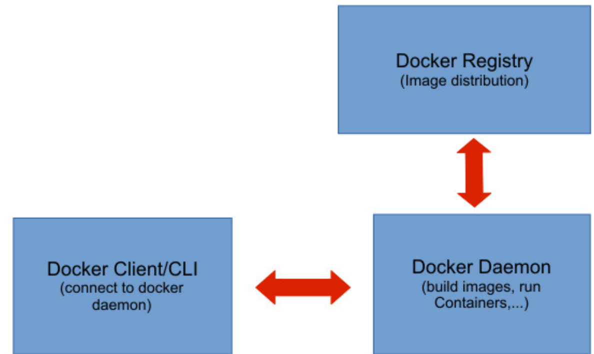


Figure 1: Basic components of Docker

adding a persistent volume or running arbitrary commands. Everytime one of these actions is performed a new layer is added on top of the current file system of the image. Basically an image consists of multiple layers that are merged together at startup. Because these layers can be shared separated, only the missing layers need to be downloaded.

Docker allows basically two ways to create an image. The first way is to use the Docker CLI. It allows to perform actions on an existing image and to save the result as a new image. This is quite similar to commiting in version control systems and therefore it is called commit. The second way to create a docker image is to use a so-called Dockerfile. Docker images can be built using a specific file that contains meta information and the action to perform when building.

The following would create a container with the current directory copied to /data and a simple HTTP server running on port 8080:

```
1 FROM ubuntu:latest
2 ADD . /data
3 WORKDIR /data
4 RUN apt-get update
5 RUN apt-get install -y python
6 EXPOSE 8080
7 CMD python -m SimpleHTTPServer 8080
```

In figure 1 the basic components of Docker are shown. The Docker daemon is responsible for running containers, building images, etc. The Docker client/CLI is used to communicate with the daemon via a socket or its RESTful API, therefore client and daemon do not need to run on the same host. Additionally, a Docker Registry can be used to distribute the images, the publicly available registry is also known as the Docker Hub.

Summary.

As we have seen there are different solutions to create machine environments. Both, Vagrant and Docker, simplify the creation of portable environments using their own DSL. Whereas Vagrant focuses on environments for development and testing, Docker containers may be used from development to production and allow great scalability for deployment in large clusters.

4. MULTI-CLOUD SYSTEMS

In today's software world, everyone can easily setup servers on demand in just a few minutes. Creating new resources on cloud services like Amazon Web Services or Google Compute Engine is a matter of minutes. All these platforms provide RESTful APIs, which allow develpers to create tools that add a new abstraction layer on top.

In [22] a pattern based deployment service is introduced that is able to deploy applications and services in different cloud systems, including AWS and local OpenStack installations. The user describes the application (e.g. a JavaEE web application) and required external services (e.g. MySQL, nginx) in a XML based language and the service is able to fully automatically deploy the application. It will startup the required server and generate chef code to provision them. Furthermore, it will create connections between different services if required.

Breitenbücher et al. introduce a technology to connect service centric platforms (e.g. AWS, Microsoft Azure) and script centric provisiong solutions, like Puppet or chef [10]. Based on so-called Planlets, the technology is able to use different solutions and combine them to create and provision resources and deploy applications. They are able to use different cloud systems and provision solutions in a single deployment.

In the following, we will show two tools that try to create a consistent view on infrastructure related cloud services. We will look at Terraform as a tool developed in industry and NixOps, an extension for NixOS, coming from academica.

4.1 Terraform

Terraform is built by the developers behind Vagrant and focuses on launching infrastructure on different service-centric platforms [8]. It supports various providers (including Google Cloud, AWS, DigitalOcean, Heroku) and makes it easy to add more. Based on its declarative description language, the user describes the infrastructure with provider specific resources:

```
1 # Cluster nodes
2 resource "digitalocean_droplet" "node" {
3   image = "coreos-stable"
4   name = "node-${count.index}"
5   region = "AMS3"
6   size = "4GB"
7   private_networking = true
8   ssh_keys = [123456]
9   count = "${var.no_nodes}"
10 }
```

The example defines a resource "node" with the type "digitalocean_droplet" (basically a cloud server). The attribute "count" defines the number of instances we want to create, which is taken from the variable "no_nodes". The name of each instance, set by the "name" attribute, includes its index. The attributes "image", "region", "size" and "ssh_keys" are specific for the type we use. "Image" and "size" describe the server and "region" defines in which datacenter we want to create it. "ssh_keys" contains a list of IDs for keys that should be added to the created instance.

A command line application is then used to apply this description. The current state is saved in text based files, so it can be shared.

4.2 NixOps

Based on Nix and NixOS, NixOps (formerly known as Charon [17]) came up in 2013. NixOps extends the idea of NixOS to a set of machines or "network of machines". Basically, it allows to provide Nix based configuration to different machines. In addition, to the configuration of every NixOS host the user describes the provider to use for deploy. NixOps supports different providers, including Amazon EC2 and VirtualBox, so one can deploy the same configuration to a production environment or a local virtual machine used for testing purposes. When deploying multiple machines to a cloud based service, NixOps will add connections between the different machines. The state of each deployment is saved in a SQLite database so NixOps will remember what has already been done.

Summary.

As we have seen, cloud provisioning tools act very similar to the previously introduced configuration management and provisioning tools, but on a much higher level. DSLs are used to create a consistent view on infrastructure built using different cloud services.

5. THE SOFTWARE PROCESS

In the previous sections we introduced the variety of tools for setting up and managing infrastructure with code, respectively creating development or testing environments. The following discusses how to integrate these concepts into a typical software development process and how it has changed the software development already.

5.1 Testing infrastructure

This section is about testing infrastructure code. We will cover different solutions to test written Puppet code and look at a new service developed by Spotify, which is able to test services in docker containers based on JUnit tests.

Puppet CLI and puppet-lint.

The Puppet command line tool already gives the opportunity to run the agent, without acutally applying the manifests. This way the developer can already catch various issues. But since a human is involved and this process cannot be automated, puppet-lint was created[5]. Puppet-lint, as all other linting tools, analyzes the code and tries to find common mistakes. Additionally, puppet-lint can find style guide issues to force a consistent format.

Rspec-puppet.

Whereas puppet-lint may already catch many issues, it does not actually test the code. Rspec-puppet tries to solve this problem. With rspec-puppet the developer can test if the Puppet mainfests get compiled the right way. It allows testing of different hosts, i.e. if each host gets the right configuration, if certain files get certain attributes and contents.

Rouster.

The engineering teams at Salesforce adopted DevOps principles recently [23]. But the quality assurance team was not satisfied with the existing testing solutions for Puppet, so they developed Rouster, which allows to write functional tests for Puppet manifests [7]. Basically the difference to testing with rspec-puppet is that the manifests are actually

applied. So the final result is tested instead of a state inbetween.

Basically, rouster is just a Ruby wrapper around the Vagrant command line interface and SSH. Additionally, there is an extension to check running EC2 instances, instead of a Vagrant machine, since Rouster only needs an SSH connection. This way you can run arbitrary commands on the corresponding machine and check the results. With predefined methods, to check for example if certain directories or files exist, this can easily be written as unit tests, but will actually test existing machines.

JUnit testing Docker container.

Spotify recently started migrating from mainly Puppet managed servers that get updates via Debian packages, to a solution backed by Docker containers. Since they did not found a solution that satisfies their requirements, they built Helios [24]. "Helios is a Docker orchestration platform for deploying and managing containers across an entire fleet"[3]. One feature of Helios is the ability to test the built Docker containers using JUnit test cases. Helios will create the needed containers temporarily (based on what is described in the JUnit test) and run tests against these.

5.2 Testing and deploying software

DevOps practices and tools also change the way how testing and deployment of software is done. We already looked at Vagrant to simplify development environment and will now see how Docker simplifies testing and deploy of software.

Soni and Hunnargikar describe how Ebay's internal CI solution works [25]. They are running a Mesos powered cluster, in which Jenkins servers are running. Each Jenkins job will then be executed in a Docker container. During the build process, a new docker image will be created containing the result of the software build. The resulting Docker image is then published to an internal private registry and can easily be brought to a production server.

At Gilt the process works similar [12]. They are currently experimenting with immutable infrastructure. Each new version of software is deployed onto a new server in a docker container. After successfull tests the new service will be brought to production side by side with older versions of the service. Instead of updating the old versions each new one gets a completely fresh environment to avoid problems with mutating the system's state.

At Google I/O 2014 Google introduced their own container cluster manager, called Kubernetes [4, 11]. This manager extends the container idea to a set of containers that belong together, a so-called pod. Kubernetes will supervise all containers to make sure the desired state is valid.

Another testing solution comes with NixOS. Van der Burg and Dolstra show how complex test case scenarios can be automated using Nix and NixOS [27]. Because of the declarative model of Nix, each configuration will be the same regardless on which host it is running. With this idea, they can spin up virtual machines that share their host's Nix store and run complex tests without corrupting the user's system in an automatable way. For example, they are able to write tests for a multiplayer game, involving a server and a client in different machines.

Summary.

We have seen how infrastructure code can be tested and how "Infrastructure as Code" concepts have changed testing and deployment of software. With Mesos, Kubernetes and Helios we have introduced tools that act on a cluster or datacenter level.

6. DISCUSSION

The basic idea of all these tools is to add an abstraction layer to give a high level interface to some low level functionality. This varies from managing configuration and provisioning servers to managing resources and services in large clusters. Basically, it is similar to concepts in software development, where code is structured into functions, modules and libraries to give a consistent, high level interface to lower level tasks and maximize reusability. Nevertheless, we can see certain differences in how these abstraction layers are achieved. The concepts can be divided into two categories, script-centric and service-centric approaches. In the following we want to explain these into more detail.

Script-centric approaches are based on some kind of code. In this category a common concept to add the abstraction layer is a DSL. So a programming language is used and the tool transforms the source code into another form. This concept is similar to a compiler of a general purpose language. Tools that belong to this category are for example Vagrant, Terraform or Nix/NixOS/NixOps.

Service-centric approaches try to give an abstraction layer by offering services. The provided interface may vary. It can be a RESTful API, a web UI or a configuration file based approach. Furthermore, the level on which these tools are implemented varies. Most often the tools itself are implemented as an application or service, e.g. Mesos, Kubernetes or Helios, but there exist other solutions. For example, in CoreOS the services are deeply coupled with the operating system itself.

These two categories are not clearly distinct. We can find various tools that belong to both categories. Puppet, for example, is one of these. The Puppet language itself is definitely a script-centric approach whereas the distribution of configuration among multiple nodes done by the Puppet master would reside in the script-centric category.

Summary.

We have seen that infrastructure management exists at various levels. In figure 2 you can see how different tools and parts of infrastructure management build upon each other and create a hierarchy. The figure does mainly include tools and techniques we introduced in this paper, so it is not complete. Furthermore, the limitations are not clearly fixed. Functionality of certain tools may spread across multiple levels.

Additionally, we have seen that infrastructure related tasks are part of all stages of the software development process, from creating development environments, over setting up build pipelines, to deployment. So the infrastructure management literally embraces the development process. Infrastructure and software are deeply coupled and can be developed and deployed simultaneously, in other words, the infrastructure is shipped with the software.

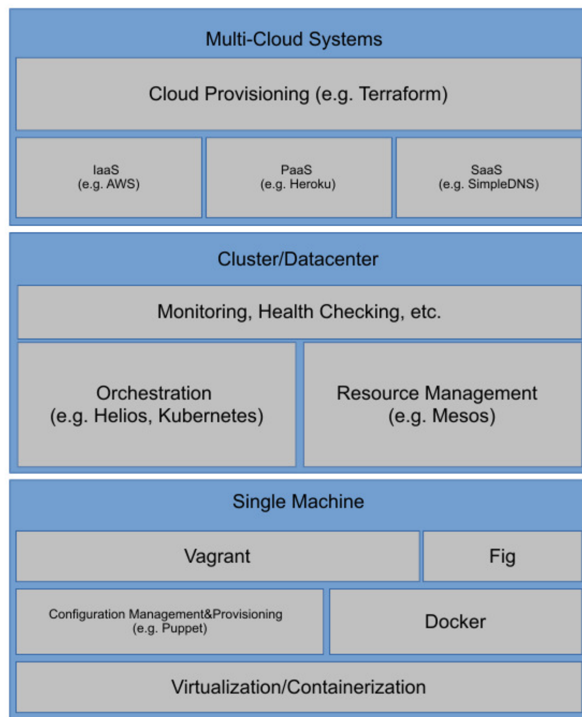


Figure 2: View on levels of infrastructure tools

7. CONCLUSION

As you can see DevOps and 'Infrastructure as Code' are hot topics. There is already a vast amount of techniques and tools to support these ideas. Especially Docker is currently gaining a lot of popularity and its ecosystem is growing quickly.

We have seen that these principles bring a lot of benefits to the software development process and have already changed the way software is developed or deployed and this shift is still going on. In the future, software will be used to create, control and manage nearly all parts of the infrastructure. Technologies like Software Defined Networks (SDN) are getting more popular and will result in ideas like Software Defined Infrastructure (SDI, [21]). It will be interesting to see where this is leading to.

8. REFERENCES

[1] Chef. `https://www.chef.io/chef/`. Accessed: 2014-12-20.

[2] Coreos. `https://coreos.com/`. Accessed: 2014-12-20.

[3] Helios. `https://github.com/spotify/helios`. Accessed: 2014-12-20.

[4] Kubernetes. `https://github.com/GoogleCloudPlatform/kubernetes`. Accessed: 2014-12-20.

[5] Puppet-lint. `http://puppet-lint.com/`. Accessed: 2014-12-20.

[6] Puppetlabs. `http://puppetlabs.com/`. Accessed: 2014-12-20.

[7] Rouster. `https://github.com/chorankates/rouster`. Accessed: 2014-12-20.

[8] Terraform. `https://www.terraform.io/`. Accessed: 2014-12-20.

[9] Vagrant. `https://www.vagrantup.com/`. Accessed: 2014-12-20.

[10] U. Breitenbücher, T. Binz, O. Kopp, F. Leymann, and J. Wettinger. Integrated cloud application provisioning: Interconnecting service-centric and script-centric management technologies. In *On the Move to Meaningful Internet Systems: OTM 2013 Conferences*, pages 130–148. Springer, 2013.

[11] C. M. Brendan Burns. Containerizing the cloud with docker on google cloud platform. Google IO (Video, 2014.

[12] M. Bryzek. Immutable infrastructure with docker and ec2 at gilt. Docker Con (Video), 2014.

[13] P. Debois. Agile infrastructure and operations: how infra-gile are you? In *Agile, 2008. AGILE'08. Conference*, pages 202–207. IEEE, 2008.

[14] T. Delaet, W. Joosen, and B. Van Brabant. A survey of system configuration tools. In *LISA*, 2010.

[15] E. Dolstra, M. De Jonge, and E. Visser. Nix: A safe and policy-free system for software deployment. In *LISA*, volume 4, pages 79–92, 2004.

[16] E. Dolstra and A. Löh. Nixos: A purely functional linux distribution. In *ACM Sigplan Notices*, volume 43, pages 367–378. ACM, 2008.

[17] E. Dolstra, R. Vermaas, and S. Levy. Charon: Declarative provisioning and deployment. In *Release Engineering (RELENG), 2013 1st International Workshop on*, pages 17–20. IEEE, 2013.

[18] B. Erk. Wie sich devops in der it etabliert - weg von der insel. *Admin Magazin*, 4:58–60, 2014.

[19] M. Hashimoto. Taming the modern datacenter. FutureStack14 (Video), 2014.

[20] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[21] M. Kuppinger. Software defined infrastructures - offeene zentralverwaltung. *ITAdministrator*, 7:50–52, 2014.

[22] H. Lu, M. Shtern, B. Simmons, M. Smit, and M. Litoiu. Pattern-based deployment service for next generation clouds. In *Services (SERVICES), 203 IEEE Ninth World Congress on*, pages 464–471. IEEE, 2013.

[23] D. Mangot and R. Mathew. On the journey of an enterprise transformation, quality is still job 1. Devops Enterprise Summit (Slides, Video), 2014.

[24] R. Singh. Docker at spotify. Docker Con (Video), 2014.

[25] M. Soni and A. Hunnargikar. Delivering ebay's ci solution with apache mesos and docker. Docker Con Video, 2014.

[26] J. Turnbull and J. McCune. *Pro Puppet*. Books for professionals by professionals : The expert's voice in open source. Apress, 2011.

[27] S. van der Burg and E. Dolstra. Automating system tests using declarative virtual machines. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 181–190. IEEE, 2010.

Software Health Management in Information Systems

Vasil Borozanov
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
vasil.borozanov@rwth-aachen.de

Horst Lichter
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lichter@swc.rwth-aachen.de

ABSTRACT

For large-scale systems it is important to detect, diagnose, predict and mitigate the errors caused by the software. Software Health Management (SWHM) is the field that comes up with the tools and technologies to automate these processes.

Safety-critical systems, such as those in the aircraft industry, machinery and cars are already implementing SWHM techniques. These systems rely heavily on sensor data and have a clear specification what the output looks like, which makes the anomaly detection easier. Here a two level strategy is applied for managing the health (health of individual components and health of the overall system).

The paper will investigate the appropriate health models and see what how they can be modeled. It will give overview of the factors that make the system ill, propose metrics to measure the health and introduce basic actions to mitigate the risks. It will also give an overview of symptoms and the factors that degrade the performance. The paper will propose an approach for health management of Information Systems. It will describe the main components and the general architecture. This approach will be complemented with an example and from it, we will see which are the potential obstacles. The paper will be concluding with the summary and it will propose further steps.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*health measures*

Keywords

Software Health Management, Information Systems, Software Health Metrics

1. INTRODUCTION

Health of a software is a question that gets importance lately. Information Systems (IS) that grow in size are getting extremely complex to maintain [2]. Adding or changing a feature requires great amount of work and patience, while trying to avoid the risk of introducing new errors. Although producing error-free software is impossible, we can take all the steps to apply techniques that minimize this problem. Software Health Management (SWHM) is an extension of the classic fault tolerant systems and typically includes the activities of anomaly detection, fault source identification, mitigation, maintenance, and fault prognosis [11] . We will focus our study on the general understanding of the health of a software, review the causes which make the software ill and briefly overview a implementation for health management. The paper is structured as follows:
Section 2 explains what is an aged system. This will give an introduction to the symptoms and the causes.
Section 3 will define the term health and it will introduce measurable metrics.
Section 4 will introduce the term SWHM and list several approaches.
Section 5 will investigate one of those approaches and adapt it for our needs.
The final section will give a summary of the paper.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

2. WHEN DID OUR SYSTEM BECOME AGED?

Software products are in constant evolution: applying changes, fixing bugs or adding features - they all lead to alteration. If a product does not undergo these changes, it becomes obsolete. While taking all the measures to prevent this, each change can potentially introduce errors. By aging, we refer to degradation of the performance or a sudden crash of a software system due to exhaustion of operating system resources, fragmentation and accumulation of errors [6].
The definition shows that effects can vary from slow response up to total failure. Software failure occurs when the delivered service deviates from the specifications of the system. Failures are not limited to the whole system; they can appear in a single component as well. Regardless of the appearance, they are much more severe than simple bugs or reduced response time: the outcome of the procedure is not acceptable. We consider the system aged when it shows one of the following symptoms: [10]

- Inability to keep up: software tends to grow in size. This leads to harder maintenance: adding methods is difficult, documentation can get cumbersome, finding the right part of the code takes time etc.

- Reduced performance: Larger systems require more resources. For example, the increased demand for storage size leads to larger database.
- Decreased reliability: Frequent changes introduce errors, which reduces the stability of our system (decreases the entropy).

Large-scale software systems suffering from these symptoms can cause enormous amount of money. Instead of investing in new features in the system, the focus will be put on fixing the problems caused by the degradation of the system.
Different factors can lead to decayed state. Resource leakage (unreleased memory, file handlers or sockets), fragmentation, round off error accumulation and data corruption are among the most common ones. [10]. All of them cause abnormal behaviour that may differ from case to case (slow calculation, out of memory exception etc.). This raises the question of classification of the aging effects. An internal vs external classification, or a technical one (OS- vs App-specific) will not help us into ranking them per severity. Volatile and non-volatile groups have been proposed [6], where the main criteria is the ability of the system to function properly after a reboot. We can view this as classification by impact of the system: more serious symptoms cannot be fixed by a simple reset command. This means that if the rating is higher, then the chances of the normal behaviour of the system are slimmer. We are going to consider this classification to serve us as a way of measuring the well-being of the system.

3. ESTIMATING THE HEALTH OF A SYSTEM

To successfully measure the health of software, we need to come up with a health model. A healthy software should behave stable, be easy to use, easy to maintain and functions without errors. These are all part of a quality attributes.
Several definitions for software quality exist. For our purposes, we take the definition from a manufacturing (engineering) point of view, which defines it as conformance to specification [8]. Deviation from the specification will be considered as a deviation from the health model (illness).
The ISO 9126 lists the following characteristics for software quality: functionality, reliability, usability, efficiency, maintainability and portability. A proposal to measure the maintainability of the system as a health has already been given ([2]), since up to 60-80 percentage of the work effort can go into maintenance [3]. To measure and quantify the maintainability of a system as a health model, we need to calculate its index. There are several possible methods for measuring the maintainability index (MI). They are based on attributes such as average lines of code (aveLOC), average Halstad Volume per module (aveVol), average extended Cyclomatic Complexity per module (aveV(g)) and number of comments per sub-module in percentage (perCM). The most simple methods are:
1. Hierarchical multidimensional assessment model (HPMAS): MI index is calculated as product of different dimension criteria (control structure, information structure and typography). By assigning an optimal trigger value range (example aveLOC ranges 10-50), each time when the metric falls out (example module with LOC size 70), weighted deviation is computed [2].
The same study showed an interesting result. The MI rises slowly with only avgLoc as an attribute. That means selecting only few attributes cannot capture the health of the maintenance good enough.
2. N - Metrics Polynomial: The goal is to create a polynomial equation, where the MI is expressed as function of metric attributes, obtained by linear regression [2]. This model requires a lot of testing and calibrating for the most suitable coefficients.
Thorough trial and error, we can end up with a four-metric polynomial example in the following form:

$$Maintainability = 1715.2 * ln(aveVol)0.23 * aveV(g) -$$
$$16.2 * ln(aveLOC) + (50xsinSQRT(2.46 * perCM))$$

Another variation widely used in practice is the three-metric polynomial. Selecting fewer arguments will not result in a reliable MI. Selecting many arguments will result in a complex function which will require a substantial amount of effort to be fitted (basically more coefficients means more try-error attempts). As stated, the weights given to the regression function will differ from system to system. The measurement is not restricted to the code only. It can include combined factors as bugs per module size, productivity of the programmer (LOC per unit of time), stability of requirements (initial number of requirements vs current number of requirements) or spoilage of the system (effort in bug fixing vs total effort) [7].
The downside of these metrics is that they exclude a fundamental part of the system. Better approach is to focus not solely on the correctness of the system, but on the economic effectiveness on the software as well. In other words, the system is healthy if the costs for maintenance are low. [12].

This quality model organizes the criteria in two dimensions: activities and facts (Figure 1). The top attributes are the activities performed on the system, each having its own cost, and left are the facts that describe the technical state of the system. Separation is important to keep the consistency of the model and to successfully describe the dependencies between different criteria. For example, on the figure we can see that the Tool fact, which consists of Debugger and Refactoring, does not influence the the coding, which is part of the implementation (for modern IDEs, this is not true, as they provide a lot of integrated tools for increasing productivity, e.g. Eclipse can automatically organise imports. The purpose of this example is to illustrate how the Cost matrix is constructed and how dependencies are viewed).
The end result is a model consisting of code-facts (copy-paste, unused code, naming quality), documentation (level of completeness, homonym ratio) and organizational facts (number of employees with required technical knowledge, number of employees with in-depth system knowledge) and relation between them.
This model was also put into practice and health check was performed. The project was over 3.5 MLOC written in different technologies (COBOL, C++, Java). The analysis demonstrated that maintenance cost could be reduced down to 30 percent [13]. If we consider the annual costs for maintenances of large scale systems, then we can have a reduction worth up to the range of millions of dollars.
In the next section we will consider a different set of tech-

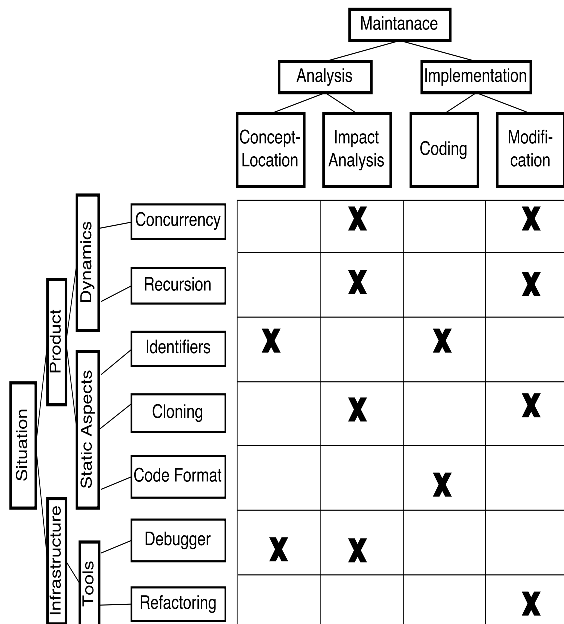


Figure 1: 2D Quality/Health model depicting maintenance effort.

niques that will be based upon the health models described here. Their goal is to keep the system stable and prevent failures before they occur.

4. SOFTWARE HEALTH MANAGEMENT

As we saw, software failures can happen easily, especially with the increased complexity. For some systems there is an acceptable failure rate, but safety critical systems zero error tolerance. The latter systems(aviation and automotive industry, hydraulics etc.) have to be prepared for any anomalies in advance and resolve them accordingly. Software Health Management (SWHM) is a modern discipline which extends the classical software-tolerance techniques, by investigating the tools and practices for automated anomaly detection, diagnosis prediction and mitigation of events due to software anomalies [5]. Similarly like the embedded safety-critical systems, SWHM for IS has the same goals. Here we lack the strictness imposed from the hardware components, such as maximum altitude, minimal speed of movement or acceptable pressure in a vault. Instead, we try to identify, isolate and mitigate the errors that occur when deviating from the health models in a system that deals with data-intensive applications. That does not mean that we lack standards - on the contrary, we have to develop them carefully and enter them in the SWHM framework. SWHM observes the entire system, not just a part of it. It needs to address the following characteristics [14] :

- continuously monitoring the software
- SWHM must be able to minimize the number of fault positives and fault negatives
- SWHM needs to be reliable
- must be integrated seamlessly with the traditional Validation and Verification techniques, but not to replace them.

To achieve this, there is vast number of SWHM techniques available Figure 2.

Here, they are analysed by the concepts:
- fault handling: how the technique deals with the fault. Originating from the software tolerant systems, it can perform prevention, removal of the error or tolerate it and allow small deviations.
- fault detection-isolation-recovery capability (FDIR): detection is the process of identifying the fault, isolation is the process of identifying the source of the error and isolating it from the rest. Recovery is the sequence of the actions taken in order to return the system to its normal functioning state.
- degree of automation: How much help from the engineer is needed.
- requirements of resources: mainly, here we focus on memory and CPU power i.e. how much computation is required for the SWHM.
- completeness: the ability to provide complete results; if that is not possible, it performs statistical estimation of the missing values).

The next section will investigate the model-based SWHM and explain the level of applicability for IS.

5. MODEL-BASED SWHM FOR INFORMATION SYSTEMS

Inspired from the ARINC component model [9], we propose a model-based approach for health management.
The approach consists of two-level hierarchy: component level with a local view (CLHM) on the problem and at a system level with the global view (SLHM) [5].

5.1 Component Level Health Management

The purpose of the CLHM is to detect the fault locally, take a suitable mitigation action and report to the high-level manager [4]. Regarding IS, the equivalence of a component is a software module with defined interfaces for communicating, pre-defined states and events. For the health status to be evaluated successfully, all the discrepancies are entered in a Monitor Specification Table. Here we define the abnormal conditions. For example, if the component is a data access object, a sample condition would be a double value which represents the time threshold to read the data from storage. The health of a component is not an isolated case: it can also depend from the status of another components. Suppose component A is in state A1, which is normal for it. Similarly, component B is in B1 normal state. For a component C which accepts as input the states from A and B, it can be in a state which is abnormal, if at the same time the components are in state A1 and B1 respectively.

To be able to describe these type of conditions, components need to communicate through their well-defined interfaces, so they can know other states. This interaction can be done in synchronous (using composition of objects) or asynchronous manner (using callbacks). Each component can be configured individually. The components are connected to monitor, which detects abnormal behaviour or violation in the constraints. For each component, a separate monitor is deployed, acting as an observable object and the connected component as an observer. When any monitor detects a discrepancy from its related monitor specification table, the status is reported to the Component Level Health Manager

Technique	Fault handling	FDIR	Automation	Resources	Completeness
Design and programming methodologies (development phase)					
Model-based design	fault prevention	N/A	N/A	N/A	N/A
Goal-based operations	fault prevention	N/A	N/A	N/A	N/A
Aspect-oriented programming	fault prevention	N/A	N/A	N/A	N/A
Recovery-based computing	fault prevention	N/A	N/A	N/A	N/A
Software configuration management	fault prevention	N/A	N/A	N/A	N/A
Verification and Validation (V&V) (testing phase)					
Testing	fault removal	N/A	manual, semi-automatic	adjustable	No
Simulation	fault removal	N/A	automatic	moderate-high	No
Debugging	fault removal	N/A	semi-automatic	varied	No
Numerical analysis	fault removal	N/A	manual	low	No
Model checking	fault removal	N/A	automatic	high	In some cases
Theorem proving	fault removal	N/A	automatic	high	In some cases
Runtime techniques (post-deployment phase)					
Redundancy-based fault tolerance	fault tolerance	isolation,recovery	automatic	varied	No
Checkpointing and rolling back	fault tolerance	recovery	automatic	varied	No
Runtime monitoring	fault tolerance	detection	automatic	minimal	No
Trace analysis	fault tolerance	detection	automatic	varied	No
Built-in tests	fault tolerance	detection	automatic	minimal	No
Software rejuvenation	fault tolerance	recovery	automatic	minimal	No
Computer immunology	fault tolerance	detection,isolation	automatic	usually minimal	No
Self-healing software	fault tolerance	detection,isolation,recovery	automatic	varied	No

Figure 2: Classification of SHM Techniques.

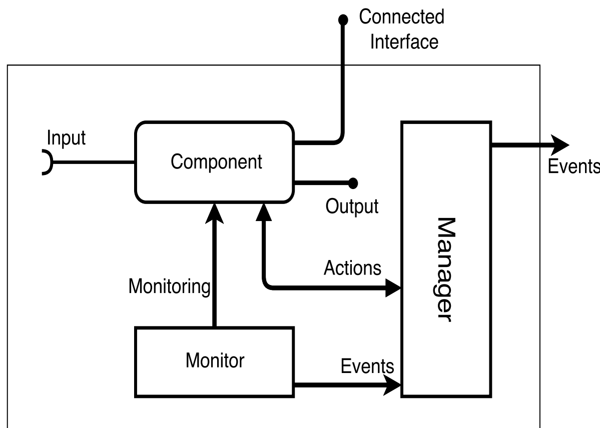


Figure 3: CLHM and monitoring a component

(CLHM). The role of the CLHM is to issue the appropriate mitigation action for that component. The mitigation actions can be of different nature. If the severity of the deviation is low, the IGNORE command is issued. If the component needs to be reverted to an original state, commands RESTART or REVERT are issued. If the severity is extremely high, then the operation must stop after issuing the STOP command. CLHM is responsible on a component level; however, if the mitigation cannot be executed there (for example, the appropriate mitigation action is unknown by manager), the call is propagated to the system level. The flow of the messages and the described organisation of components is depicted in Figure 3. We are not limited to the size of the components: the framework supports from very complex (e.g. a layer of the system) down to components with very narrowed functionality (e.g. parser). Each size comes with a trade-off: a large component will have many conditions in the monitor specification table and it will be harder to maintain, where as many small components will introduce an overhead in the system due to the increased communication between them.

5.2 System Level Health Management

The component level deals with only small part of the system. By definition, SWHM has to cover the entire system. The higher level system health management (SLHM) deals with the system as a whole. The input to this system are the discrepancy messages generated from the component managers. To identify the fault-source, an additional component Diagnosis Engine (Diagnoser) is introduced. The purpose of this component is to reason over the detected discrepancies and isolate the fault source. For diagnosis of the system, the Timed Failure Propagation Graphs (TFPG) model is used, as in the ARINC component model. This diagnose engine can be reused, due to the structure of the TFPG: the nodes represent the discrepancies (anomalies) and the directed edges describe the propagation of the failure effect. This allows the engine to detect and distinguish between the faults [1], and infer the actual deviation. The system level manager can respond with an appropriate mitigation action when needed. These actions are sent to a corresponding component, which is not necessarily the one who reported the deviation. The actions are similar as the ones issued my the component manager: IGNORE, RESTART, STOP, REVERT.

In addition, mitigation action can be issued to several component at once (several components need to be reverted). The depiction of the whole process is given in Figure 4. The existing solution can be extended with a reporting component: if the mitigation strategy fails, its job is to alert the development team to intervene manually.

5.3 Example

To understand the process of the health management, we will look at a simple scenario of a poorly indexed database. Component A tries to read the data, but the process takes a longer time than usual. The time threshold is entered in the

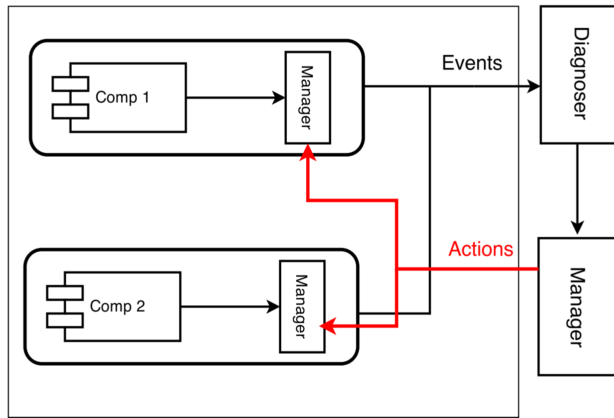


Figure 4: Event and action propagation SLHM.

monitor specification table as a deviation for that component. This triggers the monitor to report INVALID-DATA to the corresponding manager. The manager receives this event and sends a REVERT command to the component to take the default data. It also propagates the event to the system level manager. The component A accepts this and goes back into normal state.
However, component A also acts as a publisher to component B. It will report this state to component B via the pre-defined interface. The monitor specification table in B has an entry that doesn't allow for component A to have this case in specific scenario. So it sends INVALID-STATE-A signal to its manager, which in turn propagates the event to the system level manager. Now, at the system-level, we have a TFPG model with two nodes: INVALID-DATA -> INVALID-STATE-A. The diagnose engine can easily infer the cause of the problem and report to the developers the data access problems. This can be acted accordingly and prevent the decay of the system in early stages.
If we compare with the attributes from Figure 2, our framework fulfils the following criteria:
- Fault Handling: It prevents the faults. Regarding the mitigation strategies, they appear when the state enters fault mode; it is not able to do tolerance. Also there is no method to remove the fault.
- Fault Detection-Isolation-Recovery: It detects abnormal behaviour (not the error itself). This means that it cannot be isolated from the system. Recovery is also not implemented.
- Automation: Semi-automatic, it can be configured to alert the engineers when either an error cannot be reasoned or a mitigation strategy cannot be found.
- Resources: moderate to high. It depends from the code size and from the decision of the size of the components.
- Completeness: Not applicable, since we are not dealing with missing values here

Hierarchical two-level SWHM can be applied on information systems. The advantage of this approach is in its modularity. For a large-scale system, having one large complex block for health manager might only increase the effort of maintenance. Here, the encapsulation into components allows self-management without notifying the system level for minor deviations. This means that many monitors must be deployed, which results in additional resource consumption. Furthermore, the intensive monitoring and component-communication operations generate additional overhead in the system. For further research, other approaches listed from Figure 2 might be more suitable, especially Aspect oriented SWHM approach, due to the increased modularity.

6. CONCLUSION

Large-scale software systems need to have high-value of well-being. This is an important attribute due to the increased costs of maintenance. The degradation over time is inevitable, but SWHM can reduce this and rejuvenate the system (which in turn, means higher reliability and less maintenance from the human-side). We presented a modification to the original SWHM technique. Instead of inputs of signals, like gyroscope values or accelerometer readings, we re-structured the input to be suited for software health model typical for information systems: LOC, Maintainability Index and economic effectiveness and so on. SWHM requires well-established health model and creating one takes effort. It requires experience and in-depth knowledge of how the system works. Selecting the important metrics can be done through analysis and trial-error approach. Later on, the model will require tweaking and improvement to suit the software needs. Once having a model, our system is eligible for SWHM. Efforts are made into automating the detection and analysis of the faults [14]. We showed that implementing the Model Based SWHM can be an expensive step. Firstly, it requires a experienced developer, someone that can successfully design the components of the CLHM. The constant monitoring of the system has the additional need of resources. Adding the framework will sure increase the complexity of the system. These drawbacks come from the fact that SWHM is intended for safety-critical applications. To evaluate the success of the framework, the next step would be testing a prototype on a several different in size information system. Comparing the satisfactory level with the complexity of the system can show when this solution is most suitable. This paper focused on the technical factors that made the software act ill. In real environment, poor team communication, incomplete documentation and vague project description also contribute to aging. Future research proposal is to take into account the management factors as well when creating the health model. Separate problematic is to come up with measurement and corresponding model for these attributes, as their nature is rather different than maintainability. Detecting and preventing organizational issues or healing the requirement specification can further reduce the risk of reduced maintainability and lower the annual costs.

7. REFERENCES

[1] S. Abdelwahed and G. Karsai. Notions of diagnosability for timed failure propagation graphs. In *Autotestcon, 2006 IEEE*, pages 643–648. IEEE, 2006.
[2] D. Ash, J. Alderete, L. Yao, P. Oman, and B. Lowtber. Using software maintainability models to track code health. In *Software Maintenance, 1994. Proceedings., International Conference on*, pages 154–160, Sep 1994.

[3] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, Aug. 1994.

[4] A. Dubey, G. Karsai, and N. Mahadevan. Model-based software health management for real-time systems. In *Aerospace Conference, 2011 IEEE*, pages 1–18, March 2011.

[5] A. Dubey, N. Mahadevan, and G. Karsai. A deliberative reasoner for model-based software health management. In *The Eighth International Conference on Autonomic and Autonomous Systems*, St. Maarten, Netherlands Antilles, 03/2012 2012.

[6] M. Grottke, R. Matias, and K. Trivedi. The fundamentals of software aging. In *Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on*, pages 1–6, Nov 2008.

[7] C. Kaner and W. Bond. Software engineering metrics : What do they measure and how do we know ? *Direct*, 8:1–12, 2004.

[8] B. Kitchenham and S. L. Pfleeger. Software quality: The elusive target. *IEEE Softw.*, 13(1):12–21, Jan. 1996.

[9] N. Mahadevan, A. Dubey, and G. Karsai. Application of software health management techniques. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 1–10. ACM, 2011.

[10] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[11] K. Pipatsrisawat, A. Darwiche, O. J. Mengshoel, and J. Schumann. Software health management: A short review of challenges and existing techniques. In *Proc. of 1st International Workshop on Software Health Management (SHM 2009)*, Pasadena, CA, July 2009.

[12] M. Pizka and F. Deißenböck. How to effectively define and measure maintainability. *SMEF 2007*, page 93, 2007.

[13] M. Pizka and T. Panas. Establishing economic effectiveness through software health-management. Technical report, Lawrence Livermore National Laboratory (LLNL), Livermore, CA, 2009.

[14] J. Schumann, O. J. Mengshoel, A. N. Srivastava, and A. Darwiche. Towards software health management with bayesian networks. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 331–336, New York, NY, USA, 2010. ACM.

Adoption of emerging Architectural Approaches in German Software Companies

Jan Wittland
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
jan.wittland@rwth-aachen.de

Andreas Steffens
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
andreas.steffens@swc.rwth-aachen.de

ABSTRACT

Every once in a while, new architectural styles for developing, distributing and maintaining software come into the discussion of developer communities or conferences with promising arguments to make use of them. In the last decade this can be observed especially for service-oriented architectures. One of these new styles is the Microservices architectural style, which is described by Fowler [4].

Despite of the fact that new approaches arise, only a few is known about the reasons for a success or failure of them. In this study we try to identify the reasons from the perspective of software company employees. We conducted a survey with employees of German software companies and asked them about their attitude towards the adoption of new architectural styles and presented them two example architectures for a comparison.

The statistical results of the survey analysis revealed an overall positive attitude towards changes in the architecture, with a couple of limitations. Some potential problems have been identified regarding controversy opinions between employees and the company, in the question of the importance of certain architectural aspects. A proposal for the most important aspects of a new architecture has been derived.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; K.6.3 [**Management of Computing and Information Systems**]: Software Management

Keywords

Microservices, App Servers, Market Penetration, Survey

1. INTRODUCTION

The so-called *Microservices* architectural style, which is described by *Fowler* [4], is a frequent topic in the current discussion on software architectures and becomes more and more public especially in 2014 [9]. With the announcement of the *Microservices Conference in Berlin 2015* [7] it becomes even more clear that the approach gains a high attention. In addition, we already see implementations in modern companies such as *NetFlix* or *Soundcloud* [5, 1, 6, 2]. From a scientific perspective, more intense research on the topic seems to be required, since there only exist a few publications regarding the term Microservices.

With its modular approach, Microservices seem to be very promising for companies that want to move away from their heavy-weight monolithic architectures, which become more and more difficult to maintain. But what are the factors, which influence the successful adoption of a new architecture such as Microservices in companies in general?

Multiple potential problems may occur, if an established architecture is changed. We all know the saying "never change a running system", which of course is misleading in terms of being long-term competitive in the vastly growing software market. With the saying in mind, we assume that the success of a new architecture strongly depends on the readiness of the employees to invest time and brainpower into a change. This could be affected by the individual attitude towards new technologies and by the constraints given by the company. With this study, we want to get some insights on the thoughts and attitude of software company employees towards changing to a new architecture. An objective is to identify key requirements for a new architecture to be successfully adopted by companies.

In *Chapter 2*, we describe a potential problem of architectural change and discuss the idea behind Microservices. Afterwards we describe the research method used for the study in *Chapter 3*, where we give an insight on the questionnaire construction and the constraints of the data assessment, preparation and analysis. In *Chapter 4* we present the results of the study with descriptive statistics and significant differences between groups where applicable. The paper ends with a discussion of the results in *Chapter 5* and a short conclusion in *Chapter 6*.

2. BACKGROUND

2.1 Changing the Software Architecture

Compared to a building, changing the architecture of a software seems to be rather easy, since software architecture in essence is a collection of design decisions made during the evolution of the software. The shared understanding about the central components of the system and the interaction between them is kept among the people who are involved in this continuous process. Since software is immaterial, these

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

decisions may be revised later on if the decision makers are willing to [3].

As *Lehman's Law* states, there has to be a continuous change in a software system, because of the growing demands of the customers, which will in addition increase the complexity of the system over time [10, p. 532]. The required changes may also affect the architecture and therefore the developers have to be willing to accept major changes accompanied by investing time and brainpower into new technologies.

Considering this, the willingness and attitude of software company employees towards such changes has to be the key to success or failure of a new architectural style such as Microservices, which is discussed along this paper.

2.2 From a Monolith to Microservices

The idea behind conducting this study arose by the question: "Is the App Server dead - and are Microservices the future?". Since the question is very specific and bound to certain technologies it would be hard to answer it in general. Instead we decided to widen it to a more generic level of abstraction: *Monolithic Architectures* vs. *Modular Architectures*. In general, we want to know which approach is preferred by software company employees today and what the reasons for or against using them are.

As an example, we present the App Server as a monolithic and Microservices as a modular architecture to the participants of our survey. Of course we know that an App Server is not a monolith by default and a system can be built on it in a modular fashion, but it can be a monolith, if you think of the fact that you build your entire system with increasing dependencies on a single platform. As the software grows in functionality, it will more and more depend on the particular system, becomes monolithic and lacks of scalability and maintainability.

The key idea behind Microservices and its modular architecture is described by Fowler [4]. It differs from other modular approaches in a more fine grained decomposition of the software into very small functional units, which are called Microservices. Each service is independent deployable and exchangeable, runs in an own process with maybe different environments and can have its own separated databases. Microservices communicate with each other via agreed interfaces e.g. over HTTP/REST. One of the main benefits is that the system can be scaled individually by the affordances of a single service, compared to a monolith which only scales by replicating the whole monolith (see *Figure 1*). The overall maintainability of the system may increase, because each of the small services can be supervised by its development team along the whole life-cycle of the product.

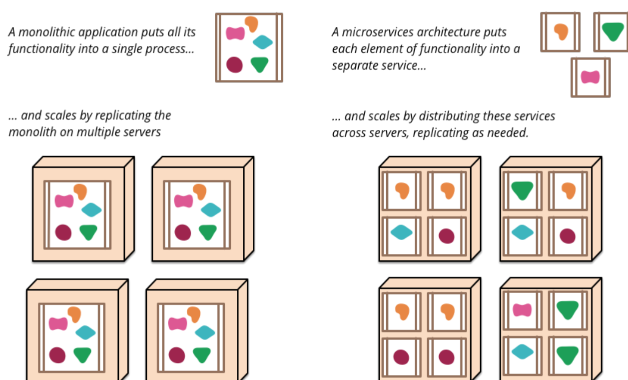


Figure 1: Monolith vs. Microservices [4]

Altogether, Microservices seem to be a promising architecture for companies that want to break up their monoliths. With our exploratory survey, we want to identify, how software company employees think about the two approaches. What is their general attitude towards changes and new technologies? Where do they see a future and what is important for them? Are there any conflicts between employee and company requirements? And finally, can the initial question be answered positively?

3. RESEARCH METHOD

3.1 Survey construction

To gain direct information from company employees we decided to construct an online-questionnaire. This form of survey was the means of choice to reach as much different companies in size, age and field of business as possible, because it is less intrusive and not very time-consuming.

The questionnaire should reveal the situation of the company the employee works for and the individual attitude towards new technologies or architectures. To be more illustrative, the questionnaire introduces the mentioned architecture examples, which will be individually rated and compared by the subject, to identify which approach is preferred in general.

These requirements resulted in 10 individual questionnaire pages. The first two pages are for introduction purposes and require the subject to read. The further pages are interactive and numbered from I to VIII. Most of the interactive blocks implement *Likert* scales with forced choice (even number of items) and represent ratio scales [11]. The default coding ranges from 0 (most negative) to 5 (most positive). The parts of the questionnaire can be described as the following:

Introduction: Welcome Page

Greeting, thank for participation, topic, privacy assurance.

Background: Two architecture samples

To clarify the topic, the opposing approaches of App Servers and Microservices are described with illustrations and a short text, to be referred to later on in some of the questions.

I. Common information

Common data of the subject such as age, gender, education and field of work are assessed and also information on the company, such as size, age, business sector, field of developments, team size and the current architecture state (monolithic/hybrid/modular). Most of the items can be used as factors for analysis, e.g. between start-ups and established companies or between younger and older participants.

II. Dealing with technology

The four items in this scale identify the general technical affinity of the subject as a factor, to analyze if technical affinity influences the other results. Since all participants had a very high technical affinity, this factor will be ignored for the analysis of the results ($\bar{x} = 4.5$, $s = 0.6$, $n = 30$).

III. Individual experience with software development

This page consists of two equivalent 8 item blocks, and asks the subjects for their individual experience in 8 areas of software development such as design, development, user support, etc. The first block asks for the *Perceived Ease of Use (PEOU)* (no experience (0), hard (1) - easy (4)) and the second block for the *Usage Frequency (UF)* (never (0), few times a year (1) to daily (4)). By combining PEOU and UF to and multiplicative index, we calculate an indicator for the general experience in software development. Example: if a subject states that designing is easy for him/her and performs the activity on a daily basis, we can assume that he/she is kind of an expert in designing. The overall experience can be expressed as what we name *Software Development Experience Index (SDEI)*, which can be used for analysis:

$$SDEI = \Big(\sum_{i=1}^{8}(PEOU_i \times UF_i)/4\Big)/8, SDEI \in [0,4] \quad (1)$$

Lower values express expertise on a narrow level (e.g. only one area) and higher values on a wider level (e.g. multiple areas).

IV. Individual attitude towards integrating technologies

This block consists of 8 items on the individual attitude towards new technologies and the learning impact of work. The subject has to rate on the default scale if he/she agrees or disagrees to the statements such as: "I think it is better to stick to established approaches instead of testing new ones.".

V. Requirements for using an architecture

This page consists of two similar blocks to rate the importance of 10 categories for a possible commitment to a new architectural approach. Categories are e.g. learnability, maintainability, scalability and costs. The first block asks the subject to sort the categories by importance from the company perspective and the second block asks for the same categories but from an individual perspective. For the individual perspective the subject can assign the same priorities, where for the company perspective a strict hierarchy is built, which enables conflict detection between both perspectives.

VI. Individual attitude towards monolithic and modular approaches

This page contains two equivalent 8 item blocks on the individual attitude towards monolithic and modular approaches. The subject rates if he/she agrees or disagrees to the statements, which enables to see how the subject rates the two approaches individually. Example: "I think that this approach will have no future in the market."

VII. Comparison of the two architectures

This question block contains 8 statements on a direct comparison of the two approaches. The subject has to assign a position to each of the statements ranging from monolithic (-3) over neutral (0) to modular (+3), which enables to see which approach is preferred in a direct comparison. Example: "I think this approach is more flexible towards changes."

VIII. Final question

The last page comes with the *Net-Promoter Score (NPS)* as a final question. This instrument was introduced by *Reichheld* [8] and tries to forecast the market success of a product by rating if the subject would recommend the product to a colleague or friend. The possible answer ranges from "Never" (0) to "Absolutely" (10) and partitions the subjects into three clusters among *Promoters* (9-10), *Passively Satisfied* (7-8) and *Detractors* (0-6). In this case we ask for Microservices, to estimate their overall chance of success.

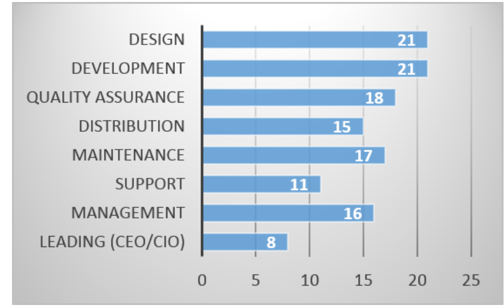


Figure 2: Employee assigned working areas ($n = 30$)

3.2 Data assessment

The online-questionnaire survey period ranges from November to December 2014. The required time to fill the questionnaire ranges from 10 to 15 minutes and varies slightly depending on reading skills and experience. For convenience, the questionnaire is provided in German and English.

The participants were selected in different ways. Personal contacts to some companies were used to pass the questionnaire to the employees. On the other hand, we acquired companies via email requests.

3.3 Data preparation

The data from the online-questionnaire was imported to the *SPSS* statistics software. For statistical tests such as F-Test for 1-factorial variance analysis we determined the significance level of $\alpha = 0.05$ as significant and $\alpha = 0.01$ as very significant. For the data analysis, descriptive statistics such as min, max, mean ($\bar{x}$) and standard deviation (s) are used. In addition, mean comparisons were made between groups using SPSS ANOVA with the calculated factors.

3.4 Description of the sample

During the assessment period $n = 30$ software company employees took part in the survey. There were 27 male and 3 female participants with an average age of $\bar{x} = 35.4$ ($s = 8.7$) years. The youngest subject is of 22 years and the oldest of 57 years, so a wide age range has been covered. Nevertheless, no significant impact on the results was attributed to the age factor. The majority of the subjects with 77% has a university or polytechnic degree.

The average number of years the subjects work for a software company is $\bar{x} = 9.8$ ($s = 7.0$) with a minimum of 2 and a maximum of 34 years. We created three groups of equal distribution with: ≤ 5 years (Beginners), 6-9 years (Established) and 11+ years (Experienced). The subjects work in a wide range of working areas, most of them in design or development (70.0%), as can be seen in *Figure 2.*

For the business sector of the companies, we also got a wide distribution with 10 of 19 provided areas. Most employees come from the Human Health/Social Work (26.7%) and the Information/Communication sector (16.7%). The companies itself develop software for a variety of platforms,

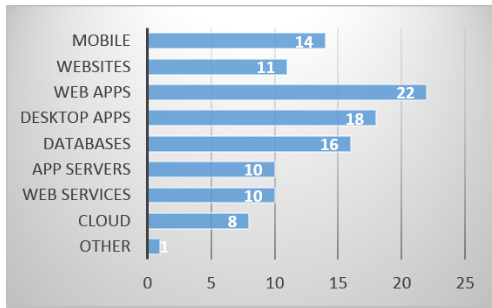


Figure 3: Company development platforms ($n = 30$)

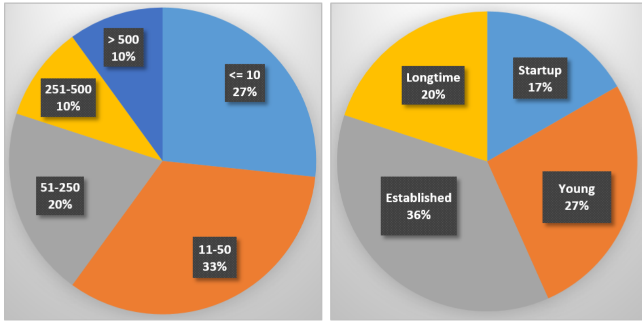


Figure 4: Company size (left) / age (right) ($n = 30$)

as can be seen in *Figure 3*. Most prominent are Web Apps (73.3%) and Desktop Apps (60.0%).

We also asked the subjects for the company's age in years and size regarding the number of employees (see *Figure 4*). For the age we classified the companies into Startups (< 5), Young (5-10), Established (10-25) and Longtime companies (> 25). The majority of the subjects works for companies of Small (27.0%) or Medium size (33.0%) and most of the companies are Young (27.0%) or Established (27.0%).

The majority of the subjects states that the company they are working for realizes monolithic (56.7%) or hybrid architectures (33.3.%). Only a small group (23.3%) states that modular architectures are implemented. The team size in the companies varies from 1 to 15, with an average of $\bar{x} = 6.6$ ($s = 4.2$). We have built three groups for further analysis: Small (≤ 4), Average (5-8) and Large (9+) teams. Most of the subjects work in small (36.7%) or average teams (40.7%).

4. RESULTS

III. Individual experience with certain working areas of software development

With the items from this section, we calculated the *SDEI* index introduced in chapter 3.1. The calculation resulted in an average of $\bar{x} = 1.6$ ($s = 0.6$, $n = 30$) for our sample. Therefore we can state that most of the subjects have experiences in multiple fields of work. We calculated three groups for further comparison with the SDEI factor: ≤ 1.00, $1.01 - 2.00$ and 2.01+. With 56% most of the subjects belong to the medium group (cf. *Figure 5*). The lowest group expresses experiences in a narrow field of work (e.g. only development) and the highest group expresses knowledge in multiple fields (e.g. design + development + ...).

We did not include a role-based analysis of the employee,

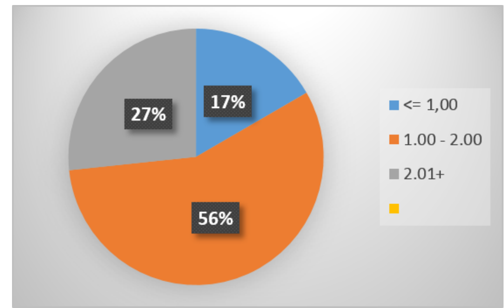


Figure 5: SDEI groups of the subjects ($n = 30$)

since this would exceed the capacity of this paper. Why? The average number of different working areas by subject, which resulted from section I, is $\bar{x} = 4.2$ ($s = 2.1$) - a very high value. Most of the subjects even work in 5 to 6 working areas (35.6%). This massive divergence makes it difficult to classify the subjects into strict groups like developers or designers. The average SDEI is much lower compared to the number of working areas and therefore seems to be more realistic, so we will chose the SDEI groups as a role factor for our analysis. In further research, a question for the main working area could simplify a role-dependent analysis.

IV. Individual attitude towards the integration of new technologies

Question	$\bar{x}$	s	min	max
1. I like to test new approaches to fulfill my tasks.	4.1	1.0	2	5
2. I think it is better to stick to established approaches.	1.7	0.7	0	3
3. It is very difficult for me to learn a new approach.	1.1	0.6	0	2
4. I have no time to address new approaches.	2.1	1.0	0	4
5. I can learn at work from a technical point of view.	3.8	1.1	1	5
6. I like to educate myself after work to learn sth. new.	3.9	1.2	1	5
7. I would like to attend more advanced training.	4.0	0.9	2	5
8. I use already learned technologies at work most of time.	3.2	1.0	1	5

Table 1: Results of individual attitude towards integrating and learning new technologies ($n = 30$)

In section IV we asked the subjects for their attitude towards integrating and learning new technologies. You find the results per item in *Table 1*. The subjects like to test new approaches and negate having difficulties with learning. They also think that it is rather bad to stick to established processes and confirm that they would like to attend more advanced training to improve their knowledge.

Significant differences were identified for the SDEI on item 8. The subjects with a low SDEI have an average of $\bar{x} = 4.0$ ($s = 0.7$), where subjects with a high SDEI result in $\bar{x} = 2.6$ ($s = 1.1$) - a neutral position ($n = 29$, $p = 0.05$, $F = 3.45$). The higher value for the low SDEI group expresses that people, who only work on a certain field, most of the time use existing technologies. This could mean that working in different fields comes with using new technologies.

V. Affordances for integrating new architectures

In section V we asked the subjects to rate 10 different categories by importance regarding the implementation of a new architecture. In the first task the subjects should sort the categories by importance from rank 1 (most important) to 10 (least important) from the company perspective. Since the items are not easy to analyze because of the large number of possible assignments, we only present the three top and flop ranks with the three most positioned categories on the related ranks in *Table 2*. The most important categories therefore are low costs, minimal migration effort and flexibility. Low cost has 15 votes in the top 3x3 and flexibility 14 votes. Learnability also seems to be important in the top 3 with 12 votes. Least important is the currentness of the architecture with 22 votes in the flop ranks.

Rank	1st most	2nd most	3rd most
1	Low costs	Flexibility	Learnability
2	Migration	Maintainability	Learnability
3	Flexibility	Low costs	Learnability
8	Currentness	Collaboration	Sustainability
9	Currentness	Collaboration	Scalability
10	Currentness	Deployment	Sustainability

Table 2: Categories for top and flop ranks with 1st, 2nd and 3rd most votes for the related rank ($n = 29$)

In the second task the subjects should rate the categories from an individual perspective. In *Table 3* the average values are listed in descending order. The most important facts are that flexibility, sustainability and maintainability seem to be most important from the individual perspective, whereas low cost is the most unimportant category - which is the opposite from the company perspective for costs and sustainability. Flexibility seems to be important from both directions.

Category	n	min	max	$\bar{x}$	s
High flexibiltiy	28	2	5	4.3	0.8
Maintainability	28	2	5	4.3	0.9
Sustainability	28	2	5	4.0	1.0
High scalability	28	1	5	3.4	1.3
Learnability	28	1	5	3.3	1.2
Collaboration	27	2	5	3.2	1.1
Currentness	28	1	5	3.0	1.3
Simple deployment	28	0	5	2.9	1.2
Migration effort	28	0	5	2.9	1.4
Low costs	28	0	5	2.1	1.4

Table 3: Categories rated individually by importance (0=least, 5=most) ($n = 27$)

We found a significant difference for the rating of scalability among beginners and established workers. Beginners therefore rate scalability on average with $\bar{x} = 2.5$ ($s = 1.0$), whereas established workers rate it with $\bar{x} = 4.2$ ($s = 1.3$, $n = 27$, $p = 0.02$, $F = 4.64$). So scalability seems to be more important to established workers than beginners, which would make perfect sense because the long-time work on a growing software project may increase the demand for scalability.

VI. Individual attitude towards monolithic and modular approaches

Section VI contains two equivalent 8 item blocks for an individual rating of monolithic and modular approaches. The average results are shown in *Table 4*. Item 1 shows that the subjects see a more probable future in the market for modular than for monolith architectures. Item 3 got the overall highest rating, even if we know from section V that deployment does not play an important role. The subjects see significant advantages in the modular approach and would prefer to develop with it (cf. item 5 and 6).

Question	Monol. $\bar{x}$	Monol. s	Modul. $\bar{x}$	Modul. s
1. I think that this approach will have no future in the market.	2.3	1.2	1.2	1.3
2. I already made satisfying experiences with such approaches.	3.0	1.1	2.5	1.5
3. I think this approach comes with flexible deployment opportunities.	1.9	1.1	4.0	0.7
4. I like this approach because it fits to our business solution.	1.8	1.2	3.0	1.4
5. I can not see significant advantages in this approach.	2.6	1.2	1.1	1.2
6. I would prefer to develop software with this approach.	1.4	0.9	3.5	1.2
7. I think this approach is easier to understand for new employees.	2.5	1.2	2.9	1.2
8. I think this approach simplifies the collaboration in the team.	1.6	0.9	3.4	1.1

Table 4: Results of attitude towards monolithic (left) and modular (right) approaches ($n = 28$)

There is a significant difference for item 1 among beginners and experienced developers. Beginners do not see a strong future in the market for monoliths ($\bar{x} = 1.9$, $s = 1.3$), where long time experienced subjects are still optimistic ($\bar{x} = 3.3$, $s = 1.3$, $p = 0.04$, $F = 3.74$, $n = 27$). Another interesting difference was found regarding the team size and item 4. Large teams state that the modular approach does not fit their business ($\bar{x} = 1.8$, $s = 1.5$), whereas average teams confirm it ($\bar{x} = 3.8$, $s = 1.3$, $p = 0.02$, $F = 5.02$, $n = 26$).

VII. Comparison of both approaches

In section VII we asked to assign statements to either a monolithic (-3) or a modular (+3) approach. The results are shown in *Table 5*. Despite of item 4, all advantages are attributed to the modular approach. The most positive answers were given for the scalability and flexibility.

There was a very significant difference for the SDEI, where subjects with a low SDEI see a promising future for modular approaches $\bar{x} = 2.8$ ($s = 0.4$) and medium SDEI subjects tend to neutrality ($\bar{x} = 1.0$, $s = 1.2$, $p = 0.01$, $F = 6.79$).

VIII. Final Question

Finally, we asked for the Net-Promoter Score of Microservices. It results in an average of $\bar{x} = 7.6$ ($s = 1.9$, $n = 28$), which is quite good, but did not reach the promoter state. Instead it represents that the subjects are passively satisfied. The original score is the relative amount of subtracting detractors from promoters. There were 10 promoters and 8 detractors, which results in 7.1% - only a small chance for the market success of Microservices, if you believe the NPS.

Question	$\bar{x}$	s	min	max
1. I think that this approach is more scalable.	2.1	1.3	-2	3
2. I think that the development is easier with this approach.	0.6	1.6	-2	3
3. I think that this approach is more flexible towards changes.	2.3	1.0	0	3
4. I think that this approach is harder to maintain.	-0.5	1.7	-3	3
5. I think that this approach is more promising for the future.	1.6	1.3	-2	3
6. I think that this approach simplifies work in teams.	1.3	1.3	-1	3
7. I think that this approach can save costs over time.	1.3	1.6	-2	3
8. If I had a choice I would rather develop under this approach.	1.6	1.3	-1	3

Table 5: Results of direct comparison of monolith and modular approach ($n = 28$)

5. DISCUSSION

The survey revealed very interesting facts on our topic, regardless of the relatively small sample.

We have seen an overall positive attitude towards architectural change, but certain requirements could also be identified. The subjects showed a broad willingness towards architectural change and state that learning new approaches is not an obstacle. Most of the participants also want more advanced training, which can be interpreted as a key requirement for a successful architecture: the employees have to be involved and trained equally.

Several potential conflicts for architectural change were identified in the comparison of category importance. The costs seem to be most relevant for companies, which is the opposite for employees. This may sound logical, but in fact it is a problem, if employees desire a change, but the company does not grant financial support. The currentness of the architecture does not seem to play a role at all, which may explain why companies stick to established structures. Instead, the most important category for both is the flexibility. So a new architecture in ideal would have: low costs, high flexibility, good maintainability and an easy migration. To increase acceptance among employees it should be easy to learn and sustainable.

The comparison between the monolithic and modular approaches showed that modular approaches are preferred in general today and a promising future is attributed to them. We also saw that monolithic approaches are still appealing if they are applicable for the situation. Even if the results of the direct comparison were very positive we cannot conclude that monoliths are going to die in the next years.

The new architectural style of Microservices, which was the activator for this study, got good ratings in the comparisons and also in the final question. Nevertheless, there also were a lot of distractors regarding the NPS, which stresses, that Microservices will not heal the world from all problems. So in the end, we can preliminary answer the initial question of the study as: nope, the App Server is not dead and hopefully will survive for some years and continue to be a reliable solution for many software companies.

To consolidate the results, further research is required. The study could be repeated on a larger scale with more preparation time, to gather more participants. The validity and reliability of the results cannot be guaranteed to be the same with another sample, since this was a pure exploratory study without any strict assumptions and conditions.

Criticism can be applied to the selection of the App Server as a monolith and Mircoservices as a modular approach. For further research, more evident examples should be chosen. Some participants argued that App Servers are also modular, which is not wrong as stated in the background chapter, because an App Server of course can be modular, too.

6. CONCLUSIONS

Altogether, we have seen an interesting and challenging study on the current distribution of monolithic and modular architectures in German software companies with several interesting results and a good potential for further research.

Future work could be applied in form of broader surveys or expert interviews with experienced software company employees, to verify the presented results.

7. ACKNOWLEDGMENTS

We thank all the participants very much for their readiness to spend some rare time on contributing to our survey!

8. REFERENCES

[1] P. Calçado. Building Products at SoundCloud —Part I-III. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith, 2014. Retrieved December 1, 2014.

[2] B. Christensen. Optimizing the Netflix API. http://techblog.netflix.com/2013/01/optimizing-netflix-api.html, 2013. Retrieved December 1, 2014.

[3] M. Fowler. Design - who needs an architect? *Software, IEEE*, 20(5):11–13, September 2003.

[4] M. Fowler and J. Lewis. Microservices. http://martinfowler.com/articles/microservices.html, 2014. Retrieved December 1, 2014.

[5] D. Glozic. SoundCloud is Reading My Mind. http://dejanglozic.com/2014/06/16/, 2014. Retrieved December 1, 2014.

[6] D. Jacobson. Embracing the Differences : Inside the Netflix API Redesign. http://techblog.netflix.com/2012/07/embracing-differences-inside-netflix.html, 2012. Retrieved December 1, 2014.

[7] microxchg2015. The Microservices Conference in Berlin. http://microxchg.io, 2015. Retrieved December 1, 2014.

[8] F. F. Reichheld. The one number you need to grow. *Harvard Business Review*, 81(12):46–54, December 2003.

[9] C. Richardson. Microservices: Decomposing Applications for Deployability and Scalability. http://www.infoq.com/articles/microservices-intro, 2014. Retrieved December 1, 2014.

[10] I. Summerville. *Software Engineering.* Pearson Studium, Munich, 2007.

[11] W. M. Trochim. Likert Scaling. http://www.socialresearchmethods.net/kb/scallik.php, 2006. Retrieved December 1, 2014.

Applicability of Test-driven development in the industry

Reyhaneh Yazdani
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
reyhaneh.yazdani@rwth-aachen.de

Horst Lichter
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lichter@swc.rwth-aachen.de

ABSTRACT

Test Driven Development (TDD) is a software development technique which tests are written incrementally before codes in the development process. Several empirical evaluations in different environments are done in order to evaluate the offered benefits and detect the downsides. Each evaluation provides results based on the existing test conditions, which lead to pros and cons opinions about TDD.

The objective of this paper is to first assess the benefits and downsides of TDD, and then prioritize them in such a way that guide a company to decide about TDD. In another view, using TDD makes a favor or an obstacle that needs hard progress to be resolved. As well, if TDD meets the company's development criteria, what should be done to apply TDD successfully in the industry?

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Test-Driven Development, TDD benefits, TDD downsides

1. INTRODUCTION

Test driven development (TDD) is an agile software development style derived from Extreme Programming (XP) [8]. According to the Scott W. Ambler, Test Driven Development (TDD) is an evolutionary approach to software development that combines the idea of writing automated tests before developing the original code and refactoring [1]. Refactoring is improving the structure of an existing body of code without changing its external behavior [11].

Using tests for ensuring the quality of software is not a new idea, but the brand new point of this approach is the importance of tests and how a development is started by writing tests. In the light of this view, designing tests are extended from being a development tool to an approach that all sectors of development, including analysis, design, and implementation, are also affected.

In the traditional development approaches with comprehensive planning, detailed documentation and expansive design [3], developers write tests after code software. The same programmers or testers may write unit tests, while in TDD, programmers write tests before the code. Traditional techniques are started with the analysis of the requirements, and then continued by designing phase. In the implementation, coding is started in several modules and units. Later in integration and test phase, all those units are integrated into a unique system and tested for verification and validation purposes. Also, refactoring occasionally occurs during the test phase when programmers address the detected software defect [14].

The process steps of TDD are the following [5] [14]:

1. Pick a piece of functional requirement
2. Write an automated test for that piece
3. Execute the automated test to make sure the new test fails
4. Write implementation code and repeat step 3 as long as the execution of automated test fails
5. Refactoring of existing code when test is executed successfully
6. Run all tests to make sure that refactoring did not change the external behavior
7. Repeat the whole process by going back to step1 and implementing other requirements.

There are several TDD attributes [4]. In the first place, TDD is test oriented. It starts by writing just enough tests for a specific functionality before coding that. In another view, the first coding task is the planning and writing automated (unit) tests that would determine whether the functional requirement is met [4].

The next attribute of TDD is continuing the development process in the incremental and iterative way. Developers add tests gradually during the development process [4].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

Another characteristic of TDD is the automation of the unit tests using frameworks such as xUnit. This means the frequent regression testing, an integral part of TDD, is automated [4].

The last is refactoring. Refactoring is part of *how we do things* in TDD but was not emphasized to the same degree in the traditional test-last experiences [4].

Recently, various empirical investigations are done to find the benefits and constraints of using TDD in comparison to the traditional approach. These are done in different industrial and academic environments which made several proponents and critics of TDD.

Although TDD is used and examined for decades, recently a keynote from David Heinemeier published on his blog, expressed his dissatisfaction with TDD, led to a sequence of conversations in a theme of *Is TDD dead?* [16]

In this paper, we present the benefits and downsides of TDD, which are determined by several investigation of evaluating Test Driven Development approach.

We organize this paper as follows. Section 2 presents advantages and disadvantages of TDD, followed by the presentation of limiting factors in applying TDD and offered solutions for the introduced limitation in section 3, and section 4 comes up with a conclusion.

2. THE ADVANTAGES AND DISADVANTAGES OF TDD

Doing various empirical investigations to find out the effect of TDD in development cycles and compare the results in order to get the general findings, is not an apt approach. Each industrial environment has its own specific characteristics, such as level of developer's experiences in writing tests and knowledge of the Test-driven approach. In addition, time and financial constraints put pressure on the developers and manager, which affect the project's outputs and manager's decisions. These are the kind of factors that can influence the project. Moreover, several empirical studies are done in controlled and isolated environments, while in reality there are some variables that may change the results.

Further, a lot of experiments and case studies were devised in different conditions to analyze the advantages and disadvantages of TDD. Varied from projects in an academic or industrial environments to being conducted by students or professional developers or mixed of them. Most of them had developed small projects and rarely real and large ones. Even though this software development method has been introduced more than ten years ago, there is still doubt regarding all benefits, which this approach claims posses [5]. By considering the fact that existing studies produced contradictory results, we are going to present the importance of those findings as advantages and disadvantages of TDD.

2.1 Advantages of TDD

1. Increased External Code Quality
   External code quality is measured using a couple of different metrics. In experiment settings, external quality means number of passed acceptance tests. In the case studies, external quality usually means number of defects found before release or defects reported by customers [13]. Most of the findings show that TDD increases the external quality, whereas there are some results that present no difference in external quality of test-first and test-last approach. TDD discipline put developers to write simple and clean source codes, while without the discipline it is easier to work less and write messy codes. Therefore, by following the discipline, we will get codes with higher quality [2].

2. Simple Design
   As design is done while tests are written, developers add classes and methods if they are required. This leads to a decrease in the complexity of the design and makes it simple. Also, developers can adapt to the changes and new features during implementing and maintenance phases [14] because of the easily understandable simple design.

3. Increased Application Quality
   Due to more testing during the implementation, software will have fewer defects in the released version. Therefore, reliability of the application is increased. As well, clients will also get the expected values more than when developing the application in traditional methods [4]. Buchan, Li and G.MacDonell [4] made a comparison between different results of various experiences regarding the quality of the application. The notable finding is that, most of the studies that students had conducted, didn't present any improvement in the quality of application. However in significant industrial case studies, they show improvement in the quality of application.

4. Improve Overall Productivity
   Although TDD increases the time on coding and developing tests on early stages, as the development progressed, writing new functionality needs less work in comparison to the test-last method [4]. Also, in TDD, errors and bugs are identified and removed in early phases, because developers remember more about the developed code than later in development life cycles. One of the interviewees in [4] described this matter as *"It is a lot cheaper in terms of resources to fix the issue immediately rather than months down the track when they may be discovered".* In addition, if the project is implemented in accordance with the TDD rules, it will be more productive than the test-last method in maintenance phase, as later in maintenance, source codes are easier to understand by others who did not write the codes.

5. Increased test coverage
   Higher test density and test coverage were also perceived to be encouraged by the use of TDD practices [4]. Due to the characteristic of the TDD, first test should be developed followed by the corresponding codes. Here, it is more probable that all tests are covered in comparison to the test-last approaches. In the test-last approaches, it is common to left some tests out because of time constraints, or it is unlikely to write tests for the parts that passed the acceptance tests of client or system tester.

6. Best understanding of the project's requirements
   Test Driven Development encourages better understanding of requirements by spending more time in develop-

ment analyzing scenarios, business requirements and more contact with the client [4]. If developers face uncertainly about a piece of the requirements during writing tests, they can ask client for clarification and more explanations about that part. In another view, the extra effort invested into understanding what the functional code had to do before writing it, resulted in a clear idea of the objects and methods required in the functional code [4].

7. Increasing developer and client confidence
One of the features of TDD is that developer can perceive the requirements more clearly. This made it possible for the developers to get more confidence as they implement what client exactly wants and the uncertainty around software releases is reduced [4]. This leads to more job satisfaction for developers. In addition, clients have more contact with the developers during the project development. This makes clients more satisfied and confident that what they get is in accordance with what they require.

2.2 Disadvantages of TDD

1. Decreased Productivity
Experiments and case studies used varied metrics for productivity. The used metric may be total development effort, lines of code per hour or number of implemented user stories [13]. Thus, it is hard to get a determined result. By considering the variety of metrics and test conditions, most of studies found out that TDD used more time in testing and coding. However, several studies report on less productivity, but at the same time they report improved external quality [13].

2. No up-front design
Proponents of TDD believe that applying TDD makes software design more simple with high quality [4]. This works properly only for a well-written and understandable source code. This means when bugs and defects are found in the maintenance phase, there is no formal design and documentation available [18] to help programmers to understand or remind the architecture and design of the software.

3. More maintenance
In maintenance phase of general software development approaches, detected bugs are removed or some enhancements and customization may be done. In a Test driven development method, these activities need to be done on test cases too. Therefore, more tasks should be done in TDD maintenance in comparison to the traditional approaches. So, TDD leads to higher maintenance costs, which makes it unattractive for the managers.

4. No ease of learning
In contrast to the findings of the research question that Kumar and Bansal presented about the ease of learning TDD approach that shows neutral view about it [14], David Tchepak has an alternative opinion. He posted an article on his blog as a TDD coacher about the difficulty of learning TDD in an effective way [19]. He minded that it is related to the wrong idea about TDD. Most of the training resources expressed TDD as a perfect way of coding and who does not apply TDD, develops complex code with low quality. The unrealistic expectations can cause people to become overly focused on the process, without understanding the rationale behind it [19]. This results in developers applying TDD in an ineffective manner [19]. In addition, it is hard to practice TDD. It has simple rules but the learner needs to have comprehensive knowledge about OO design, patterns, SOLID, DRY, etc. [19] in order to apply TDD effectively and with success. Moreover, changing the mindset and thinking model in software development from traditional to Test-driven development is challenging [17]. Most software developers have learned traditional approaches and Test-driven development is a complete different method.

3. APPLYING TEST-DRIVEN DEVELOPMENT

Based on the systematic review on 48 papers that Adnan, Daniel and Sasikummer have made [6], some limiting factors in TDD adoption are identified. Their report provides an overview about TDD and helps us to understand what should be done to apply TDD successfully in the industry. In this section, we present the limitations in applying TDD and later, propose solutions to deal with them.

3.1 Limiting factors in adopting TDD

- LF1: Long Development time
Development Time can be a critical factor for customer and organization, since by long development time, the cost of the project will be raised and customer will not be satisfied completely. Most of the experiences that have been done on evaluating TDD, have declared that in comparison to traditional development, TDD needs more time for developing. However, others claimed since the time for testing phase will be decreased in the TDD approach, the overall time in TDD is less than traditional method. How much this factor is critical, depends on the organization and its maturity [6].

- LF2: Insufficient TDD knowledge and experience
Based on the report [6], lack of the knowledge of TDD and experience in using TDD, are factors that could make problems in applying TDD in the industry. However, this subject can be applied on other approaches and techniques too. Even in developing software with traditional approach, it is expected that developers have enough knowledge and experience in them. This is a common factor that can be applied on every method. However, in comparison to the traditional approaches, the amount of the developers who have enough knowledge about the TDD and writing efficient tests, are less than test-last methods.

- LF3: Insufficient developer testing skills
TDD is a method based on test cases, in which developers use tests to guide the design of the system under development [8]. Therefore, it is essential that developers have enough testing skills to produce suitable test cases. Moreover, in traditional approach, the system can be designed by the designer and implemented by the programmer and tested by tester, while in TDD developers designed the system by implementing test cases and then corresponding source codes.

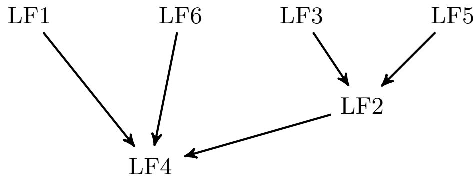


Figure 1: Causal relationships among limiting factors

Thus, TDD-developers need to have more knowledge than test-last developers.

- LF4: Insufficient adherence to the TDD protocol
  Each development technique has several steps that developers should observe them. In TDD, test cases are written, and then tested to examine their failure and after that, corresponding source code is implemented. Based on the findings of the review [6], most of the industrial case studies informed that developers were not adherence to the TDD protocol, which may lead to the low quality.

- LF5: Domain and tool specific limitations
  In a TDD project, developer needs a unit testing framework (xUnit tool) for writing test cases and a test runner to run the written tests. However, the type of the project is the matter. One of the common critics about the TDD is that it does not suit for GUI projects, as it is tough and effortful to perform automated GUI testing. The findings of the most investigation [6] on TDD shows that most of the developers have a problem in using TDD on a GUI projects.

- LF6: Lack of accepting the benefits of TDD by project team
  As TDD is not usually used in the industry, when an organization decides to apply it, its member's mindset about the TDD should be prepared. If some of the members are skeptical about the TDD usages, they can affect on the process of development and make tension in the team.

We also have identified causal relations among the limiting factors of adapting TDD in the industry. All of the causal relations lead to the same limiting factor *Insufficient adherence to the TDD protocol*. Figure 1 shows the causal relations and they are explained as follows.

- Long development time in TDD (LF1) might lead to the situation that developers do not perform TDD protocol completely (LF4). As the manager may force them to finish the project earlier, developers would not follow the TDD protocol.

- It is possible that some of the developers do not recognize the TDD's advantages (LF6). Therefore, they do not follow routine's steps of TDD (LF4).

- TDD is an approach which is based on testing. That means developers who want to obtain good experiences in TDD, are required to have testing knowledge (LF3). Thus, missing testing skills could lead to achieving less experiences in TDD (LF2), which provides a situation that developers do not perform TDD perfectly well (LF4).

- In the development based on TDD, tools and frameworks are needed. The lack of TDD-tools for some specific domains (LF5) lead to developers obtain not enough knowledge and experience in those domains (LF2). It could lead to difficulty for developers applying TDD, therefore they would not adherence to it (LF4).

Table 1: Limiting Factors and Recommended Solutions

Limiting Factor	Recommended Solution
Long development time	Select optimal approach based on received benefits
Insufficient TDD knowledge/experience	Setting lectures and lab material in TDD, training developers on site
Insufficient developer testing skills	Observe quality of written test cases
Insufficient adherence to TDD protocol	make discipline Monitor testing procedure, adapt developer's mindset
Domain and tool specific limitations	Use introduced techniques in articles, market produce and develop desired tool
Developer's skeptical about TDD	Remind developers TDD's benefits - feedback system

3.2 Solutions to limiting factors

In order to apply TDD successfully in an organization, we should remove the limiting factors. However, we can still apply TDD in the industry by accepting some of the constraints, but it is less probable to gain all TDD benefits. In this subsection, we are going to propose potential solutions for the mentioned limiting factors. The important point is that we have ordered them based on the priority. That means limiting factors with high priority have more effects on the TDD success. The rationale behind the offering priorities is the causal relations between the limiting factors that are introduced in subsection 3.1. Table 1 shows the solutions briefly.

- Priority 1: Long development time
  It should be clear from the beginning of the TDD-project that development time of the project is not only implementation time, but also writing tests, and refactoring. In this part, the manager role is important. The manager should consider this matter that implementation time in TDD is more than traditional development and should not pressure to deliver that code quickly. It does not mean that TDD is not suitable for the project that has a restrict time limitation. As a successful project that Roberto has described [15] and time was a restrict constraint, the team finished the project by using TDD on time.

- Priority 2: Domain and tool specific limitations
  Although more developers have the opinion that TDD is not suitable for GUI projects because the GUI code was not covered by any automated unit tests [12], recent articles like [9] and [10] introduced techniques that allow GUI test cases written and tested simply in TDD. However, it needs more effort to develop standards and official tools that support TDD in GUI type of projects. This is a real hard progress that is not easy to resolve, thus by considering the existing tools, TDD does not completely fit in project with GUI domain.

- Priority 3: Lack of acceptance the benefits of TDD by project team
  The best way to encourage developers to use TDD is to understand the reasons of their unwillingness to apply TDD. However, we can apply the following methods. During the development process, the outcome from TDD can be monitored and recorded, and in several time slots, expressed to the developers to remind them the benefits of their work. Moreover, if the project team consists of several groups, it is better to assign a TDD-experienced as a group leader to motivate members and believe on TDD. In addition, we should let other developers present their opinions about how TDD affects on the project. For this, we can provide a feedback system. Another way for increasing TDD acceptance is showing developers TDD benefits practically instead of using official arguments. It can be done by creating small projects to demonstrate them how TDD is helpful. Finally, we should not be dogmatic of TDD, because it leads to reluctance. We should try out to have a rational discussion with developers about TDD and show them that even without applying TDD, they are as qualified as other developers who can implement great codes.

- Priority 4: Insufficient developer testing skills
  As the quality of test cases can express the quality of designed system, it is recommended that a professional developer observe the quality of written test cases during the development. It is like the design verification in traditional test-last method.

- Priority 5: Insufficient TDD knowledge and experience
  When a developer is asked to contribute in a developing of a project, he or she should have enough knowledge on the specific method and if not, the organization should set up training for him or her. Since TDD is younger than traditional approaches, fewer developers are familiar with it. More universities and Software-Center trainings should set lectures in TDD subject and define lab materials to getting students familiar with this approach, as it is currently done for traditional ones. Also, an organization can increase the TDD-skill of its developers such as writing tests and refactoring by providing on-site trainings. Typically, it is costly and expensive to train the organization's developers, but without having TDD-trained developers, the successful project is not reachable. In addition, by developing more standards tools for using in TDD, more developers will be interested in TDD and obtain TDD-knowledge.

- Priority 6: Insufficient adherence to the TDD protocol
  For achieving the offered benefits of an approach, one must follow its phases completely. Lack of adhering to protocol is not related only to TDD. This is also possible for traditional approaches. Boby and Laurie had done an investigation about Test Driven Development in the industry [7], and in their investigation they asked control pairs who coded in a traditional fashion to write test cases after finishing code implementation. Lastly, only one group performed. There are reasons to avoid some steps of an approach such as time pressure, lack of discipline and shortage of perceived benefits [6]. Following the protocol should be monitored and controlled. Also, developer's mindset should be established to be aware of the approach's benefits. It can also be managed by working with TDD-experienced developers instead of non TDD-experienced.

4. CONCLUSIONS

In this paper, we presented advantages and disadvantages of Test Driven Development based on the different reviews of the several empirical studies. Also, existing researches and studies have indicated the limiting factors in applying TDD in the industry. These factors are:

- Long development time
- Insufficient TDD knowledge and experience
- Insufficient developer testing skills
- Insufficient adherence to the TDD protocol
- Domain and tool specific limitations
- Lack of accepting the benefits of TDD by project team

Although several empirical studies have reported many findings, those results are mixed and cannot produce a unique and particular decision about TDD. That indicates further researches need to be done about effectiveness of TDD in the industry, and for this we propose to use more TDD professionals and experts instead of students, and real projects contrary to sample and small projects.

We also provided some solutions to help removing those limiting factors and thus make it possible to apply TDD in the industry successfully. In addition, we prioritized them in a way that factor with high priority represents the constraint that leads to more difficulty on adapting TDD in the industry. For solving those constraints, we need money, time, tools and expert human resources. Test Driven Development is still new and novice in the industry and fewer professional and experienced developers exist in this area. As well, effective tools that ease developing with this approach are not plenty available in the market. These lead to apply TDD less in industry compared to traditional approaches.

We cannot decide explicitly whether TDD is dead or not. TDD is like a newcomer in the industry and still needs time to prove itself. However in some cases, lack of consideration in running TDD will lead to unavoidable failure. In case that a management team does not support TDD and puts

on pressure on developers, TDD will break down. Whenever using developers who all of them have early knowledge of TDD, it will fail or in the best case, it will be too difficult and time-consuming project. However, using high-level TDD-experience developers is not compulsory for success. Developers and managers should consider these facts and limitations, then decide about TDD by considering the trade-off, apply it at the right place and right time.

5. REFERENCES

[1] S. W. Ambler. Introduction to Test Driven Development (TDD). `http://www.agiledata.org/essays/tdd.html`, 2012. Retrieved November 8, 2014.

[2] M. F. Aniche and M. A. Gerosa. Most common mistakes in test-driven development practice: Results from an online survey with developers. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 469–478. IEEE, 2010.

[3] M. Awad. A comparison between agile and traditional software development methodologies. *University of Western Australia*, 2005.

[4] J. Buchan, L. Li, and S. G. MacDonell. Causal factors, benefits and challenges of test-driven development: Practitioner perceptions. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 405–413. IEEE, 2011.

[5] A. Bulajic, S. Sambasivam, and R. Stojic. Overview of the test driven development research projects and experiments. In *Proceedings of the Informing Science and Information Technology Education 2012 Conference (InSITE)*, pages 22–27, 2012.

[6] A. Causevic, D. Sundmark, and S. Punnekkat. Factors limiting industrial adoption of test driven development: A systematic review. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 337–346. IEEE, 2011.

[7] B. George and L. Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 1135–1139. ACM, 2003.

[8] A. Geras, M. Smith, and J. Miller. A prototype empirical evaluation of test driven development. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 405–416. IEEE, 2004.

[9] T. D. Hellmann, A. Hosseini-Khayat, and F. Maurer. Supporting test-driven development of graphical user interfaces using agile interaction design. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pages 444–447. IEEE, 2010.

[10] T. D. Hellmann, A. Hosseini-Khayat, and F. Maurer. Test-driven development of graphical user interfaces: A pilot evaluation. In *Agile Processes in Software Engineering and Extreme Programming*, pages 223–237. Springer, 2011.

[11] D. S. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer Science and Software Engineering*, page 33, 2005.

[12] D. S. Janzen and H. Saiedian. On the influence of test-driven development on software design. In *Software Engineering Education and Training, 2006. Proceedings. 19th Conference on*, pages 141–148. IEEE, 2006.

[13] S. Kollanus. Test-driven development-still a promising approach? In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the*, pages 403–408. IEEE, 2010.

[14] S. Kumar and S. Bansal. Comparative study of test driven development with traditional techniques. *Int J Soft Comput Eng (IJSCE)*, 3(1):2231–2307, 2013.

[15] R. Latorre. A successful application of a test-driven development strategy in the industrial environment. *Empirical Software Engineering*, 19(3):753–773, 2014.

[16] M.Fowler. Is TDD Dead? `http://martinfowler.com/articles/is-tdd-dead/`, 2012. Retrieved November 8, 2014.

[17] R. Mugridge. Challenges in teaching test driven development. In *Extreme Programming and Agile Processes in Software Engineering*, pages 410–413. Springer, 2003.

[18] O. P. N. Slyngstad, J. Li, R. Conradi, H. Ronneberg, E. Landre, and H. Wesenberg. The impact of test driven development on the evolution of a reusable framework of components–an industrial case study. In *Software Engineering Advances, 2008. ICSEA'08. The Third International Conference on*, pages 214–223. IEEE, 2008.

[19] D. Tchepak. Why learning TDD is hard, and what to do about it? `http://www.davesquared.net/2011/03/why-learning-tdd-is-hard-and-what-to-do.html`, 2011. Retrieved November 14, 2014.

Pioneering in Software Engineering

Tanja Ulmen
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
tanja.ulmen@rwth-aachen.de

Prof. Dr. Horst Lichter
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
lichter@swc.rwth-aachen.de

ABSTRACT

The term software engineering was used for the first time by the NATO in 1968. After that software engineering was constantly changing. Both, industry and academia invented new concepts, methods, languages, processes and technologies to make software engineering even more efficient. In this paper we want to find out if there is a general tendency to industry or academia, where more means of software engineering were developed and if there was a shifting from industry to academia or from academia to industry. To work this out we go into the history and use some innovations of software engineering. Another interesting point will be the impact of this different innovations and to find out if academia or industry had a bigger impact on the practice in software engineering. The last thing we are going to work out are the different categories of software engineering and which innovation belongs to which category. This may lead to other findings within the topic of pioneering in software engineering.

1. INTRODUCTION

In 1968 the term software engineering was used for the first time within a conference sponsored by the NATO. The following years software engineering grew rapidly. The innovations of software engineering were basically developed by academia and industry. In this paper we want to find out if there is a general tendency to academia or to industry and if there is a shifting from one to the other part. To work this out we use some innovations of software engineering. Another interesting part that we want to respect in this paper is the impact of the different innovations on the practice and if there is a connection between the impact on the practice and innovations developed by industry or academia. The last part of this paper will be a sorting of these different innovations into categories. This may lead to some other findings within the topic academia vs. industry.

Thus, this paper is split into four sections. The Section 2 is about the history of the different innovations. When the respective innovations were developed, by whom and if they were influenced by someone else. The Section 3 handles the impact of the different innovations on the practice. Section 4 deals with the categories of software engineering and if there is a connection between innovations from the academic sector or the industrial sector. Finally in Section 5 follows a short summery of the findings we made.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

2. TIMELINE

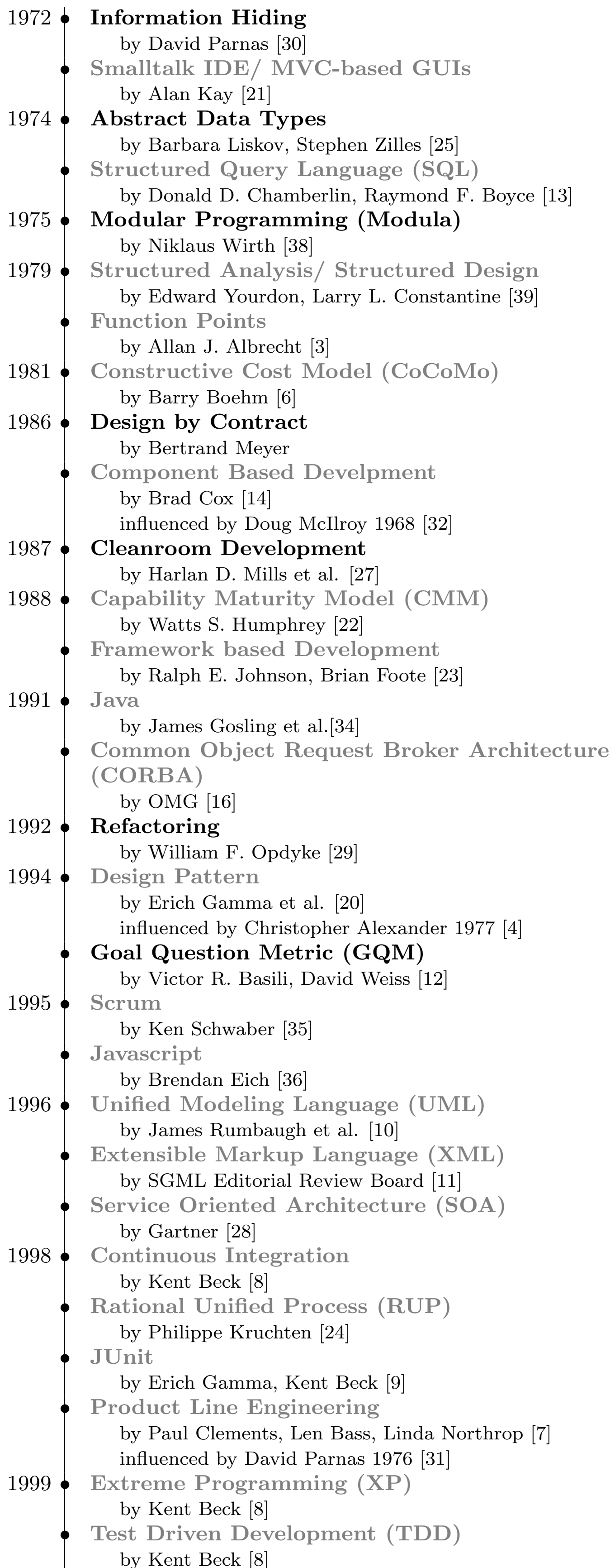
To work out which sector developed the most of the innovations, we constructed a timeline to list when each innovation was developed (Table 1). At the right side of this table are the years when the innovation was developed, the different colours distinguish if they are developed by industry (gray) or by academia (black). Below the name of the particular innovation are the names of the developers and if the development was influenced by another idea outside the software engineering sector.

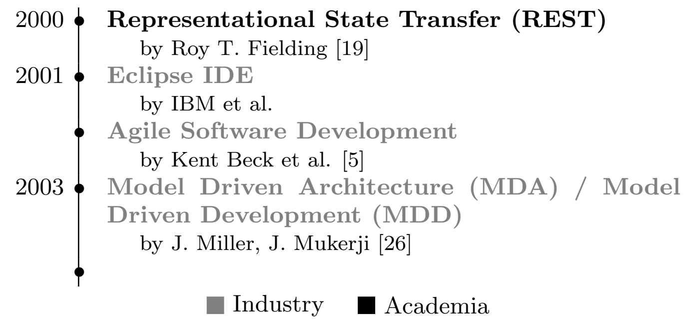
For example the design patterns were developed by the gang of four (Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides) in 1994. They all worked for different companies, thus this innovation came from the industry. The idea of the design patterns was based on the idea of pattern languages that was developed by an architect named Christopher Alexander in 1978.

Another example is structured programming that was developed in 1970 by Edsger W. Dijkstra. During this time he was Professor for mathematics at the Technical University Eindhoven thus, this innovation came from the academic sector.

The last example is the agile software development that started in 2001, after seventeen independent software developers signed the *agile manifesto*. All of the seventeen software developers came from the industry, thus this innovations is also located as developed by industry.

Year	Innovation
1967	**Object Oriented Programming (Simula)** by Ole-Johan Dahl, Kirsten Nygaard [15]
1970	**Structured Programming** by Edsger W. Dijkstra [17]
	Waterfall Model by Winston W. Royce [33] influenced by Herbert D. Benington 1956





■ Industry ■ Academia

Table 1: Timeline containing the innovations of software engineering. The left column contains the dates when the respective innovation of software engineering were developed, the right side contains the names of the developers and if the innovation was influenced by someone else. The colours distinguish between developed by industry (gray) and developed by academia (black).

2.1 Possible reasons for industrial dominance

In Table 1 we can see that the most of the innovations were developed by the industry. Out of 36 innovations only 9 innovations were developed by the academic sector. There might be several reasons:

- One reason might be that the industrial sector needs the economic competition. Industrial concerns need to be up to date to be able to bring their products to the market. This could be the reason why industry developed much more innovations of software engineering than the academic sector. Universities are not that affected by the economic competition. Of course the reputation of a university gets better if they are always up to date with their research topics but it is not as important as for the industry.

  An example is Nokia. Until 2006 they were leaders in the mobile phone market. Then Apple started with the smartphone production and one year later in 2007 the market share of Nokia fell from 50% to only 39%, in 2012 they had a market share of only 16% [1]. All mobile phone companies switched over to smartphones but Nokia missed this trend. The result is that Nokia was bought by Microsoft in April 2014. Thus, this is one example what might happen to a company if it does not follow the market. Thus, this might be one reason why the industry developed so much more innovations than academia.

- Another reason might be the capacity of an industrial company in contrast to a university. A big company like IBM for example has much more employees than a chair of a university and thus, much more capacity. Thus, they are faster and more efficient in developing new means of software engineering. For example the Software Engineering chair (i3) of the RWTH has 48 members [2], IBM in Dortmund has 230 employees [18].

- A further point is that companies nowadays are much more specialized than universities or chairs of universities. There are companies like cisco that work exclusively in the telecommunication sector. In universities

are also different chairs but usually only one for the whole sector of software engineering.

- The last reason might be, that universities have less money than industrial companies. In industry successful products are sold and the obtained money is used for the next developing process. In universities the gained money is used for several purposes, like teaching and things like that.

Another thing that is visible in the Table 1 is, that after 1994 are less innovations developed by academia than before 1994. Thus, there is no real shifting from academia to industry but it is visible that in the last 20 years academia produced less innovations than in the early years of software engineering.

One possible explanation is the growth of the software sector. There were a lot of changes in the software engineering sector, like the growth of the gaming industry and the embedded systems for example. Nowadays nearly each electrical machine contains embedded systems like cars, fridges, dish washers or even whole houses (smart homes) and like mentioned above industrial companies are more specialized than universities. Thus, companies have much more capacities to develop new products in such a wide range of software than academia has.

3. IMPACT

Another interesting point is the impact that the different innovations had on the practice. To find out how strong this influence was, we did a survey that contained all of the former mentioned innovations. In this survey the participant had to choose if the respective innovation had a very strong impact on the practice, a strong impact, a weak impact, no impact or if an innovation is unknown to him. The participants were people working for the RWTH Aachen and other computer scientists. The results of this survey are in Table 2.

At the top of the table are the innovations that had a very strong influence on the practice. Below are the innovations that had a strong influence and at the bottom are the innovations that had only a weak influence on the practice. The written values are the highest values for every innovation, the "unknown" values are left out. With this table we want to find out if there is a connection between the impact, that an innovation had on the practice and if this respective innovations came from industry or academia. Thus if it is possible that innovations from industry for example had more impact to the practice than innovations from academia.

With this table 2 it is visible that both, industry and academia produced a lot of innovations that had a strong influence on the practice and also some innovations that had only a weak influence on the practice. This table shows that there is no connection between the impact of an innovation and if it is developed by the industry or academia. It might be that the values of Table 2 would change if we worked with a lot more participants, because 32 is not very significant.

very strong influence

Innovation	highest Percentage
Object Oriented Programming	91 %
Java	53 %
SQL	50 %
Design Pattern	48 %
Modular Programming	45 %
Refactoring	45 %
Javascript	41 %
XML	41 %
Eclipse IDE	38 %
REST	26 %

strong influence

Innovation	highest Percentage
Agile Software Development	48 %
Smalltalk IDE	48 %
Continuous Integration	48 %
Structured Analysis/ Design	45 %
Structured Programming	41 %
Abstract Data Types	41 %
Information Hiding	39 %
UML	39 %
SOA	39 %
Framework based Development	38 %
JUnit	34 %
Scrum	32 %
MDA/ MDD	26 %
CORBA	24 %
CMM	23 %
RUP	20 %

weak influence

Innovation	highest Percentage
Extreme Programming (XP)	47 %
Waterfall Model	41 %
TDD	38 %
Component Based Develpment	36 %
Design by Contract	34 %
Goal Question Metric (GQM)	27 %
Cleanroom Development	23 %
Product Line Engineering	23 %
Function Points	19 %
CoCoMo	19 %

■ Industry ■ Academia

Table 2: Results of a survey with 32 participants. The survey is about the impact of the different innovations on the practice. The values are the highest values of all possibilities (no influence, weak influence, strong influence, very strong influence) except "unknown".

4. CATEGORIES

The last point that we want to focus on is the different categories of software engineering. We decided to sort the innovations into the following categories: languages, methods, concepts, processes and technologies. Thus the sorting is as follows:

	Languages
1	Java
2	JavaScript
3	Extensible Markup Language (XML)
4	Structured Query Language (SQL)
5	Unified Modelling Language (UML)
	Methods
1	Extreme Programming (XP)
2	Test Driven Development (TDD)
3	Goal Question Metric (GQM)
4	Constructive Cost Model (CoCoMo)
5	Continuous Integration
6	Service Oriented Architecture (SOA)
7	Refactoring
8	Capability Maturity Model (CMM)
9	Structured Analysis (SA)/ Structured Design (SD)
10	Function Points
	Concepts
1	Representational State Transfer (REST)
2	Design by Contact
3	Abstract Data Types
4	Structured Programming
5	Modular Programming
6	Object Oriented Programming
7	Design Pattern
8	Model Driven Architecture (MDA) Model Driven Development (MDD)
	Processes
1	Cleanroom Development
2	Component-based Development
3	Scrum
4	Agile Software Development
5	Product Line Engineering
6	Framework-based development
7	Relational Unified Process
8	Waterfall Model
	Technologies
1	Eclipse IDE
2	Smalltalk IDE or MVC-based GUIs
3	Corba
4	JUnit

Table 3: Sorting of innovations of software engineering into the categories languages, methods, concepts, processes and technologies.

Here we can observe that the most of the innovations were methods (10) followed by concepts (8), processes (8), languages (5) and technologies (4). Thus, there are a lot more methods than for example technologies. This might have the following reasons:

- One point may be that technologies like Eclipse IDE are further developed, there are a lot of new plugins every year. Thus, these plugins are no new technologies, they complete the Eclipse IDE. Whereas a development of Test Driven Development will end up in a completely new method. Behaviour Driven Development is such a method that is based on Test Driven Development [37].
- Another reason might be that new technologies are not as required as new methods.

In Table 4 we sorted the innovations with their respective categories like in Table 2. The innovations with a very strong impact on the practice are above, followed by the innovations with a strong influence and at last the innovations with a weak influence on the practice. With this sorting we want to find out if there is a connection between the impact and the category of software engineering. Thus, if it is possible to say that for example languages had more impact on the practice than methods. Another thing that might be visible is which category is developed by which sector. Maybe it is possible to say that academia developed only methods.

Innovation	category
very strong influence	
Object Oriented Programming	concept
Java	language
SQL	language
Design Pattern	concept
Modular Programming	concept
Refactoring	method
Javascript	language
XML	language
Eclipse IDE	technology
REST	concept
strong influence	
Agile Software Development	process
Smalltalk IDE	technology
Continuous Integration	method
Structured Analysis/ Design	method
Structured Programming	concept
Abstract Data Types	concept
Information Hiding	concept
UML	language
SOA	method
Framework based Development	process
JUnit	technology
Scrum	process
MDA/ MDD	concept
CORBA	technology
CMM	method
RUP	process

weak influence	
Extreme Programming (XP)	method
Waterfall Model	process
TDD	method
Component Based Develpment	process
Design by Contract	concept
Goal Question Metric (GQM)	method
Cleanroom Development	process
Product Line Engineering	process
Function Points	method
CoCoMo	method

■ Industry ■ Academia

Table 4: Sorting of the different innovations and respective categories like in Table 2. The innovations with a very strong influence are on top, followed by innovations with a strong influence and at the bottom innovations with a weak influence on the practice. The colours distinguish if the innovations are developed by industry (grey) or by academia (black).

The sorting of the categories shows that languages had a very strong influence on the practice. For example Java had a very strong influence, SQL, JavaScript and also XML. Nobody of the participants said that a language had a weak or no influence on the practice. Another interesting point is that nobody selected a process as an innovation with a very strong impact. Four processes had a strong influence and four processes only a weak influence. There is also only one method, refactoring, that had a very strong influence, whereas five methods had only a weak influence. Three technologies out of four in total had a strong influence and concepts are between a very strong and a strong influence, for both categories exists four concepts.

Thus, we can say that

- languages had a **very strong** influence on the practice (4 of 5 innovations)
- concepts are **between very strong and strong** influence (each 4 of 8 innovations)
- technologies had a **strong influence** on the practice (3 of 4 innovations)
- processes are **between strong and weak** influence (each 4 of 8 innovations)
- methods had a **weak** influence (5 of 10 innovations)

The last point that is visible in Table 4 is, that academia developed 6 concepts out of 8 in total, 2 methods of 10 and 1 process of 8. Hence, nearly all concepts are developed by academia whereas not a single technology or language came from academia, they were all developed by industry. This might have similar reasons like mentioned in Section 3: Companies are much more specialized than universities are and they have much more capacities.

5. CONCLUSIONS

To sum up it is possible to say that industry developed a lot more innovations of software engineering than academia did. This might be due to the fact that industry needs some kind of competition to get a high market share and to *survive* at the market. Companies are also more specialized than software engineering chairs and they have more employees and thus more capacities than a chair of a university.

Especially after 1994 the industrial sector developed much more innovations than academia. This might be because the software engineering sector changed a lot. The gaming industry grew for example, also the sector of embedded systems because today for instance every car has a lot of software inside, just as household aids and also whole houses. Thus, because we said that industrial companies are more specialized and because they have more capacities, they are able to develop much more innovations than academia does.

With a small survey we found out that there is no connection between the impact that innovations had on the practice and if they are developed by academia or industry. Both developed innovations with a very strong influence on the practice and ones with only a weak influence on the practice. This result might be caused by the size of the survey. We had only 32 participants and this is no significant number. Thus, with a bigger survey this result might change.

Another thing we wanted to point out are the categories of software engineering and if there is maybe a connection between the categories and the impact on the practice. We found out that languages had a very strong influence on the practice, technologies had a strong influence whereas methods had only a weak influence on the practice.

A further interesting point is that out of these 36 innovations we had only 4 technologies and 10 methods. Reasons for that may be that technologies are further developed, for example every year people develop a lot of plugins for Eclipse, but Eclipse stays the same. If a method like Test Driven Development is developed there might result a total new method out of that, such as Behaviour Driven Development.

The last finding is that all languages and technologies were developed by industry whereas nearly all concepts were developed by academia. This might also be due to the fact that industry is much more specialized and has more capacities than academia.

6. REFERENCES

[1] Nokia hat den anschluss verpasst, 2013.

[2] Members of the chair, 2014.

[3] A. J. Albrecht. Measuring application development productivity. In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, volume 10, pages 83–92, 1979.

[4] C. Alexander. *A Pattern Language*. Oxford University Press, 1977.

[5] A. Alliance. Agile manifesto, 2001.

[6] B. Barry. Software engineering economics, 1981.

[7] L. Bass, G. Chastek, P. Clements, L. Northrop,

D. Smith, and J. Withey. Second product line practice workshop report. *arXiv preprint cs/9811007*, 1998.

[8] K. Beck. Extreme programming: A humanistic discipline of software development. In *Fundamental Approaches to Software Engineering*, pages 1–6. Springer, 1998.

[9] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[10] G. Booch, J. Rumbaugh, and I. Jacobson. The unified modeling language. *Unix Review*, 14, 1996.

[11] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml). *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, 1998.

[12] V. Caldiera and H. D. Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2(1994):528–532, 1994.

[13] D. D. Chamberlin and R. F. Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[14] B. J. Cox and A. J. Novobilski. *Object-oriented programming: an evolutionary approach*, volume 2. Addison-Wesley Reading, MA, 1986.

[15] O.-J. Dahl and K. Nygaard. *SIMULA: A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual: by Ole-Johan Dahl and Kristen Nygaard.* Norwegian Computing Center, 1966.

[16] I. de Jong et al. Web services/soap and corba. *Compilation from comp. object. corba user group*, 2002.

[17] E. W. Dijkstra, E. W. Dijkstra, and E. W. Dijkstra. *Notes on structured programming.* Technological University Eindhoven Netherlands, 1970.

[18] I.-S. Dortmund. Ibm deutschland gmbh, 2014.

[19] R. T. Fielding. *Architectural styles and the design of network-based software architectures.* PhD thesis, University of California, Irvine, 2000.

[20] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[21] A. Goldberg and A. Kay. *Smalltalk-72: Instruction Manual.* Xerox Corporation, 1976.

[22] W. S. Humphrey. Characterizing the software process: a maturity framework. *Software, IEEE*, 5(2):73–79, 1988.

[23] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of object-oriented programming*, 1(2):22–35, 1988.

[24] P. Kruchten. *The rational unified process: an introduction.* Addison-Wesley Professional, 1998.

[25] B. Liskov and S. Zilles. Programming with abstract data types. In *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*, 1974.

[26] J. Miller, J. Mukerji, et al. Mda guide version 1.0. 1. *Object Management Group*, 234:51, 2003.

[27] H. D. Mills, M. Dyer, and R. C. Linger. Cleanroom software engineering. 1987.

[28] Y. V. Natis. Service-oriented architecture scenario, 2003.

[29] W. F. Opdyke. *Refactoring object-oriented frameworks.* PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[30] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15:1053–1058, 1972.

[31] D. L. Parnas. On the design and development of program families. *Software Engineering, IEEE Transactions on*, (1):1–9, 1976.

[32] B. R. Peter Naur.

[33] W. W. Royce. Managing the development of large software systems. In *proceedings of IEEE WESCON*, volume 26, 1970.

[34] H. Schildt. *Java: The Complete Reference, Seventh Edition.* McGraw-Hill, Inc., 2007.

[35] K. Schwaber. Scrum development process. In *Business Object Design and Implementation*, pages 117–134. Springer, 1997.

[36] C. Severance. Java script: Designing a language in 10 days. *Computer*, 45(2):0007–8, 2012.

[37] C. Solís and X. Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387. IEEE, 2011.

[38] N. Wirth. Modula: A language for modular multiprogramming. *Software: Practice and Experience*, 7(1):1–35, 1976.

[39] E. Yourdon and L. L. Constantine. *Structured design: Fundamentals of a discipline of computer program and systems design*, volume 5. Prentice-Hall Englewood Cliffs, 1979.

Release Engineering vs. DevOps - An Approach to Define Both Terms

Ralf Penners
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ralf.penners@rwth-aachen.de

Andrej Dyck
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
andrej.dyck@swc.rwth-aachen.de

ABSTRACT

Delivering software to the customer as fast as possible is essential for software development organizations, in order to keep in pace with competitors. As a consequence, release engineering, a software engineering discipline concerned with the delivery process of a software product, plays an important role in organizations. Furthermore, in recent years the term DevOps gained popularity in the IT world. It describes an approach to improve the collaboration between development and IT operations teams, in order to streamline software engineering processes. Until now there is no scientific definition for neither of these terms, and therefore, everyone uses his or her own definition. Thus, they are often confused or even used as synonyms.

In this paper, we will tell those two terms apart by contrasting available definitions and descriptions for both of them. Additionally, we will provide a scientific definition for release engineering and DevOps, which we developed in cooperation with some experts in these fields.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, software configuration management, programming teams*

Keywords

Release Engineering, DevOps, Definitions

1. INTRODUCTION

After improving software development processes for many years, for example, by adapting agile methods like Scrum, organizations started to recognize the importance of software delivery processes. If an organization has developed a new feature, but is not able to ship it fast and reliably to the customers, it is useless in the end. As a result, proper approaches and practices to improve the software delivery process arose.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

Release engineering is concerned with the delivery process of a software product to the customer; meaning, all activities between the point when the software is developed and the final deployment to the production environment. Since about 2009 a new approach called DevOps came up [5]. It tries to close the gap between development and IT operations, which originates through the different goals both have for their daily work, respectively. The developers want to deploy changes frequently, whereas the operators prefer less changes for stability reasons of the production environment [7]. As a countermeasure, DevOps improves the collaboration between those teams in order to reduce this barrier and accelerate the deployment. At first glance, someone could assume that both terms try to solve somehow the same problem, and thus, have the same meaning. The available definitions and descriptions of both terms so far convey the impression that there seems to be at least a big overlap [2, 3, 4, 6, 7, 8, 9, 10, 15, 20, 21, 22, 24, 25, 26]. For example, the English Wikipedia for release engineering states that it "is often the integration hub for more complex software development teams, sitting at the cross between development, product management, quality assurance and other engineering efforts, also known as DevOps" [25]. It is obviously a common problem to point out the difference due to the lack of precise definitions. Based on that, we try to find a scientific definition for both terms, release engineering and DevOps. To this end, we want to illustrate the similarities and differences between those two terms. Moreover, we involved some experts in these fields, discussing the proposed definitions.

In the first part of this paper, we will explain the terms from our point of view with the help of available descriptions and contrast them afterwards to point out the differences between them. Subsequently, we present existing definitions, our definitions and evaluate the corresponding feedback from the experts. Finally, we will give a short discussion, provide a perspective for future research, and give a short summary.

2. RELEASE ENGINEERING VS. DEVOPS

In this section, we start with the description of release engineering and DevOps, and point out the differences between both afterwards. The currently available descriptions of both terms have a strong focus on how to realize each of them in an organization, respectively. This often leads to confusion of release engineering and DevOps, since most practices and tools in this context like continuous delivery are suitable for both of them. For that reason, this section will concentrate on the meaning of both terms.

2.1 Release Engineering

Release engineering is concerned with the delivery process of a software product to the customer. So at the first glance, the word release in release engineering might suggest that it only comes into play after a software product was completely developed. But actually release engineering is part of a software project right from the beginning. There are many aspects that need to be considered beforehand and during the software development phase to establish a reliable and predictable delivery process in the end. Just like Dinah McNutt, a release engineer at Google, recommended in her keynote talk at RELENG 2014 "to keep the big picture in mind" [20, 18]; meaning, to constantly review all processes throughout the project that are related to the software development itself as well as to the delivery process of the software product.

It begins by choosing appropriate technologies and tools to compile, assembly, test, manage, and finally deploy the software. As an example the following technologies and tools can be used: Build and test tools (e.g., Maven and Jenkins), version control systems (e.g., Subversion or Git), system configuration tools (e.g., Puppet or Vagrant), and package manager (e.g., RPM or APT).

Next, the automation of recurring steps, like building and testing the software, is an important task. It improves the efficiency of those steps, and reduces failures and errors, since the manual execution is more error-prone. For instance, the following practices can be applied: Continuous delivery, continuous deployment, and infrastructure as code.

In addition to the technical aspects, the management of the software and the people involved in the corresponding project is also a central point of release engineering. A proper configuration and change management is needed to keep track of changes and quickly react on defects. That is accompanied by risk management in form of approaches like dogfooding and canary. Last but not least, coordinating the project and establishing a good overall communication between all project participants is an important task of a release engineer.

In summary, release engineering covers all aspects that have an impact or influence on the delivery process of a software product to the customer. John O'Duinn, who introduced the rapid release cycle at Mozilla, compared the task of a release engineer in his talk at GoogleTechTalks [21], which was also his keynote talk at RELENG 2013 [17], as "building a pipeline". To draw on that, a release engineer develops a construction plan about how to build that pipeline. Afterwards, they select the suitable materials to realize it. So the main task of release engineering is the construction of the processes; the practices and tools are just possible solutions to implement it.

2.2 DevOps

The term DevOps is an abbreviation of development and IT operations. Since 2009 there is a high interest in DevOps as the Google Trends analysis shows in Figure 1. While the amount of search requests for release engineering is more or less constant, the amount of search requests for DevOps increases continuously. The evolution of the term DevOps was generally influenced by several talks and papers since 2007, nicely summarized by Damon Edwards [5], co-author of the upcoming book *DevOps Cookbook* [1], in a blog post and by Michael Hüttermann in his book [7]. But one of the most determining talks was given by John Allspaw and Paul Hammond, both engineers at Flickr, at Velocity 2009 [2, 19]. This talk describes nearly everything what is today known as DevOps.

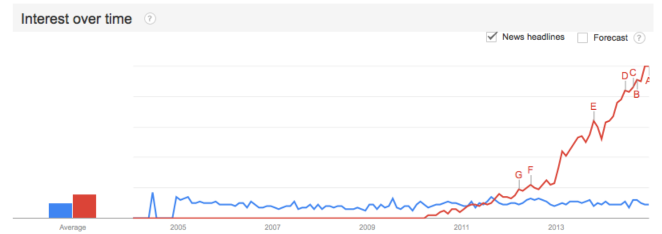


Figure 1: Google Trends statistic about the search requests of release engineering (blue) and DevOps (red) [14].

Hüttermann found out that in most bigger software development organizations the development teams and IT operations teams are separated from each other [7]. This is because of the different functions each of them have, respectively. The developers job is to evolve the software through changes, such as new features or bug fixes. Apart from that, the operators are focused on stability of the systems and therefore are anxious about changes. According to Lehman's law of continuing change, a program must change or it "becomes progressively less useful" [11]. Thus, a business can only grow further through changes. But a failure in the production environment, possibly caused by such changes, will damage the business of the organization. In principle, both teams want to increase the value to the customer through high quality software and resilient systems. However, their individual way to realize that is contradictory. Moreover, Hüttermann described the problem that nowadays the idea of agile is widely adapted in software development [7]. Whereas in IT operations still non-agile approaches are applied. It is not uncommon that the developers create a new increment for a software every two weeks, but the operators only deploy it in hard defined release windows every six weeks or even longer.

The DevOps approach aims to establish a new culture and mindset in an organization. It focuses on a closer collaboration between developers and operators, and extends the idea of agile to IT operations. Consequently, the goal is to develop high quality software and operate resilient systems. According to Puppet Labs et al. it also improves the IT performance of an organization [15]; meaning, it leads to a higher deployment frequency with lower failure rates.

The implementation of a DevOps culture comprises four areas as shown in Figure 2, which Patrick Debois, a central figure in the context of DevOps, described in a blog post [4]. The first and second areas are concerned with the feedback cycles. The developers support the operators in the delivery process, and the operators communicate the status and information of the production environment to the developers. Thereby, the operators can better build the production environment according to the needs and requirements of the developers, and the developers are able to consider the shared information during their development. Areas three and four cover the cross-functional way of working between development and IT operations; meaning, the developers will take over some operational tasks and vice versa. These

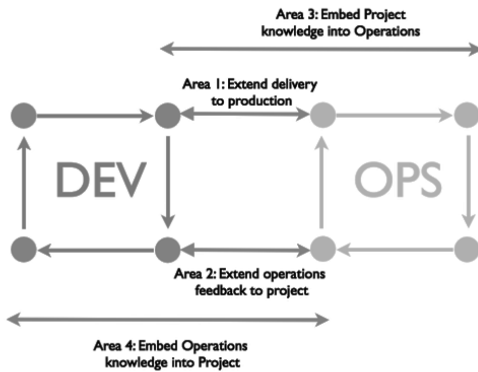


Figure 2: Four areas of DevOps [4].

four areas are then implemented with the help of practices and tools. For example, continuous delivery and deployment, infrastructure as code, and usage of a version control system are popular and important aspects for establishing a DevOps culture.

In summary, as PuppetLabs et al. explained "the heart of DevOps" are the cultural aspects like "good information flow, cross-functional collaboration, shared responsibilities, learning from failures and encouragement of new ideas" [15]. Practices and tools play only a subordinated role, since they just support the implementation. That is similar to what Katherine Daniels, an operations engineer at Etsy, stated in her talk at DevOpsDays 2014 in Minneapolis [3, 13]. According to her, tools can only reinforce culture. Instead, she interpreted the meaning of DevOps as empathy between the teams of an organization: "Empathy allows a developer to understand why the sysadmin craves reliability and [does not] want the site to go down. [...] It allows that same sysadmin to realize why the developer needs to release all their shiny new features. Empathy is what allows people to really work together" [3].

2.3 Differences

In the previous two sections, we described the meaning of release engineering and DevOps. Now, in this section we want to point out the differences between them. Both try somehow to improve the customer added value by enabling the organization to deliver high quality software. Simultaneously, both rely on common practices and tools like continuous delivery. This leads to an overlapping of both terms, and thus, often to confusion. For example, Chuck Rossi, a release engineer at Facebook, explained in his keynote talk at RELENG 2014 the concept of "on-call for the week" [22, 18], which is used in their release process: If something goes wrong during the deployment of a software, one developer is responsible that it will be fixed as soon as possible. So it involves developers into the delivery process and feedback from the production environment is pushed back to the developers. This concept also covers the DevOps areas one and two described in the previous section and shown in Figure 2.

The difference will become clear by recognizing that release engineer is a job title and DevOps not. There are many organizations that try to hire DevOps engineers or want to create a DevOps team. But like Jez Humble, an IT consultant at ThoughtWorks, explained in his keynote talk at PuppetConf 2013: "You can not hire a culture" [8, 16]. To our understanding, this is caused by the strong focus on practices and tools. However, these are only approaches for *how* the specific goals can be achieved, not *what* they actually are about.

A release engineer creates a plan about how the software can be delivered to the customer, considering all factors that have an impact on that. To realize this plan they need practices and tools. For example, they think about how to ensure high quality of the software in order to reduce failures during and after the deployment. To do so, continuous delivery could be implemented. In other words, a release engineer has to analyze the specific situation in his or her organization, create a plan tailored on that situation, and then choose the fitting practices and tools to implement the plan.

DevOps tries to establish a culture, where people in an organization do not work in functional silos [9], but instead, they collaborate with each other and work in a cross-functional way. Note that DevOps is not only limited to development and IT operations. This culture can be – or even should be – established in the whole organization to reduce barriers. In the end, this enables the productivity in form of higher deployment frequencies with less failures [15], which eventually increases the product value. There are several practices and tools, which help to implement such a culture. However, none of them is special or explicitly required for DevOps.

All in all, by focusing only on the meaning of release engineering and DevOps the differences become more clear: While release engineering is concerned with the delivery process of the software to the customer in a holistic way, DevOps establishes a culture and mindset to perform, for instance, such processes more efficiently. It could be argued, that DevOps is another approach in the arsenal of a release engineer, since their work might be easier with such a culture in an organization.

3. DEFINITIONS

With the meaning of release engineering and DevOps, and the differences between them in mind, we want to propose a scientific definition for both terms in this section. Since to our knowledge, there is no scientific definition for neither of these terms so far. As a consequence, everyone uses his or her own definition, which results in confusion about those terms. For that reason, we consolidated with experts from both fields, to find a common basis. We hope that with our definitions it will be easier to keep release engineering and DevOps apart. For each term we will start by analyzing the available definitions, present then our scientific definition and explain it afterwards by evaluating the feedback of the experts.

3.1 Release Engineering

The English Wikipedia defines release engineering as "a sub-discipline in software engineering concerned with the compilation, assembly, and delivery of source code into finished products or other software components. [...]" [25]. In this definition the aspect of improvement and efficiency is missing. After an initial software delivery process has been created, it should then be permanently improved to be more efficient and, as a consequence, deliver software as fast as possible to the customer. McNutt defined release engineering in a more abstract and figurative way: "Accelerating the path from development to operations" [20]. This corre-

sponds with our understanding as it contains the improvement factor in the word accelerating. Additionally, the word path points out the holistic approach of release engineering throughout the project and that the focus is not just on the delivery itself. However, this definition is somewhat unspecific due to a high abstract level. Based on the analysis of these definitions and the descriptions in the previous sections, we developed a scientific definition for release engineering in cooperation with some experts in this field.

> ***Release Engineering*** *is a software engineering discipline concerned with the development, implementation, and improvement of processes to deploy high quality software reliably, and predictably.*

Discussing the definition with Bram Adams (personal communication, November 14, 2014), an assistant professor at the École Polytechnique de Montréal, and Humble (personal communication, November 13, 2014), we noticed the importance of high quality software as a central goal of release engineering. A software product has to fulfill all quality requirements before it is deployed into the production environment. It should not lead to a system crash or produce errors, while the customer is using the software. In such a case just a fast, reliable, and predictable deployment process will not provide any benefit for the organization.

Adams remarked (personal communication, November 11, 2014) that the term improvement might be implied by development. We decided to itemize the terms development, implementation, and improvement to emphasize the three main tasks of a release engineer: Starting from an organization, where no defined and controlled processes exist for releasing a software product, the first task is to develop suitable processes. After that, they need to be implemented with the help of practices and tools. If there is one reliable and predictable process chain, it then can be further improved to make it more efficient. So the implementation part covers also the consideration of Professor Lichter (personal communication, November 16, 2014), head of the software construction research group at RWTH Aachen University, that methods and tools are important aspects of release engineering.

Adams suggested (personal communication, November 11, 2014) to specify the single tasks as "[...] processes to integrate, build, test, package and deliver [...]"; similar to Wikipedia's definition [25]. The problem with such a listing is, that it might suggest release engineering comprising only those tasks. As we described in the previous sections, release engineering is concerned with many different tasks and responsibilities throughout the whole software project.

Humble also commented (personal communication, November 13, 2014) to use the words delivery and deployment simultaneously in the definition. In this context, Martin Fowler, an IT consultant at ThoughtWorks, described in a blog post the difference between continuous delivery and deployment [6]: When using continuous delivery the software is always in a releasable state, since every change of a software is automatically tested in an production-like environment. The next step is then continuous deployment, where the software is automatically deployed into the production environment once all tests passed. Adapting this, we decided to even remove the word deliver in our definition, because it is already covered by the word deploy.

In conclusion, McNutt and Akos Frohner, also a release engineer at Google, stated that this definition covers what they mean at Google. Kim Moir, a release engineer at Mozilla, and Gene Kim, author and researcher in the area of release engineering and DevOps, said that they like our definition of release engineering. Moreover, Rossi noted that all keywords he considers critical in release engineering namely reliability, predictability, and delivery are covered.

3.2 DevOps

The English Wikipedia defines the term DevOps as "is a software development method that stresses communication, collaboration and integration between software developers and Information Technology (IT) professionals [...]" [26]. We disagree with this definition, because DevOps is not a software development method; it does not define any procedures or techniques like, for example, Scrum does in form of sprints. However, we partially agree with stressing on communication, collaboration, and integration between development and IT operations. Moreover, the definition is missing the cultural aspect. Hüttermann's definition of DevOps is "a mix of patterns intended to improve collaboration between development and operations [...]" [7]. We mostly agree on that. On the one hand, there are many recommended patterns or practices about how to establish a DevOps culture. On the other hand, the definition suggests that if a combination of those patterns or practices is used, then an organization has successfully implemented DevOps. In brief, this definition does not emphasize that DevOps is a mindset, and that there are no practices or tools only dedicated to DevOps. So just like for release engineering, we developed a scientific definition for DevOps based on the analysis of the available definitions, descriptions in the previous sections, and the feedback of some experts in this field.

> ***DevOps*** *is a mindset, encouraging cross-functional collaboration between teams - especially development and IT operations - within a software development organization, in order to operate resilient systems and accelerate delivery of changes.*

In general, there seems to be at least a common ground that DevOps is about improving the collaboration between teams in an organization. But the goal that DevOps wants to achieve with this improved collaboration is either completely missing or differs strongly. Adams suggested the goal of DevOps is (personal communication, November 11, 2014), for example, the acceleration of feedback, higher quality of software, and faster deployment. This is correct, because a sub-goal of DevOps is to reduce barriers and improve feedback cycles between development and IT operations in order to increase the software quality and be able to deploy faster. However, through the discussion with Jeff Sussna (personal communication, November 25, 2014), founder and principal of Ingineering.IT, we noticed that DevOps is about more than fast deploying high quality software. He stated that "it is about operating software services [and delivering] change to those services". This is similar to what also Humble used in his proposed definition (personal communication, November 13, 2014): "[DevOps is a] cross-functional community of practice dedicated to the study of building, evolving and operating rapidly changing resilient systems at scale". Those definitions also include the comment of Frohner (personal communication, October 30, 2014) to "emphasize the 'oper-

ational' part of [DevOps] [...]". So an important aspect is to focus on software and the production environment as a whole, as expressed through the word systems. Such systems need to change rapidly, in order to add value to the organization. At the same time, the systems have to be resilient, since a crash will have a negative impact on the business. Furthermore, as Professor Lichter (personal communication, November 16, 2014) remarked, developing and maintaining those systems is performed by the developers and operators jointly. Kim et al. provide another definition for DevOps in their upcoming book *DevOps Cookbook*, that corresponds with the other comments and definitions: "The term 'DevOps' typically refers to the emerging professional movement that advocates a collaborative working relationship between Development and IT Operations, resulting in the fast flow of planned work (i.e., high deploy rates), while simultaneously increasing the reliability, stability, resilience and security of the production environment" [1]. This definition is also published in an online available paper [10].

4. DISCUSSION

We mentioned in the previous sections that DevOps is another approach, which can be used by a release engineer. Moreover, we stated that DevOps can be extended to other teams in an organizations. In this section, we want to discuss these two aspects more in detail. Additionally, we came up with a question about which organizations can implement release engineering and DevOps.

Kim considers (personal communication, November 15, 2014) that DevOps is a superset of release engineering. As discussed previously, DevOps is a culture, whereas release engineering is a discipline; meaning, it would be more the other way around. In short, DevOps is one possibility for a release engineer to improve the delivery processes, because with such a culture in an organization it might be easier and more efficient. Referring to our definitions, the overlap between both terms is in the common goal. Both want to provide high quality software for the customer as fast as possible. The difference is that DevOps tries to improve the collaboration between all participants in order to reach that goal, whereas release engineering addresses this goal in a holistic way. Thus, if at all possible to define both by using sets, DevOps would be a subset of release engineering. Considering that the elements of DevOps are not new: Expect from the cultural part, all practices and tools for realizing a DevOps culture can also be used in the area of release engineering. Many organizations have been using them even before the term came up and gained popularity. For example, Rossi mentioned that DevOps gave him a name for the things they have been doing at Facebook organically in their release process even before the term came up [22]. Furthermore, release engineering exists as long as software engineering is used for developing software in a engineering-like fashion.

New terms for extending DevOps to other areas of an organization came up in recent times as mentioned in Sussna's talk [24]. Acronyms like DevSecOps or DevNetOps express the inclusion of other departments, for instance, information security or network administration, into the DevOps approach. The acronym DevOps was invented through the first DevOpsDays in 2009 and the resulting Twitter hashtag #DevOps for that event [5, 12]. The reason why it just contains the words development and IT operations is, that this conference focused mainly on the problems between those two teams in an organization. But the mindset of DevOps can be extended to any other team in order to make the whole organization more efficient. In brief, there is no need for another term to express the same content.

Another more radical approach is known as NoOps with the aim to completely replace IT operations with development. Hüttermann clarified [7], that this is difficult to achieve: On the one hand, is administrating infrastructure a completely other skill than developing software. On the other hand, even if developers replace operators, they, nevertheless, have to do operational tasks like deploying the software and setting up the production environment.

During an internal discussion we came up with a question which organizations can implement release engineering and DevOps. Release engineering can be applied by all organizations that deliver a software product to a customer. But for DevOps it depends somehow on the interpretation of the word IT operations. One the one hand, if IT operations is interpreted as operating a production environment, then only organizations, that operate their software on their own, can use it. One the other hand, if the word is meant for all activities and tasks, that are not related to development or management areas of an organization, then all organizations can implement it, since all of them will have some infrastructure that needs to be operated. Furthermore, as we explained in this section, DevOps can be extended to other teams of an organization. By focusing on the cultural part of DevOps, it could be even used in non-software development organizations. Since based on Daniel's interpretation of DevOps [3], working in a software development organization is not a precondition for improving the collaboration and developing empathy.

5. CONCLUSION

In this paper, we aimed to showcase the differences between the terms release engineering and DevOps. We summarized how both are described and defined in available books, blog posts, papers, and talks. Due to missing scientific definitions, we developed such in collaboration with some experts in these fields. Based on that, it might now be easier to keep both terms apart.

Admittedly, our research has a threat to validity, since the definition for DevOps is still discussable. ScriptRock, an organization developing cloud-based solutions to monitor dynamic data centers, stated in a blog post that it is unpleasant to find a simple definition or explanation for DevOps [23]. They explained that many DevOps definitions so far have a subjective focus, since everybody considers another aspect as most important [23]. Furthermore, they claimed that "[it is] not only necessary, but important, that DevOps be defined simply and in such a way that anyone in the office could understand" [23]. Although we already tried to remove the subjective focus by discussing our definitions with some experts, the feedback for DevOps was not as consistent as for release engineering. For a more precise definition of DevOps additional experts should be interviewed and a survey with organizations practicing DevOps could be done.

Finally, based on our definitions we want to point out the difference between release engineering and DevOps once again: Release engineering is a discipline concerned with the establishment and improvement of predictable and reliable processes in an organization to deliver high quality

software to the customer. DevOps improves the collaboration between teams in an organization through a cultural change. It enables development and IT operations to operate resilient systems, and deliver changes to them quickly at the same time.

6. ACKNOWLEDGMENTS

We give many thanks to Jez Humble, Jeff Sussna, Gene Kim, Bram Adams, and Professor Lichter for the contribution to our definitions. Additionally, we thank also Akos Frohner, Kim Moir, Chuck Rossi, Dinah McNutt, and Patrick Debois for giving us feedback.

7. REFERENCES

[1] J. Allspaw, P. Debois, D. Edwards, J. Humble, G. Kim, M. Orzen, and J. Willis. The DevOps Cookbook. http://itrevolution.com/books/devops-cookbook/, to be published. [Online; accessed: 12-07-2014].

[2] J. Allspaw and P. Hammond. 10+ Deploys Per Day: Dev and Ops Cooperation at Flickr. http://www.youtube.com/watch?v=LdOe18KhtT4, June 23, 2009. [Online; accessed: 11-10-2014].

[3] K. Daniels. DevOps Is Dead (Long Live DevOps). https://www.youtube.com/watch?v=OUUNJTq890E, July 18, 2014. [Online; accessed: 12-07-2014].

[4] P. Debois. Devops Areas - Codifying devops practices. http://jedi.be/blog/2012/05/12/codifying-devops-area-practices/, May 12, 2012. [Online; accessed: 11-14-2014].

[5] D. Edwards. The History Of DevOps. http://itrevolution.com/the-history-of-devops/, September 17, 2012. [Online; accessed: 11-10-2014].

[6] M. Fowler. Continuous Delivery. http://martinfowler.com/bliki/ContinuousDelivery.html, May 30, 2013. [Online; accessed: 01-10-2015].

[7] M. Huettermann. *DevOps for Developers*. Apress, 2012.

[8] J. Humble. Stop Hiring DevOps Experts and Start Growing Them by Jez Humble. http://www.youtube.com/watch?v=6m9nCtyn6kE, January 16, 2014. [Online; accessed: 11-10-2014].

[9] J. Humble. There's No Such Thing as a 'Devops Team'. http://continuousdelivery.com/2012/10/theres-no-such-thing-as-a-devops-team/, October 19, 2012. [Online; accessed: 11-14-2014].

[10] G. Kim. Top 11 Things You Need To Know About DevOps. http://itrevolution.com/pdf/Top11ThingsToKnowAboutDevOps.pdf, June 20, 2013. [Online; accessed: 11-20-2014].

[11] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1068, September 1980.

[12] DevOpsDays Ghent 2009. http://devopsdays.org/events/2009-ghent/, October 30-31, 2009. [Online; accessed: 12-12-2014].

[13] DevOpsDays Minneapolis 2014. http://devopsdays.org/events/2014-minneapolis/, July 17-18, 2014. [Online; accessed: 12-10-2014].

[14] Google Trends. http://www.google.com/trends/explore#q=release%20engineering%2C%20devops&cmpt=q, December 11, 2014. [Online; accessed: 12-11-2014].

[15] Puppet Labs, IT Revolution Press, and ThoughtWorks. State of DevOps Report 2014. http://puppetlabs.com/2014-devops-report, June 4, 2014. [Online; accessed: 11-13-2014].

[16] PuppetConf 2013. http://puppetlabs.com/resources/puppetconf-2013, August 22-23, 2013. [Online; accessed: 11-10-2014].

[17] RELENG 2013. http://releng.polymtl.ca/RELENG2013/, May 20, 2013. [Online; accessed: 11-09-2014].

[18] RELENG 2014. http://releng.polymtl.ca/RELENG2014/, April 11, 2014. [Online; accessed: 11-09-2014].

[19] Velocity 2009. http://velocityconf.com/velocity2009, June 22-24, 2009. [Online; accessed: 12-10-2014].

[20] D. McNutt. The 10 Commandments of Release Engineering. https://www.youtube.com/watch?v=RNMjYV_UsQ8, April 11, 2014. [Online; accessed: 11-09-2014].

[21] J. O'Duinn. Release Engineering as a Force Multiplier. http://www.youtube.com/watch?v=7j0NDGJVROI, May 28, 2013. [Online; accessed: 11-09-2014].

[22] C. Rossi. Moving to mobile: The challenges of moving from web to mobile releases. https://www.youtube.com/watch?v=Nffzkkdq7GM, April 11, 2014. [Online; accessed: 11-09-2014].

[23] ScriptRock. The Problem with Defining DevOps. http://www.scriptrock.com/blog/the-problem-with-defining-devops, December 3, 2014. [Online; accessed: 12-10-2014].

[24] J. Sussna. Promising Digital Service Quality. https://www.youtube.com/watch?v=0j4czzOOwJY, July 18, 2014. [Online; accessed: 12-03-2014].

[25] Wikipedia. Release engineering. http://en.wikipedia.org/w/index.php?title=Release_engineering&oldid=585442891, December 10, 2013. [Online; accessed: 11-14-2014].

[26] Wikipedia. DevOps. http://en.wikipedia.org/w/index.php?title=DevOps&oldid=633593434, November 12, 2014. [Online; accessed: 11-14-2014].

A Comparison of Architectural Debt Measurements

Piro Lena
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
piro.lena@rwth-aachen.de

Muhammad Firdaus Harun
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
firdaus.harun@swc.rwth-aachen.de

ABSTRACT

Over the last two decades the term of technical debt has gained much attention among the software development companies. Since the metaphor of technical debt is used for the first time, researches are conducted in order to measure it. These are mainly focused on source code and implementation anomalies. Recently, the focus has shifted into finding different methods and techniques to measure debt from the software architecture perspective. However, the studies conducted so far are new and still need improvements. In this paper we investigate different methods: Modularity metrics, Dependency Metrics and Value Oriented Architecture Analysis. Furthermore, we make a comparison mainly in terms of measurement units, economic attributes, refactoring costs and complexity of above mentioned methods.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

Technical debt, architectural debt, modularity metrics, value oriented architecting, dependency metrics

1. INTRODUCTION

"Shipping first time is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-so-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise."

This definition ([4]) is used for the first time to define technical debt. It states that neglecting the design is like borrowing money. Refactoring, it's like paying off the *principal* debt. On the other hand, developing slower because of this debt is like paying the *interest* on the loan. Another definition is by Kruchten et al:*A design or construction approach that's expedient in the short term but that creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time).* However, the concept is clear, there are still problems on finding solution to identifying, measuring, controlling and eliminating the technical debt. This comes as a consequence of the various reasons from which the debt in software development cycles is accumulated. Some of these technical debts can be identified on social, test, code or architectural level.
The scope of this paper is to focus on architectural technical debt. On this category are included inconsistent or poor design solutions, casual mistakes due to agile working or choices that turn to be wrong while gaining experience [10],[3], [8]. The difficulty in identifying the Architectural Technical Debt (ATD) is that it becomes visible later in the project. ATD is presented in Section 2. The metrics which are used to estimate or identify and reduce the impact of ATD on software projects are increasing in number, especially during the recent years. However, we focus on the Modularity Metrics [5],[9], Value-Oriented Architecture[6] and Dependency Metrics methods[7],[2]. The approaches, case-studies and contribution of each of them is described in details in Section 3. Despite the fact that these techniques are relatively new, a comparison among them in terms of architectural elements used on each of the methods, measurement units, costs, complexity, time of refactoring and maintenance details are presented in Section 4. Discussion and conclusions will adduce our remarks regarding these approaches.

2. ARCHITECTURAL TECHNICAL DEBT

Software architecture is defined as the core component of a software system. It specifies the fundamental concepts or properties of a system in its environment embodied in its elements, relationships and in the principle of its design evolution.[3]
Despite its importance, due to the fast development during agile software development, most of the time the architecture of software is neglected. For instance, at the beginning, a small software project rarely uses all the principles of software architecture. Some of them are just used by individual developers. With the passing of the time the project grows and its complexity increases, the architecture principles and structuring tend to be postponed. At the beginning all of

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

these may seem irrelevant until the whole architecture degrades. At this point there is no turning back, the whole architecture has to be repaired. The costs of this are tremendously high, and it is here when the architectural technical debt importance arises.
As it is noticed not only from companies but also from individual developers, supported also the recent studies [7],[5], the architectural debt merely effects the future project sustainability rather than the present functionality or quality. This might come as a result of the technics of software development. Agile developing usually tend to "hurry" in coming with the next release of the software product. This intentionally or not, tend to place on second place the architectural issues. When a crisis finally happens, only then the teams tend to turn back to it, trying to prioritize the backlog of restructuring tasks. As it is noticed above, the architectural debt is more related to the prediction abilities of the managerial teams. The later the intervention the higher the debt.
In contrast with normal code level decay, architectural decay is not as well defined and understood [7]. Sometimes it is possible that the two types of decay happens at the same time, on the same project. The impact of the two of them is high,but [7] the costs of architectural decay are higher. Moreover, in comparison with code decay instances that are defined based on source code analysis, the architectural decays refer to a higher level of abstraction. This increases the difficulty of analyzing its costs, especially while dealing with concepts that goes beyond the architectural definitions (for example, concerns regarding future decisions).
The complexity for correctly identifying the architectural decays is high and measuring the technical architectural debt is difficult. The recent studies have shown useful methods to tackle these issues. On these paper are presented only three of them, but there are others defined or not-fully defined methods that might be used by companies and researchers.

3. ATD APPROACHES

As already mentioned above, Architectural Technical Debt (ATD) is accumulated when intentional or not design decisions compromise the **Q**uality **A**ttributes of a project. Usually the once which are mostly effected are maintainability and evolvability. Considering that ATD does not show any observable behaviour to end users, any negative feedback from them is not expected. This increase the difficulty of detecting it. In the following sections are presented the methods commonly used to approximately measure ATD.

3.1 Indicating ATD by using Modularity Metrics approach

A way of measuring architectural technical debt is to calculate the average number of modified components per commit (ANMCC). A commit is a unit of modification to the source code of a software system [5]. The question that rises at this point is how ANMCC is related to ATD. The following give a reasonable explanation:
• ANMCC reflects the complexity and difficulty of making changes to a software system.
• The higher the complexity and difficulty of making changes the higher the accumulated ATD.
• A higher ANMCC means more ATD in a software system.
On a first glance the calculation of ANMCC seems very easy. The problem which makes it more difficult is the lack of records used to calculate the modified commits. This makes it almost impossible to directly measure ATD. [5] and [8] proposed a methodology which can be directly calculated using the available source code and is correlated with ANMCC. [5] introduced software modularity metrics which can be directly measured. Based on that they found a valuable correlation with ANMCC which on its own relates to ATD. Referring to ISO/IEC 25010 standard, modularity is a characteristic of maintainability, which is compromised by ATD. By definition (from [5]), *Modularity is the degree to which a system or computer program is composed of discrete components such that a change to one component has minimal influence on the other components.*
By having this definition and making a naive reasoning, we can come up with the idea that the modularity metrics of all the previous changes of a project source code reflect the difficulty of making changes to the project code in the future. On its own this one represents the ATD. So, if the modularity of a software system increases, ANMCC have to decrease, which means ATD decreases as well. To come up with the "scientific" approach of our intuitive result, [5] conducted a case study on 13 C# open source software (OSS) projects on Github. According to this study, the reason for choosing OSS projects is to come closer to real life context. In this way, both modularity and ANMCC can yield more realistic results than being monitored in isolation. The only restriction applied on their case study was related to the number of releases per each of the projects and the number of components per each release. This was done to guarantee that the project is "matured" enough to not have tremendous changes from the previous releases. Also, the project that was selected should have been computable, so as to generate the code maps. The measurements that was done referring to [5] were related with the following records:
• **Index of Inter-Package Usage (IIPU)**- The ratio of the number of *Use dependencies* between classes within a local package against the total number of Use dependencies between classes of the whole software system.
• **Index of Inter-Package Extending (IIPE)**- The ratio of the number of *Extend dependecies* between classes within a local package against the total number of *Extend dependencies* between classes of the whole software system.
• **Index of Package Changing Impact (IPCI)**- The percentage of the number of the non-dependency package pairs against the total number of all possible package pairs. This measures the strength of the independency of packages.
• **Index of Inter-Package Usage Diversion (IIPUD)**- The average event of how diverse the classes used by a specific package distribute in different packages.
• **Index of Inter-Package Extended Diversion (IIPED)**- The average event of how diverse the classes extend by a specific paskage distribute in different packages.
• **Index of Package Goal Focus (IPGF)**- The average extent of the overlap between the different service sets provided by the same component to other different components in a software system. This shows the average extent that the services of a specific package serve the same goal.
• **Average Number of Modified Components per Commit (ANMCC)**- The average number of components that are modified durign each commit(revision) in the studied period.
Each of the measurements from modularity metrics are compared with normalized ANMCC results using Spearman's

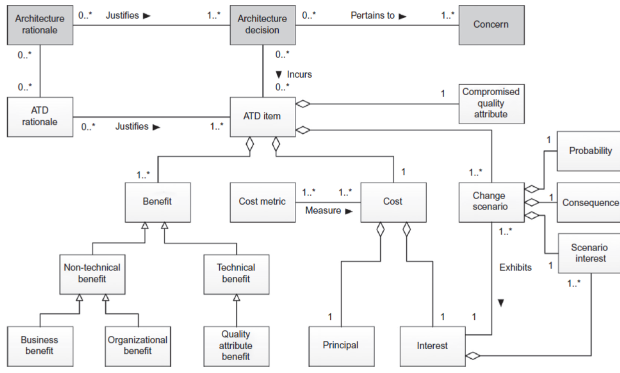


Figure 1: ATD Conceptual Model

correlation. In Table 1, below are presented the results of this tests.

	ρ -value	p-value
IIPU	-0.099	0.3741
IIPE	-0.104	0.3671
IPCI	**-0.828**	**0.0001**
IIPUD	-0.138	0.3261
IIPED	-0.028	0.4631
IPGF	**-0.522**	**0.0341**

Table 1: Correlations of modularity metrics and ANMCC

As it can be seen from the table the IPCI and IPGF have a significant negative correlation with the normalized ANMCC. The other metrics doesn't show a significant correlation with it, so they cannot be considered while measuring ATD. Furthermore, IPCI is more correlated (negatively) with normalized ANMCC than IPGF. From this follows, that it is necessary to measure only IPCI to understand the behavior of ATD. The higher the IPCI, the lower the normalized value of ANMCC, the lower the value of architectural technical debt. The same reasoning if the IPCI value is low. Actually this results give a powerful methodology to both researchers and practitioners to highly approximate the costs accumulated by ATD.

3.2 Measuring ATD through Value-Oriented Architecting

As mentioned earlier on the previous chapters, technical debt can be seen as a consequence of trade-offs made by designers while developing a software system. The costs of these decisions influence the project in all of its levels, including architectural once. In this section, is shown another method to calculate and manage architectural debt, by considering it from a value-oriented prospective. Referring to [6], the ATD conceptual model is shown on the Figure 1. In center of this model is ATD item, which is a term introduced to be used for further analysis. The other components are also useful. A full description of them could be found at [6].
The focus is mainly on the costs of long term maintenance and software evolutionary, by not taking into consideration the short term visible costs. Maintainability cover characteristics like modularity, reusability, analyzability, modifiability and testability. On the other side evolvability measures how easy is to add new or change existing requirements. ATD

ID	ATD-11
Name	Compromised modifiability due to using MS Access to store test results
Date	10-11-2012
Status	Unresolved
Incurred by	AD-12: storing test results temporarily in local MS Access database (design option 2)
Responsible	Tom
Compromised QA	Modifiability
Rationale	If MS Access as a local database is used to store test results temporarily when the remote database server is inaccessible, any change of the table design in the remote database requires the according modification to the tables in MS Access. In addition, data in the local MS Access database need to be uploaded and synchronized to the remote database.
Benefit	i. DB Handler can be reused to store data to the MS SQL Server since both MS SQL Server and MS Access support the SQL standard. ii. The performance of MS Access is better than file-based storage methods (e.g., XML). iii. This design option can enrich the development team with the experience of using MS Access as a database, which may be helpful to other ATS projects.
Cost	$1 + 2.25 = 3.25$ person-days
Principal	1 person-day
Interest	$2 \times 0.9 + 0.5 \times 0.9 = 2.25$ person-days

Change Scenarios	**#**	**Scenario description**	**Consequence**	**Scenario interest**	**Prob.**
	1	Synchronize the data in local storage files to the remote database server	Develop functions to query test results from MS Access and test the functions	2 person-days	0.9
	2	Add new tables (around 10) to record the new test results in MS SQL Server	Add the new tables in MS Access and test them	0.5 person-days	0.9
Architecture Diagram					

Test result saver
DB handler
MS SQL server
MS access

History Created: 10-11-2012 by Tom

Figure 2: ATD Item

compromises both maintanability and evolvability. In order to measure its impact on software projects, Value Oriented Architecting method, uses its own ATD conceptual model, shown on Figure 1. By analysing the ATD Item a decision is taken for the future proceedings of the project.
ATD Item is defined by [6], as a basic unit which is inquired by an architecture decision that compromises one of the system quality attributes mentioned above. Through the ATD items, all the possible scenarios are analysed, as well as estimating the interest and probability of them to happen. The result from this detailed analysis of ATD Item is used on the next step of this method, named ATD Management. First we demonstrate the usage of ATD Item by an actual use case taken by a software project decision (shown on Figure 2).
The ATD Management process consists of several steps, each of which deals with specific characteristics of the ATD Item. The following steps are the following:
•**ATD Identification -** As presented above, the corresponding ATD Item is identified. An Item is created when a problem occurs, and it starts to increase the costs and risks to the project. The Item is "labaled", and based on its *rationale* it is categorized as a threat to maintanability or evolvability.
•**ATD Measurement -** Analyzes the ATD Item considering its costs, benefits and possible changes of scenarios. The last ones are analyzed regarding the new features that are going to be added to the system, the already existing main-

tanance tasks and the new requirements. Difficulties are faced especially when new unpredicted requirements have to be added. They might change the whole architecture or a significant part of it. In these cases the measurements are based on the interest of related ATD items. This is done in order to see the scale of changes to the system.

•**ATD Prioritization -** On this step, all the ATD items are sorted according to some criteria which may varies from company to company (usually depends on the business goal and preferences). Considering that it is quite impossible to solve an ATD item without influencing the other ones, some prioritization on which one has to be solved first is necessary to be made. While performing this activity several aspects of ATD item has to be considered: 1) total costs of resolving it, 2) the ratio between cost and benefit of an ATD item, 3) the interest rate, 4) for how long has an ATD item been unresolved, 5) the complexity of solving it. What is actually measured in this step, is based on the experience and approximation of the costs. However there exists tools on which we can rely on approximately measure each of the factors used in prioritization.

•**ATD Repayment -** Sometimes making changes to the existing architecture is not an easy decision. When an ATD is identified, its negative impacts are to be eleminated or mitigated, in order to reduce the costs on the project. In such cases, when the impact is dramatically high, only several parts of architectural debt are solved, while the others are simply postponed to a second time. For the part that is solved, in a certain way it repays its costs to the project, and at the same time a less complicated item is created for the unresolved part.

•**ATD Monitoring -** This step garanties the closing loop of ATDM. Monitoring over time of all the unresolved items is crucial in estimating the costs, impacts and efforts that has to be made to fix them. Documented files of all the ATD items are keept, and compared repeatedly to check if any update or new item occurs.

As it can be seen ATDM is organized iteratively and after each iteration a release, increment or a Sprint in Scrum is created.

3.3 Dependency Metrics on measuring ATD

A software architecture is seen as a set of principal design decisions that are "connected" together. Based on this concept, the dependency metrics method transforms the architectural components and concerns related to that into an *augmented constraint network(ACN)* [7], [8]. This graph is further used to model the constraints among design decisions and environmental conditions. Design decisions cover all the decisions taken during iterative process of a software development. Environmental condition is a broaden concept which involves a set of strategies, including the work environment, the evaluation and procurement of hardware equipment, the provision for immediate access to computing resources through local area networks, the building of an integrated set of tools to support the software development life cycle and all project personnel, and a user support function to transfer new technology, [1]. Referring to [7], it is possible to extend ACN model to a new Extended ACN (EACN) model. From the new one it is possible to derive the relations among components, connectors, interfaces and concerns. Referring to [7], an ACN is defined as a set of constraint network (CN), a dominance relation (DR) and

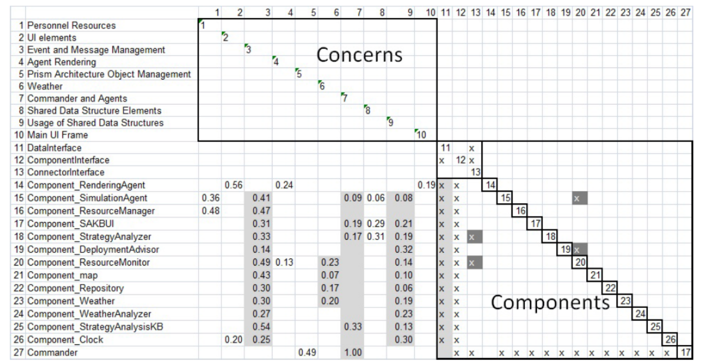


Figure 3: DSM Representation of ERS-Architecture

cluster set (CS), [so $ACN = \langle CN, DR, CS \rangle$]. CN itself represents a set of variables V, domains D, and constraints C. The variables are used to represent classes, algortithms or other concerns. The domain of a variable defines its values, by giving them a possible choice withing a certain dimension. In more formalized definition, the $CN = \langle V, D, C \rangle$. Another component of ACN is dominance relation (DR), which models an asymetric relation among decisions, formalizing the concept of design rule. The last component of ACN is cluster set (CS). In its own it introduces the modularity concept, which is essential for software designing. CS has many clusters, representing different stakeholders views of the design.

Having created the ACN (EACN), leads us to the *pairwise dependence relation (PWDR)*. This is formally defined as: PWDR $\subseteq$VxU and if (u,v) $\in$ PWDR, then v depends on u [7],[8]. This means that if u changes then v has to change in a way that the consistency of constraint network is retained. Finally, based on PWDR, ACN and a clustering of it, the design structure matrix is derived. This one is a square matrix, in which columns and rows are labeled. Each cluster of ACN is used to determine the DSM variable order. For example, if (u,v) $\in$ PWDR, then the cell in row v and column u will be marked to show their dependency. The blocks along the main diagonal are used to show the clusterings of ACN. For the EACN construction, and other formal details regarding connectors and concerns, refer to [7]. In order to show how all the architectural decay instances are modeled using PWDR, which can be visualized using DSM, we take an example from [7]. Other examples can be found on [8] and [2]. This example shows ERS - Emergency Response System, which is designed using C2 architectural style, implemented in JAVA usding PrismMW [7]. A typical characteristic of C2 architecture is its layered design. Communication among layers is done tramit message passing. For the sake of space, we are not going into the details of this architecture (the full description can be found online or on official publication by Malek). In Figure 3, is presented the corresponding DSM of ERS example. From the picture it is visible the clustering in two top level modules, the first one has 10 variables and the second one 17 ones. The second one is also clustered into 15 other clusters. These are represented with the bold squares in Figure 3. The first cluster represents the 10 concerns found on the system, whereas the next block represents 3 architectural elements and other components. The dependency between a component being involved in a

specific concern is represented by the column index. The probability of this dependency is shown by the numbers writen on the cells. There exists a language named LDA (Latent Dirichlet Allocation) which is used to make this calculations. From the DSM representation of EACN, we can easily observe multilevel of dependencies. For example: components like DeploymentAdvisor and SimulationAgent depend on ResourceMonitor, which itself depends on ConnectorInterface. Additionally, we can also observe that all the components depend on DataInterface (because at least once they have to rely on Sending/Receiving Events). So, summing up, this method is used in creating a general understanding of the dependencies among components, values and concerns. By DSM and LDA all the possible scenarios can be depicted and further analysed.

4. COMPARISON OF THE APPROACHES

The reason for focusing on each of the three above mentioned methods is because we believe that their approaches to correctly measuring technical debt is more intuitive than the other ones. Each of them is different from the other two, so we think that we cover a wide area of recent studies focusing of approximetely measuring ATD. In this section we make a comparison of them by focusing on architectural elements used in these methods, measurement units, complexity, time for refactoring and maintanance, and outputs. The reason for focusing on these attributes is that these give general understanding of each of the methods. Also, these are key attributes which are of interest of our field and within scope of this paper. For simplicity we are referring to each of the methods by using some abreviations: Modularity Metrics (**MM**), Value-Oriented Architecting (**VOM**) and Dependency Metrics (**DM**).

• **Architectural elements -** The MM method, bases its measurements in source code level of the projects. From this are analyzed the commits for ANMCC. On the other hand, DM, which is based in constructing the augmented constraint network (ACN), uses software components and connectors among them to build the pair-wise dependency relations (PWDR). This one are used to build the dependency matrix, that is used for evaluating the ATD. In contrast with the first two, VOM is based on ATD-Item concept, that is created on purpose for estimating the costs of ATD, and is further used for its management.

• **Measurement Units -** As the name suggests MM measurement units are modularity metrics (IIPU, IIPE, IPCI, IIPUD, IIPED, IPGF, ANMCC). These shows the degree of which a system or computer program is composed of discrete components such that a change to one of them has minimal impact on other components. On VOM, the measurement unit is total interst, calculated by the formula (1):

$$total_{Interest} = \sum_{k=1}^{n} I_k \times P_k \tag{1}$$

A detailed explanation of each of the terms mentioned in this formula is shown earlier on this paper. The outcome from it are monetary value of ATD. DM measurement units are basically, the ones which are used to make comparison among dependency metrics, such as probability that one component is dependent on other ones, etc.

• **Complexity -** Taking into consideration all the three methods, we believe that DM and MM have the same level of complexity in comparison with VOM, which requires less computation and data than the first two. We see complexity, as the amount of effort needed by humans or computers to collect and interpret the data needed to come up with some results. DM and MM complexity is based on respectively computation for constructing the ACN and IPGF/IPCI. Both of them are based on actual state of source code and dependencies among software components. This makes it possible to directly compute the results, leading to a fast analysis step.

• **Refactoring and Maintanance -** MM is based on source code analysis, this means that refactoring and maintanance doesn't require that much time and effort to be performed. The only problem that has to be taken into consideration is the time on which IPGF/IPCI and ANMCC is measured. Regarding the VOM method, refactoring require a lot of time and effort due to the elements that has to be taken into consideration for its analysis. If a component is replaced or updated, then the calculations, analysis, has to be recalculated from the beginning, by backtracking all the possible consiquences. The DM method doesn't require any extra efforts for refactoring. However, maintanance costs, especially when the level of dependencies between one component and the others is high, and that specific component is replaced or modified. That will cost an update of all the related components, which increase the maintanance costs.

• **Expected Result -** MM returns the values of IPGF and IPCI (recent papers suggests that IPCI is enough for determing the negative correlation with ANMCC). The VOM method evaluated all the ATD Items, and picks the one whose costs are higher, such as the overall costs of the project will not increase with the passing of the time. Regarding the outputs of DM, the dependency matrix is analysed. From this "concerns", "components" and relation among them is expressed in percentages. Matrixes are compared with each other, and the cheapest path represented by each of them is picked as solution for redusing ATD costs.

5. DISCUSSION

The purpose of this paper was not only to give a general overview of the techniques used to detect, measure and reduce architectural technical debt, but also to make a comparison among them. These three techniques were chosen so to cover a major area of recent studies in the area of architectural debt. Modularity metrics usually are used when sources are available to be analysed. It represents a fast way of estimating ATD, by telling if it is high or low, but no exact value of its costs is known. Another drawback, of this method is that it estimates the costs of only recent developments, by not giving any further information of future estimations, or ways how to reduce it. Value oriented method, as it is understood from its name, is focused only on the value part of architectural debt, by mainly focusing on the causes of this debt. Furthermore, this method describes all the possible future scenarios of the architectural changes to the software system. It evaluates the cost of ATD Item, and tries to resolve it in a specific order, depending on its priority. This methods turns usuful only when the documentation is available, the costs of individual architectural components are clearly stated, and objective of the project is well defined. For other software systems (small projects, or fast evolutive once), this method fails in estimating ATD value, as the system changes faster than the

time required to analyse each ATD issue. Last but not the least is dependency model. It is usually used to estimate the level of dependencies between the architectural components and concerns. The strong point of this method lays on its simplicity of interpretation through DSM. Also, by calculating automaticaly the probabilities of these dependencies, developing teams will be able to understand the spread of their problem, the components that will directly or indrectly conditioned by the changes and so forth. Knowing this characteristics of these methods increases the chances of chosing the appropriate technic according to the given situation. In the table below (Table 2), is presented a general overview of the above comparison:

	MM	**VOM**	**DM**
Architectural Elements	Source Code	ATD-Item	Components and connectors
Measurement Units	Modularity Metrics	Total Interest($)	Probability
Complexity	Relatively Low	Relatively High	Relatively Low
Refactoring & Maintanance*	Low	High	Moderate
Expected Result	IPGF& IPCI-which are negatively correlated with ANMCC. This estimates ATD costs.	Make a prioritization of ATD Items based on the interest and future risks of that item.	Visualises all the dependencies among components and concerns. Shows the probability that one component will depend on that conncern.

Table 2-*Comparison in terms of efforts needed for the task

6. CONCLUSIONS

In this paper we stated that Modularity metrics is a valid way to measure ATD inderectly, through IPCI and IPGF. This is a good substitute to ANMCC which is hard to calculate. Furthermore, we can test in the future studies that only IPCI will be necessary to be measured, as it is more negatively correlated than IPGF. Also, extending our results into other projects rather than C# and developing IDE tools for automatic calculations of IPCI and IPGF. Regarding the Value Oriented Architecting method, we can say that it is relatively new on itself. It introduces a completely new conceptual model based on ATD Items, and ATD Management system. It can be used for facilitating the decision making and decision evaluation in a value oriented perspective. ATDM can be used in the future as a template and documentation for the whole process of software developing, including changes of scenarios and solutions to the individual problems. Another method used to detect the architectural decays is by constructing the Dependency Models (for example DSM). Although it was proven that it is efficient on constructing DSM based on the lines of code, in the future it is possible to conduct research on constructing DSM by UML models. Based on that, we can derive dependeces directly based on architectural components, by neglecting code behind them. With this paper we think that we have covered the most important points of all these methods focusing on possible scenarios where they can be used. Further studies have to be conducted in the future to develop each of them, and increase the validity of the respective methods.

7. REFERENCES

[1] B. W. Boehm, M. H. Penedo, E. D. Stuckle, R. D. Williams, and A. B. Pyster. A software development environment for improving productivity. *Computer*, 17(6):30–44, June 1984.

[2] J. Brondum and L. Zhu. Visualising architectural dependencies. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 7–14, June 2012.

[3] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[4] W. Cunningham. The wycash portfolio management system. *SIGPLAN OOPS Mess.*, 4(2):29–30, Dec. 1992.

[5] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou. An empirical investigation of modularity metrics for indicating architectural technical debt. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 119–128, New York, NY, USA, 2014. ACM.

[6] I. Mistrik, R. Bahsoon, R. Kazman, and Y. Zhang. *Economics-Driven Software Architecture*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[7] R. Mo, J. Garcia, Y. Cai, and N. Medvidovic. Mapping architectural decay instances to dependency models. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, MTD '13, pages 39–46, Piscataway, NJ, USA, 2013. IEEE Press.

[8] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *Proceedings of the 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, WICSA-ECSA '12, pages 91–100, Washington, DC, USA, 2012. IEEE Computer Society.

[9] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 891–900, Piscataway, NJ, USA, 2013. IEEE Press.

[10] M. Shahin, P. Liang, and Z. Li. Do architectural design decisions improve the understanding of software architecture? two controlled experiments. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 3–13, New York, NY, USA, 2014. ACM.

Monitoring Heterogeneous Systems. Current State of the Art and Remaining Challenges

Nikola Velinov
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
nikola.velinov@rwth-aachen.de

Ana Nicolaescu
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ana.nicolaescu@swc.rwth-aachen.de

ABSTRACT

The current trends in software development demand products with extensive functionality while, minimizing development time and resources. This is usually achieved by reusing existing systems and libraries. As a result, the architecture of such products and its description play a crucial role in achieving the quality attributes demanded by the stakeholders. However, during the development cycle, the implemented architecture could drift away from its description due to undocumented implementation decisions. Therefore, a mechanism to verify the systems' conformance to its architectural description at every stage of the product life cycle is needed. This paper focuses on identifying the problems that are faced when trying to extract information about the interactions within and between components in heterogeneous systems. By studying the OpenWebRTC project, the paper examines current available techniques for collecting run-time tracing data and argues about their capabilities and applicability.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Software Architectures, Architecture Reconstruction, Component Interaction, Heterogeneous Systems

1. INTRODUCTION

The architecture of a software product plays a major role in achieving the qualities demanded by all stakeholders. However, it is usually defined in the beginning of the product's development cycle and tends to mismatch the actual implementation in the final phases. This usually happens because during the development phase developers make design decisions which are not part of the original architecture or are in direct contradiction with it. Also, in the advanced stages of the product development cycle, the effort is mostly focused on achieving the demanded functional and non-functional requirements and thus the architecture's documentation is not properly updated. This mismatch can easily lead to difficulties in maintenance and poor extensibility. The described problem can be tackled if one has always a clear and global understanding of the component structure and is able to trace the inter-component communications.

In this paper we concentrate on tools or frameworks that collect information from a running system to aid the process of reconstructing its architecture. Many techniques have been documented to collect such information from homogeneous systems. Custom code instrumentation is always applicable but is error prone and could drastically change the behavior of the software. Using aspect oriented programming provides a safer alternative to this, however it is very specific to the programming language used to construct the inspected system. Utilizing low level information such as bytecode or machine instructions has also been used. The DTrace framework, which will be inspected later in the paper, allows tracing software compiled to machine code without the need to instrument the code of the inspected system. Scripting languages like Python or JavaScript can easily be instrumented to provide the required data thanks to their purely object oriented nature.

Some methods of the aforementioned can be used to trace systems composed of multiple programming languages. In this paper we use the OpenWebRTC as a reference heterogeneous system. It connects a Linux daemon with JavaScript code executing in a browser through a Websocket. With such an example, we were not able to identify a tool, method or a framework that is able to trace this interaction and produce useful data. Hence, we approach this problem by identifying methods or tools that are suitable for tracing each technology and then argue about how those methods can be improved or extended in order to successfully trace the interactions in OpenWebRTC.

The remainder of this paper is organized as follows: in Section 2 the studied system is presented and the goals for the dynamic analysis are defined. In Section 3 the tools, techniques and frameworks for collecting the needed information are discussed. In Section 4 a conclusion is drawn and the benefits of collecting dynamic traces are discussed. Related work is discussed in Section 5.

2. THE EXAMINED SYSTEM AND GOALS

2.1 OpenWebRTC

Throughout this paper we will examine the OpenWebRTC

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

framework [1]. It provides WebRTC APIs to browsers and will serve as a good example of a heterogeneous system. The WebRTC standard is designed to give browsers simple JavaScript interfaces for Real Time Communication by utilizing peer-to-peer (P2P) architectures and access to audio and video streams from the host system. OpenWebRTC is developed by Ericsson as a research project. The project's goal is to create a universal module that gives access to WebRTC APIs to both browsers and native operating system components. Hence the module that realizes the WebRTC standard is a separate application from the consumers of its APIs. This differs from the approach taken by Google or Mozilla. They integrate the WebRTC realization in their browsers.

OpenWebRTC can be used via the "Bowser" browser, which is also an Ericsson developed project, as a plug-in for the Safari browser, or as a standalone daemon for Linux systems. In this paper we will examine a Linux system running the daemon and a browser that connects to it. Figure 1 describes the setup and shows a connection between two systems.

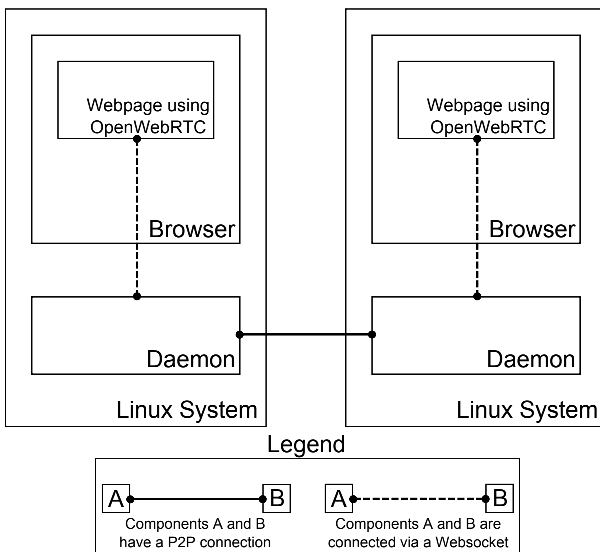


Figure 1: The OpenWebRTC Component Diagram

The dashed lines in this diagram represent the Websocket connection between the JavaScript code and the daemon. This Websocket is the only way the JavaScript in the browser can utilize the services provided by the OpenWebRTC backend. This means that every web page that requires the WebRTC API must have a specific piece of JavaScript code injected in it. This can be achieved with tools like Greasemonkey or Tampermonkey. Basically, this connection is realized by the browser, but for clarity it is indicated that it is done by the web page itself. The solid line represents a connection that is supposed to be established between two systems running the daemon if peer-to-peer services are used.

OpenWebRTC was chosen because the interaction between the different components of the system is challenging. There are a lot of techniques that can trace C programs or JavaScript scripts. But none of them can trace both simultaneously. Therefore, one must collect data separately and later process it in order to get a complete understanding of the interactions between the two components of the system.

In conclusion to trace this system we need a tool which can trace a JavaScript executing in a browser and a daemon written in C. It must also be able to link this information or generate homogeneous results in order to produce an interaction diagram. We have not yet discovered a single system capable of solving this task. Therefore we believe that identifying how current techniques that address this problem will be a valuable contribution.

2.2 Goals

Having defined the system that this paper is focusing on, now we present the goals that we would like to achieve with our run-time analysis:

- G1: Collect information about the function calls within the Linux daemon, information about the method or function calls in the JavaScript code. The collected data must allow a single operation to be traced thoroughly. By operation we mean any sequence of function calls and data packets that is a result of the JavaScript code invoking a service provided by the daemon.
- G2: We need to be able to specify pieces of the code that will be traced and pieces that will not be traced. E.g. we are not always interested in every single call to the C standard library, because some of them do not give any information for the interactions between the different functions within the program.

G2 is relevant because the collected data can be used to test if a certain interaction pattern is met. If one is interested in the internal workings of the daemon itself, one would only like to trace functions within the daemon and see the JavaScript consumer code as a single entity. On the other hand, one might need to trace all functions invoked when a specific event occurs, meaning all of the JavaScript and C interactions must be visible.

As was already mentioned before we have not identified a single framework, product or technique that satisfies the aforementioned goals. That is why the remainder of paper tackles each technology separately and then argues how to combine the results. We evaluate each technique by the following criteria:

- C1: The amount of manual code modification required to successfully gather the required information should be minimal and does not interfere with the existing functionality.
- C2: The overhead imposed by the collection of the required information is negligible compared to the runtime of the system.
- C3: It is possible to specify granularity of the data that is to be collected. E.g. trace specific functions, trace functions with common names, functions that receive a specific parameter.
- C4: The method is extendable to provide support for other technologies.

3. ANALYZING OPENWEBRTC

In this chapter the studied techniques will be presented and arguments about their applicability will be given. We

will only consider tools and techniques that are freely available or have a published paper that explains their functionality in detail. Some approaches are more sophisticated than others, but they all address our goal to trace function or method calls, or to trace the data transferred over sockets. First, tracing C code is explored, then the JavaScript.

3.1 The C Daemon

The C daemon relies heavily on external libraries for its functionality. Depending on the specific goals of the analysis, one may need to examine the libraries in detail or just identify them as black boxes. It is possible that the library plays a crucial role in the system's work flow. OpenWebRTC uses glib for its main loop. This means that this library must be traced in order to gain a deeper understanding of the function interactions within the daemon. One might also be interested in how the audio and video streams are fetched, for example, and thus need to inspect the gstreamer library as well. Henceforth, it is obvious that even before we begin our dynamic analysis we need some basic understanding of how the used libraries interact and what kind of information we need, so that we can collect data accordingly.

3.1.1 Compiler Flags

Most compilers offer the functionality to run functions at the beginning and at the end of each function executed. For this purpose the functions must be available to the linker at link time. The compiler inspected here is gcc and the flag that enables this is -finstrument-functions. The two functions that must be available are __cyg_profile_func_enter and __cyg_profile_func_exit. When called, these functions are passed the current address of the function being called and the caller's point of address. This information is not very useful because the address must be resolved in order to get the actual function name (usually done post execution). Furthermore, if one would like to inspect the parameters passed to the function, one must know the exact calling convention and perform complex stack or register analysis. Another drawback is that libraries and the whole project must be rebuilt with the relevant compiler flags. Finally, this technique can only be used with C or C++ and only with compilers that allow this. This is however a quick and easy to implement solution for small projects, which may require more post-processing of the collected data.

When we look at our defined criteria, this method is not extendable to anything else than C or C++ languages, meaning C4 is not satisfied. It is possible to selectively collect data for specific functions if the addresses passed as parameters are checked against the executable's symbolic information or if a flag is inserted to the declaration of each function to be traced. This means that our criterion for granularity of the collected could be satisfied, but it will require great programming effort. The instrumented functions will be called for every function executed unless specified otherwise in the code itself. This means there will be significant overhead involved. Hence, our criteria for the overhead C2 will not be satisfied. Finally, C1 (low manual code modification) is not satisfied, because if one wants to gather information for only a specific set of functions, one must either manually add small amounts of code to them or one must implement a lot of logic in the __cyg_profile_func_enter and __cyg_profile_func_exit functions.

Applying this technique in OpenWebRTC will first require modification of the build system for every library that shall be traced. Furthermore, once the data is collected it will have to be post-processed in order to extract meaningful information. In short, it will require a lot of effort to trace the daemon and the JavaScript scripts will remain unaddressed.

3.1.2 Aspect Oriented Programming

Another approach is to use Aspect Oriented Programming (AOP) to instrument the code. This is an approach that is researched in [2] and [11]. A few projects exist that create an additional compiler that instruments the code automatically with functions, which allow programmers to use aspects in C/C++ programs. To name some: AspectC, AspectC++, AspectX/Weaver, Arachne, aspicere etc. In this paper we will not concentrate on a specific one but we will argue about their common behavior with respect to our goals.

With this approach, one also needs to rebuild all libraries that the program depends on and are interesting for the analysis. However, this approach gives the opportunity for information gathering, without manually editing the code of the inspected system. Function names can be easily recovered as well as information about the parameters of the function calls. The instrumentation utilizes a so called weaver. This takes the source code and the aspect information as input and produces instrumented source code. An exception to this is Arachne which actually uses dynamic machine instruction insertion into a program. The latter is however not stable when used with different processor architectures simply because it is created to work with a specific one. Instrumenting the source code is a much more stable approach, because it relies in the robustness of the compiler.

The downside to using this approach is that the build system must be heavily modified, because none of the aforementioned tools provides a complete solution for the weaving and linking process. Depending on the build mechanism used for the system, this could be a big challenge.

To summarize this approach we should note that C1(low manual code modification) is partially fulfilled due to the fact that the build system will inevitably change and one must write the aspect oriented code for the collection. The overhead imposed by this approach depends on how many functions are traced and how much information is gathered. Hence, our criterion for overhead C2 could be fulfilled. Selective tracing is achievable, because enough data is available for the code in the aspect to utilize. One can extend this approach to other technologies. For example ApsectJ is an aspect oriented extension to JAVA. Languages like Python or JavaScript can be easily adapted to the aspect oriented paradigm due to their dynamic nature.

Applying this technique in OpenWebRTC will require heavy modification of the build system. However, in contrast to the previous approach, this one will generate information that can be directly fed into another tool which can be used to reconstruct parts of the architecture.

3.1.3 DTrace

The DTrace Framework was initially developed by Sun Microsystems. It is a comprehensive dynamic tracing framework originally developed for Solaris and later partially ported to other Linux-like systems [3]. It consists of a core kernel module which manages the collection of the data. The framework itself is not bound to a specific technology. Therefore it does not instrument the programs itself, but delegates

the task to the so called providers, which are loadable kernel modules for the framework. These providers publish probes in the DTrace framework, which can then be enabled to collect actual data. The probes are the actual points of instrumentation. They "fire“ when an event occurs and they collect data, which is passed to their respective provider. An example for a probe is a probe that fires when a function is entered. It provides information like the name of the function, its parameters, the thread that executes the function and so on. Figure 2 shows the architecture of DTrace.

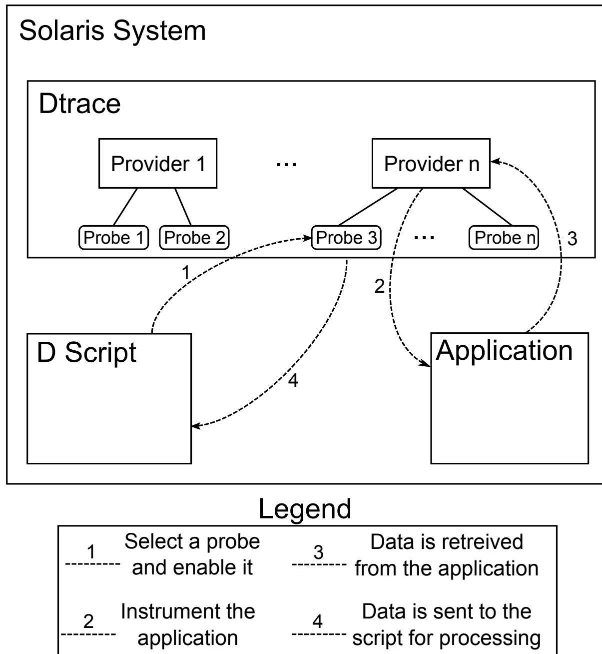


Figure 2: The DTrace Architecture

The probes that shall be used to sample information are requested by the so called D scripts. They are written in the D language, which is defined and executed by the DTrace framework. It is a language with syntax similar to C and allows arbitrary operations on the collected data. With this technology the framework allows the so called speculative tracing. This means that data can be purged of unnecessary noise in the process of collection, effectively reducing the amount actually stored.

Once a probe has been selected, the respective provider starts instrumenting the application that shall be traced. The collected information is then sent back to DTrace and passed to the script for processing.

DTrace distinguishes between two types of tracing:

- Dynamic Tracing: The providers are able to instrument the inspected system at run-time and provide the framework with the necessary data.
- Static Tracing: The inspected system contains manual code that simulates the instrumentation and manually sends the necessary data a stub provider, which forwards it to the DTrace framework.

For our use case we need to instrument a C application. A provider called "pid“ can be used. It can dynamically instrument functions by hooking to their entry and exit points. The information about the function names is retrieved from the inspected executable or library, which means that it should contain debug symbols. Furthermore, "pid" can hook to a running process without restarting it, so it requires no down time. It also does not require any changes to the code and hence it is not expected to influence the behavior of the application.

Comparing DTrace to our criteria yields promising results. No manual code modification is required if one uses dynamic tracing, hence our criterion C1 is completely fulfilled. The amount of overhead imposed by DTrace is proportional to the amount of data collected which can be easily tweaked by the probes utilized, so we consider C2 to be fulfilled. C3 is completely satisfied, because one can gather as much information from a running system as one requires and even filter it during the collection process. The concept of providers and static tracing also allows one to trace arbitrary technologies. Henceforth, we can consider C4 satisfied as well.

However DTrace has some major drawbacks. First it is only available for Solaris, FreeBSD and MacOS X. Furthermore, the providers utilize shared objects to contact the DTrace framework in the kernel. This means that if one would like to trace JavaScript for example, one must provide wrappers for the interfaces in the shared objects in the JavaScript environment. This means that each JavaScript engine that will be traceable with DTrace must have such wrappers within it. This shall be discussed in detail later, in the chapter which concentrates on the JavaScript code.

The OpenWebRTC framework is meant to be built on MacOS and Ubuntu[1]. Therefore, if we would like to test this approach we must adapt the build process to work on Solaris. However, DTrace is the only studied product that can trace the software without any code modification and, as shall be discussed later, can be adapted to work with JavaScript as well.

3.2 The JavaScript Code

JavaScript is a prototype-based scripting language that treats functions as first-class objects. This means that tracing can be achieved in a more straight forward way. One can simply create wrappers that trace the execution for each existing function by iterating and altering all the functions in the global name space. Other approaches utilize built-in capabilities of browsers and plug-ins like Firebug or standalone applications like spy-js[14]. However these methods are designed for debugging purposes and may have difficulties recording the collected data.

In this chapter we will examine two approaches that actually collect tracing data, with the purpose of storing it.

3.2.1 DTrace Providers

As already mentioned DTrace has support for arbitrary providers and can be used to trace the execution of JavaScript code. We found two sources that discuss how this is achieved: [12], [13]. [12] edits the Firefox's JavaScript engine SpiderMonkey manually by adding static probes in it. [13] uses a flag in the build configuration to include static probes for DTrace. This means that the developers of Firefox has already taken DTrace into consideration and has code that allows tracing JavaScript.

Both approaches are able to collect information about the function interactions within the JavaScript code. One can also identify functions, through which the JavaScript inter-

acts with the daemon and can relate the execution traces.

We shall not go into detail in comparing this approach with our criteria, because we have already discussed Dtrace in Section 3.1.3. We will only note that the presented methods use the static tracing mechanisms of DTrace, which means that providers must be present in the system.

With the DTrace providers one can trace the JavaScript code of OpenWebRTC without any modification of the scripts. Combining this with the previously mentioned advantages, DTrace seems to be a promising framework. However, we once more mention that it is limited to only Solaris based systems.

3.2.2 Jalangi

Jalangi is a framework that supports dynamic analysis of JavaScript scripts. It is focused on recording sequences of execution of JavaScript code and later replaying them for analysis purposes [8]. When replayed additional checks and observations can be done in order to identify problems or errors in the code.

Our interest in this tool lies in its ability to record the execution of a script. The approach taken by the authors of the tool however is tailored for their needs of replaying the recorded execution. In order to do it, they keep track of the states of variables and record the execution in a more data oriented manner. This means that extracting information about the function interactions from the records could be challenging.

In conclusion, the authors of the tool report twenty-six times slower execution speed when collecting data. Henceforth, our criterion C2 is not satisfied. This technique is addressed solely to JavaScript, so C4 is not satisfied as well. Granularity can be achieved, because the tool allows collection of data from only specific pieces of code. Finally, no manual code modification is required so C1 is satisfied.

Jalangi can be used to record the script execution of the JavaScript part of the OpenWebRTC framework without any code modifications. However, it is not certain how the collected data can be exported for further processing.

3.3 Summary

Here we present a table that compares the studied tools and techniques against our defined criteria.

Method	C1	C2	C3	C4
Compiler Flags	*	-	*	-
AOP	*	+	+	+
DTrace	+	+	+	+
Jalangi	+	-	+	-

Table 1: Summary of the Studied Tools

+	criteria is met
-	criteria is not met
*	considerable effort is required to meet criteria

As was already discussed, the compiler flags approach requires a lot of efforts, before actually producing meaningful results. It is also not extendible to to technologies other than C/C++. The aspect oriented programming still requires modifications in the build system, however it is extendible to other technologies. DTrace satisfies all defined criteria and is a promising framework. However its major drawback is that most of its functionality is only available in the Solaris operating system. Jalangi is a solution that works for JavaScript only. On one hand it provides the valuable functionality of replaying a script's execution. On the other extracting the actual data with respect to the function calls within the script is challenging.

4. CONCLUSION

Collecting run-time tracing data from applications can provide two valuable possibilities:

- It can be used to visualize the relationships between different components of a single system.
- It can be used to visualize the interactions between the components of a system for a specific test case scenario.

All the techniques and tools discussed so far provide tracing capacities. Some of them provide more sophisticated solutions, while others are more basic and require more manual work in order to produce useful data.

In this paper we were looking for solutions that require minimal manual code modification in order to collect data. This is useful for systems that are currently in heavy use, for which a change would impose risk of failing to fulfill their business goals. DTrace is valuable in this sense, because of its ability to use dynamic tracing by simply attaching to running applications. The dynamic probes however must be initially present in the system that runs the application. For example, if one would like to dynamically trace JAVA code with Dtrace, then a JAVA Virtual Machine that contains the corresponding providers must be available in the system. And as was previously noted, most of the providers of DTrace are only available on the Solaris OS.

For systems in that are being built the situation is different. They are probably not operational during their development and hence time could be invested to make the systems traceable.

There are interesting results published by Google on the topic in [9]. The authors present their experience in using the Dapper system, which is a large scale distributed systems tracing infrastructure. Their approach requires developers to use specific middleware or libraries and follow specific guidelines when coding in order to add tracing capabilities in their systems. They have also developed a "Depot API", which provides direct access to all the collected traces. With this tool they have developed numerous applications that interpret the data for different purposes, ranging from profiling to understanding system behavior.

The authors also report that during re-engineering of certain service, the results from the Dapper and the Depot API were valuable in the following aspects:

- The developers were able to trace the performance of the service as it was being built and immediately identify optimization opportunities.
- They were able to optimize their database accesses.
- They were able to redesign their dependencies and their interactions in order to minimize the load on them.
- They used Dapper for validating test scenarios.

Henceforth processing the traced data is as important as collecting it.

[4] provides an extensive overview of the methods available for this. However, they do note that a tool that processes data of heterogeneous systems is still not available. This problem is mentioned in older reports [6], meaning it is a persistent one. The different programming paradigms that are more or less imposed by the used programming language are one challenge that must be overcome. A tool that would support multiple technologies would have to identify the paradigm in order to produce a correct diagram representing the interactions that are happening.

Further research would be needed in order to identify how the traced data should be saved in a unified format, so that the different technologies can be linked together in order to provide information for the complete interactions between them.

5. RELATED WORK

Investigating the problem of tracing software execution during run-time is not new.

[10] analyzes the challenges of monitoring heterogeneous systems utilizing Java and C/C++. They manually create probes to collect the required data and then construct a framework which monitors the behavior of the software during run-time. They proceed to evaluate their method based on flexibility and performance.

[7] test the conformance of state machines in software of Electric Control Units to their corresponding models. They utilize create probes manually and insert them in the C/C++ code to collect data. They then proceed to write scripts that run tests and use the collected data to test weather the state machine implementation corresponds to their defined model.

[2] provide extensive analysis on the available possibilities for AOP in C/C++ languages. They proceed by presenting their AOP compiler for C called "aspicere". After it is presented they report their experience in using AOP techniques for recovering architectures of legacy systems. They also discuss the difficulties they meet when integrating their compiler with the existing build system of the legacy system and argue about the threats to the validity of the results that they gather.

[5] provides a survey on existing tools for analyzing collected traces. The tools discussed are: Shimba, ISVis, Ovation, Jinsight, Program Explorer, Avid, Scene, and Collaboration Browser. The argue about how execution traces can be modeled and the possible levels of abstraction of diagrams extracted from a system's traces. They proceed with discussing how the studied tools manage large traces with methods like filtering or data collection techniques.

[4] Gives an extensive overview on the process of Software Architecture Recovery (SAR). They define taxonomies and goals of the SAR process. They also categorize the SAR processes by their approach – whether it is top down, bottom up or hybrid. SAR inputs are also discussed and they give an extensive overview of the tools available for the different types of input – dynamic or static. They categorize the techniques used to perform the actual SAR process and finally discuss the outputs that are produced.

6. REFERENCES

[1] OpenWebRTC. http://www.openwebrtc.io/, 2014.

[2] B. Adams, K. D. Schutter, A. Zaidman, S. Demeyer, H. Tromp, and W. D. Meuter. Using Aspect Orientation in Legacy Environments for Reverse Engineering using Dynamic Analysis -An Industrial Experience Report. Technical report, Software Engineering Research Group, Delft University of Technology, 2008.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[4] S. Ducasse and D. Pollet. Software architecture reconstruction: A process-oriented taxonomy. *IEEE Transactions on Software Engineering*, 35:573–591, 2009.

[5] A. Hamou-Lhadj and T. C. Lethbridge. A survey of trace exploration tools and techniques. In *Proceedings of the 2004 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '04, pages 42–55. IBM Press, 2004.

[6] L. O'Brien, C. Stoermer, and C. Verhoef. Software architecture reconstruction: Practice needs and current approaches. Technical report, Carnegie Mellon, 2002.

[7] M. Saadatmand, D. Scholle, C. W. Leung, S. Ullström, and J. F. Larsson. Runtime verification of state machines and defect localization applying model-based testing. In *Proceedings of the WICSA 2014 Companion Volume*, WICSA '14 Companion, pages 6:1–6:8, New York, NY, USA, 2014. ACM.

[8] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs. Jalangi: A selective record-replay and dynamic analysis framework for javascript. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 488–498, New York, NY, USA, 2013. ACM.

[9] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[10] M. Vierhauser, R. Rabiser, P. Grunbacher, C. Danner, S. Wallner, and H. Zeisel. A Flexible Framework for Runtime Monitoring of System-of-Systems Architectures. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 57–66. IEEE, 2014.

[11] A. Zaidman, B. Adams, K. De Schutter, S. Demeyer, G. Hoffman, and B. De Ruyck. Regaining Lost Knowledge through Dynamic Analysis and Aspect Orientation - An Industrial Experience Report. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, pages 91–102, Bari, Italy, 2006. IEEE.

[12] Dtrace meets JavaScript. https://blogs.oracle.com/brendan/entry/dtrace_meets_javascript, 2006.

[13] Optimizing JavaScript with DTrace. https://wiki.mozilla.org/Performance/Optimizing_JavaScript_with_DTrace, 2008.

[14] Spy-js. https://github.com/spy-js/, 2014.

Are Code Smell Detection Tools Useful in Dealing with Technical Debt?

Johannes Krude
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
johannes.krude@rwth-aachen.de

Muhammad Firdaus Harun
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
firdaus.harun@swc.rwth-aachen.de

ABSTRACT

The term Technical Debt originated as a metaphor for shipping code of low quality. It compares typical consequences of shipping low quality code to the financial terms of interest and payback. Lately, some research focuses on identifying Technical Debt on code-level through the use of code smell detection tools. Many fundamental questions on this identification approach are still unanswered. Without answers to these fundamental questions it is unknown whether these identification approaches have the potential to provide a useful analysis.

The Technical Debt definition varies between different literature. For which of these definitions of Technical Debt is the existence of code smell actually a Technical Debt indicator? Technical Debt can be categorized according to different criteria. Examples for such criteria are the visibility to stakeholders and the intention in introducing the Technical Debt. Which kinds of Technical Debt can be identified by code smell detection tools? The Technical Debt metaphor focuses on terms like interest and payback. How does code smell relate to Technical Debt beyond identification? A lot of code smell research predates most Technical Debt research. Is code smell actually a useful indicator for low quality in software?

We have read a subjectively chosen subset of the available Technical Debt and bad code smell detection papers. We highlight promising Technical Debt identification techniques and show which techniques do not (yet) reside on solid ground.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*cost estimation*; D.2.8 [**Software Engineering**]: Metrics—*process metrics, product metrics*

Keywords

technical debt, code smell detection, literature review

This work with the exception of the figures is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

1. INTRODUCTION

The term Technical Debt originated as a metaphor by Ward Cunningham [3]. This Technical Debt metaphor compares consequences of shipping low quality code to the financial terms of debt, interest, and principal. It's early use had the intention of easing communication between technical staff and management.

Code smells are easy to spot indications for underlying problems in object oriented program code [8]. Here is an example:

```
switch (x) {
case 1:
    ...
    break;
case 2:
    ...
```

The usage of switch statements is often regarded as an indicator for a lack of using appropriate object oriented constructs. Switch statements should supposedly be replaced by object oriented polymorphism. Code smells were originally designed to be easy to spot by humans. Today, a wide range of automated tools exist to detect code smell. Lately, some research focuses on evaluating whether such code smell detection tools also identify Technical Debt [13, 6, 14].

But, many important fundamental questions on Technical Debt, code smell, and the identification of Technical Debt through code-level detection are still unanswered. For this paper, we drafted for important fundamental questions. Without answers to these questions it is unknown whether this identification approach has the potential to provide a useful analysis. We want to know whether code smell detection has the potential to usefully identify Technical Debt. It is also important to know the limitations on the analysis results of such tools. We read a subjectively chosen set of papers to evaluate the current knowledge. We investigate which kinds of analysis tools may provide which kind of useful analysis and which results are impossible to identify by such tools.

Our fundamental questions are explained in Section 2. We explain and discuss the reviewed literature and the findings on these questions in Section 3. In Section 4 we discuss the future research which is needed to definitely answer our questions.

2. DEVELOPING QUESTIONS

We want to evaluate whether code smell detection tools are useful in identifying Technical Debt. For this purpose,

we examine four fundamental questions on Technical Debt, code smell, and the identification of Technical Debt through code-level detection.

The term Technical Debt originated as a metaphor. But, Technical Debt as used in Software Engineering did not persist to be solely a metaphor. Nowadays, Technical Debt is often used as a term to describe certain aspects of artifacts in software. To distinguish which aspects are instances of Technical Debt and which aspects are not instances of Technical Debt, a Technical Debt definition is needed. In different Technical Debt literature different Technical Debt definitions are in use. These definitions vary a lot in substance and preciseness. Different authors state different positions on which aspects are instances of Technical Debt. Some definitions are rather unambiguously. Some authors state their definitions very vaguely. Some literature even omits an explicit Technical Debt definition. In these cases the reader is left with the task of implicitly deducting the authors Technical Debt definition from how he uses the Technical Debt term. For different definitions different aspects are instances of Technical Debt. An aspect which is a Technical Debt according to a Technical Debt definition may not be a Technical Debt according to another Technical Debt definition. This leads us to our first question on code smell based Technical Debt detection:

1. *For which Technical Debt definition is the existence of code smell possibly a useful Technical Debt indicator?*

This is the question of which Technical Debt definitions allow for a reasonable discussion on Technical Debt identification through code smell detection tools. This is also the question whether code smell detection can possibly be useful when interested in Technical Debt according to a specific Technical Debt definition.

Instances of Technical Debt can be categorized using different kind of characteristics. One categorization is on the stakeholders knowledge of existence of Technical Debt instances. This is the division into visible and invisible Technical Debt [10]. Another categorization is on the intention of introducing the Technical Debt instance into the software. This is the division into deliberate and inadvertent introduced Technical Debt [7]. Technical Debt is also categorized according to the development step it was introduced. Examples of these categories are architectural Technical Debt, design Debt, code-level Technical Debt, test Debt, and documentation Debt. Our questions on Technical Debt categories is:

2. *Technical Debt of which categories can possibly be detected by code smell detection tools on code-level?*

This is the question on the limits of code smell detection tools detecting Technical Debt on code-level. This is also the question on which kinds of results we can not expect from Technical Debt identification through code smell detection.

Code smells are code-level indicators for problems in software. These indicators manifest themselves on concrete parts of program code. Code smell detection tools may possibly identify Technical Debt associated parts of program code. A very important part of the Technical Debt discipline are rationales on future consequences which emerge from current states of software artifacts. These future consequences are in the Technical Debt metaphor expressed as interest and principal. An important Technical Debt question is on how to decide whether to repay a debt or not [1]. Identifying concrete parts of program code by itself does not necessarily carry any clues on interest and principal. In order to provide a useful Technical Debt analysis, a Technical Debt analysis tool should not be independent of possible future consequences. Therefore we ask the question:

3. *How does code smell relate to Technical Debt in terms of interest and principal?*

This is the question whether code smell gives indications on future problems in software. This is also the question whether code smell detection tools can help in software development management decisions.

Code smells are indicators for certain problems in software. Refactoring is usually proposed as a solution to these problems [8]. A wide range of literature discussing code smell detection tools for Technical Debt identification exist. Nonetheless, none of the literature we read gives concrete reason on why to consider code smell detection as a promising approach for Technical Debt identification. We therefore developed a small theory on why code smell detection is so widely considered for the use of Technical Debt identification. The original Technical Debt metaphor [3] speaks of low quality code. If this low quality code is not repaid, interest must be paid. We believe that certain similarities between common code smell beliefs and the Technical Debt metaphor exist. Code with occurrences of code smell is often regarded as low quality code. Refactoring this code can be viewed as a form of repaying the debt. If this code is not refactored, interest supposedly must be paid. This conclusion holds only if code smell actually is a useful indicator for problems in software. Thus, the following question arises:

4. *Are code smells a useful indicator for problems in software?*

This is the question whether code smell detection actually identifies aspects which either need to be repaid or interest must be paid. This is also the question whether Technical Debt identification through code smell detection has a chance to be a promising approach.

We phrased four fundamental questions on code smell detection in the context of Technical Debt identification. In the following we describe the knowledge related to these questions we found in existing literature, and discuss how this impacts useful Technical Debt identification.

3. FINDINGS

We read a subjectively chosen set of papers to determine the currently existing knowledge related to our four questions. This section lists these findings.

3.1 Question 1: Technical Debt Definition

Different Technical Debt literature uses different Technical Debt definitions. Some definitions are stated in very precise language, some definitions are stated rather vaguely. Regularly, Technical Debt literature even omits to specify their working definition of Technical Debt. Some random

examples of literature with various implicit Technical Debt Definitions are [13, 6, 14]. The popularity of the Technical Debt metaphor originates from folklore amongst practitioners [12] rather than from research. This may be the reason for these widespread implicit and varying Technical Debt definitions. It is rather hard to devise solid arguments based on facts which can only be read between the lines. Therefore, we ignore most implicit definitions. In the following we discuss 3 Technical Debt definitions in detail.

We start with a rather precisely stated but very broad Technical Debt definition [1]:

> *One way to understand technical debt is a way to characterize the gap between the current state of a software system and some hypothesized "ideal" state in which the system is optimally successful in a particular environment.*

There are two main concepts in this definition. Technical Debt is the divergence of a software system from an *"ideal"* state. This *"ideal"* state is characterized as enabling the software system to be optimally successful in a particular environment. These two concepts are very broad.

Not all software systems are evaluated for their successfulness in the same single standardized environment. Different software systems target very different environments. It can be easily thought of environments in which the absence of code smell is mandatory for success as well as environments in which the absence of code smell hinders success. A software development organization could be bound by a legal contract to deliver code without any occurrences of code smell. A code smell detection will indeed identify Technical Debt for this definition and environment. Let's assume a software system is developed for a highly memory constrained environment. Replacing a single switch statement in the code by a collection of polymorphic classes may introduce a high burden on the memory usage. The abstinence from code smell will prevent this software system from being successful in it's target environment. A code smell detection tool will identify aspects which are kind of the opposite of Technical Debt for this definition and environment.

The *"ideal"* state which enables a software system to be optimally successful in a particular environment may not necessarily be unique. Two different *"ideal"* states may exist which both may enable a software system to be optimally successful in the same environment. Under these circumstances, a Technical Debt identification tool or code smell detection tool may identify aspects which are Technical Debt and are not Technical Debt simultaneously.

This Technical Debt definition is too broad to aid in meaningful Technical Debt discussion. It highly depends on the software systems target environment whether code smell detection tools have the potential to identify Technical Debt. This Technical Debt definition also seems to be a synonym for low software quality [11]. There is no usefulness in evaluating the same phenomenon twice just with different names.

We want to point to one specific implicit Technical Debt Definition. We found the term *code smells Debt* in the title of a paper [6]. Neither a general Technical Debt definition nor a specific code smell Technical Debt definition is mentioned in this paper. Nonetheless we believe based on the usage of this term, the authors of that paper have some kind of Technical Debt definition for which code smell instances are Technical Debt instances. For such a Technical Debt definition, the existence of code smell instances is always a Technical Debt indicator. But, whether the existence of code smell is a *useful* Technical Debt indicator depends on whether code smell detection itself is useful for something. We outline the general findings on the usefulness of code smell detection without referring to the Technical Debt metaphor in Section 3.4.

The next Technical Debt definition does only indirectly define Technical Debt. It only lists necessary but no sufficient conditions for an aspect to be Technical Debt. It is the most useful Technical Debt definition we have encountered in our review. The following excerpt from this definition puts important burden on Technical Debt identification through code smell detection [11]:

> *A design choice that has no permanent consequence on future cost of changes, that is, incurs no form of interest payment, probably should not be labeled as technical debt, but just as an alternative choice. The presence of some form of interest, either constant [...], or in the form of a balloon payment [...], should be an important criterion for deciding if a design approach is in debt [...].*

Therefore, for code smell detection tools to be able to identify Technical Debt, the presence of code smells must imply the presence of interest. Similarly, it is stated on requirements for Technical Debt tool support [4]:

> *In practice, the system must estimate the principal, the interest, and the probability to deliver the product on-time.*

With this Technical Debt definition, for code smell detection tools to be able to identify Technical Debt, the code smell detection tool must identify some interest. Whether code smell detection tools identify interest is not intuitively obvious. The findings on the relation of code smell detection to interest are presented in Section 3.3.

Some common Technical Debt definitions are too broad or too unspecific to reasonably work with. Based on the useful Technical Debt definitions we found, interest and payback are important parts of useful Technical Debt analysis. We discuss the relation of code smell to interest and payment in the finding on our third question.

3.2 Question 2: Technical Debt Categories

Technical Debt is commonly categorized according to different criteria. Figure 1 shows some such categories. We do not know whether all of these shown categories represent Technical Debt in all common Technical Debt definitions. To simplify matters in this section we assume they all do represent some Technical Debt. The following are the findings on the relation of useful Technical Debt identification through code smell detection to Technical Debt categories.

Code smells are indicators for problems in program code. They were invented to point to opportunities for refactoring of program code. Refactoring is the activity of improving

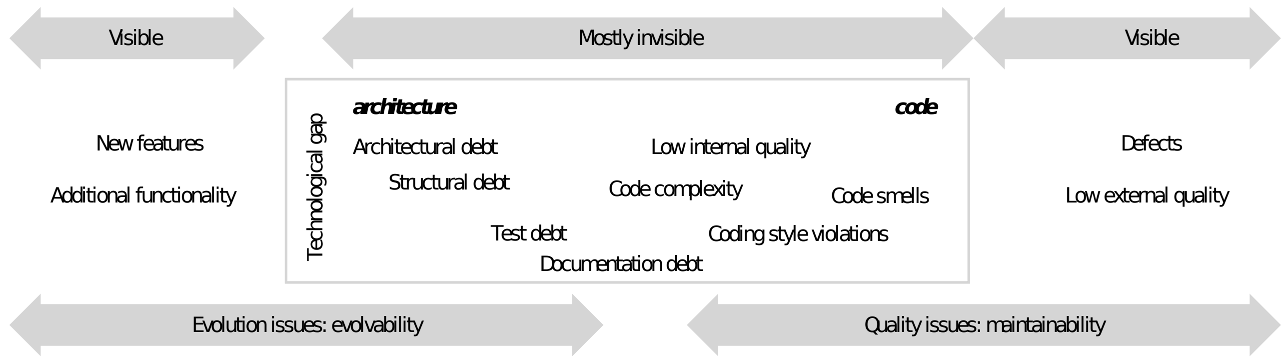


Figure 1: Some Technical Debt categories from [10].

an existing design [8]. Although code smell detection works on code-level, it primarily detects design-level problems.

Design is only one single step amongst many in the process of software engineering. Examples for Technical Debt categories corresponding to other steps in software engineering can be found in the *"Mostly Visible"* section of Figure 1. It can not be ruled out that there exist other non design related problems which typically coincide with problems detected through code smell detection. Nonetheless, it feels very unintuitive to use code smell detection for example to detect problems in documentation. We believe Technical Debt instances identified by code smell detection tools are mainly design Debt instances. This believe is also shared by others [6, 14, 13].

Code smell detection may help in improving software design. If interested in Technical Debt identification on any other level than design, other tools should be employed. The findings on the relation of code smell to design problems are discussed in Section 3.4.

One possible criteria for Technical Debt categorization is the visibility to stakeholders [1]. Whether some software lacks important features is typically visible to developers, managers, and costumers. Although a developer may know the poor design of a software, this low quality of the design is mostly invisible to manager and customers. Figure 1 shows some examples of rather visible and mostly invisible Technical Debt categories. We already covered the mostly visible Technical Debt subcategories in the previous paragraph.

Is it possible to detect visible Technical Debt through the use of code smell detection tools? Aspects like missing features or defects are deviations from explicit or implicit requirements. To check whether code and requirements match, both of them need to be examined and compared. But code smell detection is done on code-level only. Code smell detection results are independent of any specific requirements. We believe, code smell detection can typically not identify whether an important feature is not implemented. We present the findings on the relation of code smell to defect-proneness in Section 3.4.

The Technical Debt quadrant categorises Technical Debt according to two criteria [7]. The first criterion is intention. It differentiates whether a certain Technical Debt instance was introduced deliberate or inadvertent. The second criterion is caution. It differentiates whether a certain Technical Debt instance was introduced out of prudence or recklessness. Figure 2 shows how this constitutes a quadrant. Both intention and caution are mental states of human minds. They are not directly reflected in program code. We do not know of any approach to measure mental states on code-level. Therefore, code smell detection can not distinguish deliberate from inadvertent and reckless from prudent Technical Debt.

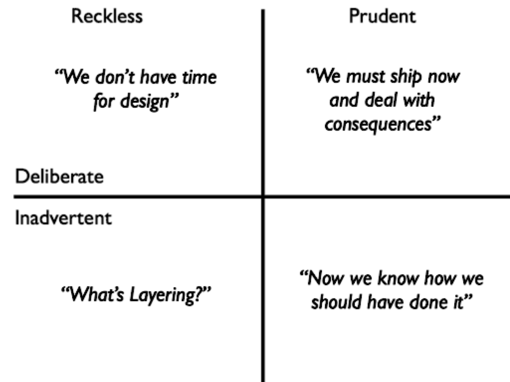


Figure 2: The Technical Debt Quadrant from [7]

Manifold Technical Debt categories exist. Code smell detection tools may identify design Debt. They may even predict defect-proneness. We do not believe in the ability of code smell detection to identify any Technical Debt which is not design or defect Debt. In our opinion code smell detection is not able to categorize the states of the minds of humans introducing Technical Debt instances. We outline the findings on defect-proneness and design Debt in the following Section 3.3 and Section 3.4.

3.3 Question 3: Interest and Principal

Reasonable Technical Debt definitions require Technical Debt instances to incur interest. An important factor in deciding whether to repay a Technical Debt is the economic difference between the interest and principal. Useful Technical Debt tool-support should therefore estimate the interest and principal. This estimation should be expressed as economic consequences [4].

We reviewed code smell detection based Technical Debt identification literature. Some of this literature measures change- and defect-proneness [14, 9]. Change- and defect-prone parts of program code are more likely to be changed in the future and involved in defects. The reviewed literature detected changes in program codes by comparing multiple revisions of the same software. Exactly those defects which were listed in defect tracking systems were examined. Both changes and defects are not directly economical consequences. None of the reviewed literature illustrates the relation of change- and defect-proneness to economical consequences. But we can imagine situations in which changes and defects lead to economical consequences.

Code smells are indicators for problems in program code. The reviewed literature [14, 9] examines whether the presence of code smell instances increase change- and defect-proneness. Neither the change- and defect-proneness of the same program code with removed code smell instances, nor the effort needed to remove these code smell instances, were evaluated. Useful Technical Debt tool-support should estimate and compare the interest and principal. Change- and defect-proneness, which are the consequences of not repaying the debt, correspond in some way to interest. We did not find any corresponding principal estimation.

Change- and defect-proneness are compared between different code fragments with and without code smell instances. These different code fragments do not implement the same functionality. Whether, a certain code smell corresponds to an underlying problem is dependent on the application domain and the chosen design patterns [5]. The reviewed literature [14, 9] compares the change- and defect-proneness of code implementing different functionality. I our opinion, code of the same functionality with and without code smell occurrences should be compared. This way, it could be evaluated whether a certain change- and defect-proneness results from presence of code smell and not from different functionality. We lack any literature which analyzes change- and defect-proneness according to our here presented criteria.

Zazworka et al. [14] investigated the impact of various code-level anomalies with respect to change- and defect-proneness. Among several other anomalies, they investigated 10 different code smells. Most of these investigated code smells are not part of the original 22 code smells from Fowler et al [8]. They found a correlation between the code smell "dispersed coupling" and defect-proneness. They did not find any correlation between any of the other 9 investigated code smells and defect-proneness as well as no relation at all between any investigated code smell and change-proneness.

Khomh et al. [9] investigated the impact of code smells on change-proneness without referring to the Technical Debt metaphor. They investigated 29 different code smells, most of which were not among the original 22 code smells. In the software they analyzed, classes with smells are more likely to be changed.

Some code smells correlate with sizes of classes. It is disputed whether code smell results should be normalized or not according to the number of lines of code [13]. The change-proneness results on big classes vary depending on whether such normalization is done.

A Commercial software exists which allegedly measures Technical Debt on code-level in terms of economical consequences [2]. The claims of doing so, are not accompanied by a substantial proof or explanation of the approach. Claims in advertisements without a proof can not reasonably be considered as findings.

Useful Technical Debt tool-support should estimate the interest and principal as economical consequences. We discovered only literature estimating the interest of code smell as the non economical change- and defect-proneness. Code smells were developed to be useful in finding opportunities for refactoring. Unfortunately, no results on comparing the costs of change- and defect-proneness to the costs of refactoring exist. For some code smells, a change-proneness correlation was shown. A correlation with defect-proneness was only shown for one single code smell. The following Section 3.4 presents the general code smell findings.

3.4 Question 4: Code Smells

Code smells are indicators for underlying problems in program code. In common software engineering belief, code smells are useful for improving code quality. For some, this maybe a reason to explore the potential of code smells in a Technical Debt context. The following are the findings on the general usefulness of code smells as code problem indicators.

Fowler et al. [8] originally introduced 22 code smells. The original code smells were intended to be combined with informed human intuition to provide inspiration for refactoring opportunities. Other literature and tool authors introduced subsequently their own code smells. Most of the code smells evaluated by Zazworka et al. [14] and Khomh et al. [9] are not among the original 22 code smells. Even fewer code smells exist wich are evaluated by both Zazworka et al. [14] and Khomh et al. [9]. Some of these code smells seem, in contrast to the original 22, to be more tailored to be detectable through software metrics. Since their code smells are mostly disjoint, their results are hardly comparable. Each of them may provide results on the usefulness of their selected code smells. None of them can claim to provide general code smell results.

Zhang et al. [15] provided in 2011 an overview of current knowledge on the usefulness of code smells. At that time they collected the current code smell knowledge through a systematic literature review. For comparability they constrained their findings to the original 22 code smells. They found, from these 22 original code smells only 7 were evaluated by researchers for their usefulness. Only 3 code smells were shown to be useful. For one of the code smells contradicting results exist. Most of the original 22 code smells are not yet evaluated for their usefulness.

Different authors use different code smells. Results on mostly disjoint code smell sets can hardly be compared. From the original 22 code smells, only 3 are known to be useful indicators for problems in software. Most code smells are unexplored by researchers for their usefulness.

4. DISCUSSION

Our fundamental questions on Technical Debt identification through code smell detection are mostly unanswered. We do not know whether code smell detection tools are use-

ful in dealing with Technical Debt. Future fundamental research is necessary to determine this usefulness. For this objective, we propose the following research tasks.

Without a useful Technical Debt definition common to most researchers and practitioners no reasonable Technical Debt discussion is possible. This holds for code smell Technical Debt as well as for general Technical Debt. Therefore:

- Establish a common useful Technical Debt definition.

Management rationales are based on economical consequences. Code smell can only aid in management discussions if it's economical consequences can be estimated. Therefore:

- Determine the economical consequences of code smell.

Most code smells are unexplored by researchers. General code smell results are only possible if most common code smells are considered. Therefore:

- Explore the usefulness of more than only a few code smells.

5. CONCLUSION

Researchers seek for automated Technical Debt identification tools. Some envision code smell detection to be a useful tool in identifying Technical Debt on code-level. We asked four fundamental questions on this approach. We reviewed a subjectively chosen set of literature to determine the current knowledge on these questions.

Multiple Technical Debt definitions are in use. Most definitions are troublesome in a code smell detection based Technical Debt identification evaluation. A reasonable Technical Debt definition should include the interest and principal terms. Technical Debt can exist for various different software engineering aspects. Code smell primarily concerns merely the design-level. The so called "visible" Technical Debts are opaque to code smell detection. Whether a Technical Debt was introduced deliberate, inadvertent, reckless, or prudent, can hardly be detected on code-level. Code smell detection may help in providing more visibility of design Debt. Useful Technical Debt tools should measure interest and principal in economical terms. No economical term based evaluation on code smell detection based Technical Debt identification exist. Instead, different code smells were evaluated with respect to change- and defect-proneness. Only few correlations were detected. Different literature evaluates different code smells. Some publications are even mostly disjoint in their code smell selection. Such results are not comparable. No general code smell detection based Technical Debt identification results exist. Only 3 of the original 22 code smells, were detected to be useful. Most code smells are mostly unexplored.

We have presented an analysis of the foundation of code smell detection based Technical Debt identification. In our opinion, a widely accepted useful Technical Debt definition is needed. We believe, automated code smell detection is mainly constrained to a few Technical Debt aspects. How code smell relates to economical consequences is not yet explored. Even the usefulness of most code smells is still unknown.

6. REFERENCES

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. K. and Philippe Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing Technical Debt in Software-Reliant Systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[2] CAST. How to Monetize Application Technical Debt. Technical report, 2011.

[3] W. Cunningham. The WyCash Portfolio Management System. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications*, pages 29–30. ACM, 1992.

[4] D. Falessi, M. A. Shaw, F. Shull, K. Mullen, and M. Stein. Practical Considerations, Challenges, and Requirements of Tool-Support for Managing Technical Debt. In *4th International Workshop on Managing Technical Debt*, pages 16–19. IEEE, 2013.

[5] F. A. Fontana, V. Ferme, A. Marino, B. Walter, and P. Martenka. Investigating the Impact of Code Smells on System's Quality: An Empirical Study on Systems of Different Application Domains. In *29th IEEE International Conference on Software Maintenance*, pages 260–269. IEEE, 2013.

[6] F. A. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *Third International Workshop on Managing Technical Debt*, pages 15–22. IEEE, 2012.

[7] M. Fowler. TechnicalDebtQuadrant. `http://martinfowler.com/bliki/TechnicalDebtQuadrant.html` (accessed 04.11.2014).

[8] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[9] F. Khomh, M. D. Penta, and Y.-G. Guéhéneuc. A Exploratory Study of the Impact of Code Smells on Software Change-proneness. In *16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[10] P. Kruchten, R. L. Nord, and I. Ozkaya. Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6):18–21, 2012.

[11] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. Technical Debt: Towards a Crisper Definiton; Report on the 4th International Worksop on Managing Technical Debt. *SIGSOFT Software Engineering Notes*, 38(5):51–54, 2013.

[12] R. O. Spínola, N. Zazworka, A. Vetrò, C. Seaman, and F. Shull. Investigating Technical Debt Folklore. In *4th International Workshop on Managing Technical Debt*, pages 1–7. IEEE, 2013.

[13] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Inestigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, pages 17–23. ACM, 2011.

[14] N. Zazworka, A. Vetrò, C. Izurieta, S. Wong, Y. Cai, C. Seaman, and F. Shull. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2013.

[15] M. Zhang, T. Hall, and N. Baddoo. Code Bad Smells: a review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, 2011.

Investigation of Code Smells in Different Software Domains

Marin Delchev
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
marin.delchev@rwth-aachen.de

Muhammad Firdaus Harun
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
firdaus.harun@swc.rwth-aachen.de

ABSTRACT

The increasing need of developing high quality software defines the necessity of deep understanding of the problems that may occur on different levels of abstraction in the software. Code smells investigate the signs of potential problems that may occur in the software. These signs may be used for automatic detection and prevention of often occurring software problems.

Unfortunately, to date there are neither clear classification nor rank indicating the code smells importance and severity. The lack of deeper understanding of the code smells prevents improvements of the current methodologies for developing high quality software.

The purpose of this work is to investigate and compare the impact and the severity of code smells in different application domains.

In order to investigate the code smells impact and severity we performed a survey among developers in various application domains then we compared statistical values of the answers. Domains that are taken into consideration are (web desktop applications, embedded systems, computer games/graphics). All results and conclusions of this work are based on the examination and comparison of the statistical values of the responses.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Refactoring; D.2.8 [**Software Engineering**]: Metrics—*quality measures, code smell measures, refactoring*

Keywords

code smells, survey, classification, software domains

1. INTRODUCTION

Martin Fowler introduces the code smells in his book Refactoring [6]. There he defines them and introduces a flat list of all 22 code smells. Since then the most wildly spread understanding of what is code smell is: "bad code". Unfortunately that is misleading interpretation. Code smells are defined as symptoms or signs that there might be a problem in the design of the software. Some of the well known design patterns described by "gang of four" contain code smells themselves. An example is the well known "Visitor" pattern, it contains the smell - "Feature Envy" [2].

Code smells are topic of various researches, some define ontologies to determine the relationship between code smells and the problems that cause them [11], others introduce logical groups of the code smells [8]. But neither of them takes into account the software domain as a factor which influences the smells.

Strong relationship exist between code smells and refactoring [6]. Refactoring is important process which improves the quality and the cost of the software systems. It is a significant part of the software life-cycle. According to [8] Microsoft reserves 20 percent of every project for refactoring. This means that projects reserve 20 percent of their development effort on reworking the weak parts of the product. Code smells may be used as an indicator when one should apply refactoring. They may be a clear sign that the code should be improved. Unfortunately the decision if the cause of a code smell is to be refactored depends only on the experience of the software developer. Our goal is to create categories and ranks of the code smells severity.

Such ranking would make the decision if a code smell should be refactored less dependent on the developers experience. It would allow better methodologies for detecting if refactoring is needed and it will reduce the required resources.

This work is organized as follows: Chapter 1 Introduces the current state of the art of detecting code smells and methodologies that use them to improve the software. Introduces the problems and benefits of possible solutions. Chapter 2 Explains the approach used for gathering information about the smells. The structure of the survey. Its target groups, questions and the included code smells. Chapter 3 Statistical approaches are used to study the answers. The results are reviewed and possible explanations about the smells severity and importance in the domains are introduced. Chapter 4 Conclusions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar KUR2014/15 RWTH Aachen University, Germany.

2. RELATED WORK

In his work [8], Mika Mantyla conducted a smell survey. He investigated the correlations between smell evaluations. He showed how the background variables affects the devel-

opers' smell evaluations. In his work he compared the smell evaluations and the source code metrics to provide more information on the reliability of smell evaluations.

[11] "defines an ontology-based methodology to provide a formalized description of the concepts of anti-patterns, code smells, refactoring, detections, and their relation. In their work they conduct a small survey to give a a priority to the most important smells."

[7] studies possible correlations existing among smells and the values of a set of software quality metrics. In their work they examine the smells frequency in different domains and their correlation with different metric values.

3. CODE SMELLS *STATE OF THE ART*

In this chapter research trends about the code smells will be described. Current tools for detecting code smells will be presented. Possible use cases of the smell classification will be described.

Since the code smells were introduced by Martin Fowlerr in his book Refactoring they were subject of various studies. The main goals of these studies is to provide effective approaches for detecting and removing smells. An example for such study is [11]. Researchers tried to introduce better taxonomies than the flat list that Martin Flower initially defined but we could not find any work that investigates how smells are perceived in the different software domains. This work continues the effort for providing better approaches for detecting smells by introducing and measuring two terms: smell severity and smell frequency.

Smell severity is defined as: "The amount of harm that a smell causes to the overall quality of the software." Smell frequency is defined as: "The number of occurrences of a smell in a software project." This work introduces various rankings and ordering of the smells by using these two terms and taking into consideration various domains, programming languages and the experience of the programmers. The smells ranking could be used for implementing better tools for smell detection. These tools may take into consideration the context. Currently there exists a lot of smell detection tools and frameworks.

- JDeodorant:[10] Is an eclipse plugin that detects code smells. This tool uses the eclipse's JDT ASTParser API to detect bad smell in the source code. It also allows adding of expert knowledge to its configurations.
- inCode:[3] Is an eclipse plugin for smell detection. It uses various object oriented metrics to detect the bad smells.
- SonarQube: is an open platform to manage code quality. It is a web-based application an it could detect code smells by using various software metrics. It allows adding rules and metrics.
- StenchBlossom:[9] is a smell detector that provides an interactive ambient visualization designed to first give programmers a quick, high-level overview of the smells in their code, and then, to help understand the sources of the code smells.
- InFusion:[4] allows to detect more than 20 design flaws and code smell.
- iPlasma:[1],[5] is an integrated platform for quality assessment of object-oriented systems that includes support for all the necessary phases of analysis.

There are various tools but they rely on the fact that the user will have the knowledge to configure them correctly. Our goal is to introduce a domain specific rankings of the smells. These rankings should summarize the expert knowledge of the software professionals in a given domain.

4. *METHODOLOGY*

In this chapter the approach used for gathering information will be introduced. The goal of the survey will be defined. The survey structure, questions, target groups, domains will be described. Smells used in the questions will be defined and explained.

To extract and compare the domain specific knowledge about the code smells the current works employs a survey. The survey was conducted via online platform. [1] The summary of the results is available at the following link. [2]

To explain the goal of the survey we need to explain some terms that we will use. Frequency of a smell is a measure which answers the following question: "How often is a smell encountered ?" That term reflects the repetition of a given smell. Severity of a smell is a measure which answers the following question: "How problematic is a smell ?" That term reflects how serious is the influence that a smell has over the overall quality of the software. Since we defined the frequency and the severity of a smell we are ready to state the goal of the survey. The main goals of the survey are the following: To measure the software professionals' perception of the smells' frequency and severity. The survey was conducted online the target group was software professional in different fields of work. The target domains were: Web/Desktop application development, Embedded systems, Computer Graphics/Games, Mobile development. The survey was distributed among various developers forums and among the following companies:

A: works in the domain of computer graphics and 3D rendering. B: works in the domain of visualization software. C: works in the domain of web applications. D: works in the domain of web banking applications. E: works in the domain of web applications. F: works in the domain of search engines. G: works in the domain of embedded automotive software. H: works in the domain of web applications. I: works in the domain of web applications.

The survey structure is the following: It consist of 26 questions. The first part consist of 6 questions asking about the experience and the background of the respondents.

- How many years of professional experience do you have?
- What is your domain of work?
- What is your role?
- Which is the main programming language that you use?
- How big is your project?

[1] https://docs.google.com/forms/d/1bqDVBfLrGKW_LMjtfD84GAnzajrjSd2w_TONgGlK_n4/viewform
[2] https://docs.google.com/forms/d/1bqDVBfLrGKW_LMjtfD84GAnzajrjSd2w_TONgGlK_n4/viewanalytics

- What software/IDE/Tools do you use for your project?

The second part consists of twenty questions. Two questions are asked for every smell :

- How often do you encounter the described code?
- How often do you change the described code?

The purpose of the first question is to measure the frequency of the smell. The scale of the answers for that question is given in the range from 1 to 5. Where 1 corresponds to "I do not encounter that smell" and 5 corresponds to "I always encounter that smell."

The purpose of the second question is to measure the severity of the smell. The scale of the answers for that question is given in the range from 1 to 4. Where 1 corresponds to "No need to change the code" and 4 corresponds to "That code is unmaintainable I always change it."

Since we wanted to maximize the number of people that fill the survey we tried to reduce the number of questions. Based on my academic and professional experience as software developer I selected a subset of 10 code smells. I consider that these 10 smells are most probable to have various severity and frequency according to the context that they are observed. The smells used in the survey are the following:

- Data Class

  I selected that smell because it is often encountered in development frameworks and it is a potential candidate for interesting results.

- Long Parameter List

  I believe that the large number of arguments is typical for some programing languages. If that is the case the frequency of that smell should be high in some domains and low in others.

- Switch statements

  That smell breaks basic principles of polymorphism it severity should be high in all domains.

- Message Chains

  A lot of frameworks contain this smell but it is interesting if the developers that use these frameworks perceive that smell as a problem.

- Primitive Obsession

  I believe that using many primitives is more often in the embedded software.

- Data Clumps

  I have encountered that smell often in the web domain and I believe it is more common there.

- Refused Bequest

  That smell violates basic inheritance principles and its severity should be high in all domains. It is an interesting question if it is encountered often in some specific domain.

- Feature Envy In my opinion this smell is often encountered when the data is being separated from the algorithm that is applied to it.

- Shotgun Surgery This smell violates the basic oop principles and I believe its severity will be high in all domains.

- Long Method This smell is really common. Based on my experience I consider that in the game domain this smell is encountered often.

5. RESULTS

In this section statistics of the survey‘s responses will be studied and compared. Possible explanations of the results will be suggested and if possible classification of the smells will be introduced.

The survey collected 73 responses. In order to classify and compare the smells: First we created a summary of the responses and studied the results. Second we searched for significant differences in the answers among the domains. Third we assigned and ordered the smells according our findings.

In the first phase of exploring the data we created data overview.[3]

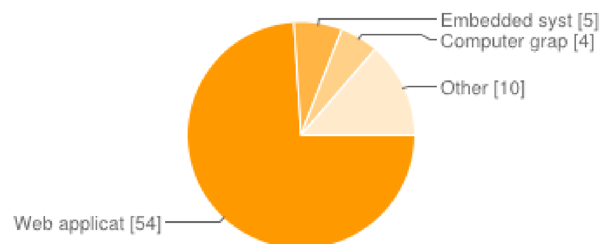


Figure 1: Distribution of the responses according the domain.

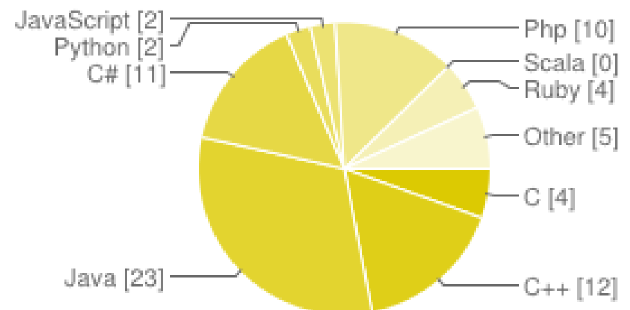


Figure 2: Distribution of the responses according the programming language.

The observations of the first part of the survey are: The experience of the respondents is evenly distributed. Twelve respondents have less than a year of working experience and 22 respondents have more that 10 years of working experience. Great part of the respondents work in the web domain: 74 percents. The rest of the respondents are approximately

[3]The detailed summary can be accessed at: `https://docs.google.com/forms/d/1bqDVBfLrGKW_LMjtfD84GAnzajrjSd2w_TONgGlK_n4/viewanalytics`

evenly distributed among the mobile, game and embedded domains. Most of the respondents are Developers (59 percents) or Senior Developers (30 percents) The most used programming languages are: Java 32 percents, C++ 16 percents, CSharp 15 percents, Php 14 percents Most of the programmers work on a large projects: 56 percents answered that they work on a project which has more than 50 000 lines of code.

Observations based on the second part of the survey are: Long Method: Half of the corespondents encounter that method often and assigned to it frequency 4 out of 5 but 51 percent consider that the smell is not a serious problem and assigned to it severity 2 out of 4.

Data class: 45 percents observe that smell rarely and assigned to it frequency 2 out of 5. The number of developers which do not consider that smell to be problem is significant. Data access objects (DAOs) and Data transfer objects (DTOs) are possible explanation for that. DAO is an object that provides an abstract interface to some type of database or other persistence mechanism. DTOs are objects that carry data between processes. DAOs and DTOs are heavily used in most of the current development frameworks to implement database abstraction. As a result most of the developers are used to it and they do not consider it as a problem.

Shotgun Surgery: Most of the corespondents assigned frequency 3 to that smell. That smell violates basic principle such as encapsulation and loose coupling and as a result it has severity 4 out of 4 according to 29 percents of the developers. No other smell has so many percents of responses assigning to it severity 4.

According to the answers the rest of the smells have approximately equally distributed severity and frequency. An interesting observation is the fact that most of the smells are refactored only if they cause a problem. If a smell is not detected in the early phase of the software development it will not be refactored later. This is a clear indicator that an effective approaches for effective smell detections are highly necessary.

In the second phase we mapped the answers of the question "How often do you encounter the described code ?" to numerical values in the range from 1 to 5. Where 1 means that the smell very rare and 5 means that the smell is very repetitive. The answers of the question "How often do you change the described code ?" were mapped to numerical values in the range from 1 to 4. Where 1 means that there is no need for a change while 4 means that the change is imminent.

We grouped the answers according three different criteria. The first criterion was the software domain. The groups according the domains were: web, game, embedded, mobile, others The second criterion was the programming language. The groups were: Java, C++, C sharp, php, C, Scripts. The script category sums up javascript, actionscript 3, python, ruby. The third grouping criterion was the years of experience that the respondent has. We created 3 groups according to the programmers' experience. The first group is developers with less than 4 years experience. The second group is for people with 4 or 5 years experience and the final third group is for people who has more than 5 years of professional experience.

The following sections explain in details the findings and the priorities that we assigned to the smell according each grouping criterion.

Using the answers that belong to the same software domain we formed 5 categories web, game, embedded, mobile, others. In each category we computed the mean for every smell. The results are in the following tables:

	Long	Data cl.	F. envy	Shotgun s	Bequest
Web:	3.403846	2.557692	2.615385	3.000000	2.096154
Game:	4.000000	2.666667	1.666667	2.666667	1.666667
Embedded:	2.750000	1.750000	2.750000	3.250000	1.500000
Mobile:	3.250000	2.500000	3.000000	3.000000	2.000000
Others:	3.000000	3.000000	1.666667	2.000000	1.333333

	Prim.Obs	M. Chain	Switch	Long list	Data clups
Web:	2.576923	3.173077	2.538462	2.865385	2.269231
Game:	2.333333	2.000000	2.333333	3.333333	2.666667
Embedded:	3.250001	3.000000	2.250000	3.250000	2.250000
Mobile:	2.000000	2.250000	2.000000	2.750000	2.500000
Others:	1.666667	2.666667	2.000000	2.333333	1.333333

Figure 3: Smell frequencies according the domain.

In this table the rows are the names of the software domains and the columns are the names of the smells. The values in each cell are the mean of the frequencies that the respondent assigned. Having that data we were able to sort in increasing order the smells' frequencies in each domain. The ordering is as follows: Web: Refused Bequest, Data clumps, Switch statement, Data class, Primitive Obsession, Feature envy, Long argument list, Shotgun surgery, Message chain, Long method. Analogously we created orderings for the other domains. An impression in the table makes the high frequency of the Long Method smell in the game domain. Possible explanation may be that the long methods are inevitable for the implementation of the game logic.

	Long	Data cl.	F. envy	Shotgun s	Bequest
Web:	2.519231	2.038462	2.692308	2.788462	2.576923
Game:	2.666667	2.333333	2.333333	2.000000	3.000000
Embedded:	2.250000	2.500000	3.500000	3.750000	3.000000
Mobile:	3.000000	2.000000	2.750000	3.250000	2.250000
Others:	1.666667	1.333333	1.333333	2.000000	1.333333

	Prim.Obs	M. Chain	Switch	Long list	Data clups
Web:	2.115385	2.442308	2.500000	2.519231	2.269231
Game:	2.333333	2.333333	1.666667	2.666667	1.666667
Embedded:	2.500000	2.000000	2.750000	1.750000	3.000000
Mobile:	2.500000	3.000000	3.000000	2.500000	2.500000
Others:	1.666667	1.333333	1.333333	2.000000	1.333333

Figure 4: Smell severities according the domain.

In the above table the means of the severities are presented. The increasing ordering that we created based on the means is the following:

- Web domain: Data Class, Primitive Obsession, Data clumps, Message chain, Switch statement, Long list, Long method, Refused bequest, Feature envy, Shotgun surgery.
- Game domain : Switch statements, Data clumps, Shotgun surgery, Data class, Feature envy, Primitive obsession, Message chain, Long method, Long list, Refused Bequest.
- Embedded domain: Long list, Message chain, Long method, Primitive obsession, Data class, Switch state-

ments, Refused bequest, Data clumps, Feature envy, Shotgun surgery.

- Mobile domain: Data class, Refused bequest, Primitive obsession, Long list, Data clumps, Feature envy, Message chain, Switch statements, Shotgun surgery, Long method.

As suggested before the fact that Data classes are heavily used by most web development framework influences the smell severity. While the web developers are highly tolerant to the Data class smell, the game developers are tolerant to switch statements and embedded developers tolerate the long list of arguments.

Next criterion that we used for grouping the responses is the programming language. We split the answers into six groups. For each group we computed the means. The results are shown into the following tables.

	Long	Data cl.	F. envy	Shotgun	Bequest
java:	3.521739	3.260870	3.043478	3.130435	3.130435
Php:	3.3	3.3	3.5	3.4	2.8
C:	2.8	3.2	2.2	3.0	4.4
Scripts:	3.1	3.2	3.2	2.7	3.2
C++:	3.416667	3.083333	3.250000	3.166667	3.000000
C#:	3.636364	2.363636	3.545455	3.181818	3.090909

	Prim.Obs	M. Chain	Switch	Long list	Data clups
java:	3.260870	3.043478	3.260870	3.000000	3.043478
Php:	2.8	4.0	3.1	3.0	2.8
C:	3.6	2.8	3.6	3.4	3.0
Scripts:	3.4	3.6	3.5	3.7	2.7
C++:	2.916667	3.166667	3.166667	3.333333	3.000000
C#:	3.181818	3.454545	3.363636	3.181818	3.090909

Figure 5: Smell frequencies according the programming language.

According to the results "Long method" is often encountered smell in Java C++ and CSharp projects. In contrast Php developers assign to "Refused bequest" and "Primitive obsession" lowest frequency and to "Message chain" the highest. While for Java projects "Long list" is with lowest frequency it has the highest frequency in projects implemented with scripting languages.

	Long	Data cl.	F. envy	Shotgun	Bequest
java:	2.391304	1.608696	2.478261	2.826087	2.304348
Php:	2.2	2.4	2.8	2.8	2.7
C:	2.083333	2.000000	2.666667	2.916667	2.416667
Scripts:	3.0	2.3	3.1	3.0	2.9
C++:	2.545455	2.090909	2.363636	2.363636	2.545455
C#:	2.2	2.8	2.6	3.0	2.4

	Prim.Obs	M. Chain	Switch	Long list	Data clups
java:	2.130435	2.608696	2.304348	2.304348	2.304348
Php:	2.4	3.2	2.7	2.6	2.3
C:	2.000000	1.750000	2.416667	2.500000	2.333333
Scripts:	2.0	2.0	2.6	2.8	2.3
C++:	2.272727	2.545455	2.454545	2.818182	2.363636
C#:	2.4	1.6	1.6	1.8	2.2

Figure 6: Smell severities according the programming language.

After we ordered the results in ascending order we observed the following: For Java developers most unharmful smells are Data class and Primitive obsession while the most severe are Message chain and Shotgun surgery. For Php Long method and Data clumps have the lowest severity and Message chain and Shotgun surgery the highest. For C++, scripts and C message chain has the lowest severity. Both C and C++ give highest severity to Shotgun surgery.

The last criterion that we used to group the data was the experience of the developers. We used the mean to assign a numerical value to the smell's severity and frequency. Since we have only three groups we did not use a table to represent the obtained values. The results are represented in the following parallel plots.

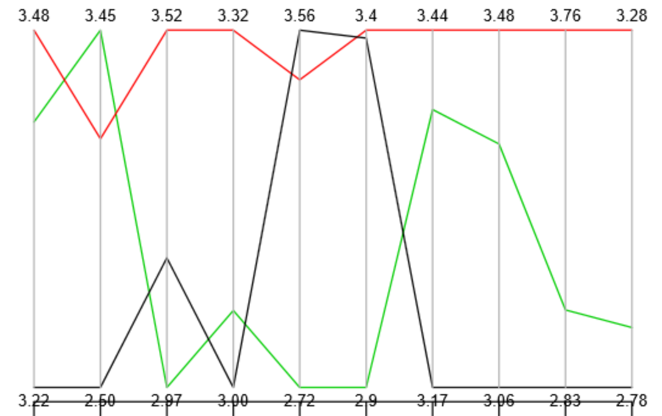


Figure 7: Smell frequencies according the professional experience.

Every vertical line in the above plot represents a specific smell. The order of the bars is: Long method, Data class, Feature envy, Shotgun surgery, Refused bequest, Primitive obsession, Message chain, Switch statement, Long list, Data clumps.

The colored lines that cross the horizontal lines are the groups of answers. The black line represents the developers with less than 4 years experience. The green line represents the developers with more than 6 years of experience. The red line represents people with 4 or 5 years of professional experience. Numbers at the bottom and at the top of every vertical line represent the minimal and the maximal mean values per group of answers. The crossing points between the colored lines and the vertical lines are the mean values of the frequencies. For example checking the third vertical line and its crossing point with the green line tells us that people with more than 6 years of experience assigned average frequency of 2.97 to the third smell. The parallel plot is suitable for demonstrating multidimensional data. In our case it clearly demonstrates the differences among the smells' frequencies. Users with high experience tend to encounter smells less often than the other two groups. Possible explanation is that they work in environments where the team members have a lot of experience and they tend to avoid introducing smells in the code.

The final plot is again a parallel plot similar to the previous one but it displays the means of the smells' severities. The vertical lines are smells and the order is equivalent to the one given above. The red line corresponds to developers who have more than six years of experience. The green line corresponds to people who have 4 or 5 years of experience

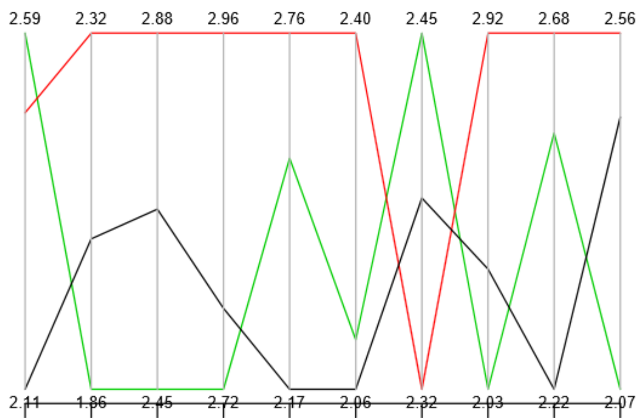


Figure 8: Smell severities according the professional experience.

and the black line indicates people with less than 4 years of experience. While we observed that highly experienced people encounter smells less often than the others, the current plot indicates that the same group is highly intolerant to most of the smells. Comparing the black and the red lines it is obvious that the less experienced developers tend to assign lower severity to the smells.

The data overview and the data means computed and presented in this chapter were used to create context dependent orderings of the smells. This orderings could be used for implementing context aware smell detection tools.

6. CONCLUSIONS

The current study included only 10 smells in future works the number of studied smells should be higher. In the current work approximately 75of the answers were from the web domain. In future works the survey should be distributed among various companies. Furthermore the examples of the smells were in Java in future works for better results pseudo code may be a better solution.

Current state of the art studies in the area of code smell were introduced. A number of smell detection tools and their qualities were presented. A possible idea for context dependent smell detection was introduced. The terms smell‘s severity and smell‘s frequency were defined. These terms were used to to assign priorities to a subset of all 22 smells. We introduced a possible scenario for improving the smell detection frameworks by taking into consideration the context of the software system. We conducted a survey among software experts in different domains. We introduced the structure and the purpose of the survey. We studied the answers and observed interesting tendencies then we used the answers to assign frequency and severity to the smells. Three different group of the answers were formed. The first group was based on the software domain, the second group on the programming language and the final group on the experience of the programmers. For each group two different rankings were presented one based on the frequency of the smells, second based on the severity of the smells.

7. REFERENCES

[1] iPlasma. `http://loose.upt.ro/reengineering/research/iplasma?_s=IUriqezunUFZjB5j&_k=f136Xjmo&_n&14`.

[2] Code smell in Visitor. `http://sourcemaking.com/refactoring/feature-envy`.

[3] InCode. `http://www.intooitus.com/products/incode`.

[4] InFusion. `http://www.intooitus.com/products/infusion`.

[5] P. M. D. R. C. Marinescu, R. Marinescu and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. *Proceedings of 21 stInternational Conference on Software Maintenance (ICSM2005), Tools Section, 2005.*

[6] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1999.

[7] A. M. B. W. P. M. Francesca Arcelli Fontana, Vincenzo Ferme. System‘s quality: An empirical study on systems of different application domains. *IEEE International Conference on Software Maintenance*, 2013.

[8] M. Mantyla. Bad smells in software - a taxonomy and an empirical study. May 2003.

[9] E. Murphy-Hill and A. P. Black. An interactive ambient visualization for code smells. *ACM Trans. Program. Lang. Syst. Proceedings of SOFTVIS‘’10 USA*.

[10] N. T. T. Chaikalis and A. Chatzigeorgiou. Jdeodorant: Identification and removal of type-checking bad smells. *European Conference on Software Maintenance and Reengineering*, 12:329–331, November 2008.

[11] D. L. C. Yixin Luo, Allyson Hoss. An ontological identification of relationships between anti-patterns and code smells. *Software Engineering Lab Louisiana State University 225-578-1378.*

Is software architecture still a shared hallucination?

Aarij Siddiqui
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
aarij.siddiqui@rwth-aachen.de

Ana-Maria-Cristina Nicolaescu
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ana.nicolaescu@swc.rwth-aachen.de

ABSTRACT

A question arose in a technical report [1] written in 1997 that claimed software architecture to be a shared hallucination. This claim was based on two observations: the architecture of the system is either not properly documented, or it is documented in such an abstract way that it is almost impossible to directly map it on the code. The manifestation of these situations was later referred to as the model-code-gap (or architectural erosion). This gap leads to problems in maintainability, and understandability of the system. The academia and industry have, over the time, realized the need to reduce this gap as much as possible. Therefore they researched and developed various tools to detect the creation of a gap in its early phases and allow corrective actions to be taken such that the code and architecture remain aligned with each other.

This paper aims to conduct a survey and reassess the answer to the same question 17 years later. The main purpose is to analyse how efficient and practical were the efforts made by academia and industry to reduce the model-code-gap.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures

Keywords

Software Architecture, shared hallucination, model-code-gap, architecture reconstruction, architecture erosion, architecture drift

1. INTRODUCTION

Architecture of a system is one of the most important and critical part of its structure, this also applies to the software architecture [2]. The importance of software architecture has long been recognized and is continuously increasing [3]. However in 1997 a technical report was published which stated that software architecture is a shared hallucination [1]. The reasons were said to be related with inappropriate documentation and the abstract nature of the architecture. Here hallucination refers to the scenario where architecture that was designed initially does not really remain the same as the development proceeds, this happens due to many reasons, which we have discussed later in the paper.

We discovered that since then, a number of mechanisms were identified and numerous tools were developed, e.g., [1], [4], [5], etc., to minimize the level of hallucination, and to reduce the difference between the architecture description and the actual system. The most practised of these concepts were Software Architecture Reconstruction (SAR), and Software Architecture Monitoring. Software Architecture Reconstruction represents the scenario where the architecture description is recreated from the existing code, so that it can be compared with what was designed initially. It should give a fairly clear idea of how much the code has deviated from the designed path. Software Monitoring is the idea where code is continuously monitored through different tools (IDE plugins, etc.) to check if the current implementation violates the rules defined in the architecture.

The aim of this research is to reassess the claim, after 17 years, if software architecture is still a shared hallucination. We decided to attempt a bidirectional approach, i.e., to search for the answer in the published literature as well as conduct interviews. For the literature review we searched the articles from "sciencedirect.com", IEEE, and ACM search engines with the following queries "Software Architecture Reconstruction", "Software Architecture Monitoring", "Software Architecture erosion", "Architecture as a shared hallucination", "Model code gap", and "Architecture Drift". We then selected the publications from 2000 till now. For the interviews our target audience were the employees, working in industry as well as in the academia. The main purpose was to understand the researchers along with the professionals and present both side by side. It was very fundamental to find out what is the importance of this hallucination in the real life scenarios. Since only then we can discuss its existence or non-existence, and go further into the reasoning. It was also very important to know if the researches done by the academia have any impact on the industry, or both of them work as different entities.

We have structured the paper in the following sections: **Approaches to reduce the model-code-gap(2):** here we presented the description of the discussed approaches used to reduce the difference between architecture descrip-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

tion and actual architecture, **Literature review(3):** here we analyse the research contribution to the under discussion, **Interview analysis(4):** here we discuss the background of our interviewees and present survey results, and **Conclusion and future work(5):** here we conclude the results from both analysis i.e. research and interview, discuss their impact on each other and also present the future work.

2. APPROACHES TO REDUCE THE MODEL-CODE-GAP

In this section of the paper we will discuss the various approaches to reduce model-code-gap. Which can also be referred to as the difference between architecture description and actual architecture of the system. The first approach we came through was Software Architecture Reconstruction. Software Architecture reconstruction (SAR) presents the idea where architecture of an existing software system is built. The SAR implementation can follow either of the following three processes: bottom-up, a top-down, or a hybrid opportunistic process as described in [6]. The bottom-up process targets first the low-level information and then proceed towards the high level knowledge [7], [8]. The top-down process aims to achieve the reconstruction starting with high-level details which includes the style of architecture and requirements, and reconstruct the architecture by understanding the conceptual background and then mapping it to the code in the implemented system [9], [10], [8]. The hybrid process combines both of the aforementioned processes [8], [11], and run them in parallel. Architecture reconstruction exists in various forms, e.g. some professionals perform it manually by analysing the code, and others use tools for the purpose 4.3.

The second approach we analysed was Software architecture monitoring. This technique refers to the mechanism where the architecture of the system is monitored, and compared with the actual architecture description, as the development of the system proceeds. It is efficient in a way that any deviation from the actual architecture is noticed immediately, and therefore effective measures can be taken without spending extra resources.

3. RELATED WORK

In this section we aim to explore deeper into the concept of architecture being a shared hallucination. Other than [1] we found another publication [12] that discusses the similar topic but in a different perspective. Booch believed software architecture to be a hallucination because it does not represent any real system yet. He referred to it as a common ground for all the stake holders of a project to agree upon, since it is very important to start the implementation. Our direction from here on was to discover the techniques and methodologies recommended in the articles to reduce the difference between architecture description and the implementation. We have divided the related work into two sections: Concepts, and Tools.

3.1 Concepts

Software architecture is usually documented using either one of the following three techniques: "Formal" architecture representation refers to a complete and thorough description of the system, following the organization guidelines and presenting it in a very sophisticated manner. On the other hand "Informal" refers to filling a shallow generic template which gives an overview of the system, it can also be a diagram on a white board. It variates in both cases how deep and detailed the description is, mainly depending on the complexity of the system which is supposed to be designed. "On the fly" depicts the situation where not even an informal documentation exists. The designing is done as the development proceeds.

The concepts presented to reduce the difference between architecture description and the implementation are all based on the assumption that some kind of architecture description already exists. Some of these concepts include Software Architecture Reconstruction, Software Architecture Monitoring, Automatic Code Generation, etc. In [6] it is mentioned that architecture reconstruction is needed for the big applications which have evolved over time and therefore it became very important to verify and compare the architecture description to the architecture of the current application, to make sure it still reflects the architecture of the implemented system. They concluded that it is complex to extract architectural components from source code, and usually they are mapped to packages or files, which cannot be compared with the architecture description because of the difference in their perspectives. Very few works take the architectural information and style into consideration. There are different viewpoints (such as data viewpoint, and capability viewpoint [13]) for which the architecture can be reconstructed. A framework was developed by [14] to reconstruct the architecture in a quality driven perspective. It discusses a framework for technical reasoning and highlights the information which is needed from the reconstruction process to map it to the business goals of the company. It was called QADSAR (Quality Attribute Driven Software Architecture Reconstruction). In [15] they provide mapping between TAXForm, which is an exchange format for frameworks at the software architecture level, and other formats used in other frameworks for the software architecture reconstruction. To support the reason for having mappings, it was said that Software engineers can use different frameworks to verify the result applied on the same input, and also use tools from various frameworks for different purposes, like a tool for parsing from one framework, and a tool for clustering from another.

The next keyword was software architecture erosion which might also be referred to as software architecture drift. But according to [16] there is a difference between these two terms. Software architecture erosion occurs when the architecture is violated, whereas architectural drift occurs when the rules implied by the architecture are not clear to the software engineers. These both are very interconnected concepts and in most of the situations might imply each other. Software architecture erosion was consider to be a common issue of software engineering by [17]. It was found to be unsolvable, irrespective of how determined the designers were. The design of the system will eventually drift from the initial version. In most cases it was considered a better option to have the application redesigned than modifying or updating the existing documentation. The main reason of this redesigning to occur is ever changing requirements. The problems that were mentioned are: Traceability of design decisions

(notations are not expressive enough to express the design), increasing maintenance cost (tasks become very complex as the system develops hence making it hard to understand and maintain), accumulation of design decisions (the dependency of one design decision on the other), iterative methods (aim to design such that it can accommodate future changes). According to [18] the main purpose of software architecture is to document all the important properties of a software application. Therefore it is highly recommended that it is being followed while the application is being developed. Whenever the actual system deviates from the designed architecture there is a possibility that critical problems will be faced. This deviation usually occurs when the system grows, making it hard to follow the design which leads to architecture erosion. The paper also discussed some tools, techniques and technologies to prevent architecture erosion. They divided the approaches into three categories namely, minimize, prevent and repair. In the "Minimize" approach it contains the following: process-oriented architecture conformance, architecture evolution management, and architecture design enforcement. "Prevent" consists of, architecture to implementation linkage, and self-adaptation. In "Repair" it includes: architecture recovery, architecture discovery, and architecture reconciliation (refers to the methods that reduce the gap between code and designed architecture of an application). It was concluded that none of the methods mentioned above single handedly provides solution to the problem of architecture erosion. But if used in combination with each other they will produce positive results.

Another technique that was discovered is software architecture monitoring. There is a very small difference between reconstruction and monitoring, in reconstruction the architecture is reconstructed after the system is developed, where as in monitoring it is continuously checked for updates and differences.

In our further researches we discovered that the under discussion phenomenon might also be referred to as "Model-Code-Gap". This addresses the gap which exists between the model that was made and the code that was developed. In [19] the concept of final software implementation to be directly generated from the model was discussed. It is said that to achieve reliable results the model is required to be detailed and complete. The concept of StateWORKS is considered to be helpful and can provide a path to "Executable UML". But since in this way all the complexity is moved towards the modelling phase, it can be hard to achieve. In [20], they explained that all the information that is required in the target code cannot be accumulated in the model, they also proposed solutions for this limitation.

3.2 Tools

Even though there is an awareness about the importance of software architecture, according to [20], [21], [22] complete and up-to-date architecture descriptions rarely exist. Keeping this in view various tools have been developed, based on different mechanisms, to regenerate the architecture description or to restrict the developers from deviating from the architecture description. In 1997, [1] developed a prototype system by the name **Dali**. This tool was developed to help the reconstruction process. Unlike other tools that were previously constructed, it does not automate the whole reconstruction process instead it extracts the information from the implemented system automatically. Then it gives user the opportunity to feed the architectural pattern and matches or link it with the extracted information. Later the resulting architecture is visualised for validation by the user.

In 2010, [5] presented an architecture monitoring tool. **Archer** is an Eclipse [23] IDE based plug-in. It is capable of identifying architectural patterns from the code and validate them against the architecture description that was initially fed to the system.

In 2013, [4] developed a software architecture monitoring tool **ARAMIS** which can produce meaningful real-time visualizations of interactions. The results were very positive and were tested on a software project in different stages of development. The tool formulates sequence diagrams on the bases of received data from the system that is being monitored. Later in 2014, the same group of researchers presented ARAMIS-CICE [24] that was able to automatically test the current architecture from the implementation and compare it to the architecture description. This too gave promising results.

To conclude the research section we can certainly state that significant efforts were made and are still being made to discover the techniques and develop tools that will help in reducing the gap between the architecture description and current implemented system. Whether it is architecture reconstruction or architecture monitoring, erosion avoiding techniques or automated code generation from models, all the approaches are being made available for the professionals to be used. Although it is something entirely different how practical the developed tools are, and how well they achieve the goal they are developed for. We expect to answer this question in our interview section, and look into the professional scenario of software development process and importance of architecture description and how they address architecture hallucination.

4. INTERVIEW ANALYSIS

For conducting the interviews we specifically targeted the professionals working in industry and academia who are related to the field of architecture design, e.g.: software architects, solution architects, etc. We followed evolutionary interview methodology, therefore all the interviews were inspired from the previous interviews. We have conducted seven interviews, out of these seven interviewees four were from the software industry. The other three are from academia but with work experience in the industry. In the following subsection we have presented a background of our interviewees, later we discuss the goals and present the analysis.

4.1 Background

In this section we aim to discuss the backgrounds of our interviewees since we targeted individuals from different environments. Two of our interviewees are senior software developers, working in the information services department of a multinational insurance company having a CMMI level 3 certification. One of them is working on a project as big as 100 to 200 man years. He uses incremental approach of software development, and believes in variable methodologies depending on the type, size, and requirements of the

project. Our second interviewee is involved in infrastructure development, he joined his current project in the middle of development phase therefore he found it a bit difficult to understand without the presence of architecture description. Hence he believes that architecture description should be there in a well documented form to assist the future development.

Our third interviewee also works in a multinational firm providing software, hardware, engineering services. He is responsible for the integration of different components belonging to one big enterprise level software system. He finds it very important that all the sub teams working on different components of the system deliver exactly what was designed, and hence emphasize greatly on the existence of architecture description that is not just documented but also understood by all the individuals. But he also believes that there should be freedom to a certain extent for the developers to modify the architecture, since in his opinion architecture is not a static entity.

Our fourth interviewee works as a solution architect in the information services department of an insurance company. The company employs 1400 employees in their information services department. Our interviewee strictly believes that in any case architecture description should be updated at all times. According to him there is no technical way of ensuring this but usually is done by emphasizing on it and including it in the company guidelines. In his scenario, solution architect is a part of development team who is not responsible for development but accessible whenever there is a need.

Our remaining three interviewees belonged to academic environment, but they have worked in industry along with their education or in previous years. One of them provides services to an organization in relation to the reconstruction of architecture of an existing system, he emphasizes on having up-to-date architecture description, mainly because of the evolving nature of big applications making it extremely hard and resource intensive to extend and maintain the application later on. The second interviewee from academic sector has an extended freelancing experience where he developed big projects as well as small ones, he believes in on the fly design of application, because he consider architecture to be an ever changing artefact. Our third academia related interviewee has managerial responsibilities for multiple small projects, he prefers an informal architecture design for the software systems. He believes there should be some documentation that can be referred to at any point in the project, because informally depicted mental models are prone to confusions and misconceptions.

4.2 Goals

In the interviews we have pursued the following goals:

1. Understand the architecting process.
2. Interviewees' point of view on architecture and its importance.
3. Analyse if the interviewees perceive a gap between the description and architecture.
4. Identify the tasks that are performed to reduce the aforementioned gap.

Table 1: Architecture Documentation done by the industrial professionals

Documentation Type	Usage
Formal	25%
Informal	50%
On the fly	25%

4.3 Analysis

Regarding the architecting process (1), in industry, as mentioned by three out of seven of our interviewees, the usual approach which is followed for the architecture design is incremental [25], i.e., they design and develop the system, component by component. This gives the architecture the opportunity to evolve. In spite of following similar methodologies for software development, the respondents had very different opinions regarding if and how the software architecture description should be documented. Sometimes, the representation of the architecture does not give the developer a complete idea of the system, and sometimes the architecture description is not documented at all. As showed in 1 industrial professionals tend to document the architecture in an "informal" way. Although there are professionals who follows the "formal" approach, equally there are individuals following the "on the fly" approach as well. The professionals who follow "on the fly" technique usually consider architecture to be a non static entity and do not believe in documenting it.

In relation to the view and importance of architecture (2) and if the gap was perceived (3), two out of seven professionals consider architecture to be a non-static entity, they find the alteration of architecture description to be very important and, sometimes, even beneficial to the system. According to them, these changes improves the system on performance and logic level and if documented, software architecture gives a very static look therefore people refrain to modify it. Both the aforementioned situations usually lead to have one common problem, when the architecture was modified in a later stage, the architecture description (if exists) is not updated to reflect the changes. Therefore architecture remains only in the minds of the individuals involved in the process. This later on (usually in the big projects) cause problems, since the architecture description cannot be referred, as everyone knows it is not up-to-date and therefore a rather informal approach, of calling and meeting with people who have worked with the system, is followed. This typically means that the knowledgeable people will have to explain the mental model which takes more time and might still not be clear. This problem was considered to be the key factor that creates the difference between the architecture description and the implemented system. The reason for the existence of this difference according to three out of seven professionals is unclear or changing requirements, for two of our interviewees it is insufficient time to design the architecture. This insufficient time to design the architecture lead to rather superficial architecture design, which is destined to change in later stages.

Table 2: Reconstruction Mechanisms used by industry and academia

Tools and Mechanisms	Industrial professionals	Academical professionals
Of-the-shelf Reconstruction Tools		33%
Self developed tools for reconstruction	25%	
Manual Reconstruction	25%	

Table 3: Opinions, if Model-code-gap has increased or decreased

Tendency	Industrial professionals	Academical professionals
Increased	25%	33.3%
Decreased	75%	66.7%

Discussing the tasks that were performed to reduce the aforementioned gap (4), as depicted here 2 three out of seven of our interviewees uses architecture reconstruction, either manually or with the help of tools, out of these three, two belonged to the industry. The choice for the tool entirely depends on which perspective (process, component, etc.) of the system is required to be reconstructed, and if there is no tool available that fulfils the requirement then one of our interviewees uses manual reconstruction and one other of our interviewees does it through self-developed tools. Two of the interviewees believe that if architecture description was made with proper consideration and analysis then it can be expected that very minimal amount of changes will happen in implementation phase. Other factor to achieve the small difference is to have architecture designed by the members of the development team but this cannot be applied in all cases.

There are also different opinions if this particular gap has increased or decreased in the last two decades 3. Five of our interviewees believe it has decreased since organizations, architectures and developers are aware of this issue and its consequences, therefore they are trying to reduce this gap. In addition to that there are also many tools available to assist in the process. Two of our interviewees believe that this gap has increased. One of them suggest that it is due to the increased usage of agile methodologies, mainly because of Agile's flexible and incremental nature [26], which usually leads to an outdated architecture description. Other believed that it is due to the increasing complex nature of the software systems.

All the interviewees expressed their concern on the functionalities provided by the tools for the reduction of this gap. They want to have something that generates the architecture from the code on the same granularity level as the architecture description so that it is comparable. But usually it generates something very detailed which cannot be compared to the description. They found the industry produced tools to be relatively more practical, but academia produced tools are gradually reaching to the point where they can also be used in the real life scenarios. At present they lack maturity and only address the most important issues.

To conclude these interviews, it can be said that all interviewees approached by us are aware of this gap and its consequences. Therefore, they are working to minimize this difference between the architecture description and the implemented system. Different people use different techniques to achieve this goal. Some use architecture reconstruction tools, others have a manual in place in the forms of company guidelines which enforces regular update of the architecture description to keep both architecture description and implemented system in accordance with each other.

We learned about different opinions, where some people consider the gap to be a good thing, since they believe that the gap indicates things were not feasible in the architecture therefore it changed during the development phase and the whole system improved. Some also believe that this architecture and code difference cannot be completely avoided, but it can of course be decreased. So what can be concluded is that people on individual as well as on organization level are working to reduce the difference but it is hard to state the effect of this effort on the end result.

5. CONCLUSION AND FUTURE WORK

To conclude the question "Is software architecture a shared hallucination" we can say that significant research has been done to develop new tools and techniques, and as we concluded above in our interview section that industrial professionals are using these tools to a certain extent, although manual methods are also in practice. During the interviews we found that all professionals have a different view regarding the difference between the architecture description and actual architecture of the system, some believe it to be a good thing, and some otherwise. The researchers, working in different universities and in research institutes are also trying to devise new and efficient mechanisms to solve the under discussion problem. What we learned is that in most cases industrial professionals are not aware of the tools developed by academia. Their main concern is that those tools are not yet in the position to contribute to the mainline industrial projects. The industrial professionals are specifically looking for something that saves time and in the case of architecture reconstruction produces an artefact which is on the same granularity level as their architecture description.

6. THREATS FOR VALIDITY

We aimed to find the concrete answer to the question, however, it is very hard to say if it is still a shared hallucination or not. To answer this question it needs quantification and statistical analysis of the software systems throughout their development cycle. Due to the lack of time we were

just able to interview 7 professionals. It would be a good point for any preceding research to continue the survey. It is possible that the tendencies change and whole new perspectives come into the equation when more people are involved in the process.

7. ACKNOWLEDGEMENTS

We would like to extend our gratitude towards all the interviewees who took the time and sat down with us for hour long sessions to discuss about our research topic.

8. REFERENCES

[1] Rick Kazman and S. Jeromy Carriere. Playing detective: Reconstructing software architecture from available evidence. October 1997.

[2] David Garlan. Software architecture: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 91–101, New York, NY, USA, 2000. ACM.

[3] L. Bass. *Software Architecture in Practice*. The SEI series in software engineering. Pearson Education, 2007.

[4] Horst Lichter Ana Dragomir. Run-time monitoring and real-time visualization of software architectures. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, pages 396–403. IEEE, 2013.

[5] Vitor Correia Alves, Rafael Henrique Santos Rocha, Rodrigo de Barros Paes, Evandro de Barros Costa, Leandro Dias da Silva, and Gustavo Robichez de Carvalho. Archer: An architectural monitoring tool. In *SERVICE COMPUTATION 2010, The Second International Conferences on Advanced Service Computing*, pages 146–152, 2010.

[6] D. Pollet S. Ducasse. Software architecture reconstruction: A process-oriented taxonomy. In *IEEE Transactions on Software Engineering*, pages 573–591. IEEE, 2009.

[7] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[8] M-AD Storey, F David Fracchia, and Hausi A Müller. Cognitive design elements to support the construction of a mental model during software exploration. *Journal of Systems and Software*, 44(3):171–185, 1999.

[9] Ian Carmichael, Vassilios Tzerpos, and Richard C Holt. Design maintenance: unexpected architectural interactions (experience report). In *Software Maintenance, 1995. Proceedings., International Conference on*, pages 134–137. IEEE, 1995.

[10] Gail C Murphy, David Notkin, and Kevin Sullivan. Software reflexion models: Bridging the gap between source and high-level models. In *ACM SIGSOFT Software Engineering Notes*, volume 20, pages 18–28. ACM, 1995.

[11] Arie Van Deursen, Christine Hofmeister, Rainer Koschke, Leon Moonen, and Claudio Riva. Symphony: View-driven software architecture reconstruction. In *Software Architecture, 2004. WICSA 2004. Proceedings. Fourth Working IEEE/IFIP Conference on*, pages 122–132. IEEE, 2004.

[12] Grady Booch. Architecture as a shared hallucination. *IEEE Software*, 27:96–96, 2010.

[13] Lih ren Jen and Yuh jye Lee. Working group. ieee recommended practice for architectural description of software-intensive systems. *IEEE Architecture*, pages 1471–2000, 2000.

[14] C. Verhoef C. Stoermer, L. O'Brien. Moving towards quality attribute driven software architecture reconstruction. In *10th Working Conference on Reverse Engineering*, pages 46–56. IEEE, 2003.

[15] R.C. Holt I.T. Bowman, M.W. Godfrey. Connecting architecture reconstruction frameworks. *Information and Software Technology*, 42(2):91–102, 2000.

[16] Alexander L. Wolf Dewayne E. Perry. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 2002.

[17] Jan Bosch Jilles van Gurp. Design erosion: problems and causes. *Journal of Systems and Software*, 61(2):105–119, 2002.

[18] Dharini Balasubramaniam Lakshitha de Silva. Controlling software architecture erosion: A survey. *Journal of Systems and Software*, 85(1):132–151, 2012.

[19] P. Wolstenholme F. Wagner, T. Wagner. Closing the gap between software modelling and code. In *Proceedings. 11th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 52–59. IEEE, May 2004.

[20] W. Hasselbring R. Reussner. *Handbook of Software Architecture (in German)*. dpunkt.verlag, 2009.

[21] D. Muthig M. Lindvall. Bridging the software architecture gap. In *Proceedings of Journal of IEEE Computer*, volume 41, pages 98–101. IEEE, 2008.

[22] C. Del Rosso. Continuous evolution through software architecture evaluation: a case study. In *Proceedings of Journal of Software Maintenance and Evolution: Research and Practice*, volume 18, pages 351–383, 2006.

[23] Eclipse. `http://www.eclipse.org/`, 2010.

[24] Johannes Dohmen Hongyu Chen Ana Dragomir, Horst Lichter. Run-time monitoring-based evaluation and communication integrity validation of software architectures. In *2014*, 2014.

[25] Galka R. Tran, P. On incremental delivery with functionality. In *Tenth Annual International Phoenix Conference on Computers and Communications*, pages 369–375. IEEE, 1991.

[26] M. Aoyama. Agile software process model. In *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*, pages 454–459. IEEE, 1997.

Evolution of Object Oriented Software Coupling Metrics

Yi Xu
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
yi.xu@rwth-aachen.de

Ana Nicolaescu
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
ana.nicolaescu@swc.rwth-aachen.de

ABSTRACT

A software metric is a quantitative measure of the degree to which a software item possesses a given quality attribute. In object oriented software systems, coupling is one of the most important factors, which affects the quality of the whole software system. With this background, we studied existing publications from the period around 1990s, when the researchers began to focus on the topic of metrics for object oriented software systems, including coupling metrics. In this paper, the research works we chose are divided into three time periods, to show how research focuses regarding coupling metrics have been evolving over the time. For each time period, we reviewed the coupling metrics in the literature, and analyzed the characteristics of the metrics in this time period and which quality attributes are measured by these metrics.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

software metrics, software coupling, software quality attributes

1. INTRODUCTION

In the 1990s, researchers realized that the traditional metrics developed for non-object-oriented software systems are not so suitable for object-oriented(OO) design. From the research of Wilde et al.[36], the traditional methods could not be adapted to OO notions such as classes, inheritance, encapsulation and message passing. The discussion of developing metrics specific for OO design came into researchers' view. Tegarden et al. and Billow were among the first authors, who thought that theoretical foundations should be taken into account in the design of object oriented metrics[35, 10]. The first OO metric was proposed in 1988 in Morris's master thesis[28]. Some years later other researchers such as Copien[16] and Pfleeger[30] implemented some initial metrics in the C++ environment. Lieberherr et al. provided in 1988 the first formal definition of object oriented programming styles based upon the concepts of coupling and cohesion[24]. Whereas Rajaraman et al. first tried to implemente coupling metrics in C++ programs in 1992[31].

Since metrics for object oriented software systems is a too wide topic, we decide to focus in this paper only on metrics measuring one specific aspect of OO software systems. Many of the existing works we found are mentioning coupling metrics, so we assume that coupling is an important factor in object oriented software systems, which will affect the quality attributes of the whole system. Thus we narrow the topic of this paper to reviewing and analyzing OO software coupling metrics.

In order to retrieve the existing research works, we use a search expression "software AND coupling OR (measurement OR metrics) AND (object OR class OR system OR component OR architecture) AND (oo OR object oriented)" and we set the time limit from 1990 until 2014. We used this search expression in 10 databases, and finally chose 6 of them according to the relevance of the results. The databases we chose are IEEE Xplore Digital Library[1], ACM Digital Library[2], CiteSeerX[3], ScienceDirect[4], Google Scholar[5] and The Collection of Computer Science Bibliographies[6]. We sorted the results according to two criteria, namely relevance based on the relevance algorithm of each database and number of citations.

First, we searched the citation databases, and got 210 matched documents from The Collection of Computer Science Bibliographies and 418 documents from CiteSeerX. We read the top 50 titles and abstracts of each database, and divided the papers into three periods, namely the papers in the 1990s, 2000s, and from 2010 until now. We observed the papers in these time periods and realized that in the beginning of the research of OO coupling metrics, researchers concentrated more on the static behaviors of the software in class level. After about 10 years, the research focus shifted to the dynamic behaviors of the software at object level. In recent years, researchers began to take the metrics in com-

[1]IEEE, http://ieeexplore.ieee.org/search/advsearch.jsp?tag=1
[2]ACM, http://dl.acm.org/dl.cfm
[3]CiteSeerX, http://citeseerx.ist.psu.edu/index
[4]ScienceDirect, http://www.sciencedirect.com/
[5]Google Scholar, http://scholar.google.de/
[6]Collection, http://liinwww.ira.uka.de/bibliography/index.html

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

ponent or system level into consideration.

Then, we queried the other four databases and sorted the results according to relevance. We retrieved 10625 results from IEEE Xplore Digital Library, 247 results from ACM Digital Library, 27300 results from Google Scholar and 42510 results from ScienceDirect. We filtered the results according to different time periods and chose 10 papers in each time periods to review in this paper.

The remainder of this paper is organized as follows. In section 2 the classical metrics in the 1990s will be discussed. Ten representative coupling metrics will be reviewed in section 2.1, following the analysis of the metrics in this period in section 2.2. In section 3, the modern metrics in the 2000s will be discussed in section 3.1. Afterwards, we will make an analysis of the metrics in section 3.2. In section 4, ten metrics in the 2010s will be described firstly in section 4.1, and then analyzed in section 4.2. Section 5 concludes the paper.

2. CLASSICAL METRICS IN THE 1990S

In this section, we chose ten influential metrics in the 1990s. The first two sets of metrics we chose is the coupling metric from the CK Metric Suite[15] and the MOOD Metric Set[20], which are top cited in the citation databases of IEEE and The Collection of Computer Science Bibliographies. The remaining 8 papers are chosen from the top 30 results from each of the database we use, judging from the titles and abstracts, which seems most relevant to our topic.

In the first part of this section, we will explain the main idea of each metric we chose. In the second part, we will make an analysis of these metrics.

2.1 Classical Metrics Review

The most influential static metrics measuring coupling is the "CK" metric suite[15]. Chidamber and Kemerer presented six metrics in their work, among which we take out the CBO (Coupling Between Object Class) metric to explain in detail. The Definition of this metric is that CBO for a class is the count of the number of other classes to which it is coupled[15].The idea behind this metric is that one class is coupled to another, when it uses the methods or instance variables defined by the other class, and measuring the degree, to which these two classes are coupled is important, because excessive coupling between classes will hinder modular design and prevent reuse.

The next metric to evaluate is the coupling metric in the MOOD Metric Set[20]. The coupling metric in this metric set is called CF (Coupling Factor) metric. The metric measures the coupling between classes, excluding coupling due to inheritance[20]. CF is calculated among all possible pairwise sets of classes by observing whether the classes in a pair are related to each other. When two classes pass massages to each other, or use the attributes or methods of each other, they are considered to be coupled.

The next three research works to investigate are the papers from Briand et al.[9, 12, 11]. They reviewed the CK Metric Suite[15] in 1996 in their work[9], and implemented an experiment of the metrics. In their experiment, they used three libraries, namely a public domain library with C++ classes, a GNU library and a C++ database library. They figured out that the CBO value is significantly small for UI classes. Then in 1997, Briand and his colleagues provided a new suite of coupling metrics which are especially suitable for OO Systems developed by C++[12]. They developed their metrics according to three facets, namely Relationship, Locus and Type. Relationship refers to the type of relationship, for example friendship or inheritance. Locus refers to the impact of change flows towards a Class (import) or away from a Class (export). Type refers to type of interactions between classes, which may be Class-Attribute interaction, Class-Method interaction, or Method-Method interaction. After two years in 1999, Briand and his co-workers continued their work, and provided a framework for Coupling Measurement in OO Systems[11]. In their Coupling Frameworks, they defined three dimensions of coupling, namely Interaction Coupling, Component Coupling and Inheritance Coupling. Two methods are interaction coupled if one method invokes the other, or they communicate via sharing data. Two classes c and c' are component coupled, if c' is the type of either an attribute of c, or an input or output of a method of c, or a local variable of a method of c, or an input or output parameter of a method invoked within a method of c[11]. Two classes c and c' are inheritance coupled, if one class is an ancestor of the other.

In Moser et al.' work, they described a formal meta-model based approach measuring class coupling[29]. In their paper, they defined two types of coupling to measure. The first one is active coupling, which measures the extent to which the second class contributes to the implementation of the first one. The values of coupling will range from 0 to 1, with larger values corresponding to greater degree of coupling. The second type of coupling is passive coupling, which measures the efficiency of the first class in implementing the second one. The values indicate to what extent one class uses the methods of another class.

In the research work of Brito et al.[13], they evaluated experimentally the impact of OO design on software quality characteristics using the MOOD Metric Set mentioned above[20]. They performed a controlled experiment, and examined the degree to which MOOD metrics allow to predict defect density (a reliability measure) and normalized network (corrective maintenance effort, a maintainability measure).

The the paper of Lee et al.[23], they first reviewed the CK Metric Suite[15], and then proposed the measurement of three forms of coupling. The first form of coupling is Coupling Through Inheritance. They used DIT (Depth of Inheritance Tree) and NOC (Number Of Children) to measure the inheritance characterization[23]. The second form of coupling is Coupling Through Message Passing, which is measured by MPC (Message Passing Coupling) metric[23]. The last form of coupling is the coupling through ADT (Abstract Data Type), which is measured by the DAC (Data Abstraction Coupling) metric[23].

The next research work we chose is the work from Allen et al.[2]. They discussed in their paper how to measure the coupling of subsystems, namely intermodule coupling. They measured coupling regarding to five properties of coupling of a modular system. The attributes are nonnegativity, null value, monotonicity, merging of modules and disjoint module additivity[2]. They used these five attributes in a measurement protocol and generated a graph representing some aspects of software design such as the design decisions.

The last research work to discuss is the paper from Eder et al.[17]. In their work, they also divided coupling into three categories like Lee et al. did in their work[23], namely

interaction coupling, component coupling and inheritance coupling. Other than that, they also showed the interplay of the three coupling dimensions taking transitive method invocations into account.

2.2 Classical Metrics Analysis

From the ten papers reviewed in the last subsection, we found out that 90% of the metrics in this time periods are static. Most of the researchers consider the metrics at class level at OO design phase, namely at the early stage of the software engineering. The well-known CK Metric Suite has a great impact on the research works afterwards. Many researchers evaluated CK metrics and implemented experiments based on these metrics. We can also come to the conclusion that Briand's works are also important in this time period, since three of his papers are cited among the top cited list.

From the static point of view, software coupling is mainly measured by how close the classes are related to each other. When a class depends to a large extent on other classes, it will increase its error density, because it may not use the other classes' methods correctly. Moreover, a minor change of one of the classes, which this class depends on, will also cause errors in this class. It will also increase the classes' cost of maintenance. Because this class is vulnerable to any changes of the other classes, on which it depends. When one of the classes changes, it has to update and be careful not to misuse the changed class. Furthermore, more interactions with other classes may reduce the usability of this class, because it will be very complicated, so that other classes will have difficulties correctly using the methods of this class.

3. MODERN METRICS IN THE 2000S

In this section, we chose ten representative metrics in the 2000s. Firstly, we choose the research works from Mitchell[25, 26, 27] and Arisholm[5], which are ranked among the top cited papers in IEEE Citation Search, CiteSeerX and The Collection of Computer Science Bibliographies. Then we limited the time range from 2000 to 2009 in all the other databases we chose, and sorted them by relevance. From the search results, we selected 8 papers with highest relevance from the top 30 papers of each database.

In the first part of this section, We will describe the metrics from the 10 papers we chose. In the second part, we will make an analysis of these metrics.

3.1 Modern Metrics Review

The first three metrics we want to mention are the research works from Mitchell et al.[25, 26, 27]. In their paper in 2004, they described a set of run-time object-oriented metrics to complement existing static coupling metrics. They used a number of statistical techniques including descriptive statistics, a correlation study and principal component analysis to assess the properties of measures, and investigated whether their metrics are redundant with respect to the CBO metric mentioned in the last section[25]. A year later, they proposed some object-level run-time metrics to study coupling between objects[26]. They used statistical techniques like agglomerative hierarchical clustering analysis to identify objects from the same class, that exhibit non-uniform coupling behaviour when measured dynamically[26]. The dynamic metrics they provided are also based on the CBO metric. These metrics seek to quantify coupling at different layers of granularity, that is at class-class and object-class [26]. In 2006, Mitchell and his colleague extended their work to explain the relationship between the CBO metric and some of its dynamic counterparts[27]. The results of their study showed that static and dynamic coupling metrics can be used independently. But they suggested that dynamic coupling metrics might be more suitable in the context of coverage measures, rather than as stand-alone software metrics[27].

The next metric set to discuss is the research work by Arishom[5]. In their work, they classified their coupling measures to three criteria, namely entity of measurement, granularity and scope[5]. Since dynamic coupling is based on dynamic code analysis, coupling may be measured for a class or one of its instances, thus the entity can either be a class or an object[5]. The scope determines which objects or classes, depending on the entity of measurement, are to be accounted for when measuring dynamic coupling[5].

In the research work of Yacoub et al.[37], they presented in their work a dynamic metric suite to measure the quality of OO designs at an early development phase. The suite consists of metrics for dynamic complexity and object coupling based on execution scenarios[37]. They provided two dynamic coupling metrics, namely EOC (Export Object Coupling) and IOC (Import Object Coupling). The export coupling for object A with respect to object B is the percentage of the number of messages sent from A to B with respect to the total number of messages exchanged between A and B during the execution of the scenario[37]. Whereas import coupling for object A with respect to object B is the percentage of the number of messages received by object A that were sent by object B with respect to the total number of messages exchanged during the execution of the scenario.

The next research work to mention is the metric from Hassoun et al.[21]. In their work, they observed object coupling as it evolves during program execution and proposed a measure which takes object interactions into account. They defined coupling as follows: two objects are coupled if either one of them can influence the history of the other. The history of an object here is defined as the sequence of its states in time. A coupling measures between two arbitrary objects P and Q, thus depends on the time during which P influences the history of Q and vice-versa. It also depends on the number of objects involved and on their complexity[21].

In the research work of Zaidman et al.[38], they used a dynamic coupling metric, which measures interaction between runtime objects, to collect and analyze the event trace of large-scale industrial application. The coupling metric they use is the EOC metric mentioned above by Yacoub[37]. They calculated the EOC for each participating object, which results in a matrix of coupling-values. Since this information is too detailed and difficult to understand, they proposed another metric to measure how many unique messages a certain object has sent, namely the Object Request For Service metric. This metric calculates the total number of messages that object O has sent during the program run.

The next research work to present is the paper from Tibor et al.[19]. This paper describes an application of the CK metric suites in open source software systems. They used regression and machine learning methods to validate the usefulness of these metrics for fault-proneness prediction.

In the research work of Jagdish et al.[8], they described

an improved hierarchical model for the assessment of high-level design quality attributes in object oriented [8]. In this model, structural and behavioral properties of classes, objects, and their relationships are evaluated using a suite of OO design metrics[8]. They mainly use two small set of metrics DSC (Design Size in Classes) and NOH (Number of Hierarchies) to assess the two design properties Design Size and Hierarchies. The quality attributes they measured are reusability, flexibiity, understandability, functionality, extensibility and effectiveness.

The last research work in this time period to describe is the paper from Abdulruda et al.[1]. In their work, they evaluated metrics to model industrial processes with Peri Net Object Oriented Data Structure. MHF (Method Hiding Factor) and AHF (Attribute Hideing Factor) metrics are used jointly by them as measures of encapsulation. MIF (Method Inheritance Factor) and AIF (Attribute Inheritance Factor) metrics are together used as measures of inheritance. PF (Polymorphism Factor) metric is used as a measure of polymorphism potential, and CF (Coupling Factor) is used to measure coupling between classes.

3.2 Modern Metrics Analysis

In this period of time, the researchers began to think that static coupling metrics may not be enough for measuring the coupling of object oriented software programs, so they started to consider to measure the run-time behavior of the software programs. For instance, Mitchell mentioned in his work, that static metrics fail to quantify all the underlying dimensions of coupling, as program behaviour is a function of its operational environment as well as the complexity of the source code[25]. Arisholm also mentioned in his paper, that the static coupling measures are imprecise as they do not reflect the actual coupling taking place among classes at run-time, because of polymorphism, dynamic binding and unused code in the software.

The metrics in this time period are mostly at object level. The research works of dynamic coupling metrics are basically divided into two main categories. One focuses on the import and export of the classes [37, 38, 25, 26, 27]and objects, the other emphasizes the evolution of the metrics over a period of time[21, 33]. Many researchers also based their metrics on the influential CBO metric mentioned in the last section, and many other researchers have implemented the dynamic metrics in experiments or even in industry.

Although the measurement of the dynamic software coupling metrics are very different with measuring the static coupling metrics, the quality attributes which are taken into considerations are basically the same, namely the maintainability, understandability, reusability and error-propagation. Some of the works also takes flexibility, functionality, extendibility and effectiveness into consideration.

4. LATEST METRICS IN THE 2010S

In this section, we choose ten prominent metrics in the 2010s. We found out that the papers in this time period are not so often cited than the older papers. Therefore, we directly sorted the result according to relevance and filtered the time period from 2010 to 2014. We chose 10 papers from the top 30 results in each database mentioned above. In the first part of this section, we will make a brief review of the ten papers in this time period. In the second part, we will make an analysis of the metrics in these research works.

4.1 Latest Metrics Review

The first metric in this time period to discuss is the metric from Aloysius et al.[3]. They presented in their paper a new cognitive complexty metric namely cognitive weighted coupling between objects for measuring coupling in object oriented systems[3]. In this metric, five types of coupling that may exist between classes are control coupling, global data coupling, internal data coupling, data coupling and lexical content coupling[3].

The second research work to mention is the paper from Kebir et al.[22]. In their work, they measure coupling metrics at component level. Firstly, they found three properties of components. The first one is that a component is autonomous if it has no required interface. The second is that a component can be composed by means of its provided and required interfaces[22]. The last one is that the component which provides many interfaces may provide various functionalities. Each interface can offer different services. Thus the higher the number of interfaces is, the higher the number of functionalities can be[22]. Then they matched the properties to metrics.

The next research work to discuss is the paper by Rathore et al.[33]. Their work is to investigate the relationship of class design level OO metrics with fault proneness of object oriented software system. They evaluated the capability of the design attributes related to coupling etc. to predict fault proneness. They defined coupling as the measure of the strength of association established by a connection from one module to another. The coupling metrics they use are CBO, RFC, CA, CE and DAM.

In the research work of Alshammari et al.[4], they realized in their work that metrics which measure the quality attribute of information security have received little attention. So they focused on the design of an object oriented application and defined a number of security metrics derivable from a program's design artefacts. In particular, they presented security metrics based on coupling and other attributes of a given object oriented, multi-class program from the pointing of view of potential information flow. The coupling metric proposed by them is called CCC (Critical Classes Coupling), which aims to find the degree of coupling between classes and classified attributes in a given design. It is calculated based on the theory of directed weighted links. This metric aims to penalise programs with high coupling.

In the research work of Gether et al.[18], the researchers found out that many existing coupling metrics lack the ability to identify conceptual dependencies, which could specify underlying relationships encoded by developers in identifiers and comments of source code[18]. So they proposed a RTC (Relational Topic based Coupling) metric at class level, which uses RTM (Relational Topic Models), generative probabilities model to capture latent topics in source code classes and relationships among them[18].

The next metric to mention is the research work from Carliss et al.[7]. They describe in their work a methodology based upon directed network graphs which can identify linkages between components in a system. They found out that most software releases are using Core-Periphery structure, where Core subsystems have been defined as those that are tightly coupled to other subsystems, whereas peripheral subsystems tend to posses only loose connections to other subsystems[7]. They also use a square Matrix DSM (Design Structure Matrix) to calculate metrics, which capture the

level of coupling for each component.

In the research work of Chen et al.[14], the authors argued that the traditional metrics are not suitable to measure coupling in component-based software system (CBSS). So they provided new metrics to specifically measure coupling and cohesion. In their paper, coupling in CBSS is defined as how one component depends on the other. They regard the CBSS as a directed graph, where the components in CBSS are vertices.

The next research work to evaluate is the paper from Babu et al.[6]. The researchers of this paper found out that metrics for centralized systems are not suitable for distributed systems with service oriented components. So they propose a hybrid model in Distributed Object Oriented Software to measure the coupling at runtime.

Poornima et al.[34] focused on measuring coupling to control the complexity level as requirements increases. They take two kinds of metrics into consideration, namely, static/class level coupling metrics and dynamic/object level coupling metrics. For the static metrics, they consider Efferent Coupling, Afferent Coupling and Depth of Inheritance Tree. For dynamic metrics, they used CBO and RFC in to measure the complexity of the system.

The last research work to discuss is the paper from Narendra et al.[32]. In this paper, the authors presented a measurement to measure coupling between objects, number of associations between classes, number of dependencies in metric, number of dependencies out metric and number of childen in OO programming. The metric values of class inheritance and interface prove which program is good to use specifically for C# users[32].

4.2 Latest Metrics Analysis

The software coupling metrics in the latest five years have some new changes compared to the last two decades. There exists many metrics which measures not only coupling metrics at class or object level, but also at component or system level. Measuring the coupling of components will not be adequate if it is measured just like classes as described in the last two sections. It will be necessary to count the interfaces as well. The researchers involve some models like directed network graphs and some kinds of structure matrix to better measure the coupling between components. Other than the granularity changes towards components, some researchers also consider the conceptual dependencies, and use some probabilistic methods to measure and identify the relationships between components.

Like the static and dynamic software coupling metrics discussed in the last two sections, the metrics in this time period also focus on measuring the quality attributes of the software artefacts regarding maintainability, understandability, reusability and error-propagation. Other than these quality attributes, some researchers also proposed metrics for measuring security at an early design phase.

5. CONCLUSIONS

In this paper, we first did a literature search according to our topic of software coupling metrics in the context of object-oriented software systems. From the results we divided the research works into three time periods. In each time period, we chose ten representative works, and made an elaborate review of the methodologies of the coupling metrics. After the literature reviews of each time period, we did an analysis about the typical coupling measurement methods and which quality attributes the researchers consider important for the software system.

We found out that despite different research focuses, most of the researchers think the quality attributes of maintainability, understandability, reusability and error-propagation are important. Thus we suppose that these four quality attributes are important in the measurement of software coupling metrics. For future work, researchers could possibly review more metrics in the existing research works, and make a coupling metrics catalogue similar to catalogues of design patterns, which will be helpful for future researchers to get some inspiration of developing their new metrics and also for practitioners to choose the proper metrics that fit in their software development.

6. REFERENCES

[1] A. A. Alfize. Metrics evaluation for industrial oo petri nets models. 2002.

[2] E. B. Allen and T. M. Khoshgoftaar. Measuring coupling and cohesion: An information-theory approach. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 119–127. IEEE, 1999.

[3] A. Aloysius and L. Arockiam. Coupling complexity metric: A cognitive approach. *International Journal of Information Technology and Computer Science (IJITCS)*, 4(9):29, 2012.

[4] B. Alshammari, C. Fidge, and D. Corney. Security metrics for object-oriented designs. In *Software Engineering Conference (ASWEC), 2010 21st Australian*, pages 55–64, April 2010.

[5] E. Arisholm, L. C. Briand, and A. Foyen. Dynamic coupling measurement for object-oriented software. *Software Engineering, IEEE Transactions on*, 30(8):491–506, 2004.

[6] S. Babu and R. M. S. Parvathi. Design dynamic coupling measurement of distributed object oriented software using trace events. 2011.

[7] C. Baldwin, A. MacCormack, J. Rusnak, et al. Hidden structure: Using network methods to map system architecture. In *Harvard Business School Working Paper*. Citeseer, 2013.

[8] J. Bansiya and C. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4–17, Jan 2002.

[9] V. Basili, L. Briand, and W. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, Oct 1996.

[10] S. Billow. Applying graph-theoretic analysis models to object oriented system models. 1992.

[11] L. Briand, J. Daly, and J. Wust. A unified framework for coupling measurement in object-oriented systems. *Software Engineering, IEEE Transactions on*, 25(1):91–121, Jan 1999.

[12] L. Briand, P. Devanbu, and W. Melo. An investigation into coupling measures for c++. In *Proceedings of the 19th international conference on Software engineering*, pages 412–421. ACM, 1997.

[13] F. Brito e Abreu and W. Melo. Evaluating the impact of object-oriented design on software quality. In *Software Metrics Symposium, 1996., Proceedings of the 3rd International*, pages 90–99. IEEE, 1996.

[14] J. Chen, H. Wang, Y. Zhou, and S. D. Bruda. Complexity metrics for component-based software systems. *International Journal of Digital Content Technology and its Applications*, 5(3):235–244, 2011.

[15] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, Jun 1994.

[16] J. Coplien. Looking over one's shoulder at a c++ program. *AT&T Bell Labs. Tech. Memo*, 1993.

[17] J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. *Technical Reprot, University of Klagenfurt, Austria*, 1994.

[18] M. Gethers and D. Poshyvanyk. Using relational topic models to capture coupling among classes in object-oriented software systems. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[19] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, Oct 2005.

[20] R. Harrison, S. Counsell, and R. Nithi. An evaluation of the mood set of object-oriented software metrics. *Software Engineering, IEEE Transactions on*, 24(6):491–496, Jun 1998.

[21] Y. Hassoun, R. Johnson, and S. Counsell. A dynamic runtime coupling metric for meta-level architectures. In *Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings. Eighth European Conference on*, pages 339–346. IEEE, 2004.

[22] S. Kebir, A.-D. Seriai, S. Chardigny, and A. Chaoui. Quality-centric approach for software component identification from object-oriented code. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 181–190, Aug 2012.

[23] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *Software Metrics Symposium, 1993. Proceedings., First International*, pages 52–60. IEEE, 1993.

[24] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: An objective sense of style. In *ACM SIGPLAN Notices*, volume 23, pages 323–334. ACM, 1988.

[25] Á. Mitchell and J. F. Power. An empirical investigation into the dimensions of run-time coupling in java programs. In *Proceedings of the 3rd international symposium on Principles and practice of programming in Java*, pages 9–14. Trinity College Dublin, 2004.

[26] A. Mitchell and J. F. Power. Using object-level run-time metrics to study coupling between objects. In *Proceedings of the 2005 ACM symposium on Applied computing*, pages 1456–1462. ACM, 2005.

[27] Á. Mitchell and J. F. Power. A study of the influence of coverage on the relationship between static and dynamic coupling metrics. *Science of Computer Programming*, 59(1):4–25, 2006.

[28] K. Morris. Metrics for object oriented software development. master thesis, MIT, Sloan School of Management, Cambridge, MA, 1988.

[29] S. Moser and V. B. Misic. Measuring class coupling and cohesion: A formal metamodel approach. In *Software Engineering Conference, 1997. Asia Pacific... and International Computer Science Conference 1997. APSEC'97 and ICSC'97. Proceedings*, pages 31–40. IEEE, 1997.

[30] S. Pfleeger and J. Palmer. Software estimation for object-oriented systems. In *1990 Int. Function Point Users Group Fall Conf*, pages 181–196, 1990.

[31] C. Rajaraman and M. R. Lyu. Some coupling measures for c++ programs. In *TOOLS (8)*, pages 225–234. Citeseer, 1992.

[32] N. Rathore and R. Gupta. A novel coupling metrics measure difference between inheritance and interface to find better oop paradigm using c#. In *Information and Communication Technologies (WICT), 2011 World Congress on*, pages 467–472, Dec 2011.

[33] S. Rathore and A. Gupta. Investigating object-oriented design metrics to predict fault-proneness of software modules. In *Software Engineering (CONSEG), 2012 CSI Sixth International Conference on*, pages 1–10, Sept 2012.

[34] P. U. S. and S. V. Significance of coupling and cohesion on design quality, 2014. Comment: 6 Pages, 2 Figures.

[35] D. P. Tegarden, S. D. Sheetz, and D. E. Monarchi. Effectiveness of traditional software metrics for object-oriented systems. In *System Sciences, 1992. Proceedings of the Twenty-Fifth Hawaii International Conference on*, volume 4, pages 359–368. IEEE, 1992.

[36] N. Wilde and R. Huitt. Maintenance support for object oriented programs. In *Software Maintenance, 1991., Proceedings. Conference on*, pages 162–170. IEEE, 1991.

[37] S. M. Yacoub, H. H. Ammar, and T. Robinson. Dynamic metrics for object oriented designs. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 50–61. IEEE, 1999.

[38] A. Zaidman and S. Demeyer. Analyzing large event traces with the help of coupling metrics. In *Proc. the 5th International Workshop on OO Reengineering, Oslo, Norway*, 2004.

Continuous Delivery in Open-source Projects

Stefan Dollase
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
stefan.dollase@rwth-aachen.de

Andrej Dyck
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
andrej.dyck@swc.rwth-aachen.de

ABSTRACT

Continuous delivery is a concept to quickly publish changes to a software project as a new release. As this is useful for every software project, it is also useful for open-source projects. It enables the developers to react fast if a bugfix is needed, but it also allows them to release new functionality frequently.

In open-source projects it is especially difficult to use continuous delivery, because it requires the developers to frequently integrate all changes into the mainline. However, in open-source project there are not only full time software developers, but also some hobby developers. These are less active which might lead to long lived branches. This paper discusses how continuous delivery can be realized even with many hobby developers in the project.

GitHub is a platform for software development and is used by many open-source projects. It offers lots of tools like a version control system, issue tracker, wiki, and code reviews, but it does not offer build services. However, Travis CI offers build services for GitHub projects. This paper describes how GitHub and Travis CI can be used together to create a continuous delivery pipeline for open-source projects. It also discusses the limits of this approach.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.9 [**Software Engineering**]: Management—*productivity, programming teams, software configuration management*

Keywords

continuous delivery, open-source, GitHub, Travis CI

1. INTRODUCTION

Open-source projects need to release their software at some point. Since many developers work on the same code base, this can become a problem, especially if the developers do not know each other personally and many developers do not work full time on the project. If the project has no real concept for the publishing of releases, it is likely that the releases become very infrequent. This also means that bugs cannot be fixed quickly and new functionality has to wait a long time until it is released. A manual release process might also lead to releases of buggy versions of the software. If the project is broken, meaning it is too unstable to be released, and the developer who publishes the new release does not check whether the project is stable, meaning not broken, an unstable release might me published. To prevent these problems, this paper examines how continuous delivery can be integrated into the common workflow of open-source projects to frequently publish new releases.

In this paper, we first describe the principles of continuous integration and continuous delivery. We investigate the common workflow of open-source projects and how it can work with continuous delivery. Afterwards, we investigate how GitHub and Travis CI help us to practice continuous delivery for open-source projects. Finally, we discuss the limits of this approach and conclude the paper.

2. CONTINUOUS DELIVERY

Integration conflicts are expected in software development. As multiple people work at the same code base at the same time they apply interfering changes to the code. These conflicts need to be resolved.

Continuous Integration helps to minimize integration conflicts. This is done by integrating all changes to the mainline as early as possible [16, p. 55]. This means that every developer integrates their own changes with the mainline at least once a day [13]. Thereby, integration conflicts consists at most of the work of one day, so continuous integration prevents big merges.

As all developers work on the mainline it is necessary to verify that the mainline is always stable [13]. To do so, the code is built and tested as soon as new commits arrive at the mainline. If the build or one of the tests failed, the developers should bring the mainline in a stable state as soon as possible, since all developers depend on the stability of the mainline. To lower the risk of committing unstable code to the mainline, each developer should run the tests before each commit. It is worth noting that the tests need to cover a good portion of the code to make the passing of the tests meaningful. Furthermore, all tests should finish quickly, to give fast feedback to the developers.

Continuous delivery helps software developers to frequently release a new stable version of their software. To do so, the build, test and release processes are automated. A build in-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

frastructure which runs these automated processes is called the continuous delivery pipeline. It consists of multiple stages.

The commit stage is the first stage of the continuous delivery pipeline. It realizes the build automation of continuous integration: It builds the application, executes a set of tests and provides fast feedback to the developers. As in continuous integration, the test suite of the commit stage should cover most of the code, so the passing of the tests is meaningful.

The build process and test execution should finish fast, so the developers get fast feedback from the commit stage. This does not mean, that long running tests cannot be used with continuous delivery, but the commit stage is the wrong place for them. Fast feedback is important for the developers, so they are able to react fast if they break the mainline.

The last step of the commit stage is to make the build artefacts available for the other stages of the continuous delivery pipeline. Hereby, it is ensured that the application is built only once per pipeline execution, so all stages of the pipeline use the same artefacts.

All build artefacts should be traceable. The reason for this is, that each build artefact might be part of a release, if it passes all stages of the continuous delivery pipeline. To make all artefacts traceable, the commit stage adds the build number and commit hash to the name of each build artefact before it makes the artefacts available to the other stages.

The second stage of the continuous delivery pipeline is automated acceptance testing. This means the project is verified to provide functionality that is valuable to the users [16, p. 187]. So, as opposed to unit tests, acceptance tests do not verificate requirements of developers but validate requirements of the users. There are tools available for automated acceptance testing. One of them is FitNesse [3].

The third stage of the continuous delivery pipeline tests non-functional requirements. This stage is the right place for long running tests, for example performance tests. If it is possible, the application should be installed on production like environments as this produces more realistic test results.

The last stage publishes the new release. As this is the last stage of the continuous delivery pipeline, a new release is only published if all other stages passed. It is worth noting, that the release of a new version of the application is not the same as the deployment of the application. A release makes a new executable version of the application publicly available, while a deployment installs for example a web application to a publicly available web server.

Continuous delivery is an extension of continuous integration. Continuous integration ensures, that the mainline is always stable. However, the tests that are executed with continuous integration cannot provide sufficient feedback on the production readiness of the application, because they should finish fast. Therefore, mainly unit tests are executed, but just a few or no acceptance and non-functional tests. The automatic execution of these long running tests is added in the continuous delivery pipeline.

The introduction of continuous delivery simplifies the process of releasing the application. If all stages of the pipeline pass, a new release is published automatically, so a release is no longer a significant event. This is an advantage of continuous delivery over continuous integration. With continuous integration the mainline is always stable, but the latest stable version of the application is not available as executable download most of the time.

3. WORKING IN OPEN-SOURCE PROJECTS

Developer communities of open-source projects can be diverse. The developers may not know each other personally, they may not even live on the same continent. One may work on the project as its full time job while another may want to gain first experiences in software development. So it is a challenge to work with all the different people.

In general, it is possible to separate the developers in two main groups [16, p. 411]. One group is very active and has good knowledge of the project. These are the main developers. The other group has less knowledge of the project and is less active. These are the hobby developers and they mainly consist of users which might for example report a bug, provide a bugfix, ask for a feature or provide a user interface translation. As time goes by, a hobby developer might become a main developer and vice versa.

In open-source projects a strategy is needed to work with the main and hobby developers on the same code base. Since the main developers are very active, they can use continuous delivery. This is not so easy for the hobby developers since they are not so active and their changes might take a long time, so these changes cannot be integrated in the mainline once a day. Furthermore, it is also a security risk for the project to allow write access to the mainline to everyone.

For a long time patches have been sent via email to propose changes to open-source projects. One of the main developers of the project had to review the changes and apply the patch manually. This enabled everyone to collaborate, but also protected the mainline from malicious changes at the same time. Additionally, there was a chance to detect bugs during the code review.

Since distributed version control systems like git and mercurial are popular, nobody needs to send patches via email. Instead, a hobby developer can clone the main repository, commit their proposed changes and then allow read access to the main developers. A main developer is able to pull the changes from the repository of the hobby developer, review the changes and try to integrate them. If it is possible to integrate the changes, the main developer will push the integrated changes of the hobby developer to the main repository. So, using distributed version control systems has the same advantages as the sending of patches (code review, no direct write access to mainline), but they are also more convenient for the developers to work with.

In distributed version control systems, the development of a feature in a separate branch is called feature branching [14]. A feature is not necessarily new functionality, but it can also be for example a bugfix. The advantage of feature branching is, that the developers do not disturb each other during the development of different features. They can work on the feature until it is done and integrate it with the mainline afterwards. The simultaneous development of different features in separate repositories is also feature branching.

The problem with feature branching is the long living branches, which may live for several days, weeks, month or even years. These branches can occur when too big features should be implemented. Another reason for the occurrence of such branches is, that the development of the feature has been paused for a while and should be resumed later. The problem with these branches arises as soon as they are merged back into the mainline, because not only the feature

Figure 1: The GitHub Flow [5]

branch has changed but also the mainline has changed a lot. Functionality that used to be in the mainline might be removed or the behaviour might have changed. If the new feature uses such functionality, the integration becomes very difficult. In general, the integration conflicts become bigger as the lifetime of the feature branch increases.

Feature branches are not always bad to use. They are useful, if each feature is kept small and all feature branches have a short lifetime. This prevents big integration conflicts, because the mainline has not changed significantly since the branch was created and the branch itself does not introduce a lot of changes. However, it requires experienced developers that understand the risk of long lived feature branches to successfully use feature branching.

Feature branching and continuous delivery seem to represent two different concepts [16, p. 410-412]. A key part of continuous delivery is to integrate the changes of all developers as early as possible, while feature branches help to separate the changes until the implementation of a feature is done.

However, to use continuous delivery in open-source projects we need to combine continuous delivery with feature branching. By using only short lived feature branches continuous delivery can still be realized, because we are still able to integrate often with the mainline.

This concept works for the main developers and the hobby developers. The main developers can work directly on the main repository since they have write access. They can use short lived feature branches during the development of small features. The hobby developers can clone the repository and offer their proposed changes to the main developers by giving them read access. The main developers can review the code, integrate it and push it to the main repository. Since hobby developers do not work full time on the project it might happen that their feature branches exist for a longer period of time. However, that should not cause too big integration conflicts, since hobby developers will do minor changes most of the time [16, p. 411].

4. GITHUB

GitHub is a platform that provides a set of tools for collaborative software development [4]. Some of the features of GitHub are a version control system, an issue tracker, a wiki, support for code reviews and an API for third parties.

On GitHub each user owns a set of repositories. By default everyone can read public repositories, but only the owner has write access. However, the owner can add other users as collaborators, which enables them to write to the repository as well. The owner and the collaborators of a repository are the main developers.

Since not all users have write access to all repositories, GitHub provides the forking mechanism. If a hobby developer wants to write to a foreign repository, they can fork it and then commit changes to the fork. A fork clones the main repository to the hobby developer's own user account. Thereby, the hobby developer can make changes without having write access to the main repository.

GitHub Flow is a workflow that is proposed by GitHub to contribute changes to an open-source project [5]. It is visualized in figure 1. If a hobby developer wants to contribute a change they should follow these steps:

1. Create an issue.
2. Fork the repository and create a branch.
3. Implement a first version.
4. Create a pull request.
5. React to feedback.
6. Wait for the merge.

Before a developer starts to implement a new feature, they should create an issue and describe what they want to implement. This enables the main developers to keep track of all changes that are currently in progress. Furthermore, the main developers can help to refine the idea and to find the files for the changes. Also, they might not be interested in our changes. In this case the developer that proposes the changes can decide to do the changes anyway and provide them in an own repository, or they can decide to drop the changes which saves a lot of probably unnecessary work. It is important to keep track of the size of the feature. If it grows to big, it should be split into multiple features.

Now, that the changes are specified the implementation can begin. Therefor, we create a fork of the repository if necessary, create a new branch and implement a first version of the new feature in the new branch. A main developer can skip the step of forking the repository as they have write access to the main repository. All changes that are made to implement the new feature should be committed in the new feature branch. This enables the main developers to review the changes before they are integrated into the mainline. Before the next step is started, the developer that implements the feature should do this as good as they can. Ideally, they completely implement the feature and the corresponding test cases. A good first version of the feature implementation helps the main developers to focus on code reviews that really need help.

As soon as the first version is implemented, the developer that implemented the feature can create a pull request from the feature branch. Thereby, they ask for a code review of the implementation and for feedback about what still needs to be changed before the feature is integrated into the mainline. To support this discussion, GitHub provides a discussion page for each pull request. The pull request is bound to the branch from which it was created, so the changes that are made in this branch after the pull request was created are reflected on the pull request's discussion page. This helps to keep track about which changes are implemented as a result to which suggestions. If the branch that is associated with the pull request can be merged with the mainline without any conflicts, GitHub offers a button to merge the branch with the mainline, which also closes the pull request. It is a good idea to link the issue from step one and the pull request to each other to know what belongs together.

A key part of this workflow is, that the developer which implemented the feature keeps track of the pull request's discussion page until it is merged into the mainline or finally rejected. They need to react to feedback and adjust the implementation as it is requested by the main developers. If the developer that implemented the feature stops working on it before it gets integrated into the mainline, the feature might never get merged at all and the time that was spent on it becomes wasted time.

This workflow realizes the concept described in section 3. It allows us to work with feature branches and to review the code before it gets integrated into the mainline. However, it is important to always implement only small features and to break bigger features into multiple small ones to be able to quickly finish the feature implementation which results in short lived feature branches, so big integration conflicts are prevented. Up to this point, we ensured that we integrate often. The other part of continuous delivery is to have an automated build, test and release process.

5. TRAVIS CI

Travis CI is a continuous integration service which integrates with GitHub [6]. It is able to automatically build, test, release and deploy software. It supports multiple languages, build tools, deployment providers, and notification mechanisms. Every change in a GitHub project can trigger the execution of the build lifecycle in figure 2.

First, Travis CI prepares the host on which it will execute the build. It restores a snapshot of a virtual machine, to ensure that the environment is clean for each build. Afterwards, it clones the GitHub project that should be built,

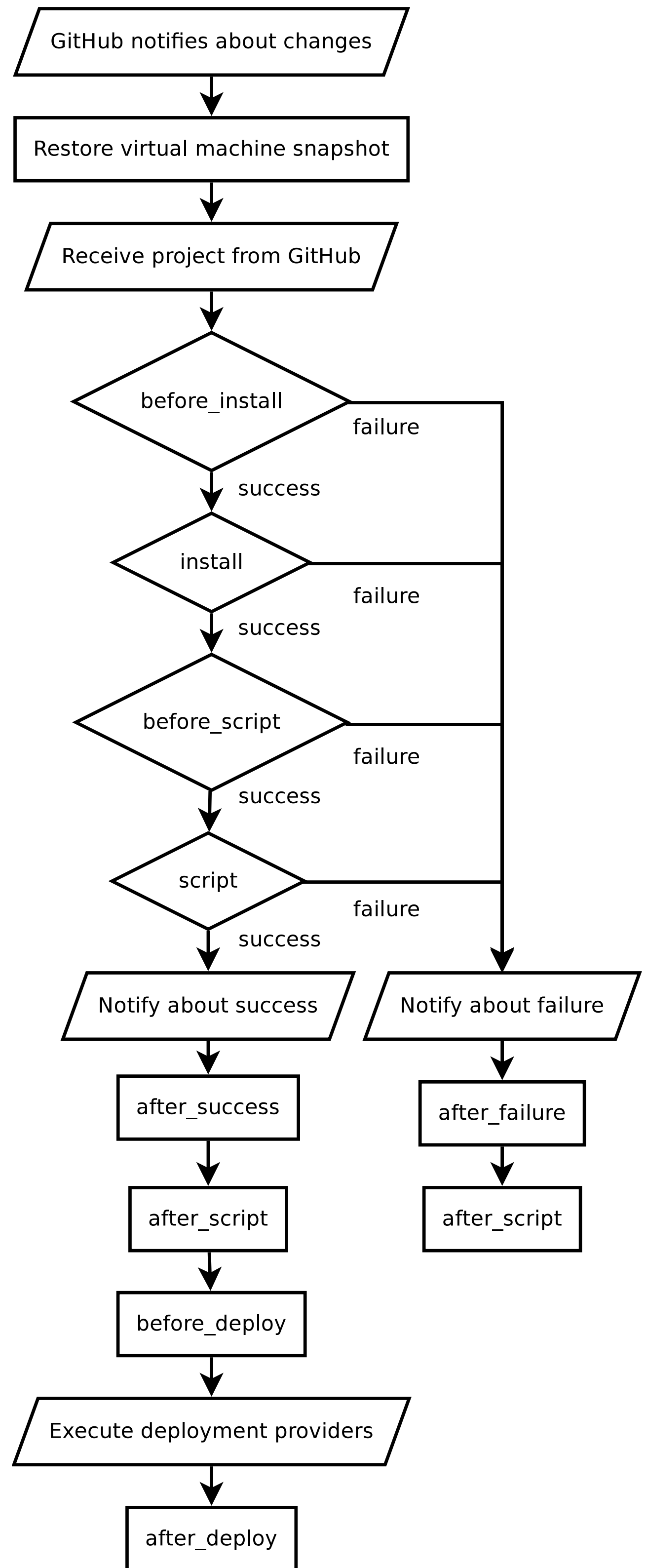


Figure 2: Travis CI flowchart, created with the description in [8] and [12].

changes the current directory to the repository directory and checks out the correct commit.

Next, the project dependencies are installed. Therefor, Travis CI executes the *install* commands. What is executed depends on the language of the project. For example, if Travis CI is configured to build the project as a Java project and a `pom.xml` file is found in the root directory of the project, Travis CI will use Apache Maven to install the project dependencies. Many other languages and build tools are supported [11]. However, it is also possible to overwrite the *install* commands with self defined commands. It is worth noting that Travis CI can be configured to execute custom *before_install* commands before the *install* commands. These can be used to prepare the system without overwriting the default *install* commands.

After the installation of dependencies, the build and test process is triggered. Travis CI uses the *script* commands for this task. Similar to the *install* commands, Travis CI automatically determines which build tool should be used, based on the configured language and the available configuration files. Also, the developer can overwrite the *script* commands and configure *before_script* commands.

Travis CI also allows to define a more complex operation as a build step. Since the operating system which is used by Travis CI is a Linux distribution, it is possible to set each of the commands to a shell script that is placed in the GitHub repository. In the shell script the developer can define a set of custom commands.

This build step can also be used to ensure the traceability of all artefacts, as it is recommended by continuous delivery. Therefore, a shell script can rename the artefact files, so each filename contains information like the branch, tag or commit hash which was build and the build number of the Travis CI build. For this purpose, Travis CI offers environment variables like `TRAVIS_BRANCH`, `TRAVIS_TAG`, `TRAVIS_COMMIT` and `TRAVIS_BUILD_NUMBER` [1].

The further build lifecycle is determined by the success or failure of the previous steps. As depicted in figure 2, the build fails if any of the *before_install*, *install*, *before_script* or *script* commands failed. The success or failure of a command is determined by its exit code. The exit code zero means success, while all other exit codes mean failure.

If the build failed, the developers are notified about the failed build, the *after_failure* commands are executed and the *after_script* commands are executed. By default, Travis CI sends emails to notify about the build status. However, is can be configured to send messages to an IRC channel and to call webhooks to inform a webserver about the build status [7]. Custom notification mechanisms can be executed via the *after_failure* commands.

If the build succeeded, the developers are notified about the successful build, the *after_success* commands are executed and the *after_script* commands are executed. The notification mechanism works as described above for the failure of a build.

To publish a new release after a successful build, Travis CI offers deployment providers. For example, it can upload the build artefacts directly to Amazon S3 or Google Cloud Storage. Furthermore, Travis CI can attach the build artefacts to a release on GitHub. Currently, there are about 20 different deployment providers available [9]. Furthermore, the developer can configure a custom deployment via the *after_success* commands. There exists a setting in the configuration file to specify conditions when a deployment should be executed. For example, the developer can restrict deployments to a specific branch or to tagged commits. This is useful to have a better control over the publishing of releases. Also, the developer can configure custom *before_deploy* and *after_deploy* commands.

To realize the continuous delivery pipeline with Travis CI, the main developers need to create a configuration file for Travis CI, named `.travis.yml`, and place it in the root directory of the repository. They need to configure the proper language, deployment providers and notification mechanisms. The main part of the continuous delivery pipeline is the building and testing process. To get this to work properly, the main developers have multiple options. First, they can use a build tool like Apache Maven to execute the build, unit tests, acceptance tests and non-functional tests. The other option is to use a shell script to execute different tools for building and testing separately. This might be the simpler way if the build tool has limited functionality. Note, that the exit code of the shell script has to reflect the success or failure of the build and test runs.

A great thing about Travis CI is, that it is able to build all branches of a GitHub repository. This also includes branches of pull requests from forks. This simplifies it to detect integration conflicts with feature implementations of hobby developers. Furthermore, it is possible to try to automatically merge the feature branch with the master branch before the build and test process starts [16, p. 79-82]. This can be realized with a *before_install* command. If the merge was successful, the next steps are executed directly on the integrated feature branch. If the merge fails, the build fails. However, this merge is only performed locally on the Travis CI host systems and not pushed back to GitHub.

Travis CI has no user interface to configure the build process. Instead, it uses a configuration file in the root directory of the repository, as described above. This method of configuration has advantages and disadvantages. An advantage is, that no extra rights management is needed for Travis CI. Everyone with write permissions to the GitHub repository, and thereby all main developers, can adjust the build configuration. Another advantage is, that Travis CI always takes the configuration from the commit that it currently builds. Thus, each build should be reproducible and it is easily possible to have different build configurations for different branches. One disadvantage is, that the commit history of the project is cluttered with changes of the build configuration.

The configuration file might contain confidential information, such as API keys for the deployment providers, email addresses, or other notification settings. This might be a problem, because it is placed in the public GitHub repository. Therefor, Travis CI can read encrypted information from the configuration file [10]. To encrypt this information, Travis CI generates a pair of private and public RSA keys for each repository. The private key is only known to Travis CI, so the main developers can encrypt confidential information in the configuration file with the public key. Travis CI offers the command line tool `travis` to perform the encryption. Note, that all encrypted information is not available for forks of the repository as those use another private key. Furthermore, encrypted information is unavailable for builds of branches of pull requests from forks.

6. DISCUSSION

There is a problem with performance tests on Travis CI. Since Travis CI only provides their build servers, there is no production like environment available to execute performance tests that would actually produce meaningful results. However, since the software project is an open-source project, it can be run on any machine, so it is difficult to define a goal for performance tests. However, the Travis CI Team is working on faster and more reliable builds, which can also be execute on own hardware, especially for their enterprise customers [2].

As described at the end of section 5 it is possible to integrate each feature into the mainline before the build is executed. This idea can be extended to not only merge the feature branch that triggered the build, but to always merge all feature branches before a build is started [15]. If the integration of at least one feature fails, the build fails. This behaviour helps to quickly identify integration issues between different features. This idea can also be realized with Travis CI by adding commands to the *before_install* section. However, while this is useful for teams in the business world, it is not realistic to use this in open-source projects, as this implies that the build status of the whole project depends on feature implementations of hobby developers.

7. CONCLUSION

After implementing the continuous delivery pipeline as described in this paper for an open-source project on GitHub, the creation of a new release becomes a no event. After the development of a feature is finished and the feature branch passes all tests, a main developer will merge the feature branch into the mainline. This triggers a new build in Travis CI. As soon as this is finished, Travis CI automatically publishes a new release. Thus, we reached the goal to describe how to realize continuous delivery for open-source projects, so manual releases are no longer necessary. However, we only described the concrete realization for GitHub and Travis CI, since these are very popular platforms for open-source projects. How this can be done on other open-source platforms like SourceForge, Google Code and Bitbucket needs further investigation.

8. REFERENCES

[1] The build environment. `http://docs.travis-ci.com/user/ci-environment/`. Accessed: 2015-01-12.

[2] Faster builds with container-based infrastructure and docker. `http://blog.travis-ci.com/2014-12-17-faster-builds-with-container-based-infrastructure/`. Accessed: 2015-01-12.

[3] Fitnesse. `http://www.fitnesse.org/`. Accessed: 2014-11-20.

[4] Github. `https://github.com/`. Accessed: 2014-11-06.

[5] Github flow. `https://guides.github.com/introduction/flow/`. Accessed: 2014-11-19.

[6] Travis ci. `https://travis-ci.com/`. Accessed: 2014-12-12.

[7] Travis ci: Configuring build notifications. `http://docs.travis-ci.com/user/notifications/`. Accessed: 2014-12-12.

[8] Travis ci: Configuring your build. `http://docs.travis-ci.com/user/build-configuration/`. Accessed: 2014-11-20.

[9] Travis ci: Deployment. `http://docs.travis-ci.com/user/deployment/`. Accessed: 2014-12-12.

[10] Travis ci: Encryption keys. `http://docs.travis-ci.com/user/encryption-keys/`. Accessed: 2014-12-12.

[11] Travis ci: Getting started. `http://docs.travis-ci.com/user/getting-started/`. Accessed: 2014-11-06.

[12] Travis ci: The lifecycle of a travis ci build. `http://docs.travis-ci.com/user/build-lifecycle/`. Accessed: 2014-11-20.

[13] M. Fowler. Continuous integration. `http://www.martinfowler.com/articles/continuousIntegration.html`. Accessed: 2015-01-12.

[14] M. Fowler. Featurebranch. `http://martinfowler.com/bliki/FeatureBranch.html`. Accessed: 2015-01-12.

[15] S. Goff-Dupont. Story branching and continuous integration: a swords-to-plowshares tale. `http://blogs.atlassian.com/2012/07/feature-branching-continuous-integrationgit-bamboo/`. Accessed: 2014-11-21.

[16] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation.* Addison-Wesley Professional, 1st edition, 2010.

Does canary releasing lead to better software and less downtime?

Philipp Franke
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
philipp.franke@rwth-aachen.de

Andrej Dyck
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
andrej.dyck@swc.rwth-aachen.de

ABSTRACT

In our rapidly changing world, where complex software handles almost everything and downtime means losing money, release engineers need strategies to keep the costs as little as possible. This paper introduces canary releasing, which is a way to roll out software for a small subset of users in order to test in production/real-life. The main thesis is why companies should use canary releases and why it leads to better software. This is proven by statistics and leading authorities in fields of releasing engineering.

Keywords

Canary releasing, complex software, beta testing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

Towards Systematic Logging

Jan Simon Döring
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
jan.simon.doering@rwth-aachen.de

Andreas Steffens
RWTH Aachen University
Ahornstr. 55
52074 Aachen, Germany
andreas.steffens@swc.rwth-aachen.de

ABSTRACT

With logs being a rich source of information about a software system, logging plays a key role for many software system management tasks like debugging & troubleshooting, anomaly detection & security and performance monitoring. As different studies discovered, logging is not performed in a systematic way which negatively affects the log quality. Often developers are either not aware of the importance of logging or they simply do not know what to log as they don't get any requirements. First of all, this paper motivates the need for logging and analyzes current logging practices. As the main reason for today's problematic logging practices, a missing logging formalization is identified. By understanding logging as a discipline and not as a requirement, it proposes a logging process that is aligned to the testing process as defined in ISO 29119-2.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids, Diagnostics, Monitors, Debugging*

Keywords

Logging, Guidelines, Logging Formalization

1. INTRODUCTION

Logging, the process of recording information during a program's runtime, is a well-known programming practice [23]. The recorded artifacts are called logs, which, according to the National Institute of Standards & Technology, are composed of log entries containing information related to a specific event that has occurred [12]. Every log entry can be classified as a security, operating system, or application log entry [12]. Independent of their type, every log entry's lifecycle typically consists of four phases, which are performed by different stakeholders. By specifying log statements, developers define the phase of creation. Since developers often are responsible for configuring the logging system, they also influence the second phase of log collection & storage. The third phase which comprises analysis of log entries is generally performed by system operators (except for application debug logs). In the fourth phase system operators typically specify a retention policy determining when to remove the log entry.

Based on the stakeholder's perspective, the log producer (developer) differs from the log consumer (system operator). For this reason, it is important that developers know what information they are required to record. Another challenge is introduced by the unstructured nature of log messages. For example, developers can use arbitrary field identifiers (e.g. date vs timestamp vs creationDate) or different message formats (e.g. XML vs JSON vs plain text) introducing conflicts regarding the processing and comparability of log messages. In that case, system operators would be forced to continuously adapt and reconfigure their log analysis tools. As recent work by Yuan discovered, these challenges are not tackled. Logging is done in an ad-hoc and arbitrary way, which results in log quality problems [27]. Due to the log quality problems this paper improves current logging practices in a systematic way. First of all, section 2 establishes the need for logging in order to motivate these logging improvements. Section 3 analyzes current logging practices and identifies concrete problems, which are addressed in section 4 and section 5 outlining a logging process and tool support. Section 6 concludes this paper.

2. NEED FOR LOGGING

96% of the participants of a survey by Fu et. al [9] strongly agree that logging is important for system development and maintenance, which requires an understanding of the system's runtime behavior (dynamic program analysis). Although there are other possibilities for dynamic program analysis, like automated software instrumentation techniques [5], these techniques often introduce high overhead. As Shang stated, software instrumentation and profiling are often performed by non-system experts with limited domain and system knowledge after the system has been build [24]. For this reason, enormous volume of data is produced, since the non-system expert instrument the code in a blind manner due to their limited system knowledge. Overall, system operators and developers typically only rely on the software system's logs to understand the system's run-time behavior and to diagnose bugs [24].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SWC Seminar 2014/15 RWTH Aachen University, Germany.

Since the unstructured, textual nature of logs provides a rich source of information, logs are multipurpose. As recent work demonstrated, logs can not only be used to diagnose bugs [10] or to understand the system behavior [8], they can also be used for anomaly detection [14], to generate workload information for capacity planning [16] or to diagnose performance problems ([18]). Moreover, they can be used to improve the software quality in general [23]. Overall, logging often is the only feasible way to provide information for different software system management tasks. The importance of logging can also be derived from its commercial acceptance. It is an common industrial practice to request logs upon system failures (i.e. "send error report") or their systems even automatically send logs periodically (i.e. "callhome") [27]. Beside these "convenient tasks", there are also regulations and laws (like PCI DSS [21], FISMA[19]) which systems and their logs need to comply with. Overall, it is crucial not only to log, but also to log the "right" information to make use of them. For this reason, the following section analyzes current logging practices to identify possible problems.

3. CURRENT LOGGING PRACTICES

As J. Arrasz and J. Buch discovered, there are many problems regarding current logging practices [1]. Todays logs often don't have any meaningful content, their huge amount hides important information ("needle in a haystack") and they are rarely index-able and therefore hard to search or evaluate. To analyze current logging practices in a structured way, the following subsections examine if developers log at all respectively which events they logged (section 3.1). Moreover, they analyze which data is included (section 3.2).

3.1 Which events are logged

According to Yuan et al. [27] there is one line of logging code every 30 lines of code (on average and at least in Open Source Software) meaning that developers do log. Fu et. al discovered, that half of their considered log snippets were logged due to unexpected situations [9]. In addition, 57% of their interviewed developers considered exception types and function calls related to exceptions (46%) as important factors for logging decisions. Other factors like security (20%) are considered less important for logging decisions. Overall, developers do log but mainly events they are personally interested in (like exceptions for debugging purposes).

Yuan et al. also analyzed the churn rate of logging code. The code churn measures changes made to a component [17]. In this context, added, modified or deleted logging statements from one version to another version are examined. Yuan et al. discovered that the churn rate of logging code is almost two times higher (1,8) than the churn rate of of the entire code [27]. This fact implicates an active maintenance of logging code, as well as some uncertainty of what events should be logged (since statements are added, moved, etc.). Moreover, 26% of all log modifications are log level adjustments meaning hat developers spend significant effort to (re-) prioritize log messages [27]. Overall, they don't seem to get log specific requirements. Instead, they implement log messages after a failure happened and logs are required. Same goes for prioritizing log messages. Since they seemingly don't get any definition when to use which log level, they have to estimate the cost (importance vs overhead) on their own [27].

Concluding the findings of Yuan and Fu ([27], [9]) regarding which events are logged, developers log in an arbitrary way. It's their subjective decision which events to log. Mainly they log events that are valuable to them, meaning they only consider their debugging context. Missing events or information are added as "after-thoughts" [27]. The following sections analyzes, which concrete information they are missing to record.

3.2 Which data is (not) included

In general, a basic log message syntactically consists of time stamp, source and data. Semantically it comprises state (any program information contained in variables, return values, stack information etc.) and context and should tell what happened and why it happened. According to Chuvakin's five W's of logging [4] and many blog posts ([6], [22], [13]), respectively articles ([3]), it should answer the following:

Who was involved? The log message should include information (*user identify*, *source address*) about the involved user or machine.

What happened? The log message should include the affected system component (*object*), the event importance (*priority*), the cause *action*, the action result (*event status*) and a description.

Where did it happen? The log message should answer where the event happened by specifying the *system*, *application*, *component* or code location.

When did it happen? The log message should include a *timestamp* and *timezone*.

Why did it happen? By providing a *reason* the log message helps to answer why the event happened.

Therefore an example of an useful log message would look like the following:

```
2015/09/01 10:58:00AM GMT+1,
priority=3,
system=mainserver ,
module=authentication ,
source=127.0.0.1,
user=mustermann ,
action=login ,
object=database ,
status=failed ,
reason="password incorrect"
```

Although these guidelines exist and even different organizations published their logging guidelines (Microsoft [15], Open Web Application Security Project [20], IBM [2]), some information are missing in real world logging. Shang noticed that around 80% of the logging statements provide the meaning of the log lines. Other aspects, like the cause, context, and impact are often omitted [24]. As a result, developers modify 36% of their log statements at least once (according to a study by Yuan et al. [27]. The majority of these log modification (27%) are a result from adding new variables to gather more context information. Fu et al. explain this fact by developers only consider function (69%)

and block level (61%) as the scope for logging decisions [9]. As a result, Petersson depicts the following log message as an real-world example [3]:

```
Aug 11 09:11:19 xx null pif ? exit! 0
```

This example not only illustrates that developers often log data only valuable and understandable (!) to them, but also demonstrates the difficulty of parsing the log message's content, since developers don't follow a consistent format but use an arbitrary textual representation. For this reason, every log message content modification might forces system operators to adapt their log processing applications.

Overall, developers are not only not aware of the system operator's log importance, but also subjectively decide which information they need. For this reason, even today the power of logs cannot be fully leveraged. To tackle these problems a systematic logging approach is required, which the following section introduces.

4. LOGGING PROCESS

To introduce an organization-wide awareness about the importance of logging and to improve the log quality, an formalized logging process is needed, which this sections proposes. As Yuan et al. identified, logging plays a significant part in software evolution, because of its ad-hoc practice. Thus, a formalized logging process would not only improve log quality, but also shorten development time.

Since Logging is structural similar to testing, the outlined logging process is influenced by the Testing Process as defined in ISO-29119-2 [11]. Figure 1 illustrates the proposed logging process. In general, the process comprises different stages like the ISO testing process. The first stage is about creating and maintaining an **Organizational Logging Policy** which defines the scope of logging in an organization by specifying high-level principles and goals. Because the Logging Policy should be understandable on executive management level, it doesn't contain any technical details. For example, an organizational logging policy could state "All logging should follow the CEE 1.0 specification". Based on the Logging Policy, the second process stage defines an organizational-wide **Logging Strategy**, which is a technical document that details the expected logging practice across the organization. It provides guidelines and reusable templates for project specific logging plans which are created on the third stage of the logging process. Both the Logging Policy and the Logging Strategy ensure that a consistent logging approach is maintained across all projects of an organization.

While the first two logging process stages are performed at organizational level, the third stage is performed on project-level. Thinking of smaller organization, not every organization requires to establish a custom logging policy and strategy. Instead, existing guidelines and best practices ([6], [22], [13], [15], [20], [2]) can be adopted. No matter what is used as a basis for the logging strategy, the third stage is heavily influenced by the logging strategy. Since this third stage is at the core of our logging process, it is described separately in the following subsection.

4.1 Project Logging Process

The project-level logging process is illustrated in figure 1. It comprises three major activities (Project Log Plan Development, Implementation, Log Analyse) which are performed in a continuous manner. Prior to describing these activities, the following introduces main roles and responsibilities in the logging process.

4.1.1 Roles and Responsiblities

As logging is a cross-functional team activity, different stakeholders are involved in the logging process. Using their relationship to log messages as a taxonomy, two different stakeholder can be identified.

Developers implement log statements. Normally they use logging only for debugging purposes and are not aware of other stakeholders and their log related interests or system limitations (e.g. resource limitations). To improve their awareness and to ease implementation, the logging process requires them to communicate with other stakeholders and plan logging upfront for defining concrete requirements. Since they are expected to deliver high-quality software, they require field-knowledge answering how the software performs in operation. As Shang stated, this field-knowledge can be gathered by using logs if developers and system operators cooperate [23]. Therefore developers are not only responsible to realize logging specific requirements, they are also responsible to specify them in a cross-functional manner.

System operators use logs for management tasks of their system, which they are responsible for. They analyze the log messages content to gather information dealing with performance, security or other run time aspects. Since they do not implement log statements, their analysis possibilities heavily depend on the logged information specified by developers. To use these information at all, they require developers to log in a consistent way. To support developers and to gather development knowledge to enhance their log message understanding, operators are responsible to plan and define concise logging related requirements and to communicate field knowledge to developers.

4.1.2 Activities

The project-level logging process comprises several activities, which are detailed below.

Log Plan Development To tackle the current ad-hoc logging practice (see section 3), the proposed logging process relies on planning. Before implementation, the project's logging approach and the logging requirements are specified in the log plan, which basically is a project specific (tailored) logging strategy. Thus, the log plan is a document that specifies both technical and functional log related aspects. For the technical aspects it defines logging mechanism to be used throughout the project, message encapsulation and transport formats. In addition, other technical aspects like log message storage and disposal are specified. The functional aspects comprise concrete requirements which events to log, how to exactly log (e.g. use dedicated loggers for different contexts) and define reusable log

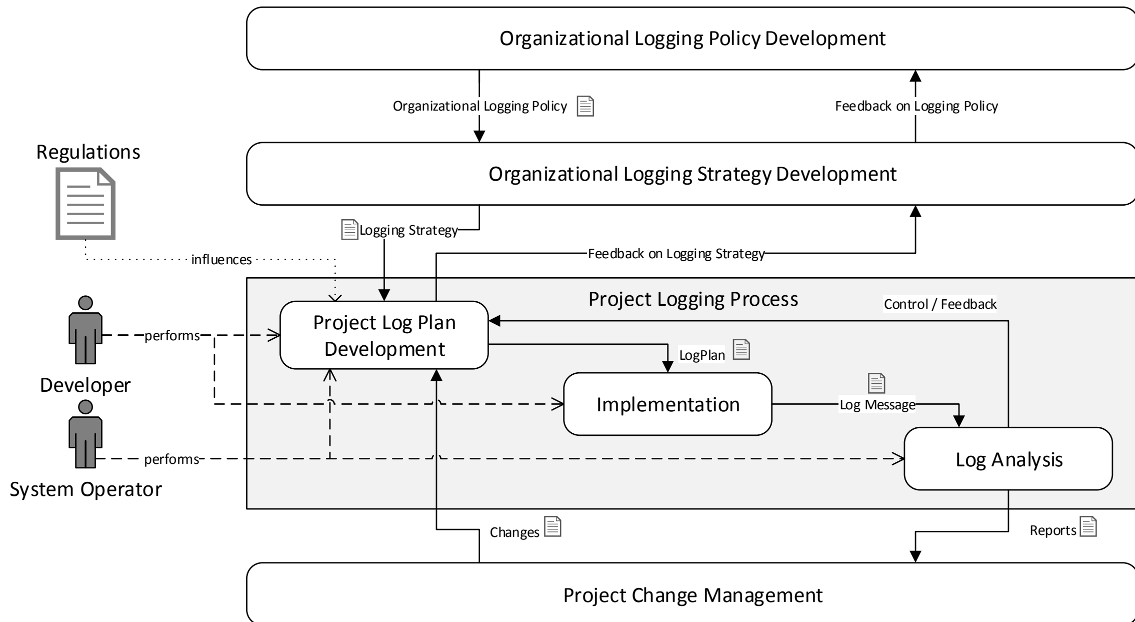


Figure 1: Logging Process

statement templates. Both, developers and system operators perform this activity together. This ensures not only their logging commitment and awareness of each "world" but also enables to specify and communicate(!) their stakeholder specific requirements. That way different logging contexts (Debugging, Security, Performance, Compliance etc) can be addressed. As stated, the log plan is a tailored logging strategy. Since the tailoring not only bases on the stakeholder requirements, but also on (external) regulations and laws the projects needs to comply to, figure 1 explicitly illustrates that the log plan development is influenced by regulations like PCI DSS (Data Security Standard [21]) or FISMA (Federal Information Security Management Act [19]).

Implementation After one iteration of project specific log planning is done and a log plan has been created, developers can begin to implement logging and the required log statements. One important aspect of coding these statements is documentation. As Shang identified, one way to improve the log quality and analysis possibilities is to provide system operators with development knowledge [23]. Therefore, developers are required to document their log statements to persist development knowledge. Referring to the high churn rate of logging code of current logging practices (see section 3), it is important to note that evolving log statements must be communicated upfront and of course still need to comply with the log plan.

Log Analysis While run-time, log processing applications which assist in storing, querying and analyzing logs (e.g.Splunk [25] or LogStash [7]) collect occurring log events. System Operators then gain field knowledge by using these tools to monitor the log event producing application and to evaluate the data. Most importantly, system operators need to share this field knowledge with developers enabling them to improve the software quality. In addition, they have to use any gained information as feedback to reiterate the logging process to adjust the log plan. For example, they could experience that the amount of performance related log messages with an informative priority, i.e. Log Level INFO, is too huge such that these messages cannot be leveraged. For this reason, they reiterate the log planning phase reducing priority and filtering mechanisms or they vote on an adaptive logging approach (logging on-demand [9]), which only emits a log message, when certain (performance-related) criteria a met, e.g. response time > 200ms.

Log Strategy Feedback In addition to the log plan feedback originating from the Analysis, there is feedback for the organizational log strategy. Because Log Strategy Feedback is only applicable if an organizational log strategy is employed, which must not be the case thinking of smaller organizations, this feedback activity is described separately, although it is part of the log planning activity (since log planning is about tailoring the organizational log strategy and therefore the appropriate place to perform the feedback activity). Overall, developers and system operators perform this log strategy feedback activity. Based on the experiences they made while realizing logging for a specific project, they are able to give generalized guidelines or improved practices. Taken an organization into account that mainly develops software for U.S. federal agencies, every projects needs to comply with the Federal Information Security Management Act (FISMA),

which also states logging requirements [12]. In this case it would be reasonable to facilitate compliance with FISMA by specifying a project independent way in the logging strategy and not to plan the way of reaching compliance for every single project over and over again.

Change Management System Operators analyze collected log data for example to detect application failures. They report these failures as bug reports enabling developers to trace and fix problems. Of course there are other kind of reports. According to Chuvakin, popular report categories beside failure and critical error reports are: Authentication and Authorization Reports, System and Data Change Reports, Network Activity Reports, Resource Access Reports and Malware Activity Reports [4]. In reaction to these reports, code modifications might be required. For example there might be bugs, security vulnerabilities or some resources are exhausted because of inefficient implementations. Fixing these problems requires code modifications and possibly added or modified log statements. Moreover, software evolves over time (thinking of new releases / features). To this reason, developer and system operator perform change management whenever a code modification is required. They plan and agree upon code modifications and by reiterating the logging process they also consider required logging modifications and ensure a consistent logging approach.

Overall, these activities respectively the log planning ensures that developers don't have to log in an ad-hoc manner. Assuming the log plan is accurate, they don't need to decide which information and events should get recorded. They just have to follow the log plan. With the logging process being defined, the following section shortly analyzes if there is tool related logging support.

5. TOOLING

Considering the proposed project logging process, only log message analysis related tools are available. As one participant of a survey made by Fu et al. explicitly stated ("*...more automatic for writing logs, instead of writing all by myself*") [9], developers need and want support regarding the log statement implementation. With the log plan offering reusable log statement templates and defining concrete rules for log message priorities etc., developers get some support. Nevertheless, they would benefit from tool support regarding the realization of the log plan.

Some tool related research has been made. Yuan et al. presented a tool for enriching existing log statements [28] which could be integrated to an IDE providing developers with valuable feedback on their log statements. Park et al. introduced a tool to automatically insert log statements [26], but this approach is limited to failure diagnosis leaving the need for future work in this area. In addition, both approaches are not designed to incorporate the project-specific log plan.
As logging might produce huge amounts of data resulting in a performance overhead, operators should have tools to easily configure the log level during run-time to reduce the produced data amount. Ideally, this configuration can be automated in a proactive manner dynamically adapting the log level based on certain conditions. But to the best of our knowledge, these tools don't exist yet.

6. CONCLUSION & FUTURE WORK

Logs are a means to communicate important run-time behavior. Their rich nature introduced many log processing applications (e.g. Splunk, LogStash) that assist system operators in storing, querying and analyzing logs. As research demonstrated with a wide variety of use cases for log data, logs are multipurpose. But leveraging their rich but textual nature introduces many challenges. Based on the stakeholder perspective, the log producer (developer) often differs from the log consumer (system operator). To this reason it is important that developers know which information system operators require them to record for analysis purposes. The unstructured nature of log messages introduces another challenge. For example, developers can use arbitrary field identifiers (e.g. date vs timestamp vs creationDate) or different message formats (e.g. XML vs JSON vs plain text) introducing conflicts regarding the processing and comparability of log messages. In that case, system operators would be forced to continuously adapt and reconfigure their log analysis tools to make use of the log data.

Even today these challenges are not tackled. Logging is done in an ad-hoc and arbitrary manner resulting in log quality problems. Feeling they own the logs, developers often change and update log statements making an useful analysis really difficult. In addition they rarely get any logging specific requirements leading them to record only information they need or they understand. Although if all required information are included, understanding these messages often requires development knowledge which system operators don't have. Thus, system operators can not make use of these messages. Interestingly, there are many guidelines and tips supporting developers in deciding what, when and how to log, but developers rarely follow these guidelines. Either they are not aware of them or consider logging only for debugging purposes. Since the importance of logging with more and more large scale applications will only grow, it is important to improve current logging practices. To this reason a logging process is needed which formalizes and instrumentalizes logging. This paper outlined such a logging process which is aligned with the testing process as defined in ISO 29119-2.

To cope with many stakeholder related logging challenges, the outlined logging process focuses on communication and planning. In a cross-functional manner, developers and system operators create a project specific log plan which continuously is adapted and details the expected logging practice. Based on this plan, developers implement and realize logging which system operators then make use of during run-time and continuously improving the log quality by giving feedback regarding log messages content. To ease the adoption of the outlined logging process, which needs to be formalized in future work, tool support is required. While tools supporting system operators in log analysis already exists (e.g. Splunk [25], Logstash [7]), tools supporting developers in implementing log statements or enabling system operators to configure logging during run-time don't or barely exist yet, which gives plenty of opportunity and need for future work. Given the logging process and appropriate tools in

future, the power of logging then can be fully leveraged.

7. REFERENCES

[1] J. Arrasz and J. Buch. Die Kunst des Logging. http://jaxenter.de/artikel/logging-production-ready-176454, 2014. Retrieved 31/12/14.

[2] C. Chan. Effective logging practices ease enterprise development. http://geekdetected.files.wordpress.com/2013/12/effective-logging-practices-ease-enterprise-development.pdf, 2005. Retrieved 09/12/14.

[3] A. Chuvakin and G. Peterson. How to Do Application Logging Right. *IEEE Security and Privacy*, 8(4):82–85, 2010.

[4] A. Chuvakin, K. Schmidt, and C. Phillips. *Logging and Log Management: The Authoritative Guide to Understanding the Concepts Surrounding Logging and Log Management*. Syngress Publishing, 2013.

[5] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A Systematic Survey of Program Comprehension through Dynamic Analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, Sept. 2009.

[6] C. Eberhardt. The Art of Logging. http://www.codeproject.com/Articles/42354/The-Art-of-Logging, 2014. Retrieved 09/12/14.

[7] Elastic Search Inc. Logstash. http://www.elasticsearch.org/overview/logstash, 2015. Retrieved 12/01/15.

[8] Q. Fu, J.-G. Lou, Q. Lin, R. Ding, D. Zhang, and T. Xie. Contextual Analysis of Program Logs for Understanding System Behaviors. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 397–400, Piscataway, NJ, USA, 2013. IEEE Press.

[9] Q. Fu, J. Zhu, W. Hu, J.-G. Lou, R. Ding, Q. Lin, D. Zhang, and T. Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, ICSE Companion 2014, pages 24–33, New York, New York, USA, 2014. ACM Press.

[10] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 103–116, New York, NY, USA, 2009. ACM.

[11] ISO. Software and systems engineering – Software testing – Part 2: Test processes. ISO 29119-2, International Organization for Standardization, 2013.

[12] K. Kent and M. P. Souppaya. SP 800-92. Guide to Computer Security Log Management. Technical report, Gaithersburg, MD, United States, 2006.

[13] I. Levent. Application Logging: What, When, How. http://java.dzone.com/news/application-logging-what-when, 2009. Retrieved 09/12/14.

[14] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 24, Berkeley, CA, USA, 2010. USENIX Association.

[15] Microsoft. Logging Guidelines (Windows). http://msdn.microsoft.com/en-us/library/windows/desktop/aa363667%2528v=vs.85%2529.aspx. Retrieved 09/12/14.

[16] M. Nagappan, K. Wu, and M. A. Vouk. Efficiently Extracting Operational Profiles from Execution Logs Using Suffix Arrays. In *2009 20th International Symposium on Software Reliability Engineering*, pages 41–50. IEEE, Nov. 2009.

[17] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering - ICSE '05*, page 284, New York, New York, USA, May 2005. ACM Press.

[18] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, page 26, Berkeley, CA, USA, 2012. USENIX Association.

[19] NIST SP. 800-53. *Recommended Security Controls for Federal Information Systems*, pages 800–853, 2013.

[20] Open Web Application Security Project (OWASP). Logging Cheat Sheet - OWASP. https://www.owasp.org/index.php/Logging_Cheat_Sheet, 2014. Retrieved 09/12/14.

[21] PCI. Data Security Standard. https://www.pcisecuritystandards.org/documents/pci_dss_v2.pdf, 2010. Retrieved 11/12/14.

[22] R. Sethi and N. Bhalla. Building Secure Applications: Consistent Logging. http://www.symantec.com/connect/articles/building-secure-applications-consistent-logging, 2010. Retrieved 09/12/14.

[23] W. Shang. Bridging the divide between software developers and operators using logs. *2012 34th International Conference on Software Engineering (ICSE)*, pages 1583–1586, June 2012.

[24] W. Shang. *Log Engineering: Towards Systematic Log Mining to support thedDevelopment of ultra-large scale software systems*. PhD thesis, Queen's University, 2014.

[25] Splunk Inc. Splunk. http://www.splunk.com/, 2015. Retrieved 12/01/15.

[26] D. Yuan, S. Park, P. Huang, Y. Liu, M. M. Lee, X. Tang, Y. Zhou, and S. Savage. Be Conservative: Enhancing Failure Diagnosis with Proactive Logging. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 293–306, Berkeley, CA, USA, 2012. USENIX Association.

[27] D. Yuan, S. Park, and Y. Zhou. Characterizing Logging Practices in Open-source Software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 102–112, Piscataway, NJ, USA, 2012. IEEE Press.

[28] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving Software Diagnosability via Log Enhancement. *SIGPLAN Not.*, 47(4):3–14, 2011.