Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications

SQAMIA 2019
Ohrid, North Macedonia, 22 – 25. 09. 2019

Proceedings

Department of mathematics and informatics
Faculty of Sciences
University of Novi Sad, Serbia
2019
Volume Editors

Zoran Budimac
University of Novi Sad
Faculty of Sciences, Department of Mathematics and Informatics
Trg Dositeja Obradovića 4, 21000 Novi Sad, Serbia
E-mail: zjb@dmi.uns.ac.rs

Bojana Koteska
SS. Cyril and Methodius University,
Faculty of Computer Science and Engineering
Rugjer Boskovikj 16, P.O. Box 393, 1000, Skopje, North Macedonia
E-mail: bojana.koteska@finki.ukim.mk

Publisher:
University of Novi Sad,
Faculty of Sciences, Department of mathematics and informatics
Trg Dositeja Obradovića 3, 21000 Novi Sad, Serbia
www.pmf.uns.ac.rs

Typesetting: Doni Pracner

Papers are copyrighted © 2019 by the papers’ authors. Copying permitted only for private and academic purposes. This volume is published and copyrighted by its editors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0). Proceedings also appeared in ISBN 978-86-7031-476-4, Faculty of Sciences, University of Novi Sad. The contents of the published papers express the opinions of their respective authors, not the volume publisher or the editors.

Typeset in \LaTeX and Microsoft Word by Doni Pracner and the authors of individual papers.

ISBN: 978-86-7031-476-4
Preface

This volume contains papers presented at the Eighth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications (SQAMIA 2019). SQAMIA 2019 was held during 22 – 25. 09. 2019, at Hotel “Tino Sveti Stefan” in Ohrid, North Macedonia.

SQAMIA 2019 continued the tradition of successful SQAMIA workshops previously held in Novi Sad, Serbia (in 2012, 2013 and 2018), Lovran, Croatia (2014), Maribor, Slovenia (2015), Budapest, Hungary (2016) and Belgrade, Serbia (2017). The first SQAMIA workshop was organized within the 5th Balkan Conference in Informatics (BCI 2012). In 2013, SQAMIA became a standalone event intended to be an annual gathering of researchers and practitioners in the field of software quality.

The main objective of the SQAMIA series of workshops is to provide a forum for presentation, discussion and dissemination of the latest scientific achievements in the area of software quality, and to promote and improve interaction and collaboration among scientists and young researchers from the region and beyond. The workshop especially welcomes position papers, papers describing work in progress, tool demonstration papers, technical reports, and papers designed to provoke debate on present knowledge, open questions, and future research trends in software quality.

The SQAMIA 2019 workshop consisted of regular sessions with technical contributions reviewed and selected by an international program committee. In total 18 papers were accepted and published in this proceedings volume. All published papers were at least double reviewed, and in some cases even quadruple reviewed. We are grateful to all PC members for submitting careful and timely opinions on the papers.

Our special thanks are also addressed to the members of the SQAMIA Initiative (sqamia.org) without who this and previous workshops could not be possible: Zoran Budimac (Novi Sad, Serbia), Tihana Galinac Grbac (Pula, Croatia), Marjan Heričko (Maribor, Slovenia), Zoltán Horváth (Budapest, Hungary) and Hannu Jaakkola (Pori, Finland)

We extend special thanks to the SQAMIA 2019 Organizing Committee from the Faculty of Computer Science and Engineering, Ss. Cyril and Methodius University, Skopje, North Macedonia and the Department of Mathematics and Informatics, Faculty of Sciences, University of Novi Sad, especially to Bojana Koteska for her hard work and dedication to make this workshop the best it can be. Further we’d like to thank Doni Pracner for his patience and diligent work on making the proceedings.

And last, but not least, we thank all the participants of SQAMIA 2019 for their contributions that made all the work that went into SQAMIA 2019 worthwhile.

September 2019

Zoran Budimac, Bojana Koteska

iii
Workshop Organization

**General and Organising Chair**
Bojana Koteska (*Ss. Cyril and Methodius University, North Macedonia*)

**Program Chair**
Zoran Budimac (*University of Novi Sad, Serbia*)

**Program Committee**
- Nuno Antunes (*University of Coimbra, Portugal*)
- Tihana Galinac Grbac (co-chair, *Juraj Dobrila University of Pula, Croatia*)
- Jaak Henno (*Tallinn University of Technology, Estonia*)
- Marjan Heričko (co-chair, *University of Maribor, Slovenia*)
- Zoltán Horváth (co-chair, *Eötvös Loránd University, Hungary*)
- Sami Hyrynsalmi (*Tampere University of Technology, Finland*)
- Hannu Jaakkola (co-chair, *Tampere University of Technology, Finland*)
- Harri Keto (*Tampere University of Technology, Finland*)
- Vladimir Kurbalija (*University of Novi Sad, Serbia*)
- Goran Mauša (*University of Rijeka, Croatia*)
- Anastas Mishev (*Ss. Cyril and Methodius University, North Macedonia*)
- Zoltán Porkoláb (*Eötvös Loránd University, Hungary*)
- João Saraiva (*University of Minho, Portugal*)
- Valentino Vranić (*Slovak University of Technology in Bratislava, Slovakia*)

**Organizing Committee**
- Monika Simjanoska (*Ss. Cyril and Methodius University, North Macedonia*)
- Nataša Sukur (*University of Novi Sad, Serbia*)
- Doni Pracner (*University of Novi Sad, Serbia*)

**Organizing Institution**
*Ss. Cyril and Methodius University*
Faculty of Computer Science and Engineering

**Technical Editor**
Doni Pracner (*University of Novi Sad, Serbia*)
# Table of Contents

- A Survey On Secure Container Isolation Approaches for Multi-Tenant Container Workloads and Serverless Computing ................................................................. 1:1– 1:10  
  Christian Bargmann and Marina Tropmann-Frick
- The Impact of Refactoring on Maintability of Java Code: A Preliminary Review ............. 2:1– 2:11  
  Mitja Gradišnik, Sašo Karakatič, Tina Beranič, Marjan Heričko, Goran Mauša and Tihana Galinac Grbac
- Survey on Static Analysis Tools of Python Programs .................................................. 3:1– 3:10  
  Hristina Gulabovska and Zoltán Porkoláb
- Using Multiplayer Games to Create Locally Secure Communication .......................... 4:1– 4:13  
  Jaak Henno, Hannu Jaakkola and Jukka Mäkelä
- The Impact of Students’ Pre-Knowledge on Learning Computer Programming .............. 5:1– 5:8  
  Marko Hölbl and Lili Nemec Zlatolas
- On Measuring Learning Success of Students with Disabilities in Virtual Environments ...... 6:1– 6:12  
  Mirjana Ivanović, Amelia Bâdică, Maria Ganzha, Marcin Paprzycki, Costin Bâdică and Aleksandra Klašnja-Miličević
- Dynamic Testing of Executable UML Models with Sequence Diagrams ........................ 7:1– 7:10  
  Tamás Jánosi, Krisztián Mózsi, Péter Bereczky, Dávid J. Németh and Tibor Gregories
- Unit and Performance Testing of Scientific Software Using MATLAB .......................... 8:1– 8:9  
  Bojana Koteska, Monika Simjanoska, Ivana Jacheva, Frosina Krsteska and Anasats Mishev
- Software Metrics as Identifiers of Defect Occurrence Severity .................................. 9:1– 9:9  
  Goran Mauša, Tihana Galinac Grbac, Lucija Brezočnik, Vili Podgorelec and Marjan Heričko
- The Technical Debt Management Tools Comparison ................................................. 10:1–10:9  
  Luka Pavlič and Tilen Hliš
- Recent Trends in Software Testing – A Case Study with Google Calendar ....................... 11:1–11:9  
  Bojan Popov, Bojana Koteska and Anasats Mishev
  Ivan Pribela, Gordana Rakić and Zoran Budimac
  Maja Pušnik, Katja Kous, Andrej Godec and Boštjan Šumak
- An Insight on Standardized Patterns in Model Driven Software Development ............... 14:1–14:9  
  Petar Rajković, Ivan Petković, Aleksandar Milenković and Dragan Janković
  Alen Rajšp, Gregor Jošt, Viktor Taneski, Saša Kuhar and Luka Pavlič
- Complexity Analysis of Decentralized Application Development Using Integration Tools ... 16:1–16:10  
  Patrik Rek, Blaž Podgorelec and Muhamed Turkanović
- Fitness Functions and Transformations in an Automated Process .................................. 17:1–17:10  
  Nataša Sukur, Doni Pracner and Zoran Budimac
A Survey On Secure Container Isolation Approaches for Multi-Tenant Container Workloads and Serverless Computing

CHRISTIAN BARGMANN and MARINA TROPMANN-FRICK, Hamburg University of Applied Sciences

Container virtualization has become the tool of choice for running isolated applications in cloud environments. Linux-Containers virtualize at the operating system level, with multiple containers running atop the operating system kernel directly. Therefore, threats to one container are potentially threats to many others. Especially for PaaS and Serverless providers, the secure execution of untrusted workloads on their platform in order to mitigate software vulnerabilities from spreading has high priority. Containers face a variety of different threats, vulnerabilities and historical weaknesses that need to be considered and defended against. This paper presents current approaches to securing container workloads. gVisor, Kata Containers and Firecracker are presented and compared with each other. Although sandbox containers have different attack surfaces such as the container daemon process, network, or storage, this paper focuses on the Linux kernel itself as a vulnerability in sandbox containers and examines how each approach implements protection.

1. INTRODUCTION

Due to their flexibility and scalability, containers have gained popularity in recent years. Despite their name, containers are not completely closed. The guest system of each container uses the same host operating system and its services. This reduces overhead and improves performance, but can cause potential security or interoperability problems. The degree of isolation provided by the Linux kernel combines process isolation with namespaces. This concept works well, but does not close all possible security gaps due to construction, so malware can break out and gain access to the host directly or other containers sharing the same host system kernel. This is particularly critical in multi-tenant scenarios where multiple clients run containerized workloads in a shared cloud environment and the containers can run as isolated processes belonging to different clients on the same shared host. The isolation of runtime environments is also becoming increasingly important for Serverless Computing where the cloud provider provides a runtime environment for executing server-side logic which can be potentially harmful to other runtime environments running on a shared host system.

The Linux kernel itself provides a number of mechanisms to isolate processes from each other and restrict system calls. Although a high level of isolation can be achieved with these mechanisms, the shared kernel remains a security risk [MITRE Corporation 2019] [Heise Online 2019]. The rise of container virtualization led to the discussion whether and how containers could inherit classic hard-
ware virtualization. Time has shown that container virtualization and hardware virtualization are not opposed to each other. Rather, the technologies are complementary, especially when the focus is on security. Hardware virtualization is able to isolate computing resources while Containers focus on isolating processes instead of resources. A fully virtualized system gets its own set of resources assigned to it and minimal sharing takes place. Virtual Machines (VMs) offer a comparatively high degree of isolation and thus security for workload isolation. However, the replication of a hardware environment including the operating system leads to an overhead and the start up time of a virtual machine is significantly slower than with container virtualization. Compared to VMs, containers offer a significantly less isolation level but can be deployed immediately without complex installation processes, therefore container virtualization is not suitable for implementing security concepts [Docker 2016].

To secure container workloads, multi-layered security is necessary and issues from both approaches are needed. On the one hand, the performance of container deployment and on the other hand the isolation of the host system, which a virtual machine offers. In this paper, we have reviewed which approaches are currently being driven forward for the implementation of an additional security layer. We have addressed the question of whether these approaches can be distinguished from each other on an abstract level. First, scenarios for container isolation and currently used security mechanisms for containers are presented and the question why an additional security layer is necessary is answered. On the basis of this, gVisor, Kata Containers and Firecracker are presented. In conclusion, the approaches are compared with each other and an overview of the current development approaches is given.

2. BACKGROUND

This section covers backgrounds for container isolation. It starts with use cases and scenarios for container isolation. It then describes the status of containerized runtime environment security and the need for layered security.

2.1 Use Cases & Scenarios

2.1.1 Sandbox untrusted/vulnerable code. Containers are ideal for isolation on application level. There are use cases where a user wants to run an application with a number of previous vulnerabilities on a host machine. There is a risk that an untrusted attacker may be aware of these vulnerabilities and new vulnerabilities may be easily found. To protect the host system, the use of a highly isolated container environment for the application is useful. Another use case would be a scenario in which code is to be executed on a host machine that cannot be trusted. This may be relevant if access to the source code is not given. The lack of transparency can lead to a security risk. Container isolation is also suitable for protection here.

2.1.2 Defense in depth scenarios. For the processing of sensitive workloads and data, defense in depth is of high relevance. In use cases where the code is trusted, it may still be important to protect it from possible attacks. This can be especially important for medical or financial services.

2.1.3 Serverless Computing Scenarios. Serverless Computing is a cloud computing paradigm where the cloud provider provides a runtime environment for executing server-side logic. Server-side logic is implemented by means of short-lived, stateless functions called serverless functions. The execution environment for these functions is called Function-as-a-Service (FaaS) [Spillner 2018]. Serverless Computing is the ideal of an event-based application architecture - an application architecture that emerges from the current trend in container and microservices architectures [Baldini et al. 2017]. As a result, functions are executed in a serverless application in response to the triggering of certain events. If a serverless function is triggered, the cloud provider dynamically allocates the computing capacity required to execute the function. After the execution has been completed, the computing capacities are
released again. In this scenario, the cloud provider does not know the code executed by users in the runtime environment provided. This code can be potentially harmful to its environment. For serverless platform providers, it is therefore of great interest to isolate the individual runtime environments of users strongly.

2.1.4 Multi-Tenancy Scenarios. There are more than one type of multi-tenancy, and multiple definitions of a tenant. Depending on the tenant scenario, the required degree of isolation may vary. Assuming a multi-tenancy model in which multiple untrusted users (Tenants) may execute untrusted code in a shared cloud environment, thus code deployed by tenants can potentially be executed on the same host machine on which code from other tenants is simultaneously executed. Compared to Serverless Computing, this means that the cloud provider makes the compute resources available to the user, but the user himself can use his own runtime environments, e.g. in the form of his own containers. The cloud provider does not know the user’s application and runtime environment in advance. Unlike multi-tenant scenarios, where users of a cloud environment trust each other and also trust the executed code, e.g. within a company with different departments as tenants sharing a compute cluster, the described model requires a strict isolation of resources belonging to different clients.

2.2 Current Mechanisms used for Container Isolation

Today many mechanisms based on Linux technologies are used to isolate container runtime environments.

2.2.1 AppArmor / SELinux. AppArmor / SELinux are security frameworks for Linux. As Mandatory Access Control (MAC) systems they control applications individually. For these, access rights can be defined in profiles that are finer than the general file rights. Beside the predefined ones, own profiles can be set up. The purpose of AppArmor / SELinux is to protect security-critical applications, i.e. primarily applications / processes with network access, but also office applications which could possibly compromise the system by loading infected documents [Debian man-pages 2018] [Linux man-pages project 2018d].

2.2.2 Capabilities. The basic idea behind Linux capabilities is to break up the monolithic root privilege that Linux systems have had, so that smaller more specific privileges can be provided where they’re required. This helps reduce the risk that by compromising a single process on a host an attacker is able to fully compromise it [Linux man-pages project 2018a].

2.2.3 Namespaces. An instance of a namespace defines a new environment that virtualizes certain operational resources such as process users, the file system, or the network in a very lightweight way. They abstract the respective global system resource in such a way that it looks like an independent isolated instance for a process within the respective namespace. Lightweight means that no hypervisor is needed, but the processes simply do not see the other instances of a resource, but they continue to run in the same kernel [Linux man-pages project 2018b].

2.2.4 User Permissions. For the isolation of container runtime environments the Linux own user system is used. Discretionary Access Control (DAC) is restricting access to objects based on the identity of subjects and groups to which a certain subject belongs to [Rusling 1999].

2.2.5 cgroups. Using "control groups" a user can combine several processes into one group. The user can then provide these processes and all child processes with parameters for specific subsystems. For example, a subsystem is a resource controller that manages the available memory. cgroups can be used to define limits that must not be exceeded by resources, such as a lot of memory. Users can give priority to some resources over others, such as more frequent processing by the CPU. cgroups can also
measure how many resources have been consumed. With the help of cgroups, entire groups of processes can also be controlled, e.g. they stop or continue to run [Linux man-pages project 2018c].

2.2.6 seccomp-bpf. "Secure computing mode" is a simple and effective sandboxing tool. It allows the user to attach a system call filter to a process and all its descendants, thus reducing the attack surface of the kernel. Seccomp filters are expressed in Berkeley Packet Filter (BPF) format [Mozilla Foundation 2018] [Linux man-pages project 2018c].

2.3 Why is Multi-Layer Security is needed for Container Isolation?
Attacks aimed at breaking out of a containerized runtime environment are particularly critical and require a lot of attention. There is always a risk that processes will break out. If this happens, not only the individual container instance is affected, but also all other processes that also run on the host system.

Fig. 1. Multiple layers are needed to form a trust boundary around untrusted code. Containerized applications should have an additional security layer above the container boundary.

The primary target of such an outbreak is the operating system kernel. If an attacker controls it, he has the entire system in his hands. Container technologies like Docker [Docker Inc. 2019] or Rkt [CoreOS 2019] change much faster than the underlying kernel technologies, so newly discovered vulnerabilities in the kernel can affect a number of different kernel versions. The presented security mechanisms are excellent to ensure defense in depth, but the Linux kernel itself remains the biggest vulnerability [Bettini 2013]. It is sufficient for an attacker to exploit a known vulnerability in the Linux kernel to compromise all security mechanisms.

Using a container runtime is therefore not sufficient for isolating applications. To safely isolate applications from each other, one or more additional security layers beyond the boundaries of the container runtime environment are required as shown in Fig. 1.

3. SECURING CONTAINER WORKLOADS
This section introduces gVisor, Kata Containers, and Firecracker as approaches to isolating container workloads. It is being discussed whether the approaches meet the requirements of multi-layer security.

3.1 gVisor
On May 2, 2018 Google released their container runtime sandbox gVisor [Lacasse 2018]. gVisor is a user-space kernel specifically for containers, written in the Go programming language. The software implements a substantial part of the Linux system interface and thus emulates the Linux system call
API in a userspace process. gVisor draws a boundary between the application running in a container and the host kernel by using the runtime environment of the Open Container Initiative (OCI) called runsc.

Applications that run in containers on the host system access system resources in the same way that applications that are not in containers do. System calls are sent directly to the host kernel. Even without gVisor, the kernel limits the access that a containerized application can make to the system resources, but such a kernel still offers a large attack surface. gVisor does not use hardware virtualization to isolate containers, i.e. containers are not packed into virtual machines to provide an additional security boundary and therefore no virtualized hardware is passed to a guest kernel via a Virtual Machine Manager. Instead gVisor uses a kernel that is active as a normal, non-privileged process and supports most Linux system calls. gVisor intercepts application system calls and acts as the guest kernel, without the need for translation through virtualized hardware. Just like in a normal virtual machine, an application running in a gVisor sandbox gets its own kernel and a selection of virtualized devices isolated from the host and other sandboxes. Unlike containers packed in virtual machines, gVisor is more lightweight, but provides a similar level of isolation and therefore security [gVisor 2018]. A major disadvantage of gVisor is that at the time of this paper not all system calls of the Linux System Call API are implemented, which is why not all applications run with gVisor yet. Also, applications that are sandboxed with gVisor create a higher per-system call overhead.

With gVisor, multi-layer security can be implemented. The emulated kernel in the userland represents the first security boundary that an attacker must overcome. The second security boundary is the process isolation mechanisms of the host kernel. Kernel features such as seccomp filters can be used to provide better isolation between host and gVisor kernels. However, they require the user to create a predefined whitelist of system calls. Due to the additional abstraction layer offered by gVisor, filter rules can be used much more universally. Creating system call filter rules for applications that are not known in advance is often a difficult task. The application of filter rules on gVisor as an interface between host and application is easier to configure.

### 3.2 Kata Containers

Kata Containers version 1.0 was released on May 22, 2018 [Bertucio 2018]. The technical foundations of Kata Containers are two other projects that have been in existence for quite some time. The first are the Clear Containers, which Intel launched in 2015 [Clear Linux Project 2019]. Number two is runV from Hyper [Hyper 2018]. Kata Containers is the fusion of both initiatives. The objective of
Kata Containers is to develop a standard implementation of lightweight virtual machines that feel and function like containers, but offer the workload isolation and security benefits of hardware virtualization. It connects both worlds and lets containers run inside slim virtual machines to create a sandbox environment.

The architecture of Kata Containers consists of six elements [Kata Containers 2019a]. The isolation and insertion of another kernel is done by virtualization software using a minimal operating system. A slimmed down version of QEMU [QEMU 2018] is used on which Clear Linux runs. Clear Linux provides the kernel and is optimized for running containers. QEMU and Clear Linux together form the Kata framework. The remaining four components are responsible for the container structure, all developed in the Go programming language and available under Apache license. The runtime environment corresponds to the specifications of the Open Container Initiative (OCI). It can be installed and used parallel to the standard container runtime `runc`. The Kata runtime and the so-called shim form the interface to container tools such as Kubernetes or OpenStack. The shim as a kind of auxiliary construct enables the classic container tools to get an interior view of the processes of the virtual machines. Without this help, tools like Kubernetes would not be able to look behind the scenes of QEMU-Lite. Kata Containers uses the proxy and agent to connect to the virtual machine. They communicate via gRPC using QEMU’s serial port. The agent is running as a process in the QEMU instance. It acts as a supervisor for managing containers and processes running within those containers. There is one agent per virtual machine. The proxy is the counterpart on the host side and is used for communication between the hypervisor on the host system and the virtual machine running containers. Also here the Kata runtime environment starts exactly one proxy per QEMU instance. Kata Containers combines the properties of virtual machines and containers. Virtualization protects the kernel of the host system. However, a disadvantage is that the high number of components required for managing and communicating containers within the virtual machine is another potential attack surface. The kernel and operating system used by Clear Linux must also be kept up to date in order not to be affected by possible security vulnerabilities.

With Kata Containers multi-layer security can be implemented. The hypervisor provides an additional security boundary and thus increases the security of the sandbox. This makes it more difficult for a harmful process to break out. A sandboxed process must first convince the guest kernel to trigger a malicious virtual machine exit. This also significantly reduces the attack surface at the same time. Afterwards the hypervisor has to be exploited to break out of the sandbox and gain control over the host system. The degree of isolation by the first security boundary is reduced with the number of rights a container receives within the virtual machine. If the user is able to start his own guest systems in the virtual machine, the first security boundary can be omitted.

3.3 Firecracker

On November 27, 2018, Amazon Web Services introduced Firecracker, a virtual machine manager that enables secure, multi-tenant, minimal overhead execution of container and function workloads, as an open source project [Arun Gupta 2018]. Amazon’s own serverless computing platform Lambda uses Firecracker as a foundation for deploying and operating sandboxes that run untrusted code from customers. Firecracker boots a minimal kernel config without relying on an emulated bios and without a complete device model [Firecracker 2018b].

The main component of Firecracker is its own hypervisor which is directly based on Linux Kernel-based Virtual Machine (KVM) capabilities. The foundation for this is the Crosvm project [Chromium 2018] by Google’s Chromium team, written in Rust programming language and used for Linux applications in Chrome OS. Firecracker is also completely written in Rust. The Firecracker application utilizes the hypervisor to set up and run minimal virtual machines. Firecracker itself is started by an
application called Jailer, which provides and configures system resources such as cgroups. The Jailer also manages permissions, or withdraws them from the system as soon as they are no longer needed. In addition, Jailer sets seccomp-bpf filters to restrict guest code access [Firecracker 2018c]. Kernel namespaces are used as an additional barrier. The communication with Firecracker runs via a Restful API, with which the virtual machines can finally be started and controlled. By default, the virtual machines use a virtualized CPU core and 128 MB RAM, which can be adjusted via the API endpoints. In addition, the VMs only use a Virtio-provided network device and a block device, both of which have rate limiting. There is also a serial console and a minimal keyboard driver to reset the virtual machines. To start a VM, users need an unpacked kernel image and a root file system. Alpine Linux is a Linux distribution based on musl and BusyBox, which focuses on security, simplicity and resource efficiency and also enjoys great popularity as a basic image for containers. At the time of this paper, Firecracker still only uses Intel hardware virtualization, but support for AMD and ARM hardware is expected to follow. Work is also underway to integrate Firecracker into existing container ecosystems such as Kubernetes or OpenStack [Firecracker 2018a].

Firecracker implements multi layer security by defining different nested trust zones. The jailer process sets up system resources that require elevated permissions, drops privileges, and then executes into the Firecracker binary, which then runs as an underprivileged process. In addition, seccomp-bpf filters are used to limit system calls to the guest kernel. This represents the first security boundary. In order to take over the host system, an escape from the virtual machine must be successful and thus represents the second security boundary.

4. COMPARISON OF CONTAINER ISOLATION APPROACHES

Although their concrete architecture is quite different, all three projects follow a similar approach. They build an additional layer between the application to be isolated and the kernel of the host system. The main difference between the three projects presented lies in the virtualization technology used to form the isolation layer.
Kata Containers chose QEMU as the foundation for its virtualization technology using "QEMU light". A sandboxed container runs within a minimal, lightweight virtual machine. Via an internal agent process and an external shim, runtime commands and IO request are intercepted. Firecracker follows a similar approach. Again, virtual machines are used to separate sandboxed applications from host kernels. Unlike Kata Containers, Firecracker does not use QEMU, but implements its own virtual machine manager that uses Linux Kernel-Based Virtual Machine (KVM). It emulates a minimal device to achieve low latencies when starting the VM and low memory footprint on the host system. At the same time, a trusted sandbox environment is provided for an isolated application within the virtual machine. Unlike Kata Containers, Firecracker does not start any containers in the virtual machine. Google's gVisor, on the other hand, has a completely different strategy than the other two projects presented. Neither microVMs like Firecracker nor a combination of virtual machine and container like Kata Containers are used. Instead, a dedicated buffer layer is created for each individual container to provide an additional security boundary between host system and application. The layer created by gVisor runs as a process in userspace and allows the container to perform system calls without directly accessing the host kernel.

Although there are currently several projects with different ideas for the realization of container isolation, it must be mentioned that they are all more or less experimental at the time of this paper. Even though large cloud providers such as Google or Amazon are behind some of the concepts presented, all approaches are at a very early stage of development. Although Firecracker or gVisor are used in the cloud environments of the providers, it is not certain whether the solutions are also suitable for general production environments outside the provider-specific offerings.

An important step for the productive use of sandbox environments is the compatibility to common container tools like Docker or orchestration tools like Kubernetes and OpenStack. Both Docker and Kubernetes use containerd as container runtime. While gVisor and Kata Containers can already be used as plugins for containerd and are therefore compatible with Docker [Docker 2016], this is not yet the case with Firecracker. Currently there is a project to run Firecracker as a plugin for containerd [Firecracker 2019]. At the same time Kata Containers supports the Firecracker hypervisor since the release of version Danke 1.5 on January 16, 2019 [Kata Containers 2019b]. gVisor is also working on
the further implementation of Linux system call API, so that more applications can be sandboxed with gVisor in the future.

5. CONCLUSION AND FUTURE WORK

This paper presented and compared approaches to isolate container workload. Different use cases and scenarios were presented in which the secure handling of containerized applications is of high priority. Furthermore, state of the art mechanisms for the isolation of containers were investigated and the question why multi-layer security is relevant to provide a secure sandbox environment was answered. Based on this, gVisor, Kata Containers and Firecracker were presented and explained whether the concepts can implement multi-layer security. Finally, the concepts were compared with each other and an outlook on further development was given. In future we seek to develop a comparison criteria catalogue that allows the presented, as well as emerging approaches to be classified and differentiated on the basis of an evaluation framework.

In summary, securing container workload is becoming increasingly important. Not only for providers of big public cloud environments, but for every cloud environment with multi-client capability and the execution of untrusted code on host machines. The development of the approaches presented remains exciting and it will be shown which of the approaches will prevail for general productive use.

REFERENCES


The preservation of a proper level of software systems quality is one in the cornerstones of making software evolution easier and sustainable in the long run. A good design allows complex systems to evolve with little effort and in an economically efficient way. When design deviations are detected, refactoring techniques are applied to eliminate or at least reduce the identified flaws. A number of studies show that not all refactoring techniques contribute to improving the quality of different software systems equally. Therefore, effective approaches to measuring the impact of refactoring on software quality are required. In this study, we examine approaches to estimate the effect of applied refactoring techniques on the maintainability of Java based software systems. Since refactoring primarily affects the system’s internal structure, maintainability was put in the focus of the study. We conducted a brief literature review, limiting our study on quantitative metrics. The results show that researchers use different approaches for evaluating the impact of refactoring on the observed Java based software systems. In some studies, researchers measured the effect of refactoring on the internal structure attributes measured by software metrics, e.g. C&K metric suite but the scope of our research was limited to the effects of refactoring on maintainability. In other studies, the effects of refactoring are estimated through external quality attributes, e.g. maintainability, readability, and understandability. Additionally, some researchers observed the impact of refactoring indirectly, e.g. through the defect proneness of classes of observed systems.

1. INTRODUCTION

Software quality is one of the most important issues in software engineering, drawing attention from both practitioners and researchers [Alkharabsheh et al. 2018]. Insufficient quality leads to an unacceptable product. Likewise, as the level of quality increases, the product becomes more useful, and the highest levels of quality can give the product a competitive advantage. Although, higher product quality decreases software evolution and maintenance costs, beyond a certain point the level of quality becomes excessive [Barney et al. 2012]. Thus, increasing the quality level of a software system even further does not bring any additional competitive advantage. This is why, software quality should maintain a balance between development velocity, efficiency and resources required to build a useful software products that satisfies the expectations of stakeholders. As software quality is a complex and multifaceted concept, user satisfaction is not its only aspect [Barney et al. 2012]. Software quality also
includes non-functional attributes, such as reliability and maintainability [Gorla and Lin 2010]. Improving maintainability is one of the cornerstones of making software evolution easier [Alkharabsheh et al. 2018].

The design of a software system is one of the most influential factors in its quality, and a good design allows the system to evolve with little effort and less money [Stroggylos and Spinellis 2007]. Code smells have become an established concept for patterns or aspects of software design that may cause problems to further development and maintenance of the system [Yamashita and Leon 2013]. The term code smells was first coined by [Riel 1996] and [Brown et al. 1998] to refer to any symptom for poorly designed parts of code that can cause serious problems while maintaining the software [Lafi et al. 2019]. [Fowler 1999] provided subsequently a set of informal descriptions for twenty-two code smells. Each code smell in the catalog is associated with a corresponding refactoring technique that can be applied to remedy it. The main motivation for using code smells for system maintainability assessment is that they constitute software features that are potentially easier to interpret than traditional object-oriented software measures [Yamashita and Counsell 2013]. Refactoring is a valuable tool that can be used to improve the design of software and consequently reduce the negative effects of software quality degradation. Our definition of refactoring comprises only those changes in the software system’s program code that change its structure but do not add any additional functionalities. Although there is a general belief among software developers that refactoring always leads to improved software quality, research shows all refactoring techniques do not necessarily improve software quality nor do they improve all aspects of software quality equally. Often, the improvement of one aspect of software quality leads to deterioration in another aspect of software quality. Since we do not have a clear and quantitative instrument for measuring software quality it is often hard to evaluate the benefits of such quality improvement activities.

Therefore, the general research field looking at the impact of refactoring due to code smells on software quality is not yet entirely clear. The goal of this paper is to study which approaches to the quantitative measurement of changes in software quality have been used in recent studies by researchers who are studying the effect of refactoring techniques on software quality attributes. Since refactoring primarily affects the maintainability, the study was focused on maintainability and its related external attributes, such as reusability, analyzability, and modifiability. To answer our research goal, we performed a brief literature review. We only considered studies published within the last five years.

The main objective of this paper is to review measurement approaches used to evaluate improvements in software design achieved by refactoring techniques. Hence, we conducted a brief literature review of the last five years and extracted approaches for evaluating the change in maintainability of two observed software versions.

This paper is structured as follows. The introduction establishes the scope and purpose of the paper and gives the necessary background information relevant to the research. Following the introduction, a second section explains the procedure of the literature review. In sections 3 and 4, refactoring and maintainability are defined using definitions provided by the authors of the analyzed studies. Section 5 outlines the key approaches in measuring the effects of refactoring on maintainability. Finally, we summarize our research and give concluding remarks in the last section.

2. METHOD

The main research goal of this paper is to explore the measures used to quantify the refactoring impact. In addition, we looked at how refactoring is defined in the literature and searched for the reason that guided the refactoring activities. Since we focused our study on maintainability, our aim was to detect different measures related to maintainability and to summarize all the tools used for measuring these measures. To answer these questions and gather available research contributions, a preliminary review
was done using five digital libraries and research search engines: IEEE Xplore, ScienceDirect, ACM Digital Library, arXiv, and Google Scholar. The selection of the relevant papers is based on the title and the abstract relevance. We only considered papers published between 2015 and 2019. The search was done using the following search query: refactoring AND maintainability AND software. The results obtained from digital libraries are shown in Table I. In our research, we only included studies available in the selected digital libraries and written in the English language. The search was conducted for both journal and conference papers and all of the studies used in our paper were published in the software engineering domain. In total, more than 158 (and many more from Google Scholar, which were not relevant) results were obtained from five research digital libraries. Based on the title and the abstract 39 papers were read and the final set of 27 papers was formed which served as the basis for this research. There is a rising trend in the number of published relevant papers, whereas the conferences still constitute the most popular medium for publishing papers on the topic.

3. REFACTORING

The definition of refactoring as an activity of software quality assurance is unanimous among researchers, mainly taken from [Fowler et al. 1999]. They all agree that refactoring is basically the restructuring of source code used to improve existing code [Fontana et al. 2015], i.e. the internal structure of software systems [Kádár et al. 2016; Rathee and Chhabra 2017], thereby improving its understandability and readability [Tawani and Chug 2016] while preserving their external behavior [Fontana et al. 2015; Malhotra et al. 2015; Ouni et al. 2016; Vidal et al. 2018; Szőke et al. 2017; Kannangara and Wijayanayake 2013; Hégedűs et al. 2018; Gatrell and Counsell 2015; Bashir et al. 2017], or more generally, to reduce technical debt [Kouros et al. 2019]. Another view on refactoring, from the developers’ perspective, was given by [Szőke et al. 2017]. They rely on findings by [Kim et al. 2012], supported by their own research [Szőke et al. 2014], that developers’ use refactoring primarily to fix coding issues and not for the refactoring of code smells or antipatterns. A different definition was given by [Kaur and Singh 2017] where the authors described refactoring as an approach that decreases the complexity of software by fixing errors or appending new features. For [Mens and Tourwé 2004] the aim of refactoring, adopted by [Mehta et al. 2018], is to redistribute classes, variables, and methods across a class hierarchy in order to facilitate future adaptations and extensions. Refactoring activities may include operations like [Ouni et al. 2017; Szőke et al. 2017; Steidl and Deissenboeck 2015]: (1) class - level: move, inline, rename, extract class, subclass, superclass or interface, (2) method - level: move, pull up, push down, extract, rename, parameter change and (3) field - level: move, pull up, push down.

On the other hand, a study done by [Shatnawi and Li 2011] identified a different subset of ten refactoring techniques with the highest impact: Introduce Local Extension, Duplicate Observed Data, Replace Type Code with Subclasses, Replace Type Code with State/Strategy, Replace Conditional with Polymorphism, Introduce Null Object, Extract Subclass, Extract Interface, Form Template Method, and Push Down Method.
Code Smells are thought to be the primary technique for identifying refactoring opportunities [Fontana et al. 2015; Kádár et al. 2016; Rathee and Chhabra 2017] and refactoring depends solely on our ability to identify them [Rathee and Chhabra 2017], which may be done by tools based on predefined metrics [Steidl and Deissenboeck 2015]. Code smells are defined as symptoms of problems at the code or design level [Fontana et al. 2015], specific types of design flaws [Kouros et al. 2019] or certain structures which violate the design principles [Malhotra et al. 2015] that originated from poor design choices applied by programmers during the development of a software project [Palomba et al. 2018]. Although code smells do not always represent direct problems to a software system (faults or defects) [Fontana et al. 2015], they are known to degrade quality, and impact the legibility, maintainability, and evolution of the system [Vidal et al. 2018], thus motivating developers to remove them through refactoring.

The identification of refactoring may be done by self-reported refactorings by programmers [Nayebi et al. 2018] or by looking into manual refactoring, but the support of a static source code analyzer tool like SourceMeter is found to be helpful for developers [Szöke et al. 2017]. However, the dominant approach is to use different tools to detect the refactoring. [Hegedüs et al. 2018] searched for refactorings done in seven open-source Java systems with RefFinder [Kim et al. 2010], a tool for refactoring extraction. The same tool was also used by [Kaur and Singh 2017] to extract refactoring tasks from the source code of four versions of the open-source Junit project. On the other hand [Gatrell and Counsell 2015] used the automated tool Bespoke to detect the occurrence of 15 types of refactoring in C# programming language.

4. MEASUREMENT OF MAINTAINABILITY

Within the performed review, we focused on software maintainability since it represents an important aspect within source code refactoring. Maintainability can be understood differently. For example, maintainability is one of the characteristics defined in the ISO 25010 [ISO/IEC 25010 2011] that presents a software product quality model, "the degree of effectiveness and efficiency with which a product or system can be modified to improve it, correct it or adapt it to changes in the environment, and in the requirements". This is not the only definition used in the literature. The definition of maintainability can also be found within ColumbusQM quality model [Bakota et al. 2011] and others individual definitions are frequently found. Additionally, in the software product quality model, maintainability is composed of more subcharacteristics defined and measured in a prescribed way. To unify the understanding of maintainability, the definition of the investigated attribute and possible subattributes within each work is crucial.

[Tarwani and Chug 2016] define software maintainability as the ease, with which software can be modified, corrected or updated. To measure software maintainability, the authors relied on the relation between metrics and software maintainability by using the definition proposed by [Dubey and Rana 2011], which states that the relation between metrics and software maintainability is always inverse. [Malhotra and Chug 2016] adopt the term maintainability by [Aggarwal et al. 2006], who define it as the ease with which software can be modified. It is measured in terms of Line of Code (LOC) added, deleted and modified in the maintenance phase of SDLC. [Bashir et al. 2017] define software maintainability as the level of ease in extension, fixing bugs and performing certain maintenance-related activities. [Szöke et al. 2017] and [Hegedüs et al. 2018] adopt the definition of maintainability provided in the ColumbusQM probabilistic quality model [Bakota et al. 2011], which is based on ISO/IEC 25010 quality characteristics and reduces software quality to one single value.

Still, the majority of researchers [Steidl and Deissenboeck 2015; Ouni et al. 2017; Nayebi et al. 2018; Kannangara and Wijayanayake 2013; Kádár et al. 2016; Ouni et al. 2016; Vidal et al. 2018], did not provide the definition of the maintainability or targeted subcharacteristic. A special case was the paper by [Gatrell and Counsell 2015], since the main focus of their paper was not in measuring...
the maintainability but following the change and fault proneness of classes. In the paper by [Kaur and Singh 2017] the explicit definition of software maintainability is not provided, but they refer to software metrics as a measure of software maintainability of observed software projects.

Only a small number of papers look into the subcharacteristics of maintainability. For example, [Kannangara and Wijayanayake 2013] measured the analysability, changeability, resource utilization and time behavior for each participant in their experiment doing the refactoring. They also measured the maintainability index, cyclomatic complexity, depth of inheritance, class coupling, and lines of code for refactored code and code without refactoring. [Nayebi et al. 2018] used two metrics to measure software quality, especially the maintainability. First, the decoupling level by Mo et al. [Mo et al. 2016], which says how well a software system is decoupled into independent modules, using Baldwin and Clark’s design rule theory as the underlying theoretical foundation: the more active, independent, and small modules there are, the more option values can be produced. They also used the propagation cost proposed by [MacCormack et al. 2006] to measure how tightly coupled a system is, based on the dependencies among files. They also identified the following design flaws and architecture smells in accordance with [Mo et al. 2015]: package cycles, improper inheritance, modularity violations, crossings, and unstable interfaces. They combined this with the DRSpace tool [Xiao et al. 2014] which identifies most error-prone files.

5. EVALUATING THE CHANGE IN MAINTAINABILITY

Researchers agree that once refactoring is applied, the developer should assess its effects in terms of software complexity, understandability, and maintainability or in terms of productivity, cost, and effort for the undertaken process [Ouni et al. 2016; Vidal et al. 2018]. This could be done, for example, by looking at different software metrics to measure the attributes of coupling, complexity, cohesion, or other aspects of the system, before and after refactoring [Vidal et al. 2018]. Some of the established software metrics used later in the paper with abbreviations are summed up in Table II.

5.1 Assessment of the internal design

In some studies, the assessment of the change in maintainability relies on the software’s internal design. For instance, to assess a change in maintainability of software products, researchers [Chug and Tarwani 2017] measured nine software metrics, namely CBO, LCOM, RFC, WMC, NOC, DIT, Ocavg, AHF, and MHF. According to the authors’ understanding of maintainability adopted by [Dubey and Rana 2011], maintainability is in inverse relation to the values of the measured software metrics, i.e. an increase in value of the metrics results in a decrease in the maintainability value. An identical set of metrics was also used by researchers [Tarwani and Chug 2016], who acquired the metric measurements using the IntelliJ IDEA metrics plug-in. Furthermore, a similar approach was taken by the researchers [Malhotra and Chug 2016]. The researchers refer to the work of [Singh and Malhotra 2012] which states that the metrics of the C&K suite are negatively correlated with maintainability. In the study, the researchers empirically evaluated the repercussions of refactoring on maintainability with the help of their measurable effects on the internal quality attributes as well as the external quality attributes. While internal attributes were assessed using the C&K metric suite, the external attributes of observed software projects were assessed through expert opinion. The researchers [Kaur and Singh 2017] estimated the internal structure of software products by measuring its complexity. According to researchers, complexity can be efficiently estimated via an assessment of size (TLOC, and TNOS), coupling (RFC, and NOI), clone (CI), complexity (WMC) and comment (TCLOC) of software products. To measure the required code metrics for the complexity estimation, the authors used the Halstead Metrics tool. Furthermore, [Ouni et al. 2017] measured the number of code smells, number of design patterns and the hierarchical model of evaluating software with 11 low level metrics by
Mitja Gradišnik et al.

[Bansiya and Davis 2002]: design size in classes, number of hierarchies, average number of ancestors, data access metric, direct class coupling, cohesion among methods of class, measure of aggregation, measure of functional abstraction, number of polymorphic methods, class interface size, and number of methods. The authors of the study did not provide the information on which tool was used to measure the metrics.

Table II. List of metrics and their descriptions.

<table>
<thead>
<tr>
<th>Metric name</th>
<th>Metric description</th>
<th>Metric name</th>
<th>Metric description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHF</td>
<td>Attribute Hiding Factor</td>
<td>NC</td>
<td>Number of Classes</td>
</tr>
<tr>
<td>CBO</td>
<td>Coupling Between Objects</td>
<td>NGen</td>
<td>Number of Generalizations</td>
</tr>
<tr>
<td>CI</td>
<td>Clone Index</td>
<td>NGenH</td>
<td>Number of Generalization Hierarchies</td>
</tr>
<tr>
<td>Connectivity</td>
<td>Connectivity</td>
<td>NOC</td>
<td>Number of Children</td>
</tr>
<tr>
<td>DIT</td>
<td>Depth of Inheritance Tree</td>
<td>NOI</td>
<td>Number of Incoming Invocations</td>
</tr>
<tr>
<td>EPM</td>
<td>Entity Placement metric</td>
<td>OCAvg</td>
<td>Cyclomatic Complexity of a Class</td>
</tr>
<tr>
<td>LCOM</td>
<td>Lack of cohesion of methods</td>
<td>RPC</td>
<td>Response For Class</td>
</tr>
<tr>
<td>maxDIT</td>
<td>Maximum DIT</td>
<td>TCLOC</td>
<td>Total Comment Lines of Code</td>
</tr>
<tr>
<td>MHF</td>
<td>Method Hiding Factor</td>
<td>TLLOC</td>
<td>Total Logical Lines of Code</td>
</tr>
<tr>
<td>MI</td>
<td>Maintainability Index</td>
<td>TNOS</td>
<td>Total Number of Statements</td>
</tr>
<tr>
<td>MPC</td>
<td>Message Passing Coupling</td>
<td>WMC</td>
<td>Weighted Methods for Class</td>
</tr>
<tr>
<td>NAggH</td>
<td>Number of Aggregation Hierarchies</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[Kannangara and Wijayanayake 2013] also focused on the internal attributes of studied software products while estimating a change in software maintainability after refactoring the project's code. The authors of the study did an experiment with students of computer science, where they split the students into two groups: the control group with C# without the refactoring and the experimental group of students which got the refactored C# code. They measured the students’ performance on fixing bugs and answering questions about the code. An analysis of the internal metrics showed that there was improvement in the maintainability index, while other metrics (cyclomatic complexity, Dept of Inheritance, Class Coupling and Lines of Code) stayed the same.

In their study, researchers [Steidl and Deissenboeck 2015] used a simplified model to estimate the change in the software quality of studied software by measuring the change in code size. The researchers did use a change in lines of code of observed method as a metric (measured by their own tools), besides their own growth quotient, which showed how methods got bigger. According to the quality model used in the research, longer methods mean less maintainable program code.

5.2 Assessment of external software attributes

In contrast to the approaches that estimate a change in software maintainability through a change of values of internal attributes, in some studies the estimation of external attributes of software products is applied. [Kouros et al. 2019] argued that metrics are not reliable indicators of software quality when comparing different products or even different versions of the same system and that quality can be objectively assessed only by measures that evaluate a design against the optimum design that could be achieved for a particular context. Hence, they used the Entity Placement metric from [Tsantalis and Chatzigeorgiou 2009] which encompasses interclass coupling in the numerator and intra-class cohesion in the denominator, making it a suitable fitness function for their search-based approach to propose activities like refactoring to achieve an optimal architecture and thus reduce technical debt. In the literature, it is common to assess the maintainability of a software system by calculating its maintainability index (MI). One of the works that focuses on the maintainability index, is a study performed by [Mehta et al. 2018]. The authors of the study proposed an approach to improving software quality by removing relevant code smells from the source code of observed software systems. The effectiveness
of the proposed approach is demonstrated by measuring the Maintainability Index and Relative logical complexity, first measured by the JHawk tool and subsequently measured by the Eclipse Metrics plug-in. To reduce maintainability measures to a single assessment value the authors introduced the Maintainability Complexity Index (MCI), calculated according to the formula $MCI = MI \times RLC$. According to the authors, the combination of MI and RLC does better at estimating the maintainability of a software system than the maintainability index itself.

[Bashir et al. 2017] adopted the MOMOOD quality model, proposed by [W. A. Rizvi and Khan 2010]. The model defines maintainability through the following formula: $Maintainability = -0.126 + 0.645 \times Understandability + 0.502 \times Modifiability$, where understandability is calculated via the formula:

$Understandability = 1.166 + 0.256 \times NC - 0.394 \times NGenH$, and modifiability is calculated via the formula:

$Modifiability = 0.629 + 0.471 \times NC - 0.173 \times NGen - 0.616 \times NagH - 0.696 \times NGenH + 0.396 \times MaxDIT$.

The authors do not state how the metrics required for maintainability assessment were measured. Furthermore, [Szőke et al. 2017] measured the effect of refactoring on the software projects by the Columbus QM probabilistic software maintainability model [Bakota et al. 2011] which is based on the quality characteristics defined by the ISO/IEC 25010 standard. The maintainability of the software project was measured by SourceMeter, a tool developed by the authors of the study. The same tool, QualityGate SourceAudit, was used by [Hegedűs et al. 2018], within which the Relative Maintainability Index (RMI) is measured. RMI expresses the maintainability of a code element and is calculated using dynamic thresholds from a benchmark database instead of fixed formulas [Hegedűs et al. 2018].

In a similar study, [Szőke et al. 2017] identified refactoring commits based on the tickets and analyzed the maintainability of the revision before and after the commit. The measurement was done with QualityGate SourceAudit $^1$. The effect of refactoring is measured as: $MaintainabilityChange = Maintainability(t(i)) - Maintainability(t(i) - 1)$. [Hegedűs et al. 2018] did not deal with maintainability change itself, but researched the differences in relative maintainability index between refactored and non-refactored elements. [Kádár et al. 2016] used a metric named the Relative Maintainability Indices of source code elements, calculated by the QualityGate, an implementation of the ColumbusQM quality model. Like the well-known maintainability index, the Relative Maintainability Indices reflects the maintainability of a software module, but is calculated using dynamic thresholds from a benchmark database and not via a fixed formula. Thus, it expresses the relative maintainability of a software module compared to the maintainability of other elements in the benchmark.

Similarly, [Han and Cha 2018] propose a two-phase assessment approach for refactoring identification based on the calculation of the delta value in maintainability. The authors took into consideration several aspects that can affect maintainability. To assess maintainability as accurately as possible the metric values of the Entity Placement Metric (EPM), Connectivity and message Passing Coupling (MPC) were calculated. There are also novel measures used for this purpose. For example, [Malhotra et al. 2015] defined the measure Quality Depreciation Index Rule (QDIR) that is calculated by considering both bad smells and the C&K Suite of Metric, while [Rathee and Chhabra 2017] focused on improving the cohesion of different classes of object-oriented software using a newly proposed similarity metric based on frequent usage patterns. Ouni et al. [Ouni et al. 2017] measured the gain in different QMOOD quality factors (reusability, flexibility, understandability, effectiveness, functionality and extendibility) ratio changed as defined by [Bansiya and Davis 2002]. Also, they used Ph.D. students to evaluate the refactoring and compare the results of their approach to professional recommendations.

While most researchers evaluated maintainability via metric measurements, the researchers [Malhotra and Chug 2016] estimated understandability, level of abstraction, modifiability, extensibility, and reusability through expert opinions.

$^1$https://www.quality-gate.com/
5.3 Indirect assessment of external quality attributes

Last but not least, in some studies researchers focused on the reliability of a software system in order to assess a change in the software quality of an observed software system after refactoring had been applied. A change in reliability can be detected by the increased frequency of defects that are rooted in poor software design. However, the measure of quality is often expressed in terms of a number of defects, before and after applying refactoring [Fontana et al. 2015; Ouni et al. 2016]. For example, maintainability was not measured by [Gatrell and Counsell 2015], however, the researchers looked into the change and fault proneness of classes, which is a commonly used predictor of a system’s reliability.

6. CONCLUSION

To maintain a proper level of software quality in the long run, the detection of deviations in a system’s design should be responded by improvement actions, usually performed by refactoring the affected part of the software system. Despite the liveliness of refactoring research field in the past, there is no complete consensus about the definition of refactoring. The majority of researchers in the analyzed studies understand refactoring as a restructuring of the code used to improve existing code, i.e. the internal structure of software systems, its understandability and readability, while preserving their internal behavior. Hence, refactoring improves the quality of the internal structure of the software, without adding any extra functionalities. In general, refactoring can be understood as a set of corrective activities that contribute to their longevity by improving the internal structure of software systems.

In existing studies, maintainability represents a software quality aspect of the software that is mostly affected by source code refactoring. In general, maintainability can be best described as the ease, with which software can be modified, corrected and updated. Researchers do not agree completely on when the maintenance phase start. For most of them, maintenance is a phase that starts once the software is delivered to customers. Regardless of the interpretation of maintenance activity, maintainability represents an important aspect of the quality of the software overall. Consequently, it is also an important aspect of software quality models, i.e. ISO 25010 standard and ColumbusQM quality model. Despite the fact that maintainability is defined as a compound quality attribute in software quality models, only a small number of papers look into the sub-characteristics of maintainability, e.g. understandability, readability, and changeability.

The results of the study show that researchers use different approaches to evaluate the effects of refactoring on observed software systems. In some studies, researchers are focused on measuring the effect of refactoring on attributes of a system’s internal structure. According to the studies, a change in the quality of the system’s internal structure can be detected by a set of software metrics or metric suites. Often, the effect of refactoring on maintainability can also be estimated through the estimation of some external quality attributes, e.g. maintainability, readability, and understandability. Last but not least, some researchers observe the impact of refactoring indirectly, e.g. through the defect proneness of the classes in the observed software systems.

A study of the effects of the defected anomalies in software design on software quality attributes has remained a lively field of research over the last decade. One of the main objectives of the research field is to objectively assess how improvements in software design achieved by refactoring techniques contribute to higher software quality. The goal of this study was to review the literature of the last five years and extract approaches for evaluating the changes in maintainability of two observed software versions. In the literature, maintainability was most commonly associated with software’s internal design. Therefore, this quality attribute was the focus of our study.

REFERENCES


K. Stroggyllos and D. Spinellis. 2007. Refactoring—Does it Improve Software Quality?. In Fifth International Workshop on Software Quality (WoSQ’07: ICSE Workshops 2007).


Survey on Static Analysis Tools of Python Programs

HRISTINA GULABOVSKA and ZOLTÁN PORKOLÁB, Eötvös Loránd University

Static program analysis is a popular software technique performed by automated tools for verifying large scale software systems. It works without executing the program, just analyzing the source code and applying various heuristics to find possible errors, code smells and style discrepancies. Programming languages with strong static type system, like C, C++, Java are the usual targets of static analysis, as their type system provides additional information for the analyzer. However, there is a growing demand for applying static analysis for dynamically typed languages, like Python. In this paper we overview the current methods and tools available for static analysis on Python code base and describe some new research directions.

1. INTRODUCTION

Maintenance costs take a larger part of the estimated price of the software systems. Most of these expenses are spent for bug fixing. The earlier a bug is detected, the lower the cost of the fix [Boehm and Basili 2001], therefore various efforts are applied to speed up the development–bug detection–bug fixing cycle. The classical test based approach do not fit well to this pattern. Writing meaningful tests requires large code coverage and takes substantial development workload and certain time. Another approach, using tools, like Valgrind [Nethercote and Seward 2007], or Google Address sanitizer [Serebryany et al. 2012] which work run-time, evaluates the correctness only those parts of the system which have been actually executed. Such dynamic analysis methods therefore require carefully selected input data, and they easily can miss certain corner cases.

Contrary to testing and dynamic analysis methods static analysis works at compile time, based only on the source code of the system, and does not require any input data. It is a popular method for finding bugs and code smells [Bessey et al. 2010]. Compiler warnings are almost always based on various static analysis methods. As we will see in Section 2 many of the applied techniques are fast enough to be integrated into the Continuous Integration (CI) loop, therefore they have a positive impact on speed up the development–bug detection–bug fixing cycle. This makes static analysis a useful and cheap supplement to testing, built into the continuous integration loop during development.

The type system provides extra information for the compiler as well as the static analysis tools. As a consequence, static analysis is most popular for programming languages with static type system, like C, C++ and Java. A number of commercial [Synopsys 2019; Roguewave 2019] and open source tools [Clang SA 2019; Clang Tidy 2019; GrammaTech 2019; Calcagno and Distefano 2011] exist with considerable large developers’ community to support these languages [Dergachev 2016; Zaks and Rose 2012]. However Python, one of the most rapidly emerging programming languages with dynamic type
system is still not sufficiently supported by static analysis tools. Python is very attractive implementing Machine Learning and Cloud based applications among others, and in July 2019 it is already the third most popular language by the Tiobe index [Tiobe 2019].

In this paper we investigate how static analysis methods could be adapted for Python based software systems. The authors believe that in spite of the difficulties caused by the dynamic type system many of the methods applied for the classical languages could be utilized for Python too.

This paper is organized as follows. We discuss the most important analysis techniques in Section 2. We take a survey on static analysis tools used for the Python programming language in Section 3. In Section 4 we evaluate the tools according some typical use cases. This paper concludes in Section 5.

2. METHODS OF STATIC ANALYSIS

In this section we will overview the most frequently used methods of static analysis for modern programming languages. It is not our goal here to go to very deep technical details, but we intend to compare the advantages and the limitations of each methods. We will cover only methods which are (1) implemented at least for one programming language (2) based only on the source code, i.e. we will not discuss methods, where the user has to supply invariants or semantics of the program.

All static methods apply heuristics, which means that sometimes they may underestimate or overestimate the program behavior. In practice this means static analysis tools sometimes do not report existing issues which situation is called as false negative, and sometimes they report correct code erroneously as a problem, which is called as false positive. Therefore, all reports need to be reviewed by a professional who has to decide whether the report stands.

Here we face an interesting human challenge. Unlike the theoretical program verification methods, we do not strive to minimize the false negatives, i.e. try to find all possible issues, but we aspire to minimize the false positives. If the tool produces a large number of false positives, the necessary review process takes unreasonable large effort by the developers, meanwhile they also lose their trust in the analyser tool.

2.1 Pattern matching

In this method the source code is first converted to a canonical format, i.e. all branches are converted to simple one liner if-else format, loops to a simple while-command format, comparison expressions are reordered to less-than and less-equal format. The developers express the various issues to detect as regular expressions. The tool tries to match this regular expression to every line in the source and reports each matches.

Earlier versions of CppCheck [Marjamäki 2013] used pattern matching to find issues in C and C++ programs. Experiences with the tool report relatively low level of false positives, as well-written regular expressions have easy to predict results. Apart of the low false positive rate, additional advantage of pattern matching is working on non-complete source, even if we cannot recompile the software system. This means that the method can be applied to software under construction, or when we have only partial information on the system.

In the same time this method has several disadvantages too. As regular expressions are context free grammars, we are restricted to find issues based on information in the close locality of the problem. Having no knowledge about declarations we could not use type information. We cannot use name and overload resolution, cannot follow function calls and generally we have weak understanding on the overall program structure. As a summary, we can consider pattern matching based approaches as easy entry-level methods [Moene 2014].
2.2 AST matchers

To provide the necessary context information to the analysis we can use the Abstract Syntax Tree (AST). AST is the usual internal data structure used by the front-end phase of the compilers [Aho et al. 1986]. Basically, the AST is a lossless representation of the program; it encodes the structure of the program, the declarations, the variable usages, function calls, etc. Frequently, the AST is also decorated with type information and contains connections between declarations (types, functions, variables) and their usage.

This representation is suitable for catching errors that the simple pattern matching is unable to detect. These issues include heap allocation using the wrong size, some erroneous implicit conversions, inconsistent design rules, like hurting the rule of three/five in C++ and it proved to be strong enough to detect even some misuses of the STL API.

Such AST based checks are usually relatively fast. Some rules can even be implemented using a single traversal of the AST. That makes it possible to implement such checks as editor plug-ins. External tools, like the Clang Tidy [Clang Tidy 2019] uses AST matching for most of its checks.

While the AST matcher method is more powerful than the simple pattern matching, it has some disadvantages too. To build up a valid AST requires a complete, syntactically correct source file. To resolve external module dependences we need some additional information not represented in the source itself, like include path for C/C++ programs, CLASSPATH for Java or BASE_DIR in Python. That usually means, we have to integrate the static analysis tool into the build system which can be painful. Another shortage of the AST matcher method, that it cannot reason about the possible program states which can be dependant from input values, function parameters.

2.3 Symbolic execution

Executing abstract interpretation [Cousot and Cousot 1977] the tool reasons about the possible values of variables at a certain program point. Symbolic execution [Hampapuram et al. 2005; King 1976] is a path-sensitive abstract interpretation method. During symbolic execution we interpret the source code, but instead of using the exact (unknown) run-time values of the variables we use symbolic values and gradually build up constraints on their possible values. Memory locations and their connections (e.g. a pointer pointing to some variable, structs and their fields) are represented by a sophisticated hierarchical memory model [Xu et al. 2010]. A constraint solver can reason about these values and is used to exclude unviable execution paths. Most of the high end proprietary analysis tools, like CodeSonar [GrammaTech 2019], Klocwork [Roguewave 2019], and Coverity [Synopsys 2019] as well as open source products like the Clang Static Analyzer [Clang SA 2019], and Infer [Calcagno and Distefano 2011] use this method.

Symbolic execution is the most powerful method for static analysis as it makes profit from the program structure, the type system, the data flow information and is able to follow function calls. However, there is a price for this. To represent the internal state of the analysis, the analyzer uses a data structure called the exploded graph [Reps et al. 1995]. Each vertex of this graph is a (symbolic state, program point) pair. The edges of the graph are transitions between vertices. This graph is exponential in the number of control branches (conditions) of the program. This could be critical, especially for loops, which are represented as unrolled set of conditions and statements. This factor makes symbolic execution also the most resource expensive (in memory and execution time) method.
3. THE CURRENT TOOL LANDSCAPE OF PYTHON STATIC ANALYSIS

Here we give a brief overview of the current most commonly used static analysis tools in Python.

3.1 Pylint

Pylint [Logilab 2003] is so far the most popular Python static analysis tool which is free and capable of not only catching logical errors, but also warns regarding specific coding standards, offers details about code complexity and suggests for simple refactoring. It is working by building an abstract syntax tree with the help of the Astroid [Logilab 2019]. Also, Pylint allows developers to write own plugins for specific checks which enables them easily to extend the tool. It is used by several popular IDEs and frameworks mainly for in-time static analysis of the code, some of which: PyCharm, VSCode, Django, Eclipse with PyDev etc. Pylint popularity and reliability is shown through its use at Google as one of the leader tech companies. For the static analysis of its Python codebase, Google is mostly relying on Pylint, while having a common decision how to avoid some of its imperfections. [Google 2018].

3.2 Pyflakes

Pyflakes [PyCQA 2014] focuses only on the logical errors and does not complain about code style or standards. It works by parsing the source file, instead of importing and it is a bit faster than Pylint, since it examines the syntax tree of each file individually. As a consequence, it is more limited in the types of issues that it can check. Its main focus is not to emit false positives which has its advantages, but as a disadvantage of this strict focus, some typical errors are not reported at all.

3.3 flake8

Flake8 [Cordasco 2016] is a wrapper around Pyflakes, pycodestyle and mccabe [Cordasco 2017]. It is mostly used by those who like Pyflakes but also want to get stylistic checks. The code style is checked against PEP8.

3.4 Frosted

Frosted [Crosley 2014] is a fork of pyflakes which focuses on more open contribution development from the outside public.

3.5 Pycodestyle

Pycodestyle [Rocholl 2006] strictly checks only the code style against the style conventions in PEP8. It is usually used in combination with the other tools which are looking for logical errors and warnings.

3.6 Mypy

Even though it is still considered experimental and optional static checker, Mypy [Lehtosalo 2016] proves itself to be highly effective, especially in type checking. It focuses on combining the benefits of dynamic typing and static typing. It has a powerful type system that in some cases, as discussed later in this paper, it catches type errors that are not caught by any of the common Python static analysis tools.

3.7 PySym

PySym [Dahlgren 2016] is an experimental Python package capable of a symbolic manipulation on a minimal scope. None of the previously mentioned tools are in contact with this package, nor with symbolic execution in general as a method of static analysis. Therefore, it is decided to be considered in this paper in the further investigation of symbolic execution in Python static analysis tools.
3.8 PyExZ3

PyExZ3 [Ball and Daniel 2015] is a tool which as part of the research on “Deconstructing Dynamic Symbolic Execution” [Ball and Daniel 2015], adds to the exploration of the dynamic symbolic execution (DSE) through a minimalist implementation of DSE for Python. With symbolic execution not being a common topic in the present static analysis field of Python, the authors of this paper believe they can rely on the PyExZ3 tool for further experimentation on dynamic symbolic execution in Python with the goal to improve the Python static analysis tool set.

On Figure 1 we summarize the connections between the Python static analysis tools.

4. EVALUATION

Here we evaluate the tools by (1) which methods they use, (2) how mature they are, (3) what kind of problems they detect.

4.1 Typical Python bugs and error tested

It is true that the compiler relies on static analysis, similar to these tools in order to generate its warnings during compilation time. However, its primary task is to translate a compilable code from high-level programming language to a lower level language, and the attention to its essential characteristic – the compilation time, makes it less capable of doing static analysis than the “third-party” static analysis tools. Python compiler misses all of the following discussed bugs and errors. Here is a short overview of the dummy code snippets used as a test examples of typical logical errors and bugs, used in this study to analyze and test the power of the static analysis tools in Python.

4.1.1 Referencing undefined variables. Very often in Python it happens that due to a typing mistake, an undefined variable is referenced. However, due to the dynamic nature of the language, such bugs do not get discovered up until runtime and when they are referenced. Example code for testing this bug with the static analysis tools of Python:
It can be noticed that at the line 6 the variable “mesage” is mistyped, and does not refer to the string “Hello there!” even though it looks similar. It is possible that error like this might not appear even during runtime. For example if this Python code is run without the “–greetMe” as an argument, then the code will just print out “This code is fine, no problems.” without errors or warnings.

4.1.2 Too many positional arguments. The following dummy code beside the “too many positional arguments” error in the call of the method getAge, has 3 more issues. The getAge method itself does not have a “self” argument. Further, the platform variable is undefined, this is similar to the previous example. And the last issue the sys package is imported but never used. Unused imports might slow down the program.

4.1.3 Passing parameter of a different type then intended. Since Python is a dynamic language variables do not have types in Python, but rather values do, and the type of the values as well as the purpose for what an object can be used is decided during runtime. Python annotations were introduced to add hints for the type of a variable, however the type annotations issues are not caught by the compiler, so it is of a big importance for the other static analysis tools to discover that. The following code snippet is used to test the tools against and describes the importance of type annotations in Python:

Considering the sum function here, Python compiler does not report any issue, but during runtime the program throws an error which explains function sum can not be called with string since int and str types can not be sum up together with the + operator in Python.
4.1.4 Non-existing class member. The following example is used to test the tools against a code which refers to an unexisting class member. As it can be seen, the program tries to print the PERSON1's height, but height does not exists as an attribute in the class Person.

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
PERSON1 = Person("Hristina", 23)
print(PERSON1.age)
print(PERSON1.height)
```

4.1.5 Nested function call. A call to a nested function is a big logical error, but still the Python compiler does not generate any warning about it. The following dummy code is used to test the static analysis tools against call of a nested function outside its scope:

```python
def outer():
x=3
def inner():
    print(x)
inner()
outer()
inner()
```

4.1.6 Closure bugs. In Python, Closure is a concept that allows function object to remember values in enclosing scopes even if they are not present in memory.

```python
#Python Closure
def outer():
x=3
def inner():
y=3
result = x + y
return result
return inner

a = outer()
print(a()) #this is the same as a call to the inner()
print(a.__name__) # prints the name of the actual function # which is called – inner.
```

The following is an example code used to test the Python static analysis tools against a closure bugs in Python which is not reported by the Python compiler and the program runs just fine, with an unexpected output:

```python
def greet(greet_word, name):
    print(greet_word, name)
greeters = list()
names = ["Kiki", "Riki", "Joe"]
for name in names:
greeters.append(lambda x: greet(x, name))
for greeter in greeters:
greeter("Hi ")
```
We may expect that this code would printout:

```
1 Hi Kiki
2 Hi Kiki
3 Hi Joe
```

But instead it is printing:

```
1 Hi Joe
2 Hi Joe
3 Hi Joe
```

### 4.2 Test results discussion

<table>
<thead>
<tr>
<th>#</th>
<th>Typical error</th>
<th>PyLint</th>
<th>Pyflakes</th>
<th>Flake8</th>
<th>Mypy</th>
<th>Frosted</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Referencing undefined variables</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2.</td>
<td>Too many positional arguments</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3.</td>
<td>Passing a parameter of a different type then intended</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>4.</td>
<td>Nonexistent class members</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>X</td>
<td>-</td>
</tr>
<tr>
<td>5.</td>
<td>Nested function call</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>6.</td>
<td>Closure bugs</td>
<td>X</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

On Table I we summarize our test results. The accent of the discussed code tests is mostly put on the logical errors that are slipping into Python developers everyday work, thus at this point, code style focused static analysis tools (such as Pycodestyle) are not considered in the result table. This result discussion is based on catching-coverage of the logical errors, confirming if the tool’s behavior is as it is described in their documentation as well as a general overview of spotted pros and cons of each of the tools used against the test code examples. Currently, just a short general overview of the output format of the tools is given, but since this is also an important aspect of the tool’s functionality, especially when they are used against a bigger Python code based, more detailed analysis of this will take part further in the study. To clarify, all the tools are tested while in their default configuration. The results might differ if the tool’s configuration is separately tuned according the code tested.

Upon initial inspection of the table results, Pylint outperformed the other tools when run against the described test examples. It gave the most accurate reports and it was also the most descriptive in the generated output. However, if we look at the output format and style (which is not the focus of the table results at present), it is apparent that in many cases Pylint is “too talkative”, meaning the output can quickly get noisy and not so easily readable as the code base expands. Yet, the convenience of Pylint is again proven as it is the only tool that reports the errors “Too many positional arguments” and “Closure bugs”, which are easily made mistakes and the first one is especially trivial, but left to be caught during runtime, which is much more expensive. As such, they are of a great importance to the static analysis and the tools clearly need a support in their improvement. The “Closure bug” is a bit more complex. Closure in Python is an important concept and it is prone to bugs which in many cases, shown in the dummy test example as well, is hardly caught even during runtime. Such errors can take even days of debugging and manual reviews by the developers. This is notably true for bugs hiding in the Python closure concept. The authors believe that with the implementations of symbolic execution, the Closure is one of the concepts in Python that might see great improvements regarding static analysis by moving it ahead from the current benefits of the AST approach.

There is only one test case where Pylint did not report any warning nor error, and to a positive surprise, Mypy was the only tool that caught it. This was a type annotation related bug where Mypy
proved that even though it is still considered as an experimental tool and far from the commonly preferred Pylint, it justified its focus to perform well on type annotation errors. This makes Mypy an essential static analysis tool to be considered in a Python code that uses type annotations gain on the benefits of the predefined variable types, which is not in the core of Python as a dynamic programming language. The importance of type annotation static analysis is shown in the “Passing parameter of a different type then intended” example and its “sum” function. Seeing that analyzing type annotations is one important point of the static analysis tools that needs to be improved, and at the same time noticing that the tools lack the symbolic execution as a method technique behind the scenes which seems to be a great factor of this improvement, encourages author’s further directions of the research to go towards implementing and testing symbolic execution in Python static analysis. On the brighter side of the results table, there are two tests that were covered with a warning report by all of the static analysis tools under test. These are “Referencing undefined variables” and “Nested function call”, both of which are quite trivial errors. The example “Referencing undefined variables” in a strictly typed language such as C, C++ and Java, this will be immediately reported by the compiler, but since Python is a dynamically typed language, its compiler does not catch it. Thus, this is one of the crucial moments for the Python static analyzers to shine up above the Python compiler. Anyhow, the strong believe that symbolic execution can improve the performance even in the most crucial and trivial examples, stays alive and will be considered in the future of this research.

Since the focus now is in the logical errors caught by the static analysis tools, it is important to notice that Pyflakes and flake8 are expected to, and they do give, the same results in the table. This is mainly because flake8 just combines pyflakes, pycodestyle and mccabe, so they have the same approach in catching the logical errors. However, noticing that Pyflakes did not catch many of the logical errors that were caught by the other tools, knowing that as a tool it focuses mostly on the logical errors avoiding the style reports, it raises a lot of questions. Moreover, leaves the believes that its very strict avoidance of false negative error, adds more cons than pros to its convenience as a static analysis tool, since because of this, many errors were left not reported at all. Frosted as a tool during this test examples did not show that catches any different errors than what Pyflakes and flake8 caught, which is to a certain degree expectable, since it is a fork of pyflakes, with the main difference being that its open to a public development.

To sum up this brief introductory discussion and analysis of the existing Python static analysis tools, it can be said that there is no tool presently that can stand alone as the best of all. Best results at the moment would be achieved when using several of the existing tools at the same time, thanks to their manually configurable and customizable capabilities. Exploring Python symbolic execution further in the research, promises that would bring great improvements in the currently existing Python static analysis world.

5. CONCLUSION

Static analysis is a powerful technique to validate large software systems in the early phase of the Continuous Integration (CI) loop. There are sophisticated methods for analyzing mainstream languages with static type system, like C, C++ and Java. The current support for the Python programming language is way behind. Tools are in experimental phase and lacking large development communities. The main reason is the dynamic nature of the language which makes harder to apply the usual analyzer techniques. However, this can be improved as tools can step forward from the current AST based methods. The most promising direction is symbolic execution, which can handle the dynamic language features in a more straightforward way. The authors suggest future researches to implement further symbolic execution techniques and test them in industrial environment.
REFERENCES


Using Multiplayer Games to Create Secure Communication

JAAK HENNO, OY Hypermeedia
HANNU JAAKKOLA, Tampere University
JUKKA MÄKELÄ, University of Lapland

Massively multiplayer online games (MMOGs) and social networks are very popular communication and entertainment formats, where millions of players from all around the world interact in a shared environment and exchange and trade different types of digital media: texts, videos, sounds and music. The communication systems implemented in these virtual environments are in increasing number encrypted to prevent fraud and user impersonation. Ubiquity of virtual forums with massive participation and participants communication systems has raised several questions about their security – gaming servers are suitable as potential exploitation tools for terrorist groups use to conduct on-line operations. The prevailing on Internet top-down security organization where trustworthiness of an object (computer, program) is established by a strong hierarchical system of security certificates does not work, since trusted high level certificates can already be bought online in dark web marketplaces.

For many local communities (players of an online multiplayer game, local social networks etc.) is more advantageous a different, local ‘sand-box’ organization of a communication system where anonymous participants (initially identified only by their generated username without using even e-mails) actively interact using messages with strong encryption. For encryption they need entropy/randomness, but this they can create themselves in their interactions. In competitive interactions (e.g. they play a competitive game) all participants (trying to compete each other) behave differently, try to create thus the sequence of their actions is random and can be used as the secure key for symmetrical encryption for communication among participants.

Here is considered a class of games where expectation of payoff is the same for all moves, thus players cannot get from results any additional information about the game, thus their best strategy is to select their moves randomly (non-learnable games). It is shown, that in a sense all games of this class are similar, can be created with the same procedure and can be reduced to each other using introduced here operation of rectangular modification of the game state matrix. The best strategy for moves in this class of games is uniform randomness, thus (if they are competing and try to beat each other) in the play they create with their moves a random sequence. This sequence can be used for generating a key for symmetric encryption. In a non-local (server-based) multiplayer game players know only their own moves, about moves of other players they get only results of game (not the actual moves made by other players). Thus for generating a key server sends players sequence of all moves from which the player's own moves are removed. This sequence is different for players and contains only partial information, thus an eavesdropper (man-in-the-middle) cannot use it. Players insert into this holey sequence their own moves and get all the same sequence of moves which will be used as the common key for symmetrical encryption of communication: the procedure allows several enhancements for further randomness and/or speed of key generation. The key generation from player's moves removes need for use of public-key systems and all communication (and keys) remain inside the virtual community, whose security thus becomes self-sustainable.

1 INTRODUCTION

1.1 Communication and Communication Security

Our communication is increasingly encrypted – currently already more than 72 % of Internet communication is encrypted and the encryption is growing rapidly, with nearly 20 % increase in a year [Encrypted Traffic 2018].

Leading advisory company Gartner predicts that already in this year (2019) 80% of Internet traffic is encrypted [Cisco 2019]. Encryption is the main/only technology which enables communication privacy and data security, but for encryption is needed entropy/randomness.

Author's addresses: Jaak Henno, Hypermeedia OY, Paldiski matt 157-8, 13518 Tallinn, Estonia, email: jaakhe@online.ee; Hannu Jaakkola, Tampere University, Pori Campus, P.O. Box 300, FI-28101 Pori, Finland, email: hannu.jaakkola@tuni.fi; Jukka Mäkelä, University of Lapland, 96101 Rovaniemi Finland, email: jukka.makela@ulapland.fi.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
1.2 Encryption

The encrypted communication proceeds usually in two steps. In the first step, interested parties use public-key (asymmetric) encryption scheme for creating a common for both parties key for a symmetric encryption. The reason for such a two-step technology is in different properties of non-symmetric and symmetric encryption: the non-symmetric encryption system allows to send the first message using a publicly available long (currently a typical value is 2048 bits) key: the message can be decrypted only with the receiver’s private key (much shorter, e.g. only 256 bits). Using a publicly available key is convenient, but the method is slow, used only for creating a ‘shared secret’ – the key (random string of 128/256 bits) for symmetric encryption, which is later used in the (quick) symmetric encryption (the key is secret, created in the first open-key round) of the main communication.

Using a publicly available resource (the public encryption key) is problematic – to whom does this key actually create an understandable (decipherable) message, who is the ‘real’ owner of this key?

1.3 The ‘top-down security’ organization - certificates

To authenticate keys owners, for creating new keys and managing their use has been created a hierarchical system of security certificates. These certificates are like passports which authenticate the identity of the certificate’s holder and grant permissions to use encrypted communication using a public key infrastructure. The certificates are issued and managed by a strong hierarchical system of security principals using multistep validation processes – every passport/certificate should have only one valid fixed owner.

But all public secrets have shot lifetime (computers could crack any secret) and this ‘top-down’ security model leaks – many top-level certificates can be acquired from dark web marketplaces [Maimon at al 2019], authentication keys have been used to authenticate malware [SecurityWeek 2019] and/or to encrypt malware [Hacker News 2017], [Kaspersky 2018].

1.4 Growing need for entropy

Introduction of new Internet-connected devices - Internet of Things (IoT), virtual/cloud servers, Internet-connected mobile devices (cars, scooters etc.) increase the need for randomness used for encryption. There are already proposals for special services to serve entropy, i.e. random data [NIST 2016]. In order to deliver provided entropy to users is proposed a special new ‘Entropy as a Service’ protocol [EaaSP 2018]. But for delivery this entropy also should be encrypted, thus it is not clear, whether this service will reduce the need for entropy or increase.

The current hierarchical, top-down security scheme introduces many unnecessary checks which slow down encrypted traffic. If I want to send an encrypted message to my co-worker sitting in the next room then with the current hierarchical top-down security system would be evoked many upper-level security authorities (up to the Heavenly God Microsoft – we are using Outlook). Specialists estimate, that the share of network traffic that really should be checked by security measures is still relatively small [Effect of Encryption 2017] and local traffic should be secured locally – local communication participants (the communication’s local context) know best what are the possible dangers and thus also can invent best measures to eliminate them. This is also the main idea of the behavioural security measures which popularity is rapidly growing [Terada 2017], [Leventhal 2018].

1.5 BYOK – Bring Your Own Keys

The growing in popularity trend to move security concerns down, closer to communication participants is also the ‘Bring Your Own Keys’, where the user manages the encryption keys for their data. This is offered (or proposed) by many global data/communication services: Google Compute Engine, Amazon, Adobe Creative Cloud, Microsoft’s Key Vault and the number is growing.
Using Multiplayer Games to Create Secure Communication

One of areas where local communication could be secured with local measures and locally generated encryption keys is communication in (limited) on-line communities – chat/exchange for players of online multiplayer games and (local) on-line social forums.

2 GAME AND GAME COMMUNICATION (CHAT, EXCHANGE/TRADE)

2.1 Economy of Video Games

Video games have become a significant part of the global entertainment economy and their significance is constantly growing.

The first video games were products – you paid and got a game, either stored on disk (CD, DVD) or right to download/play it from some (streaming) server. But with the global marketing trend „tie your customer forever” video games are increasingly creating their own economic values and trading space for these values. A game could be rather cheap, but then appear the „extreme edition”, „additional downloads”, new items could be bought in-game etc. You can enrich your game with DLC (DownLoadable Content) from a virtual marketplace and add to your game various digital content: songs, skins, characters, modes, levels, weapons, cars, expansions, etc. Value of DLC’s already exceed value of games, with DLC video game industry has become a multi-billion dollar-a-year business [WePC 2019]. Such an on-line market needs good security.

Multiplayer Online Games (MOG) introduce a virtual in-game economy where players can find or earn virtual objects – coins, arms, skins for weapons etc. Games supports in-game trading of these virtual items and even have their own virtual currency that can be used in trading within the game (“Ultima Online”, “Second Life”, “League of Legends” etc.), but many players want to trade game items also outside the game. This economy is driven by virtual supply and demand, involves the exchange of real world money and is usually not regulated by game developers. However, many game creators support ‘outside-game’ economy, e.g. player of MMOG ‘Counter-Strike: Global Offensive’ (created by Valve) get sometimes after finishing the game a crate which could contain several skins: some (e.g. gold tier skins) are extremely rare, thus many gamers want to get one, but to open the crate gamer should buy from Valve (for real 2.49 USD) a key. In last year this game had more than 4 million players [SteamCharts 2019], thus many potential buyers and this ‘in-game’ purchases (for real money) strategy is increasingly used. Demand for these virtual objects has created alternative economy for enterprising players who have enough time to play these games to "farm" the game by earning in-game currency and rare items. For farming, level up player characters and other repetitive, no-skill based and time-intensive tasks have appeared many bots, which perform these tasks while player sleeps [Hackerbot 2019 ], [MPGH 2019 ].
These virtual goods are then converted to real money using Real-Money Trading (RMT) sold to players using chat rooms or dedicated forums or on online auction sites, e.g. eBay; profits for this business are growing in thousands of dollars [TrendMicro 2016].

Besides virtual values are for cybercriminals attractive also player’s ‘real-world’ values working e-mail, country/language, mobile phone’s number. Although many on-line communities (Fortnite, Google Play, EA) have introduced double verification of personality, statistics shows that theft of player’s personal information is constantly growing. Thus it is better not to use player’s personal information and make the game world and game communication closed. Game designers should not hope for profits from selling players information, but make their games ‘inner world’ with DSL better.

When participants of online multiuser communities (multiplayer games, social networks) want to establish a direct communication with fellow participants (a chat system), which allows to exchange text messages and game’s virtual values (media files - text, images, video), then this new communication system should be closed and not to burden game servers which are already busy with the game communication. To ensure security of game (game players and game server) the communication system should be ‘sand-boxed’.

![Diagram showing security surfaces of information flows in a multiplayer game with communication (chat system).](image)

The game communication system is also a marketplace for exchange of game’s virtual values, thus all the communication here should be encrypted. In order to close the game communication (chat) system from outside it should generate the encryption keys itself (i.e. this is another example of BYOK). For this it could use the entropy/randomness, created by player’s themselves and not to use any outside sources of entropy/randomness.

3 ENTROPY FROM GAMES

For players of on-line multiplayer games keys for symmetric encryption can be generated as a part of gameplay, thus eliminating the need for higher-level security authorities and by-passing the use of public-key encryption step in key generation.

"Randomness has been part of games since their earliest inception -- and when I say 'earliest inception,' I mean deep into the unwritten Neolithic past" stated game design/author /classic Greg Costikyan [Costikyan 2009], [A. Chalk 2009].

Randomness in games appears from two sources – players decisions (actions/moves) and game setup, e.g. in a shooting game – movement/appearance of targets, precision of calculating hits (collisions of player’s bullets with targets) etc. The properties of the second source (randomness caused by game setup) is nearly impossible to measure exactly, game designers always want to create random outcomes within
parameters that can be influenced to fit the game design. Thus also the randomness appearing in game mechanics is difficult (in a good game - nearly impossible) to evaluate and estimate its dependence on game algorithms (e.g. showing/moving/hiding targets) – this is just a noise injected between a player’s choice and the result.

Therefore in the following is presented a setup, where this second source is eliminated - game is a simple deterministic algorithm and the randomness is created only by player’s actions.

Randomness created by humans is an arguable topic – many game designers consider humans very predictable [Costikyan 2013]. Statement “Multiple competing players create randomness” is not a provable mathematical statement. This statement is similar to Gödel’s incompleteness results – it can’t be proved, but since nobody has presented a counter-example, we believe it. Humans are playing (video) games more and more; if the process of playing does not introduce randomness (surprise, entropy), if this were just a process of repeating game’s moves over and over, playing were not fun and the number of players would not increase. But more than one third of human population is already playing video games [Video Gaming 2019], thus this should be true – players create randomness in the process of gameplay, this creates the fun of playing.

3.1 Non-local simultaneous games

A game is non-local, if the only information what players get about actions of other players are the results of those actions and they do not get information about actions/moves of other players what caused these results. This is the typical situation in server-based on-line video games. The situation is more complicated with e.g. direct Peer-To-Peer (P2P) communication [House 2018] which is here not considered.

A (simultaneous) game is a 5-tuple

\[ G = \langle P, M, A, R, \bar{\rho} \rangle \]

Here

- \( P \) is the set of players, \(|P| = n, n \geq 2\) is the number of players; e.g. for two-player game \( P = \{p^1, p^2\} \);
- \( M = \{0,1,...,m-1\}, m \geq 2\) is the set of player’s moves (legal actions, the same for all players), here they are labelled as m-ary natural numbers; all players move simultaneously, i.e. select an element from the set \( M \); denote by \( m^i \in M \) the move of player \( p^i \in P \) in gameplay at moment (move) \( t, t = 0,1,...,l \), ( \( l \) - the length of gameplay); \( \bar{m}_t = (m^1_t, m^2_t, ..., m^n_t) \in M^n \) is the (global) move vector of all (simultaneous) moves made by all players in move \( t \).
- \( A \) is a deterministic finite automaton, which on players move \( \bar{m}_t = (m^1_t, m^2_t, ..., m^n_t) \) decides the rewards (payoff) \( \bar{r}_t = (r^1_t, r^2_t, ..., r^n_t) \in R^n \) for all players after the move and changes its state (i.e. players rewards can/will change), thus automaton \( A \) implements m-ary mapping \( \bar{\rho} : M^n \rightarrow R^n \).

The mapping \( \bar{\rho} \) could be different for different states of automaton \( A \) (during gameplay);
- \( R \) is the finite set of rewards. In ‘real’ video games rewards can be of several different types, numeric and/or symbolic (badges, ammo, health etc., see e.g. Phillips 2018 for typology); they represent the utility, thus should be comparable: here rewards \( R \) are shown as integers. In ‘real’ video games number of available actions (thus also rewards) may change in different states of the game (players can...
get e.g. new weapons or skills/spells), in the following number \( m \) of available actions and rewards remains constant.

Players make moves (create vectors \( \vec{m}_i = (m^1_i, m^2_i, ..., m^n_i) \)) simultaneously and after a move get information only about their own result, i.e. player \( p^j \in P \) gets the value \( r^j_1 \), but not information about moves what were made by other players, i.e. they do not get moves \( m^j_i \in M \), \( j \neq i \).

In order to extract randomness from players moves the mappings \( \tilde{p} \) should be resilient to game-bots – programs which play the game on behalf of human players or help player to advance in game. They try to extract information from players’ moves and predict play; producers of on-line games are developing methods to discover and disable them [Ah Reum Kang et al 2013].

Development of game-bots was started by dishonest players, but currently their development has become a direction of study in machine learning [Luzgin, R. 2018].

However, not every game can be ‘cracked’ by machine learning. Games where expectation of payoff is the same for all actions of player are non-learnable even for Tensorflow [Shai Ben-David et al 2019].

Game payoffs (in current state) can be described by the state’s payoff mapping, which for players \( p^1, p^2 \) is as in the economy-based game theory a matrix, where in rows are moves and corresponding rewards for player \( p^1 \), in columns – moves and rewards for player \( p^2 \); rewards for move \( m_i^j \) by player \( p^1 \) and move \( m_j^i \) by player \( p^2 \) are \( r^j_1, r^i_2 \):

Table 1. Payoffs table for a two-person game (in some state)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>...</th>
<th>( m )</th>
<th>1</th>
<th>...</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( r^1_{00}, r^2_{00} )</td>
<td>( r^1_{01}, r^2_{01} )</td>
<td>...</td>
<td>( r^1_{0m-1}, r^2_{0m-2} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
</tr>
<tr>
<td>1</td>
<td>( r^1_{10}, r^2_{10} )</td>
<td>( r^1_{11}, r^2_{11} )</td>
<td>...</td>
<td>( r^1_{1m-1}, r^2_{1m-1} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
</tr>
<tr>
<td>...</td>
<td>( r^1_{m-10}, r^2_{m-10} )</td>
<td>( r^1_{m-11}, r^2_{m-11} )</td>
<td>...</td>
<td>( r^1_{m-1m-1}, r^2_{m-1m-1} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
</tr>
</tbody>
</table>

\[
\text{Game is non-learnable, if all sums for all rows and columns are equal:}
\]

\[
\sum_{i} r^1_i = \sum_{i} r^2_i = ... = \sum_{i} r^1_{m-1} = \sum_{i} r^2_{m-1} = ... = \sum_{i} r^1_{m-1} = \sum_{i} r^2_{m-1} = V
\]

(1)

The value \( V / m \) is the (average) expected payoff of a move.

Games satisfying (1) have been considered in several earlier studies, e.g. in [Shapley 1963], [Plan 2017], [Shih-Fen CHENG et al 2004], but mostly only for their symmetry properties and not for the aspect considered here – extracting randomness from gameplay.

In the (economic) game theory publications are in the game payoff (for the current state) matrices usually shown only payoffs and not labels for player’s moves/strategies (every player can relabel them). Usually only values for \( r^j_1 \) are shown (especially if the game is zero-sum, i.e. \( r^j_1 = -r^j_2 \)), as e.g. a game from [Shapley 1963]:

---

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>( m )</th>
<th>1</th>
<th>...</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( r^1_{00}, r^2_{00} )</td>
<td>( r^1_{01}, r^2_{01} )</td>
<td>...</td>
<td>( r^1_{0m-1}, r^2_{0m-2} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>( r^1_{10}, r^2_{10} )</td>
<td>( r^1_{11}, r^2_{11} )</td>
<td>...</td>
<td>( r^1_{1m-1}, r^2_{1m-1} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>( r^1_{m-10}, r^2_{m-10} )</td>
<td>( r^1_{m-11}, r^2_{m-11} )</td>
<td>...</td>
<td>( r^1_{m-1m-1}, r^2_{m-1m-1} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>( m )</th>
<th>1</th>
<th>...</th>
<th>( m )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>( r^1_{00}, r^2_{00} )</td>
<td>( r^1_{01}, r^2_{01} )</td>
<td>...</td>
<td>( r^1_{0m-1}, r^2_{0m-2} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>( r^1_{10}, r^2_{10} )</td>
<td>( r^1_{11}, r^2_{11} )</td>
<td>...</td>
<td>( r^1_{1m-1}, r^2_{1m-1} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>( r^1_{m-10}, r^2_{m-10} )</td>
<td>( r^1_{m-11}, r^2_{m-11} )</td>
<td>...</td>
<td>( r^1_{m-1m-1}, r^2_{m-1m-1} )</td>
<td>( \sum r^1_i, \sum r^2_i )</td>
<td>...</td>
</tr>
</tbody>
</table>

\[
\text{Game is non-learnable, if all sums for all rows and columns are equal:}
\]

\[
\sum_{i} r^1_i = \sum_{i} r^2_i = ... = \sum_{i} r^1_{m-1} = \sum_{i} r^2_{m-1} = ... = \sum_{i} r^1_{m-1} = \sum_{i} r^2_{m-1} = V
\]

(1)

The value \( V / m \) is the (average) expected payoff of a move.

Games satisfying (1) have been considered in several earlier studies, e.g. in [Shapley 1963], [Plan 2017], [Shih-Fen CHENG et al 2004], but mostly only for their symmetry properties and not for the aspect considered here – extracting randomness from gameplay.

In the (economic) game theory publications are in the game payoff (for the current state) matrices usually shown only payoffs and not labels for player’s moves/strategies (every player can relabel them). Usually only values for \( r^j_1 \) are shown (especially if the game is zero-sum, i.e. \( r^j_1 = -r^j_2 \)), as e.g. a game from [Shapley 1963]:

---
Table 2. Payoff table for a non-symmetric non-learnable game

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
</tr>
</tbody>
</table>

3.2 Mappings of games

A game is non-learnable if the (expected) value of all moves for all players is the same, i.e. (1) holds in every state of the game. Such a non-learnable games have many properties in common and already in [Shapley 1963] was shown, that the payoff matrices for such a game could not be symmetric, but could be made skew-symmetric with re-arrangement (re-labelling) of player’s moves using blocks of size power of two. Here it is shown, that the matrix of such games could be transformed to totally non-symmetric form.

In the following is considered the class of all such games (for the fixed number \(m\) of player’s actions) and is shown, that all these games can be constructed from one vector of payoffs, using transformations of blocks of \(2 \times 2\) elements, thus all these games belong to a class closed for mappings of zero-sum vectors and \(2 \times 2\) rearrangements.

In the ‘real’ (i.e. economy-based) theory of games mappings (isomorphism, endomorphism) are considered only for games with the same number of players. For video games this is nonsense – in a (server-based) multiplayer game new players could appear at any moment and just the same way, already playing players could drop the game or just miss a number of moves. Nobody considers that (dis)appearing players introduce a new game, the game remains the same, thus we need a definition of games same-ness (similarity).

A game \(G = \langle P, M, A, R, \bar{P} \rangle\) is simpler (a homomorphic image) of a game \(G_1 = \langle P_1, M_1, A_1, R_1, \bar{P}_1 \rangle\) iff:

1. \(|P| \leq |P_1|\) · it is defined for a smaller set of players (but still \(|P| \geq 2\) · otherwise this is not a multiplayer game), let \(\iota : P \leftarrow P_1\) be an (arbitrary) mapping (function, renaming) of players;
2. \(|M| \leq |M_1|\) · players have less actions (but sill \(|M| \geq 2\) · otherwise this is not a game);
3. \(|R| \leq |R_1|\) · the number of (different) rewards is smaller (but still \(|R| \geq 2\) · otherwise this is not a game);
4. there exist mappings (with component wise application to vectors)

\[ \mu : M \leftarrow M_1 \]
\[ \nu : R \leftarrow R_1 \]

such that

\[ \nu(\bar{P}(m_1, m_2, \ldots, m^n)) = \bar{P}_1(\mu(m_1), \mu(m_2), \ldots, \mu(m^n)) \]

A game \(G = \langle P, M, A, R, \bar{P} \rangle\) is similar (nearly isomorphic) to a game \(G_1 = \langle P_1, M_1, A_1, R_1, \bar{P}_1 \rangle\), if in the above definitions \(\iota, \mu\) are 1-1 (i.e. renaming). The mapping \(\nu\) could not be 1-1, but for a games for humans who want to see differences in their actions it should preserve entropy, i.e. satisfy

\[ (\forall r_1, r_2, r_3 \in R_i) ((r_1 \neq r_2 \neq r_3 \neq r_1) \rightarrow (\nu(r_1) \neq \nu(r_2) \neq \nu(r_3) \neq \nu(r_1))) \]

Under this definition instances of a multiplayer game with different number of players are similar even with different dispersion of expectations of player awards.
3.3 Symmetric group-like games

Here are considered games, where randomness is generated by player’s decisions and not by game decision mechanism and structure. This randomness is similar to biological physically unclonable functions [Wali et al 2019] – it is the result of built-in hardware, first of all limits of player’s brains. Information is not created en masse, but in interactions of individuals. Therefore the game decision function

\[ \tilde{\rho}: (m_1^n, m_2^n, \ldots, m_n^n) \rightarrow (r_1^n, r_2^n, \ldots, r_n^n) \]

decides rewards \( r_1^n, r_2^n, \ldots, r_n^n \in \mathbb{R} \) after considering all interactions between all players, i.e. it uses a sub-function \( \rho \) to decide local interactions

\[ \rho: (m', m') \rightarrow (\tilde{r}'', \tilde{r}'') \]

which uses actions \( (m', m') \) of players \( p', p' \in \mathbb{P} \) and decides their (local) rewards \( \tilde{r}'', \tilde{r}'' \): the mapping \( \rho \) is deterministic, thus instead of mapping more suitable is the functional notation \( \rho(m', m') = (\tilde{r}', \tilde{r}') \)

The global rewards function \( \tilde{\rho} \) is cumulative, i.e.

\[ r_i^j = \sum_{i \in \mathbb{R}^{\mathbb{R}}} \rho(m_i^n, m_j^n) \]

In games of chance nobody wants to have worst chances by design of the game and all actions of players should be significant, i.e. could change the result (have maximal entropy – players want to have maximal fun!), thus the local decision function \( \rho \) should satisfy the following conditions.

1. For any actions \( m', m' \in \mathbb{M} \) if \( \rho(m', m') = (\tilde{r}', \tilde{r}') \), then \( \tilde{r} + \tilde{r}' = 0 \) – the game is zero-sum
2. For any actions \( m', m' \in \mathbb{M} \) if \( \rho(m', m') = (\tilde{r}', \tilde{r}') \), then \( \rho(m', m') = (\tilde{r}', \tilde{r}') \) – the game is symmetric (in some publications is used different symmetry \( \rho(m', m') = \rho(m', m') \) – payoffs do not depend on who made a move, such a game is e.g. ‘Zero-One’ considered in [26])
3. For any actions \( m', m' \in \mathbb{M} \), if \( \rho(m', m') = (\tilde{r}', \tilde{r}') \), then there exists an action \( m' \in \mathbb{M} \) such that \( \rho(m', m') = (\tilde{r}', \tilde{r}') \), i.e. game is group-like – any player could reverse the result of local interaction (if player had information about opponents move \( m' \) – but players make moves simultaneously and even after the move they have information only about result, i.e. reward – not about the move what caused it)
4. For any actions \( m'^1, m'^2 \in \mathbb{M} \) there exists an action \( m'^1 \in \mathbb{M} \) such that \( \rho(m'^1, m'^2) \neq \rho(m'^2, m'^2) \), i.e. all actions are efficient and one player can always change the reward of the second player.
5. For any actions \( m'^1, m'^2 \in \mathbb{M} \) there exists \( 1 \cdot 1 \) cyclically monotone mapping (see e.g. [CalTech 2004]) between sequences of rewards \( \rho(m'^1, 0), \rho(m'^1, 1), \ldots, \rho(m'^1, m' - 1) \) and \( \rho(m'^2, 0), \rho(m'^2, 1), \ldots, \rho(m'^2, m' - 1) \); from the property 2. it follows, that \( 1 \cdot 1 \) mapping exists also between the above sequences and the sequences \( \rho(0, m'), \rho(1, m'), \ldots, \rho(m' - 1, m') \) for any \( m' \in \mathbb{M} \).

Let \( NL \) be the class of all games with properties 1-5.

From the properties 1-4 of the local decision function \( \rho \) follow several properties of this function.
(1) From 1. it follows, that \( \rho(m', m') = (\vec{r}', \vec{r}') \) could be presented as \( \rho(m', m') = (\vec{r}', -\vec{r}') = (-\vec{r}', \vec{r}') \). The expressions, e.g. \( \rho(m', m') = (\vec{r}', -\vec{r}') \) could create impression, that \( \rho \) depends only on one argument (here – on the first, in the second expression \( \rho(m', m') = (-\vec{r}', \vec{r}') \) · on the second) thus in the following we will sometimes use notations \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \) to simplify presentation denote \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \). Thus \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \). Then for any \( i, j \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \). Thus \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \).

(2) \( \rho(m, m) = (0, 0) \)

Proof. By 2. for any \( m, m' \subseteq \mathbb{R} \), \( \rho(m, m') = m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) or \( m, m' \subseteq \mathbb{R} \). Thus \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \). Thus \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \). Then for any \( i, j \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \), \( m, m' \subseteq \mathbb{R} \) and \( m, m' \subseteq \mathbb{R} \) by 2. \( f \subseteq \mathbb{R} \)

(3) The elements of the set \( \mathbb{R} \) can be renamed so that \( \mathbb{R} \) has cyclic structure. Denote \( k = (|\mathbb{M}| - 1) / 2 \) if \( |\mathbb{M}| \) is odd and \( k = \mathbb{M} / 2 \) if \( |\mathbb{M}| \) is even and let \( \vec{r} = (0, r_1, ..., r_{k-1}, r_k) \) be a (strictly) monotony increasing sequence, i.e. from \( i < j \), \( 0 \leq i < j \leq k \) follows \( r_i < r_j \). First \( \vec{r} = (0, r_1, ..., r_{k-1}, r_k) \) be a (strictly) monotony increasing sequence, i.e. from \( i < j \), \( 0 \leq i < j \leq k \) follows \( r_i < r_j \). Then \( \vec{r} = (0, r_1, ..., r_{k-1}, r_k) \) be a (strictly) monotony increasing sequence, i.e. from \( i < j \), \( 0 \leq i < j \leq k \) follows \( r_i < r_j \). For \( k \) odd

\[
\vec{R} = \{0, r_1, ..., r_k, r_{k-1}, ..., r_1\}
\]

For \( k \) even

\[
\vec{R} = \{0, r_1, ..., r_k, r_{k-1}, ..., r_1\}
\]

3.4 Constructing a non-learnable game from a vector of rewards

In both cases the sequence \( \vec{R} \) has \( m \) elements, thus to simplify presentation denote \( \vec{R} = \{\vec{r}_0, \vec{r}_1, ..., \vec{r}_{m-1}\} \) and let \( \pi: \vec{R} \rightarrow \vec{R} \) be any substitution of the sequence \( \vec{R} \) without sub-cycles, i.e. for any \( \vec{r}_i, \vec{r}_j \in \vec{R} \), \( \exists t < m \) such that \( \pi'(\vec{r}_i) = \vec{r}_j \); then for any \( \vec{r} \in \vec{R} \) elements \( \vec{r}_0, \pi(\vec{r}_0), \pi^2(\vec{r}_0), ..., \pi^{m-1}(\vec{r}_0) \) cover the whole set \( \vec{R} \).

A game matrix for selected vector \( \vec{R} \) can be the following:

<table>
<thead>
<tr>
<th>( \vec{r}_0 )</th>
<th>( \vec{r}_1 )</th>
<th>...</th>
<th>( \vec{r}_{m-1} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi(\vec{r}_0) )</td>
<td>( \pi(\vec{r}_1) )</td>
<td>...</td>
<td>( \pi(\vec{r}_{m-1}) )</td>
</tr>
<tr>
<td>( \pi^2(\vec{r}_0) )</td>
<td>( \pi^2(\vec{r}_1) )</td>
<td>...</td>
<td>( \pi^2(\vec{r}_{m-1}) )</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( \pi^{m-1}(\vec{r}_0) )</td>
<td>( \pi^{m-1}(\vec{r}_1) )</td>
<td>...</td>
<td>( \pi^{m-1}(\vec{r}_{m-1}) )</td>
</tr>
</tbody>
</table>

The conditions (1), (2) hold – in all rows and columns are just the elements of the set \( \vec{R} \) (but in different order), i.e. this is a matrix for a non-learnable game constructed from the sequence \( \vec{R} \).

3.5 Modifications of games from \( \mathbb{NL} \)

The class \( \mathbb{NL} \) of non-learnable games allows modifications of their matrixes preserving condition (1):

(a) Multiplication of all elements with a constant (even with 0 – the property (1) will hold, but the game will not have any entropy – nobody would like to play):
(b) Adding a constant to all elements the – in `real’ video games players rewards are thousands or even bigger – this seems to make the game more fun;
(c) Manipulating the corner values of any rectangular block: if \( x_{i,j}, x_{i,k}, x_{i,j}, x_{i,k} \) are corner elements of a rectangle in a game matrix, then they could be replaced with \( x_{i,j} + c, x_{i,k} - c, x_{i,j} - c, x_{i,k} + c \) for any constant \( c \) (the rectangle adjustment); similar property was considered already in [Shapley 1963]).

Transformations (a), (b) change the value of \( V \), i.e. the average expected payoff \( V/m \), thus can be used for transforming/comparing non-learnable games with different value of \( V \). Transformation (c) does not change \( V/m \) and with induction on greatest difference in matrix it is easy to see, that with transformation (c) all games from the class \( \text{NL} \) with the same \( V \) could be reduced to a constant game (all elements of the matrix are the same \( \cdot V/m \)). Since the rectangle adjustment transformation (c) is reversible it follows, that all games of the class \( \text{NL} \) with the same number of actions \( m \) can be transformed to each other, i.e. are all similar (nearly isomorphic).

3.6 Examples
Below is the decision table for the function \( \rho \) for a 3-ary game: this is the classical rock-paper-scissors game with encoding of player’s actions rock=0, paper=1, scissors=2 and the set of rewards \( R = \{-1,0,1\} \) (but using property (b) it could be changed to \( R = \{-100000,0,100000\} \cdot \) human players would like this much more!); in the table are indicated rewards \( r' \) of player \( p' \in P \) (rewards of player \( p' \in P \) are \( p' = -p' \)); in the last row/column is indicated direction of monotonicity and the sum for the corresponding row/column.

Table 4. Decision table for a rock-paper-scissors game with encoding of actions rock=0, paper=1, scissors=2

\[
\begin{array}{cccc|c}
\rho & 0 & 1 & 2 & \Sigma \\
0 & 0 & 1 & -1 & \rightarrow 0 \\
1 & -1 & 0 & 1 & \rightarrow 0 \\
2 & 1 & -1 & 0 & \rightarrow 0 \\
\Sigma & 1 & 0 & 1 & 0 \\
\end{array}
\] (4)

The same for a 4-ary game \( (m – \text{even}) \) with \( R = \{-2,0,2\} \):

Table 5. Decision table for a 4-ary group like game with \( R = \{-2,0,2\} \).

\[
\begin{array}{cccc|c}
\rho & 0 & 1 & 2 & 3 & \Sigma \\
0 & 0 & -2 & 0 & 2 & \leftrightarrow 0 \\
1 & 2 & 0 & -2 & 0 & \leftrightarrow 0 \\
2 & 0 & 2 & 0 & -2 & \leftrightarrow 0 \\
3 & -2 & 0 & 2 & 0 & \leftrightarrow 0 \\
\Sigma & 1 & 0 & 1 & 0 & 0 \\
\end{array}
\] (5)
From conditions 2.4. in 3.3 it is clear, that if m is odd then these games are fair ([Aten 2019]), i.e. 
\[|\rho^{-1}(r)| = |\rho^{-1}(r')| = m \] for any \( r, r' \in R \); if m is even, then 
\[|\rho^{-1}(r)| = |\rho^{-1}(r')| = m \] for any \( r, r' \in R, r \neq 0 \neq r' \), 
\[|\rho^{-1}(0)| = 2^* \] (every row/column in the table has 0 twice).

The property (c) of the class \( \mathcal{NL} \) allows to modify the presented above example (2) with rectangle adjustment to totally non-symmetric form:

Table. 6. Modified with transformation \( x_{0,1} - 2, x_{0,2} + 2, x_{2,1} + 2, x_{2,2} - 2 \) matrix (3)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>-1</th>
<th>4</th>
<th>-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>-2</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>-1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Particular cases of this class of games were considered in [Henno et al 2018], [Henno et al 2019]: games belonging to this class were (for odd values of m) considered also e.g. in [Akin 2018].

The above minuscule examples could be combined to create a ‘real’ game, adjoining the matrices to a game (automaton) transitions graph, e.g. using as the transition table the graph of a 4-bit digital clock from [Henno 2017]: for rewards are applied matrixes from above (modified with transformations (a), (b)):

Fig. 3. A non-learnable game from a 4-bit lock automaton: inputs without transformations keep state.

4 CREATING ENCRYPTION KEY

In the above was presented a class on non-learnable games, i.e. where competing players always have to select their moves randomly and presented some examples of such games: some of such games (‘Odd·Even’, ‘Matching Pennies’) were considered in [Henno et al 2017] and above is shown, that such a game could be created for any vector of rewards. In the following is described a procedure for creating a shared secret (encryption key) using the randomness created by players when playing such a game where the best strategy for players is total randomness.

The server records sequence of player’s moves, e.g. for a game with two players Alice and Bob with \( l \) moves this could be \( m_1, m_{12}, m_1, m_{22}, m_{2}, m_{2n}, m_{2n}, m_{2n}, m_{2n}, ... \); here \( m_1, m_{12} \) are respectively moves of Alice and Bob in gameplay move \( t \). Both players know only their own moves and from every result the probability any move of opponent which caused this result is the same (the above property (4)), thus players cannot guess moves of other players: if the situation is local, players e.g. can see each other moves, then they do not need any elaborated communication – they can speak directly with each other.

To generate a key server send to players the sequence of all moves from which the player moves are removed, e.g. server sends to Alice the sequence \( \ast m_{12}, \ast m_{22}, ... \ast m_{2n}, \ast m_{2n}, \ast m_{2n} \ast \) this information with holes does not give to an eavesdropper any information (it is assumed, that the game server communication with players is secure, so here is the only time when the game communication is used for the chat system). When players replace in the received sequences with holes their own moves they all get the same random sequence which could be used as the secure random key for symmetric encryption.
Fig. 4. Key generation combining a sequence with holes from server with sequence of player moves.

If all the following messages are secured/encrypted with the means used in game for sending player's moves and rewards, players store their moves in the order they made them and when player enters the game (logs in) he receives together with his gameplay client also (a link to) a cryptographic library, e.g. some member from the W3C list or (when using node.js) Node.js Crypto module, then the procedure to create a secret key could be described by following steps:

1. Player1 sends to server message: chat(playerList) here playerList is a list of players with whom he/she would like to communicate or the word all - player wants to start global chat
2. Server checks whether there is already enough entropy (at least e.g. 12-20 moves have been made) and Player1 does not look like a bot - has comparable rewards
3. Server sends invitations to start chat to all members of playerList (with time limit for answer)
4. After time limit server removes from playerList all players who did not answer the invitation
5. If the remaining list contains some players, server adds to list also the Player1 and sends to all members the global list of moves from which the moves of player himself are removed (replaced by '*') - with time limit for answer
6. Players replace the '*' in received list with their own moves (in the correct order); the corresponding procedure could also be delivered to all players already together with the game client
7. To check, players send to server (encrypted with the new key) message "Encryption OK"
8. Server removes from playerList the players, whose messages were not decipherable
9. If there still remain players (>1), server sends them all list of addresses of other members, thus they can start a chat forum.

To increase security of the key server could use some filters, e.g. for a randomly-selected value \( r \in \mathbb{R} \) remove all moves which produced result \( r \); to speed up the game could be used multi-moves, i.e. participants send in every move a fixed-length sequence of moves etc.; several test implementations are in work.

5 CONCLUSIONS

In above is considered a class of non-learnable games, where competing players should be maximally random and presented a framework for generating from players moves a secure key for symmetric encryption. The process of generating a key (shared secret) uses randomness generated in gameplay and does not use public-key step; thus it also does not have to distribute any personal information of players nor need any higher-level security authorities, i.e. the use of randomness and securing communication is self-sustained.
REFERENCES

Shapley 1963. SOME TOPICS IN TWO-PERSON GAMES. The US Department of Defence Rand Corporation Memorandum RM-3672-PR
The Impact of Students' Pre-Knowledge on Learning Computer Programming

MARKO HÖLBL and LILI NEMEC ZLATOLAS, University of Maribor

Students taking courses of computer programming have different pre-knowledge on the topic, which not only includes their knowledge of a specific computer programming language but also their ability for algorithmic thinking and the understanding of the concept of computer programming. At the beginning of the course, they were asked to self-evaluate their knowledge of computer programming and take a pre-course test of their knowledge in JavaScript. In the scope of the research, we have compared the results of the pre-course test of students with technical and general High School education, as well as the results they achieved at the post-course test. Altogether, 55 students collaborated in both pre and post-course tests. The results indicate that students with technical and general High School backgrounds scored similarly on a pre-course test of computer programming and that students with general High School backgrounds acquired more knowledge in the course compared to the ones with a technical High School background. The results also indicate that the students with lower self-evaluation of computer programming have acquired more knowledge during the course compared to the ones with higher self-evaluation scores.

1. INTRODUCTION

Learning the principles of computer programming is an essential skill for an IT professional. Teaching computer programming is a challenge, as the algorithmic way of thinking is uncommon for people [Matthew et al. 2007, Futschek 2006, Knuth 1985, Sleeman 2002]. There are significant differences between experts and novices in computer programming, as well as many challenges on how to teach computer programming in order for the students to learn as fast as possible [Cooper et al. 2000, Fowler and Cusack 2011]. Becoming a good programmer is a cumbersome process, requiring a lot of studying and practising.

Students with different secondary education level backgrounds are joining Universities to study Informatics and Computer Science or an ICT related study. A programming course usually serves as a fundamental course in any ICT-related Study Programme, due to its importance in understanding the concept of ICT systems. Additionally, the development of ICT solutions and/or services requires knowledge of computer programming.

Many students come to the university with little or no pre-knowledge of programming and often tend to overestimate their knowledge.

The studies have shown that pre-knowledge can have an effect on students so that they believe they already have enough knowledge on the topic and do not need to learn any more [McMillan and Hearn 2008, Sewell 2002]. This might lead to the effect that such students do not achieve the level required by university standards. Additionally, previous studies also employed different approaches to teaching programming [Janzen and Saiedian 2008, Kalelioğlu 2015].

The authors acknowledge the financial support from the Slovenian Research Agency (Research Core funding No. P2-0057) and from the European Union’s Horizon 2020 Research and Innovation program under the CyberSec4Europe project (GA No. 830929). We would also like to thank the participants of this research project.

Author's address: M. Hölbl and L. Nemec Zlatolas, Faculty of Electrical Engineering and Computer Science, Koroska cesta 46, 200 Maribor, Slovenia; email: marko.holbl@um.si, lili.nemeczlatolas@um.si

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

In this work, we present a study where we analysed if students with different secondary education level backgrounds, thus with different pre-knowledge of computer programming and their self-perception of this knowledge, influenced their results in learning computer programming. The analysis was performed with active students before taking the course, and after they had taken the course lessons and completed the practical laboratory work. The experiment was conducted with students who took the course Programming for Media, which involves the basics of JavaScript programming.

We set up the following research question: How does the pre-knowledge of computer programming affect the knowledge of computer programming after taking lectures from the computer programming course?

The structure of the paper is as follows. The description of research methods is provided in Section 2 and the main contribution of the paper in Section 3, where the results and a discussion are given. Finally, the conclusions are presented in Section 4.

2. RESEARCH ON STUDENTS’ KNOWLEDGE OF COMPUTER PROGRAMMING IN JAVASCRIPT

We have used a test type of evaluation of pre and post knowledge of students in programming. Further details are presented in the following sub-sections.

2.1 Data collection and participants

At the beginning of the semester, students attending the Programming for Media Course were asked to fill in a test to assess their skills on computer programming in JavaScript. After the course was finished and they had attended lectures as well as the exercises of the course, their knowledge and acquired skills in computer programming in JavaScript were tested again. The sample of demographics is presented in Table I. Altogether, 55 students collaborated in both parts of the survey – the pre-knowledge test and the test after they obtained knowledge at lectures and exercises. Most of the students were in their first or second year of studies. There were more female students collaborating in the survey, and the majority of students were enrolled in the Study Programme Media Communications. The previous secondary education (High School) of students was either general or technical.

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>SAMPLE RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gender</td>
<td>Male 43.6%</td>
</tr>
<tr>
<td></td>
<td>Female 56.4%</td>
</tr>
<tr>
<td>Study Programme</td>
<td>Media Communications 74.5%</td>
</tr>
<tr>
<td></td>
<td>Informatics and Technologies of Communication 25.5%</td>
</tr>
<tr>
<td>Previous education</td>
<td>General 60.0%</td>
</tr>
<tr>
<td></td>
<td>Technical 40.0%</td>
</tr>
</tbody>
</table>

2.2 Measures

Measurement items were tested with a 7-point Likert-scale ranging from 1 to 7. The measurement items are presented in Table II. There were altogether 25 questions in the survey. To connect the pre-course test of each student and the post lectures test, we used a unique ID number that a student had to enter when filling in the pre-course test on his/her knowledge and on the exam at the
end of the course. The first part of the test (demographical questions and questions 1-11 in Table II) were only asked in the pre-course test. In the post-course test (after the students had already done the tasks and taken the lectures) only the questions 12-18 were asked. The questions were chosen to suit the lectures concept. To connect the pre-knowledge of each student and the grade for Database Modelling, we used a unique ID number that a student had to enter when filling in the self-evaluation questionnaire on his/her knowledge and on the exam at the end of the course.

Table II. Measurement of variables.

<table>
<thead>
<tr>
<th>QUESTIONS</th>
<th>POSSIBLE ANSWERS</th>
<th>Present in pre-course test</th>
<th>Present in post-course test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Have you already attended courses on computer programming before?</td>
<td>Y/N</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>I have a lot of pre-knowledge on computer programming.</td>
<td>1 – Strongly disagree 2 – Disagree 3 – More or less disagree 4 – Undecided 5 – More or less agree 6 – Agree 7 – Strongly agree</td>
<td>Y</td>
<td>N</td>
</tr>
<tr>
<td>I know how to use the JavaScript computer programming language.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use some computer programming language.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use variables.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use arrays.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use conditional expressions / statements.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use loops for computer programming.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use functions in computer programming.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use Objects in computer programming.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I know how to use DOM.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>What is a variable?</td>
<td>A box into which we save the data. A box into which we save unchangeable data. JavaScript expression. JavaScript number. I don't know.</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>If the user enters &quot;10&quot; in a prompt window, what will be the result?</td>
<td>24 34 4 Undefined</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>var x = Number (prompt(&quot;Insert value:&quot;));</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>var res = x * 3 - 1;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>res += 5; console.log(res);</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>If the user enters &quot;10&quot; in a prompt window, what will be the result?</td>
<td>&quot;A&quot; &quot;B&quot; &quot;C&quot; &quot;A&quot; in &quot;B&quot;</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>var x = Number (prompt(&quot;Insert value:&quot;));</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (x &lt;= 10) { if (x &gt;= 5) { console.log (&quot;A&quot;); } console.log(&quot;B&quot;); } else { console.log(&quot;C&quot;); }</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Which operator needs to be in the place where &quot;???&quot; is now so that the expression makes sense?</td>
<td>AND operator: &amp;&amp; OR operator:</td>
<td></td>
<td>NOT operator: ! No operator.</td>
</tr>
<tr>
<td>var num1 = Number (prompt(&quot;Insert number 1:&quot;));</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3. DATA ANALYSIS AND RESULTS

After collecting the data using the test, we have used SPSS for data analysis in this study. The results are presented in the following sub-sections.

3.1 Testing the programming pre-knowledge of students with different educational backgrounds

First, we compared the students with technical and general High School background. We conducted the pre-course test of knowledge with questions 12-18, which are given in Table II. On average, both the students with technical and general high school educational background scored similarly on a 7 question test for pre-knowledge of computer programming in JavaScript. The average for a group with a technical High School background was 3.68; while students with a general High School background scored a bit better in the pre-course test that is 3.79 points on average out of 7. A more detailed analysis is presented in Table III.

Table III. Comparison of pre-knowledge between students with different High School backgrounds.

<table>
<thead>
<tr>
<th>High school education</th>
<th>Pre-course test score (0-7)</th>
<th>TOTAL</th>
<th>AVERAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.00</td>
<td>1.00</td>
<td>2.00</td>
</tr>
<tr>
<td>Technical High School</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>General High School</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>TOTAL</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
Additionally, we analysed how much knowledge students with general and technical High School backgrounds acquired during the course by calculating the difference in the pre and post-course tests. These were the same in both cases (questions 12-18 in Table II). Altogether, 7 points could be achieved in repeated tests. If a student scored more points in the post-course test than in the pre-course test, the value was positive. For example, if a student scored 4 points in the pre-course test and 6 points in the post-course test, the difference would be +2 points. As Figure 1 indicated, students with a technical High School background improved on average by 1 point, and students with a general High School background on average by 2 points between the pre- and post-course tests.

![Fig. 1. A box plot of the difference in pre- and post-course tests between groups with different High School backgrounds.](image)

### 3.2 Comparison of pre- and post-course test results from students with different pre-knowledge on programming

An analysis of the variance between different groups was conducted next. Questions 2 to 11 indicate the level of knowledge of JavaScript programming according to the self-evaluation of students. The following 7 questions test basic users’ knowledge on JavaScript programming (questions 12 – 18 given in Table II). After calculating the difference, we ran a Univariate analysis of Variance and compared the different groups.

Next, we divided students based on their self-evaluation in computer programming. Group 1 had an average lower than 4 out of 7 when doing the self-evaluation (questions 2-11), while group 2 self-assessed their computer programming knowledge higher than 4 out of 7. As can be seen from Table IV, students who had lower pre-knowledge of computer programming improved their score on
average by 2.53 points after taking lectures, compared to the ones with higher pre-knowledge. Those improved their score just by 0.91 points. However, students with higher pre-knowledge scored better in both cases – in the pre-course test and in the post-course test. Yet, in the post-course test, the difference was not as high. The results of the univariate analysis of variance also indicate that the significance of Levene’s test [David et al. 2006] is <.005, which rejects the null hypothesis, where we tested if the error variance of the dependent variable was equal across the groups. This indicates a significant difference between groups with low pre-knowledge and high pre-knowledge in computer programming. We conducted another test, the “Test of Between-Subjects Effects with dependent variable Difference in pre- and post-course tests”, and the difference between the two groups was also significant, and lower than 0.005. Furthermore, the partial eta squared is .210, which means that 21% of the variance is explained in the dependent variable.

<table>
<thead>
<tr>
<th>Table IV. Univariate Analysis of Variance of groups with low and high pre-knowledge.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Descriptive statistics for groups</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Group 1: low pre-knowledge</td>
</tr>
<tr>
<td>Group 2: high pre-knowledge</td>
</tr>
<tr>
<td><strong>Descriptive statistics with dependent variable Difference in pre- and post-course tests</strong></td>
</tr>
<tr>
<td>Mean (difference)</td>
</tr>
<tr>
<td>Group 1: low pre-knowledge</td>
</tr>
<tr>
<td>Group 2: high pre-knowledge</td>
</tr>
<tr>
<td><strong>Levene’s Test of Equality of Error Variances with dependent variable Difference in pre- and post-course tests</strong></td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>7.473</td>
</tr>
<tr>
<td><strong>Test of Between-Subjects Effects with dependent variable Difference in pre- and post-course tests</strong></td>
</tr>
<tr>
<td>Mean Square</td>
</tr>
<tr>
<td>35.042</td>
</tr>
</tbody>
</table>

Fig. 2 depicts the box plot of the two different groups and the differences between the pre and the post-course tests. As can be concluded from the boxplot, some of the students have even scored less in the post-course test than in the pre-course test. Some with low pre-knowledge even scored up to 6 points (out of 7) more in the post-course test, while the others with high pre-knowledge scored mostly up to 3 points more in the post-course test in comparison to the pre-course test.
The Impact of Students’ Pre-Knowledge on Learning Computer Programming

3.3 Discussion

In this study, we investigated how the pre-knowledge of computer programming affects the knowledge of computer programming after taking lectures from the computer programming course. We found that the students with low pre-knowledge in computer programming gained more knowledge during the lessons than the ones with high pre-knowledge of programming. Additionally, students with a more technical background in computer programming did not present better knowledge of computer programming in the pre-course test than students with no technical backgrounds. Students having luck in the pre-course test, since the answers were a, b, c, d and they could have chosen the right one, which could explain this. However, it would be expected that, if they did not know the answer, they would choose the “don’t know” option. Students from general High School backgrounds also learned more while taking the lessons than the students from technical High Schools.

4. CONCLUSION

At the Faculty of Electrical Engineering and Computer Science at the University of Maribor, students were taking the course Programming for Media in the first or second year of their studies. We asked the students at the beginning of the course to self-evaluate their knowledge of Programming in JavaScript, and, at the same time, test their computer programming skills using a questionnaire. After taking the course, the students again took a test of their computer programming skills. Our results indicate that the students with no, or very little, pre-knowledge of computer programming had acquired much knowledge from the basic course of computer programming, and
had achieved almost the same results as students with a lot more pre-knowledge. We could conclude
that the self-esteem of the students with pre-programming skills is not a positive thing, because the
students put less effort into the course. Our statistical analysis showed that students with technical
and general High School backgrounds scored similarly on a pre-course test of computer
programming, which was quite surprising. We expected that students with a technical background
would have more knowledge of computer programming. Students with general High School
backgrounds also acquired more knowledge while taking the course than students with a technical
High School background. The limitation of this study is a small sample, but the study could be tested
on a larger sample.

Additionally, it would be expected that students with a technical background would acquire more
knowledge after attending the lessons of the Programming for Media. However, our results indicate
the opposite.

REFERENCES
Matthew Butler, Matthew Butler, and Michael Morgan. 2007. Learning challenges faced by novice programming students
Isecon, 506–539.
in Honor of Harold Hotelling. DOI:https://doi.org/10.2307/2984135
Adrian Fowler and Brian Cusack. 2011. Kodu Game Lab: Improving the motivation for learning programming concepts. In
Foundations of Digital Games, 238–240. DOI:https://doi.org/10.1145/2159365.2159398
Gerald Futschek. 2006. Algorithmic Thinking: The Key for Understanding Computer Science. In International conference on
informatics in secondary schools-evolution and perspectives, 159–168. DOI:https://doi.org/10.1007/11915355_15
532. DOI:https://doi.org/10.1145/1352322.1352315
Filiz Kalelioglu. 2015. A new way of teaching programming skills to K-12 students: Code.org. Comput. Human Behav. 52,
(November 2015), 200–210. DOI:https://doi.org/10.1016/J.CHB.2015.05.047
DOI:https://doi.org/10.2307/2322871
J H McMillan and Jessica Hearn. 2008. Student self-assessment: The key to stronger student motivation and higher
achievement. Educational Horizons 87: 40–49
DOI:https://doi.org/10.1145/6592.214913
On Measuring Learning Success of Students with Disabilities in Virtual Environments

MIRJANA IVANOVIĆ, University of Novi Sad
AMELIA BÂDICA, University of Craiova
MARIA GANZHA, Warsaw University of Technology and Systems Research Institute Polish Academy of Sciences
MARCUS PAPRZYCKI, Warsaw Management Academy and Systems Research Institute Polish Academy of Sciences
COSTIN BÂDICA, University of Craiova
ALEKSANDRA KLÂŠNJÂ-MILIČEVIC, University of Novi Sad

During more than the last three decades, the interest in educational arena supported by technological advancements has been growing systematically. New, different, concepts and tools including: Innovative Educational Environments, Future Classrooms, and Virtual Laboratories emerged constantly, introducing innovative ways of e-learning. On the other hand, Inclusive Education is getting more attention and importance, as contemporary classrooms must include students of diverse abilities, to learn and socialize together. The modern classroom should not discriminate between students with disabilities and students without disabilities by offering them equal attention and opportunities. In this position paper, our intention is to consider possible technological influences on Virtual Environments/classrooms having in mind Science, Technology, Engineering and Mathematics (i.e. STEM) education and, accordingly, propose some possible measures for learning success of students with disabilities. The key issue in these activities is to make all students feel welcomed and properly supported in their efforts to gain adequate knowledge and skills, while collaborating with their peers and interacting with e-learning environments.

1 INTRODUCTION

During the last several decades (As a very basic example of considerations concerning use of technology in education that have been formulated almost 30 years ago see, for instance, this references list?), the interest in education supported by technological advancements has grown enormously. New, creative learning environments emerged constantly, exhibiting highly promising features, for advancing the educational arena. Virtual, multi-functional environments and classrooms determine the more active involvement of teachers and students, supported by advanced pedagogical approaches enabled by modern digital technologies. Challenging technological, pedagogical and methodological approaches, in educational processes, promote positive impact on students’ academic knowledge, skills, interaction and levels of technological literacy. Technology

Authors address: Mirjana Ivanovic, Faculty of Sciences, Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia; e-mail: mira@dmi.uns.ac.rs; Amelia Badica, Faculty of economics and business administration 13, A.I. Cuza Street, Craiova, 200585, Romania; e-mail: ameliab@ameliab@yahoo.com; Maria Ganzha, ul. Newelska 6, 01-447 Warsaw, Poland; e-mail: maria.ganzha@ibspan.waw.pl; Marcin Paprzycki, ul. Newelska 6, 01-447 Warsaw, Poland; e-mail: paprzyck@ibspan.waw.pl; Costin Badica, Department of computers and information technology, Blvd. Decebal nr. 107, RO-200440, Craiova, Romania; e-mail: cbadica@soft.edu.ro; Aleksandra Klasnja Milicevic, Faculty of Sciences, Trg Dositeja Obradovica 4, 21000 Novi Sad, Serbia; e-mail: akm@dmi.uns.ac.rs;

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
3 http://www.ibspan.waw.pl/~paprzyck/mp/cvr/education/neted.html
enhanced learning assure flexible, responsive, and effective use of digital technology [Patrícia and Neuza 2018].

Technological advancements are also directly influencing Inclusive Education. It is important that students of diverse abilities and backgrounds get learning and socializing together in the same classroom. However, such new way of learning and socializing needs significant educational reforms, at least in the area of adequate teacher and staff training, as well as the availability of technical support and adequate learning materials tailored to specific students’ needs. Inclusive education, in essence, means that students with diverse abilities and backgrounds learn together, in the same classroom (real or virtual/distance) by receiving high quality support that is needed to achieve success in the core elements of the proposed curriculum. The modern classroom should treat the students with disabilities (SwD) as being, fundamentally, as competent as students without disabilities. Technological advancement in ICT, especially in domains like robotics, mechatronics, and artificial intelligence, together with innovative instructional design and novel pedagogical approaches, are essential premises for the successful inclusive education, regardless of student differences and of their diversity in cognitive, academic, physical, social, and emotional traits [Savin-Baden 2015].

Students with disabilities have specific, individual learning needs, as well as restricted learning abilities. So the pedagogical methods have to be oriented towards developing unconventional teaching practices, adequate educational resources, and they should apply specific assessments supported by adjusted measures of learning success. Distance learning, as well as mobile learning supported by the use of tablets, smartphones, and similar devices, may offer innovative solutions for adequate education of SwD. Mobile applications and special educational tools with speech-to-text and text-to-speech functionalities can highly support SwD by enabling their fair engagement in the learning/teaching process, regardless if they are in real or virtual classrooms.

In their early days, distance learning and e-learning were concentrated predominantly on narrative disciplines, without the need to use laboratories and hands-on activities that characterize (Science, Technologies, Engineering and Mathematics) STEM disciplines. Modern technologies strongly contributed to the inclusion of these disciplines into the focus of distance and e-learning education, by providing adequate technological support, like for example the development and application of online virtual laboratories for students of science and engineering disciplines (mathematics, informatics, physics, robotics, mechatronics, control systems and so on). In such specialized virtual laboratories a student can exercise specific practical tasks regardless of time or location boundaries, fear of improper handling of equipment, requirement for a live instructor (who will be replaced with Pedagogical Agents and Chatbots) [Terracina et al. 2016]. Here, the good news is that important early issues related to, broadly understood, computer literacy (see, for instance [Paprzycki 1992]) have almost disappeared across the “developed world”.

Additionally, great potential lies in use of Internet-of-Things (IoT) and Internet-of-Everything (IoE) that enable virtual use of various smart and specialized devices, thus making learning easier, faster, and safer. These innovative approaches bring about incredible potential for students with disabilities as well, by enabling them to learn from homes and, at the same time, to interact and collaborate with their peers and teachers.

In this position paper we concentrate our attention on possible technological influences on Virtual Environments and on some possibilities to measure the performance of such systems, in order to increase learning success of students with disabilities.

The remainder of the paper is organized as follows. Different contemporary technologies and their influences on learning of students with disabilities are presented in Section 2. Deeper considerations of measures for learning success of SwD and proposition of most appropriate characteristics of
Pedagogical agents in social aspects of learning, are given in Section 3. Last Section brings concluding remarks.

2 TECHNOLOGICAL INFLUENCES ON LEARNING OF STUDENTS WITH DISABILITIES

Continuous technological influences on improvement of classroom flexibility, development of modern teaching methodologies, usage of e-services, mobile/smart devices and social media should be adapted for students with disabilities. Current worldwide revolution that is happening in the education (in both real and virtual classrooms) is initiated by numerous applications of Internet of Things (IoT) and Internet-of-Everything (IoE). For example, the paper [Farhan et al. 2018] shows that activities of students in an e-learning environment can be effectively measured using an attention-scoring model (ASM). The model is based on the observation of students’ faces and eyes in order to discover their attention and emotions. The IoT can have implications on the overall delivery of the educational material in highly innovative manner in all aspects of students’ activities. IoT, intelligent technologies and new concepts, such as cloud computing, educational and learning analytics, wearable technology, etc. promote the materialization of smart education.

Availability of a wide range of multimodal educational resources that are at students’ disposal and that can enhance teaching and learning rapidly increases in the emerging world of the IoE. While IoT represents the networked connection of different physical objects, IoE represents a network of smart objects, i.e. interconnected things where the difference between the physical object and the digital information augmenting them is blurred [Selinger et al. 2013]. The huge number of connections of people (including students and teachers), processes, data, and other things brings about a completely novel concept of Internet of Learning Things [Selinger et al. 2013]. Inclusion of IoE in educational activities, with the aim to improve learning and assessment capacities is seen as highly promising, more sustainable, and challenging future direction. Different universities and companies, all over the world, strive in development of IoE-based smart classrooms that include numerous highly heterogeneous devices: smart tables, interactive whiteboards, 3D printers, sensor gloves, eye-trackers, headsets attention monitoring systems, Human-Computer interfaces, and other digital laboratory devices. Such devices support reducing different obstacles and barriers faced by students with disabilities, such as physical, cognitive, social and organizational barriers. Human-Computer interfaces, supported by suitable technical devices, are an essential element to support all students in virtual educational environments. The developed IoE-based smart educational environments can enable a completely revolutionized learning and teaching practice, for students with disabilities, in STEM disciplines.

2.1 Role of E-Learning Environments in Educational Processes

During the last decade, a wide range of software systems, enriched by including numerous and diverse aspects of multimedia and Web technologies, and seamless, multimodal, user-friendly Human-Computer Interaction, have been developed for promoting innovative and smart learning.

One such general purpose Learning Management System that is probably mostly used nowadays is Moodle. Its development was based on sound pedagogical principles, and it can support diverse e-learning approaches, including distance education, flipped classroom and blended learning. Since its introduction, about two decades ago, many improvements and enhancements of Moodle have been developed [Open Source Technology 2014], [Link1] to follow requirements of ever-changing educational demands. Constantly innovated Moodle environment helps educators to build multimodal, multifunctional, and interactive e-lessons. There are a lot of additional educational services and components that can enhance Moodle in order to improve classroom flexibility and offer pleasant learning atmosphere, for all diverse categories of students.
Apart from technological advancements, pedagogical aspects are of paramount importance for modern distance educational environments. The preparation of educational resources, for Virtual Environments and students with disabilities, to support accessibility with wide range of functionalities, must be based on non-conventional hardware and software components such as upgraded keyboards, speech-to-text and text-to-speech functionalities, scalable fonts, and so on, by augmenting the Virtual Environment with:

- multiple formats (HTML, RTF, PDF, etc.), to ensure acceptability by a wider range of students,
- multiple modalities (visual, auditory, kinesthetic, or tactile) to increase students’ motivation and interest,
- more flexible course materials to accommodate students’ differences and diversity,
- innovative use of modern multimedia technology supported by educational data mining and learning analytics to obtain high level of personalization and tailored recommendations.

Additionally, today’s innovative learning models and concepts, like flipped classrooms [Lage et al. 2000], serious games, and massive open online courses (MOOCs), are constantly progressing and enable more students (especially for students with disabilities) to engage from more and diverse locations at a wider scale in higher quality education using adequately prepared educational resources.

Regardless of these general purpose systems, worldwide research and academic institutions have been developing their own, specific-purpose, usually intelligent and personalized educational systems/environments. They are extremely important in facilitating higher-quality STEM e-learning [Klašnja-Miličević et al. 2017]. An essential feature of such systems is personalization that provides open, flexible, and tailored learning to students with diverse abilities.

Key desirable features of a wide range of e-learning systems include new models for intelligent personalized interaction and teaching material recommendation. These particular intelligent personalized interactions are usually required to address the specific and personal needs of each student, including: learning style [Klašnja-Miličević et al. 2011], personal learning characteristics of the learning style (like: Discussion forums, Simulations, Roles and serious games, Case studies and so on), as well as the most suitable electronic media for representation of educational resources (e-book, Forum, Wiki, Weblog, Podcast, and so on). These numerous possibilities to enhance traditional learning provide students with disabilities better access to information based on visual, auditory and kinesthetic means. At the same time, the teacher and peer’s interaction and collaboration with SwD can be significantly improved. With the availability of numerous online collaboration tools, existing today on the market, student-teacher and student-student interaction has never been easier. Innovative ICT technologies (cutting-edge multimedia, speech and mobile technologies, IoT and IoE) improve accessibility and contribute to the transition of teaching from a traditional “one size fits all” approach to more individualized “one size fits one” learning solutions that are more appropriate for students with disabilities.

In order to provide accessibility to literature for students with sensory, physical, cognitive and psychosocial disabilities, the development of multimedia libraries and enhanced virtual laboratories based on speech technologies is a challenging task [Lynch and Ghergulescu 2017]. Virtual laboratories offer new way to students’ participation and interaction in inquiry-based classes. In such classes, students can perform their own experiments, learn from anywhere (by using virtual objects). As another opportunity, multimedia libraries can enhance standard lecture presentations, with accompanying explanations in both textual and audio forms (using, for example, Pedagogical agents – see, the next sub-section). With the help of IoT, the audio form, in general, can increase the accessibility of the educational resources to the visually impaired, while the textual and visual parts
of educational resources will make it more accessible to the hearing impaired students. The utilization of speech-enabled mobile applications, for example, helps students with reading related disabilities to access educational resources and assists students with writing difficulties to finish their writing tasks. IoE-enabled devices and technology help students to access their courses at any time, from anywhere in the manner most appropriate to them.

A very challenging issue, in this area that was recently taken into consideration is the appropriate capture, stimulation and use of human senses. It is regarded as one of the prominent practices of the educational process. Multi-sensory instruction is described as teaching that involves all the senses: seeing, hearing, tasting, touching and smelling [Aleksandra Klašnja-Milićević et al 2018a]. Multi-sensory instructions, together with abovementioned innovative approaches, enhanced by augmented reality technology, offer highly promising elements that can obtain new functionalities in virtual learning environments for students with disabilities. For example, NEWTON project [Lynch and Ghergulescu 2018] encompasses augmented reality with personalized learning and virtual reality with gamification and emphasis on developing virtual laboratories tailored to the specific needs of students with disabilities.

2.2 Role of Pedagogical Agents in E-Learning Environments

Research in Inclusive Education area show that the presence of SwD gives non-SwD students new kinds of learning opportunities [Savin-Baden 2015], [Savin-Baden et al. 2019]. In such learning organization, one significant opportunity occurs when non-SwD serve as peer-coaches. In fact, by trying to help another student, the helper peer can improve his/her own performances (see, also [Paprzycki and Vidakovic 1993] and references collected there).

Another challenge is connected to the teachers’ duties. To take care of their more diverse audience, including students with disabilities, teachers must be able to provide instruction in a wider range of learning modalities (visual, auditory, and kinesthetic) bringing benefits also to their non-SwD. This task is definitely not easy for majority of teachers. However, using contemporary technological advancements this problem could be successfully solved in near future using intelligent software agents.

Autonomous, intelligent software agents, used in a learning context, are usually known as Pedagogical Agents. Their aim is to support learners across a wide range of subjects. Pedagogical agents are especially valuable to guide students through multimedia, multimodal learning environments, by exploring their motivations and by assessing the learning effects and outcomes. The use of Pedagogical Agents ranges from supplementing existing human-driven instruction with expert features, to entirely replacing human teachers. “What is required is the use of such agents in places of widening access, increasing diversity, and spaces that work against standardized models of learning,” [Savin-Baden et al. 2019]. In the paper [Rickel et al. 2002] authors presented Autonomous (Pedagogical) Agent as a kind of software that conveniently interacts with the user, possibly using natural language, in form of: conversation, coaching to achieve solution of particular task or posing questions to assess acquired knowledge. The desirable form of these agents is to be realized as virtual visual assistants. Additionally, Pedagogical Agents have been found to improve motivation and reduce cognitive load among the students [Bowman 2012], [Ivanović et al. 2015].

Pedagogical Agents are used for learning purposes in different domains and courses. However, in the context of this paper, our intention is to present their advantages in STEM domains, with emphasis on students with disabilities. For this purpose, interesting experiments were performed in subjects spanning STEM with the AutoTutor system [Graesser et al. 2014], and for learning programming with the PROTUS system [Ivanović et al. 2015]. In these systems, students have to gain adequate knowledge and develop their problem-solving skills. Pedagogical Agents used in these systems are aimed at providing students with a virtual tutor that can respond intelligently to their
inquiries and that has the ability to emulate the teacher by offering students immediate feedback and hints, thus helping them to improve their knowledge and skills.

Virtual Pedagogical Agents/Tutors raise difficult challenges and offer great opportunities to be faced and exploited in virtual e-learning environments and laboratories that provide technologically supported inclusive education. One of very important aspects, during learning for students with disabilities, is attitude – a complex psychological concept that characterizes the mental and/or emotional state of a person [Perloff 2016]. Therefore, the use of Pedagogical Agent as a motivator that demonstrates positive attitudes towards the task and the desired levels of performance, helps students to cope with situations where they feel as novices or with some level of anxiety.

One-to-one communication between student and system is valuable, especially for students with disabilities, and it can help enormously in acquiring curricular knowledge and skills. However, other personal diversities (like social and emotional) also play important role in multiple educational activities. Moreover, recent approaches that target the realization of mixed group interactions and conversations between agent(s) and students are getting more and more attractive for students with disabilities. For example, a notable approach presented in [Graesser et al. 2014] concentrates on triologues, i.e. some simple way of group conversations. In fact “The incremental value of multiple agents is that the student can learn by observing how the agents interact. A student can learn vicariously by observing one agent communicating with another agent, showing how actions are performed, and reasoning collaboratively with the other agent.”[Graesser et al. 2014]. This paper highlights situations where two agents can behave in different circumstances such as: disagreement, contradiction, and holding an argument, thus providing the students' with the opportunity to face different situations that can appear in real classrooms.

This very interesting approach can represent an excellent starting point for further investigation in this area, aiming at bringing useful consequences for students with disabilities, in supporting their social and emotional aspects of learning. The development of future e-learning environments must investigate if such virtual multi-conversational Pedagogical Agents actually encourage or discourage the development of problem solving, reasoning, and, in-depth learning of students with disabilities. Future investigations and experiments also have to include different configurations of students in both situations: (1) real classroom with blended learning style or in (2) Virtual Environments, where the use of chat forums is highly encouraged in both situations. For example, it will be interesting to consider the following learning scenarios in which mixtures of both types of students (SwD and non-SwD) are included in the same group:

- **Scenario1**: Real classroom with a blended style of teaching/learning with the possibility to use chat forums.
  - group of several students (including students with disabilities) communicate with one Pedagogical Agent;
  - one student with disabilities communicates with several Pedagogical Agents;
  - group of several students (including students with disabilities) communicate with several Pedagogical Agents;

- **Scenario2**: Virtual Environment with direct communication or by using chat forums:
  - group of students can communicate with one Pedagogical Agent
  - group of students can communicate with several Pedagogical Agents.

Students can either communicate directly or exchange messages and then expect the opinions and suggestions of their peers, as well as of Pedagogical Agents.

Such scenarios and experiments, with different mixed groups of students, could bring valuable insights into the pedagogical, methodological and motivational aspects of inclusive education.
supported by contemporary technologies. However, currently this task is not easy, in spite the fact that different frameworks for communication using natural languages offer great opportunities, like for example Amazon Alexa and IBM Watson.

3 MEASURING LEARNING SUCCESS – STUDENTS WITH DISABILITIES

Measuring students’ learning success is one of key activities in all educational processes. One can find a lot of different definitions (on the Internet, in books, and research papers) of learning success. Some of them are rather complex and comprehensive, so in the paper we will concentrate only on several key aspects important for students with disabilities. We focus on learning success that considers the level and quality of acquired curricular knowledge, including social aspects i.e. interaction and collaboration between students and teacher (real or virtual). Diverse measures of learning success have been developed constantly, with main motivation to help in improving methods, pedagogies and adjust learning activities to specific and particular students' needs [Davis 1993], [Lorenzo et al. 2013], [Byers 2017]. These measures are oriented towards obtaining subjective, or objective, evaluation of educational practices. Triggered by the contemporary necessity to take special care of inclusive education, existing measures have to be re-considered and re-evaluated under new circumstances, such that updated, as well as novel instruments must be proposed.

Modern trends and technologies like educational data mining (EDM) and learning analytics (LA) offer instruments to answer increasingly important, but very complex, questions: what is the current student knowledge level and whether a student is actively engaged in the learning process together with her peers. Scientists, from different disciplines, connected to educational processes have actively considered and experimented with new techniques, based on machine learning and data mining from system-generated data that have shown promise for predicting students’ learning achievements and outcomes. They analyzed students’ behavior in learning environments, trying to recognize their different learning patterns, for possible later use in predicting further students’ learning activities and achievements, in order to increase quality of learning [Klašnja-Miličević and Ivanović 2018b].

3.1 Measures for Learning Success of Students with Disabilities

In this sub-section we will consider some, possible, “general-purpose/standard” measures that can contribute to the advancement of technology enhanced learning of students with disabilities.

**MEASURING LEARNING SUCCESS - CASE 1:** Rather than a standard way to measure learning success and effects of learning, considering 3 specific learning situations is presented in [Savin-Baden et al. 2019]. Authors considered students with different abilities and background and they allocated them into one of three conditions. First group consisted of students that used support of Pedagogical Agent. Second group consisted of students that used on-line teaching material. Third group consisted of students that have traditional face-to-face session. Qualitative data was collected through semi-structured interviews, while quantitative data was collected through both objective (target subject attainment) and subjective (technology acceptance and learning approaches) self-reporting measures. Technology Acceptance Model form (TAM) [Davis 1993] [Lorenzo et al. 2013] was used to assess the usability and perceived usefulness of the Pedagogical Agent. The ASSIST Questionnaire form [Tait et al. 1998] was used to evaluate students’ learning approach, i.e. to check if the approach with the agent engagement is more effective. Analysis and final drawn conclusions showed that, at the moment, students from the three groups prefer more the Online and traditional F2F approach as compared to the Pedagogical Agent approach. Specifically, “for the Pedagogic Agent groups, scores on the measure of technology assessment, the TAM, were highest for computer playfulness and lowest for computer anxiety. For the Online group, scores were highest for the perception of external control and also lowest for computer anxiety.”
As Pedagogical Agents can offer additional methodological advancements and support in variety of educational settings, they must be carefully considered and especially used in inclusive education. Accordingly, we can suggest and conclude: to use Pedagogical Agents in inclusive education, students need to be supported in understanding their preferred learning strategies, as well as to be able to build on individual self efficacy to promote more effective engagement.

MEASURING LEARNING SUCCESS - CASE 2: In this case, a rather innovative approach of measuring students’ learning success is discussed. A specific kind of learning observation metric entitled Linking Pedagogy, Technology and Space (LPTS) was developed by Terry Byers [Byers 2017]. It supports “real-time empirical evidence of spatial interventions by teachers through their practices and subsequent impact on students.” This, rather comprehensive, measure covers five aspects of learning: Pedagogy; Learning Experiences; Communities of Learning; and Student and Teacher Use of Technology. The measure was designed to determine the duration of each activity and its behaviors associated with 36 indicators. The time spent in each activity and associated behaviors are recorded as they occur during the learning process [Patricia and Neuza 2018].

For Learning Experiences, the following indicators are assigned: formative assessment, receive instruction, remember/recall, understand, apply, analyze, evaluate, creation/practical activity, students disengaged. For Communities of Learning, the following indicators are assigned: individual, small groups (the same number), whole class, mixed groups (different numbers), mixed-class/year-level. For Student and Teacher Use of Technology, the following indicators are assigned and they are the same for both participants: mode1: teacher-centered, mode2: student-centered, mode3: informal, outside classroom, substitution, augmentation, modification, redefinition, pen and paper, tablet/laptop (typing), tablet/laptop (touch or stylus), front data projector, additional visual display/screens, whiteboards (writeable walls), camera or recording equipment, equipment or tools. Obviously, it is possible to find indicators in each category that could be considered for use in inclusive education. However, because of limited space of the paper, from the point of view of inclusive education and success of students with disabilities, we focused here only on the Pedagogy Aspect of the measure. This aspect includes the following indicators [Patrícia and Neuza 2018]:”

- **Didactic Instruction**- when the teacher is engaged in presenting/disseminating content, concepts or information to students through a didactic/direct instruction mode;
- **Interactive Instruction**- when the teacher is engaged in demonstrating a process/ability or skill through an interactive/dynamic instruction mode (using equipment and/or tools through a series of interactive steps);
- **Facilitating**- when the teacher is moving about the room to observe/monitor/regulate students’ progress and behaviors;
- **Providing Feedback** - when the teacher provides feedback (advice, direction or suggestions) on an individual, pair or small groups progress in a particular learning activity;
- **Class Discussion**- when the teacher promotes the instruction/discussion with the students/between the class to provide input to a particular topic of discussion that they or the whole class are participating in; when students interact/discuss with each other
- **Questioning**- when the teacher asks the student(s) (individual, pair, small groups, whole class) to answer or respond to either closed or open questions about the thematic contents/activities.”

Original definitions of indicators have to be adapted depending on specific characteristics of experiments and educational circumstances. Concerning the role of Pedagogical Agents and the possibilities to use them in different communication scenarios, during learning processes of students with disabilities, the last three indicators can play a significant role from our point of view.

The social aspect of learning is generally very important, especially for majority of STEM disciplines. It seems additionally valuable for students with disabilities. Concerning the indicators
mentioned above: Providing Feedback, Class Discussion and Questioning, the value of student-teacher dialogue for students with disabilities has to be considered with special attention. Concerning Scenarios proposed in Section 3.1, all explanations of Pedagogical Agents depend on two basic processes that students must be engaged in, especially when on-line discussions are in use:

- “speaking”: externalizing ideas/opinions by posting messages to the discussion forum;
- “listening”: consuming the externalizations of others by accessing existing posts.

Speaking, in on-line discussions, is visible to other participants, while listening is invisible, and it is in fact, the critical issue for discussions here. The different kinds of listening behaviors represent step ahead in productive use of discussion forums. It is important to motivate students to be actively engaged and support better connections in student-student and student-teacher listening and speaking behaviors [Wise et al. 2013]. Pedagogical Agents engaged in virtual learning environments can highly motivate students to actively participate and regulate/decide how they will speak and listen in online discussions. Having appropriate communication skills, Pedagogical Agents can positively influence students with disabilities for their active and productive participation in Class Discussion and Questioning.

Analysis of data collected from discussion forums (especially if we use educational data mining and learning analytics) could be extremely useful in Providing Feedback from Pedagogical Agents and tailoring and personalizing actions suggested for each individual student with disabilities.

For achieving this constructive supervision and tutoring of Pedagogical Agents, some basic measures can be considered in data collection phase like: Range of participation, Number of sessions, Average session length, Number of sessions with posts, Number of posts made, Average post length, Number of posts read, Number of reviews of own posts, Number of reviews of other's posts.

Additionally, as interaction/conversation has visual and audio nature, we can valorize the additional power in the identification of frequently used words/phrases and the basic elements of text and speech analysis that can significantly improve the personalized feedback provided by Pedagogical Agents.

3.2 Measures for Empowering Students’ Interaction and Motivation

Different methodologies for qualitative and quantitative measurement of learning success are necessary in order to increase the quality of learning. They are especially important in technology enhanced learning and, in particular, when employing Pedagogical Agents in Virtual Environments/classrooms. Recent investigations in the area of inclusive education show that, when mixing both students with and without disabilities, both groups have the opportunity to learn more. Many studies carried out over the past three decades have found that students with disabilities obtain higher achievement and improved skills through inclusive education, while their peers without challenges can benefit, as well [Bui et al. 2010], [Alquraini and Dianne Gut 2012].

Adequate measures (regardless if they are objective or subjective) must be used, to help in empowering personal communication, to provide better recommendations of appropriate educational resources and to increase the motivation of students. To summarize previously presented possibilities to measure students learning success, we can suggest several possible main domains for measuring learning success of students with disabilities in virtual learning environments.

**Measure1: Learner-centeredness**—provides the students with the opportunity to actively participate in the teaching and learning process; supports learning; students are regarded as contributors to their own learning [Makoelle 2014].

**Measure2: Learning preferences**—students’ learning approaches and preferences highly affect their engagement with the Pedagogical Agent(s); such preferences influence the process of teaching material tailoring and personalizing, in order to serve the special needs of students with disabilities.
**Measure 3: Virtual interaction**—these measures should offer opinions and suggestions for further learning steps given by Pedagogical Agent and they are crucial for participation of students with disabilities in virtual classrooms. For SwD, additional important measures should be oriented towards assessment of visual appearance and interaction with Pedagogical Agent like:

- **General characteristic** - age, gender, clothing, weight, etc.;
- **Quality of voice** - high, medium, treble, etc.;
- **Emotion expressions** - compassionate, pleasant, strict, polite, etc.;
- **Additional emotional factors** - boredom, pride, pleasure, shame, etc.;
- **Monitoring and directing motivation** - arouse interest, highlight the relevance of the topic, strengthen the student’s confidence, etc.;
- **Capabilities** - behave as expert (strict), motivator (friendly), and mentor (supportive);
- **Human vs. non-human characters** - appearing to be as static or animated;
- **Communication mode** - students can freely choose if and when to chat with their Pedagogical Agents.

**Measure 4: Team dynamics**—Teamwork is an important way of organizing manpower in activities leading to producing solutions, in majority of STEM disciplines. Grouping students in Virtual learning environments is an additional motivational factor for all group members regardless of how diverse they are.

**Measure 5: Cognitive abilities**—characteristics of the students who interact with the Pedagogical Agent(s) include several cognitive factors like: prior knowledge, ability to integrate the new information into the existing cognitive structure, ability to share knowledge and so on.

**Measure 6: Information processing**—provides explicit information about prerequisites, conditions, relationships or outcomes of the learning content, enables students to decompose new information into smaller units, synthesizes them and is able to extract similarities and differences.

**Measure 7: Transfer of information**—the ability of students to apply the new knowledge, to transfer it to other topics, and to use it for solving new problems.

Contemporary learning, usually represents a challenging and unique mash-up of home-school-work-media-peer-collaboration in both real and virtual classrooms/environments. It includes also the following significant practices (based on [Savin-Baden 2015]) that are applicable to students with disabilities, granting them equal opportunity in educational processes:

- **Mentorship**—using mobile devices to keep in touch with various educational players through different means of communication including ubiquitous social media.
- **Co-operative online learning**—cooperation with peers and virtual agents to guide and support completing homework, assignments, tests. Similar measures as abovementioned could be applied here.
- **Gaming**—isolated or combined in order to share, teach, learn, offer advice, negotiate, and give and receive hints, tips and solutions.
- **Teaching technology**—teaching and sharing experiences with peers and virtual agents about applications, services, new devices, and helpful sources of information.
- **Emotional learning**—using digital media for peer to peer support to manage personal issues and difficulties, and to receive hints and advice.
- **Playful learning**—trying things out and fiddling around, in order to experiment and discover.

All these practices additionally attract research community attention, influence further research and raise a lot of interesting research questions in order to find and propose adequate new measures.
or evaluate and adjust existing measures to meet requirements of inclusive education and higher learning effects of students with disabilities.

In this position paper we pointed out initial considerations and proposals for establishing measures for virtual learning environments with Pedagogical Agents that can help students with disabilities to achieve better learning success and interaction with peers and teachers (real and/or virtual).

4 CONCLUDING REMARKS

Continuous and rapid technological advancement essentially changes our traditional perceptions of education. Numerous emergent technologies appear “on the monthly basis”. Impressive growth of availability of IoT and IoE smart devices with sensing / actuating capabilities and applications that can take advantage of them bring enormous potential in education and can significantly change its pedagogical and methodological aspects. They are excellent facilitators for contextual, personalized and seamless learning in smart environments, suitable for students with disabilities.

Having SwD and non-SwD in the same classrooms (real and/or virtual) is challenging for the wide range of educational stakeholders. Henceforth, investment in designing and establishing appropriate success measures of such systems can benefit students learning success and thus is an important task. Moreover, active participation of students in these efforts is crucial for full success. Involving students, asking for their opinion and suggestions is unavoidable and highly relevant. Brookfield and Preskill [Brookfield and Preskill 2012] suggested and interesting approach and method for helping students to create their own ground rules. These suggestions also can represent a good starting point in preparation of specific measures of virtual learning environments (based on Pedagogical Agents) for students with disabilities, allowing them to assume a more active role towards increasing their learning success.

Acknowledgement. This paper is a part of the Serbia-Romania-Poland collaboration within multilateral agreement on “Agent systems and applications” and Romania-Poland collaboration within bilateral project “Semantic foundation of the Internet of Things”.

REFERENCES


Aleksandra Klašnja-Miličević, Boban Vesin, Mirjana Ivanovic, Zoran Budimac. (2011). E-Learning personalization based on
hybrid recommendation strategy and learning style identification. Computers & Education 56(3): 885-899
Terracina Annalisa, Berta Riccardo, Bordini F., Damilano R., Mecella Massimo. (2016). Teaching STEM through a Role-Playing Serious Game and Intelligent Pedagogical Agents, ICALT 2016: 148-152
Alyssa Friend Wise, Yuting Zhao, Simone Nicole Hausknacht. (2013). Learning Analytics for Online Discussions: A Pedagogical Model for Intervention with Embedded and Extracted Analytics, LAK ’13, Leuven, Belgium
Dynamic Testing of Executable UML Models with Sequence Diagrams

TAMÁS JÁNOSI, KRISZTIÁN MÓZSI, PÉTER BERECZKY, DÁVID J. NÉMETH and TIBOR GREGORICS, Eötvös Loránd University

Executable UML makes it possible to define high-level models of software systems, which then can be inspected independently of the target platform or translated to efficient platform-specific code. This not only provides early validation, but enables modeling to even be considered an additional layer of abstraction over programs written in a general-purpose language. In both cases, reasoning about the properties of constructed models plays an important role in the development process.

To advance this area, in our paper we propose a framework supporting the definition and evaluation of sequence diagram-based test cases for executable UML models. As part of this, we introduce a textual language capable of describing assertions about the communication in and between, and the state of, system components. We also discuss a test evaluation strategy characterized by the idea of executing sequence diagrams and models alternately, and show how arising challenges like synchronization or the quantification of possible divergence can be tackled. Finally, we illustrate the feasibility of our approach by presenting our prototype implementation, embedded into an open-source xUML modeling toolchain.

1. INTRODUCTION

As software systems gradually grow both in size and complexity, their design, development and maintenance become increasingly difficult tasks. Although modern general-purpose programming languages offer a rather high level of abstraction, the structure and behavior of large-scale architectures can still be hard to manage in conventional source code. One way to address this issue is the use of UML [Object Management Group 2017], a language which enables us to model several aspects of a system in the form of high-level diagrams. With the help of dedicated tooling, appropriately defined, potentially platform-independent models can also be executed directly or translated to runnable platform-specific representations. This is the essence of executable UML (xUML) modeling [Mellor and Balcer 2002] – a method for early functional validation, or even a paradigm for developing software on a higher layer of abstraction.

Whichever aim of xUML we consider, the ability to reason about properties of constructed models is of great importance. Static analysis is an emerging possibility, however, in this paper we focus on monitoring the runtime behavior by dynamic testing. To realize this, one could observe the interpreted model or the generated platform-specific code in a black-box fashion. Yet we argue that formulating test cases as part of, and on the same layer with, defined models has the advantage of keeping the specification and implementation coupled together closely enough for simplified comparison and main-
Tenance. Inspecting the feature set of UML, sequence diagrams appear to be a suitable candidate for expressing expected interaction in and between components. Regarding the evaluation of described test cases, we optimize for traceability and interactivity. To achieve these goals, our solution executes sequence diagrams directly and in parallel with models under test. Building upon these ideas, in the following we propose a framework supporting the definition and evaluation of sequence diagram-based test cases for executable UML models.

The main contributions of this paper are the following:

—a textual language for defining sequence diagram-based test cases for executable UML models,
—a methodology for evaluating these tests by executing sequence diagrams and inspected models synchronously,
—a prototype implementation of the proposed framework embedded into an open-source xUML modeling toolchain.

The rest of the paper is structured in the following way. As introduction, Section 2 summarizes related work. In Section 3 we present general decision concerns of the proposed framework, including the main ideas behind the testing language and its intended evaluation strategy. The latter are discussed in detail in Section 4 and Section 5, respectively. Section 6 introduces our prototype implementation, demonstrating the feasibility of our approach. Finally, Section 7 briefly evaluates our results and Section 8 concludes.

2. RELATED WORK

Model-based testing (MBT) [Utting et al. 2012] builds upon the idea that test cases can be generated automatically from abstract, simplified representations of the system under test, which encode its intended behavior. When the system to be tested is a UML model itself, for example interaction diagrams can be seen as the higher-level specification descriptions MBT promotes. This approach helps keep the system and its test cases close, however, the abstractionsal difference between tested systems and generated test cases makes traceability challenging.

Executable UML models can also be evaluated by manually examining their behavior during direct execution, a method called model debugging. Two xUML modeling tools providing this feature are BridgePoint [One Fact 2012] and txtUML [Dévai et al. 2018], where the latter even offers a visual debugger which highlights transitions in state machines during execution. One great disadvantage of this approach is that instead of automatically evaluating predefined test cases, the model has to be actively monitored while it is running.

In conclusion regarding xUML testing, currently we have to choose between well-defined test cases (MBT) and interactive traceability (model debugging). A new approach would be desirable to combine the advantages of the two discussed methods. In the following we show how this can be achieved with executable sequence diagrams.

3. METHODOLOGY

Our approach is to create executable sequence diagrams to test existing models written with an xUML tool. In this way, users could test their programs by expressing message interactions during a specific execution. This could eliminate the need for writing test programs, because the sequence diagrams themselves would serve as test cases. The level of abstraction of these test cases would be the same as of the models'.
3.1 Abstractions required to construct meaningful test cases

To achieve the abovementioned goal, a textual domain-specific language is needed to provide a way to define and execute sequence diagrams. In this language, there should be a language element which makes it possible to create the lifelines of the given diagram. Each lifeline will represent a model element. As we also want to draw sequence diagrams from the written format, we need to provide a way for users to manipulate the drawn position of the lifelines inside the textual notation.

Before executing the sequence diagram, some initializing steps are needed. Model elements have to be created and users have to define explicitly which model element is represented by each lifeline. This means that each sequence diagram needs an initializing part, which will be performed before we execute the sequence diagram itself.

During the execution, we want to be capable of investigating message exchanges and state assertions. Users should be able to define the order of sent messages during an execution with the sources and the targets of the messages. Combined fragments from the UML standard can be used in the sequence diagrams too, and messages can be defined in these fragments.

State assertions are not part of the UML standard, but as our model elements in an executable UML tool are state machines, we wanted to provide a way to check the state of a given model element at a given point of the execution.

3.2 Advantages of making the test description language textual

Textual descriptions have numerous advantages over graphical approaches [Grönniger et al. 2007]. In a graphical solution, models are usually persisted in a format that is hard or impossible to edit directly. Experience shows that the existing tools still need to evolve a lot. Furthermore, while graphical notations help understand software more than textual representations, editing graphics is usually less productive than editing text. At the same time, many high-quality text editors with editing and search-related features, as well as merge and compare tools exist, which can help the user during the development process. Moreover, graphical diagrams can be generated easily from textual descriptions to visualize the defined diagram.

3.3 Evaluating test cases

To test a model with sequence diagrams, we have to execute both the model and the diagram. To be able to test a model based on message exchanges, the execution strategy of the host xUML tool has to be message-driven. When running a sequence diagram, a given model will be executed. Each time a message exchange happens in the executed model, we have to check whether it is equal to the next exchange in the sequence diagram. Two exchanges are said to be equal if their sources and targets are respectively the same and their included messages are equal as well.

4. THE TEST DESCRIPTION LANGUAGE

As it was mentioned earlier, to create sequence diagrams the language has to provide a way to declare the lifelines of the diagram. This can be achieved with the following syntax:

```
lifeline <identifier> <positive number>
```

The positive number is an optional part of the declaration. It defines the order of the lifelines on the diagram generated from the textual description – therefore positions will be ignored during the execution.
There should be an initialization part too, where model elements can be created and linked with the lifelines. While the syntax of the model element creation is based on the host xUML tool, connecting the elements with the lifelines should look like the following:

\[
\text{<lifeline> = createLifeline(<model element instance>)}
\]

The lifeline will represent the given model element instance during the execution.

After the initialization, the execution itself can be described. Message exchange expectations have two different syntaxes, because we differentiate messages between model elements and between an actor and a model element.

\[
\text{assertSend(<lifeline>, <message>, <lifeline>)}
\]
\[
\text{fromActor(<message>, <lifeline>)}
\]

In the first case we have to define the source, the message and the target. In the second case the message comes from an actor, so the source does not have to be specified. If a message exchange happens during the model execution, we try to match the source, the message and the target with the currently expected message in the diagram.

State assertions can also be made with the help of a lifeline and a state as parameters.

\[
\text{assertState(<lifeline>, <state>)}
\]

If the sequence diagram execution starts with state assertions, the parameter model element instances have to be in the given states before the execution. Whereas if a state assertion is placed after an expected message exchange, the model element has to be in the given state after the exchange happens.

Sequence diagrams may also contain combined fragments. Currently, our language supports only the \text{loop}, \text{opt}, \text{alt} and \text{par} fragments from the UML standard. The \text{loop}, \text{opt} and \text{alt} fragments can be defined with conventional loop and conditional statements, and they can contain further message exchange expectations, state assertions and combined fragments. The \text{par} combined fragment can be used with the following language element to express parallel execution of child sequence diagrams.

\[
\text{par(<sequence diagram>+)}
\]

By using the language defined above, not all models could be tested. Model element instances should be observable from outside the model, as their references are needed to connect them with lifelines. This is not always feasible, since some references may only be accessible from inside the model. Our idea is to introduce dummy objects, called proxies to substitute such instances. Proxies can be thought of as special lifelines, as they behave quite similar. The language is extended with proxy creation, where the only information needed for this is the type of the object to be substituted.

\[
\text{<proxy> = createProxy(<type of model element>)}
\]

4.1 Example diagram

Let us demonstrate our language with an example model. The model has two elements: the factory and the customer. If a customer gets an \text{Order} message from the actor, it orders from the factory by sending it a \text{DoOrder} message. If the factory gets a \text{DoOrder} message and the number of orders reached a limit, it starts working until it fulfils all orders. It has a counter which indicates the number of the active orders. Every time an order is ready, it sends an \text{OrderReady} message to the customer who gave the order. This model can be tested for example with the following sequence diagram.
Dynamic Testing of Executable UML Models with Sequence Diagrams

seqdiag factorySequenceDiagram {
  lifeline factory 1;
  lifeline customer1 2;
  lifeline customer2 3;
  lifeline customer3 4;

  initialize {
    Factory f = createFactory(3);
    Customer c1 = createCustomer();
    Customer c2 = createCustomer();
    Customer c3 = createCustomer();

    factory = createLifeline(f);
    customer1 = createLifeline(c1);
    customer2 = createLifeline(c2);
    customer3 = createLifeline(c3);
  }

  run {
    assertState(factory, WaitingForOrder);
    fromActor(Order, customer1)
    assertSend(customer1, DoOrder, factory);
    fromActor(Order, customer2)
    assertSend(customer2, DoOrder, factory);
    fromActor(Order, customer3)
    assertSend(customer3, DoOrder, factory);
    assertState(factory, Working);
    assertSend(factory, OrderReady, customer1);
    assertSend(factory, OrderReady, customer2);
    assertSend(factory, OrderReady, customer3);
  }
}

Fig. 1. Example textual sequence diagram describing a test case for a producer-consumer model

Fig. 2. Graphical sequence diagram corresponding to the textual example (generated with PlantUML [Arnaud Roques 2009])
5. EVALUATION STRATEGY

One of the problems which have to be solved to create executable sequence diagrams is synchronization. When running a sequence diagram, we have to execute the model which is under test, and the sequence diagram which tests the given model. The reason behind this is that we want to be able to check the state of model elements for state assertions and guards of the interaction operators. This can be achieved either by saving the states of the model elements after every message exchange during the model execution; or by stopping the execution of the model at certain points to run a part of the sequence diagram. The latter is much more efficient if we have a complex model and a long execution, so we chose this solution.

5.1 Synchronization

The execution of a sequence diagram must start with initializing the needed elements, as it was described in Section 3.1. After that, we have to execute the main part of the diagram, which describes the expected exchanges, until we arrive at a message exchange. If the exchange is between an actor and a model element, the message has to be sent before we can check if the exchange happened during the model execution. At this point, we have to stop the execution of the sequence diagram and start executing the model. This is because the model has to run to send messages which we can test in our sequence diagram. Here is where the most difficult question regarding synchronization arose. When do we stop the execution of the model and continue executing the sequence diagram?

We came to the conclusion that we have two options. The model execution can be stopped after a message is sent – which we call sending semantics – or after a message is processed – which we call processing semantics. Both solutions has its advantages and disadvantages. Synchronizing after message sending is probably more intuitive, but this way we can only check if the given message was sent – we do not know whether it was processed or not. This is problematic because if a message is sent during the execution, most of the time we want that message to be processed by the target. Furthermore, with this solution if we want to write a condition in the sequence diagram which checks the number of the orders in our previous example, we have to know whether the counter is increased before or after an OrderReady message was sent by the factory. If the order of events changes here, the sequence diagram has to be altered too.

Synchronizing after processing a signal is not so intuitive, and writing conditions can be tricky. The condition has to reflect to the state after the signal was processed. On the other hand, if the behavior of our model does not change, the sequence diagram does not need to be altered either, as opposed to the previously discussed solution. Last but not least, if we synchronize after processing a signal, the message processing can also be checked in addition to the sending, which is a huge advantage.

We concluded that stopping the model execution after processing a message has more significant advantages than synchronizing after message sending. Therefore, this is our chosen solution for the described synchronization problem.

5.2 Error counting

Counting errors during the execution is not a trivial problem either. Diagrams can be executed with lenient and strict execution mode and both of them require a different error counting logic. Lenient execution mode means that a test passes if there is a subsequence of the actual exchanges which matches the diagram. Additional messages can be sent, and they are not counted as an error. The test fails only if there is at least one message exchange in the diagram which did not happen during the execution, or there is a state assertion which is not correct. In strict execution mode the test passes only if the diagram describes exactly the same exchanges that happened during the model execution.
In lenient mode the error counting problem is simple. We have to execute the model, and when a message exchanges happens, we check the next exchange in our diagram. If the two exchanges are not equal, we keep checking the same exchange from the diagram after every message in the model execution, until we get a match. Then we can continue to execute the rest of the diagram. After the execution terminates, every expected message exchange which did not happen during the model execution is an error.

In strict execution mode counting errors is more complicated. If an actual exchange is not equal to the next exchange in the diagram, should we proceed with the sequence diagram execution – which we call \textit{continuous semantics} –; or keep trying to match the missed message exchange from our diagram until it occurs in the model – which we call \textit{recurrent semantics}.

First consider continuous semantics. If we encounter a mismatching pair of expected and actual message exchanges, but after that all remaining exchanges match, only one error will be shown. This is desirable. However, if we only \textit{miss} (and not \textit{mismatch}) a message exchange from our test case, and we proceed with the execution, every remaining exchange processed during the execution will raise a new error. Here, it would be more intuitive to report only one.

Now consider recurrent semantics. If the only problem is that actual exchanges were left out of the diagram, the number of reported errors will be the same as the number of missed messages. This is an advantage compared to the other possible solution. However, in cases where the diagram contains an exchange which does not occur in the model, we will keep trying to check the same expected exchange until the model execution terminates. With this semantics there is also another open question: should we count errors individually or group them between matching exchanges? In the first case it can happen that we have a diagram with one expected message exchange and we get for example twenty errors after the execution, which may be considered unexpected.

This problem is still under discussion, as the most suitable solution seems to be different for every diagram and model. Currently, we use recurrent semantics and we raise a new error for every mismatch. This may change in the future.

5.3 Proxy objects

Introducing the concept of proxies brings additional questions about the execution. First, in some cases during test execution it is not trivial to determine which proxy substitutes which instance. Another question is, what are the similarities and differences between concrete and proxy objects, as it would be desirable to handle them as similarly as possible. Limitations and effects should be considered as well.

As proxy objects and concrete model objects are considerably similar, let us generalize and call them lifelines. The only difference between them is that in case of model objects, it is known initially which instance is denoted, while with proxies only the type of the object is known. Therefore proxies do not specify a concrete instance when they are created.

To determine the concrete instance denoted by a proxy, the concept of binding should be introduced. Binding can only happen during message comparison, thus it should be defined how to introduce proxies into the existing comparison method of expected and actual communication. Proxies without concrete meaning may be encountered as senders or targets of expected messages. If both sender and target is an unbound proxy, furthermore types of these proxies match with the types of the actual objects, binding happens for both sender and target. If only one proxy is bound and the types are matching, one binding happens. Otherwise expected and actual exchanges are not matching. If both participants are bound proxies, the original matching method is appropriate, as they behave similarly to concrete objects. Note that message sending assertions with unbound proxy participants not only
assert communication, but a binding might also happen in the background. This concept is hidden from the sequence diagram code.

Let us consider corner cases. If a proxy is created to substitute an object that can be referenced from sequence diagram code, the proxy behaves as an alias after it is bound to the object. Two or more distinct proxy objects can be bound to the same concrete object as well. In this case, similarly to the previous one, the concrete object will be accessible via multiple different proxies.

6. PROTOTYPE IMPLEMENTATION

We created an implementation of our language in the txtUML tool. It is an open-source textual, executable and translatable UML tool, implemented in Java. We chose txtUML as the host of our prototype because it is open-source; fulfills the criteria of message driven execution; and its model executor was created in a way that only minor changes are needed to implement the synchronization of two executor threads. The latter makes the technical part of integrating the capability to execute sequence diagrams into the tool relatively easy. Furthermore, unlike in most xUML tools, users can create models in a textual representation instead of a graphical one. This fits into our concept that sequence diagrams should be defined in text because of the advantages mentioned in Section 3.2.

```java
public class FactorySequenceDiagram extends SequenceDiagram {
    @Position(1) Lifeline<Factory> factory;
    @Position(2) Lifeline<Customer> customer1;
    // ...

    @Override
    public void initialize() {
        Factory f = Action.create(Factory.class, 3);
        Customer c1 = Action.create(Customer.class);
        // ...

        factory = Sequence.createLifeline(f);
        customer1 = Sequence.createLifeline(c1);
        // ...
    }

    @Override
    @ExecutionMode(ExecMode.STRICT)
    public void run() {
        assertState(factory, Factory.WaitingForOrder.class);
        fromActor(new Order(), customer1)
        assertSend(customer1, new DoOrder(), factory);
        fromActor(new Order(), customer2)
        assertSend(customer2, new DoOrder(), factory);
        fromActor(new Order(), customer3);
        // ...
    }
}
```

Fig. 3. The previous example diagram embedded in Java (excerpt)
6.1 Embedding the language in Java

Our sequence diagram language prototype is capable of executing and testing txtUML models. As both txtUML and its domain-specific modeling language are embedded in Java, we also created the sequence diagram language in this way. Users can define their own sequence diagrams by extending the provided `SequenceDiagram` class. To specify expected message exchanges, state assertions and `par` fragments, they can use the static methods of the Sequence API.

Lifelines can be declared as attributes of the diagram. Their position on the visualized diagram can be specified with the `@Position` annotation. The initialization and the execution part can be described in the `initialize` and the `run` methods of the diagram, respectively. The execution type (strict or lenient) can be defined with the `@ExecutionMode` annotation on the run method.

6.2 Synchronized execution

Diagrams can be executed with an executor (`SequenceDiagramExecutor`) which runs the model and the sequence diagram on two different threads in a synchronized way. The synchronization is implemented with a `BlockingQueue` which contains the current message in the sequence diagram. After the model processes a signal, we pop the current message from the queue, and the model execution waits until a message is placed into it. The queue contains a message only if the sequence diagram thread is waiting or terminated, indicating that the model executor thread can continue its execution.

6.3 Error counting

First consider the lenient execution mode. If an expected message in the diagram differs from the actual message processed in the model, we keep trying to match subsequent actual messages with the same expected message until the two are equal or the model execution stops. At the end, missed expectations are stored as errors in a list. If the execution mode is strict, an additional error is recorded each time an expected and an actual message differs. After the execution is finished, the executor provides the list with the errors which were found during testing.

7. EVALUATION

txtUML comes with demo projects. We created tests for these projects with the help of our sequence diagram implementation. One of the demos is a producer-consumer model, which is excellent for demonstrating the usage of the `par` combined fragment. Another demo project implements the model of a train with a gearbox: every time a gear is shifted, the train goes into a different state. Here we were able to write a test case with a lot of useful state assertions.

One of the demo projects contains model instances which cannot be accessed from outside the model. This project provided a good opportunity to demonstrate the usefulness of proxy objects: here we were able to create much more meaningful test cases than it would have been possible without proxies.

8. CONCLUSION

In this paper we proposed a novel approach to test xUML models with sequence diagrams. We aimed for the goal of providing a framework which offers model tests that are convenient to write, run, trace and maintain. We presented our methodology consisting of a textual test description language and an evaluation strategy that executes sequence diagrams and models under test synchronously.

Considering the possible realizations of interaction-based testing, we derived abstractions suitable for constructing meaningful test cases. We mainly built these elements upon the UML specification but we also extended it with state assertions and proxy objects. We argued why it is desirable to design the language as text-based and then intuitively defined its syntax with representative code examples.
As for test evaluation, we elaborated two alternatives depending on whether synchronization happens at sending or at processing signals. We showed that the latter has more advantages, which can be considered the informal semantics of our language. We also discussed possible ways to quantify divergence between actual and expected communication in the form of continuous and recurrent semantics. Regarding assertions concerning objects that are not directly accessible in a test case, we presented a binding strategy to realize the previously introduced concept of proxies.

Finally, we demonstrated the feasibility of our approach by elaborating details from our prototype implementation that is integrated into an xUML modeling tool. We presented how the designed language can be embedded in Java using custom classes and annotations; how alternating execution can be realized with standard synchronization features; and how we implemented the formerly discussed error counting methods.

By publishing our framework and its open-source prototype [ELTE-Soft 2019], we intend to make a useful contribution not only for the modeling community, but also for large-scale software development in general.

REFERENCES

Unit and Performance Testing of Scientific Software Using MATLAB®

BOJANA KOTESKA, MONIKA SIMJANOSKA, IVANA JACHEVA, FROSINA KRSTESKA and ANASTAS MISHEV, Ss. Cyril and Methodius University

In this paper we report our activities on performing testing of scientific software for calculating ECG-derived heart rate (HR) and respiratory rate (RR) by using Matlab®. The aim of this software is to aid the triage process in the emergency medicine which is crucial for ranging the priority of the injured victims in mass casualty situations based on the severity of their condition. One challenge of this paper is to perform unit testing in Matlab by using the Input Space Partitioning method. For that purpose, we created sets of test values for the ECG signals and we modified them according to our needs. By using the Profiler tool we tested the performance of the algorithm functions.

1. INTRODUCTION

Scientific software solves problems in various (scientific) fields by applying computational practices [Kelly et al. 2008]. Its multidisciplinary nature makes it more complex, but it provides great opportunities and advantages for scientists in many different scientific fields. Scientists usually have a large amount of data to process [Wilson et al. 2014], many calculations, lots of requests to handle, and automating the process by creating software makes their work easier, increases productivity, quality and sustainability [Wiedemann 2013].

Scientific software is based on models, experimentation and observation of the results [Joppa et al. 2013]. Tests are very much like experiments and the obtained results are observed later. That is how the scientists test their hypotheses. They run experiments, measure results and analyze the data.

Scientific software testing is a hard and challenging task due to the complexity and the lack of test oracles [Kanewala and Chen 2018; Lin et al. 2018]. Challenges are categorized according to the specific testing activities: test case development, producing expected test case output values, test execution, test result interpretation, cultural differences between scientists and the software engineering community, limited understanding of testing process, not applying known testing methods, etc [Kanewala and Bieman 2014].

In this paper, we report our activities on performing testing of a scientific software for calculating ECG-derived heart rate (HR) and respiratory rate (RR) by using Matlab®. We try to identify specific challenges, proposed solutions, and unsolved problems faced when testing scientific software. The aim of this software is to aid the triage process in the emergency medicine which is crucial for ranging the priority of the injured victims in mass casualty situations based on the severity of their condition.
whether they are man-made, natural or hybrid disasters. When performed manually, an efficient triage process takes less than 30 seconds. In order to optimize the process when there are hundreds of injured people, the challenge is to reduce the triage time and the number of medical persons needed. The software we describe in Section 2 extracts heart rate and respiratory rate from ECG signal. This optimization can be achieved by using the benefits of the biosensor technologies to extract the vital signs needed for the triage.

The other sections are organized as follows. Section 3 provides a comprehensive explanation of the testing methodology and results: definition of test cases, testing preparations and execution and analyses of the results of the executed tests. The final Section 4 concludes the paper.

2. DESCRIPTION OF SOFTWARE FOR CALCULATING ECG-DERIVED HR AND RR

The software is developed as a part of the triage procedure for determining a patient’s condition [Gursky and Hrečkovski 2012]. It is designed for low-power wearable biosensor and it uses only an ECG signal to estimate automatically the HR and RR as crucial parts of the primary triage. The goal of this software is to perform efficient real-time processing of ECG data in terms of the power-demanding Bluetooth connection with the biosensor and the data transmission. The accuracy of the algorithm is published in [Simjanoska et al. 2018].

As depicted in Fig. 1, the input of the algorithm is a raw ECG signal. The calculation of the HR is performed by R-peak detection performed by using the the Pan Tompkins algorithm [Pan and Tompkins 1985]. HR is calculated according to the following equation:
The obtained R-peaks are used to estimate the RR. The kurtosis computation technique is used for measuring the peakedness of the signal's distribution. The locations of the local maxima are needed for the smoothing method upon which a peak finder method is applied to find the local maxima (peaks) of the ECG signal. The peaks represent a number of respirations according to which the respiratory rate is calculated:

\[ HR = \left( \frac{\text{signal length}}{\text{number of R peaks}} \right)^{-1} \times 60 \]  

(1)

\[ RR = \left( \frac{\text{signal length}}{\text{number of respirations}} \right)^{-1} \times 60 \]  

(2)

The implementation of the proposed algorithm is realized in Matlab. It has 493 lines of code in total. The software contains two main functions: h3r and pantompkin. The first one, h3r, has two arguments needed for the calculation of the estimated values of RR and HR. The first argument is a raw ECG signal, represented in vector format, which contains an array of decimal values and the second argument is the measurement frequency (number of measurements in a second).

The second function, pantompkin, uses three arguments for calculating the qrs_amp_raw - amplitude of R-waves, qrs_i_raw - the index of R-waves and delay - a number of samples in which the signal is delayed due to the filtering. The arguments used by pantompkin are ECG - raw ECG vector signal, fs - sampling frequency and gr_flag - flag for plotting.

3. METHODOLOGY AND RESULTS

3.1 Unit Testing

h3r and pantompkin functions are the two main software units. Unit testing purpose is to assess the software units produced by the implementation phase. It represents the "lowest" level of testing [Ammann and Offutt 2016]. Unit testing improves the quality of science and engineering software and it focuses on small units of code (class, module, method, or function). In order to have effective unit testing, the procedure should be automated so that entire test suites can be run quickly and easily. The verifying of the individual units is helpful for verifying overall system behavior. The focus on the smaller software part leads to modular and more maintainable code [Eddins 2009].

When testing a scientific software, it is very important to think about the nature of the software and to be familiar with the scientific field. The metrics should be well defined and most often it is easily noticed that they are surreal. For example, a respiratory rate can’t be 1000 breaths per minute. That’s why the testers should know the possible result value ranges, so that they can determine the input domain. The input domain should consists of as much as possible inputs (valid and invalid) that could be taken by the program. The tester should choose test cases wisely since the input values could be infinite. Program correctness can be proved by testing all possible input values, but we can only test limited set of inputs (known as test cases). One method to determine the inputs for a specific variable is to use Input Space Partitioning (ISP) - input or output data is grouped or partitioned into sets of data that we expect to behave similarly and will help us with creating test cases. We used Functionality-based ISP [Ammann and Offutt 2016].

Matlab provides multiple tools for unit testing [The MathWorks, Inc 2019].

h3r function’s first argument is the raw ECG signal vector. According to the database of ECG signals, the usual values are floating points values in range 0-10. The purpose of ISP is to create partitions of the input values and to test the software behavior when the input values are outside the usual range also. For the ECG vector, the following test cases were tested:
—empty vector;
—vector of zeros;
—vector with negative float values;
—vector with positive floating values;
—vector with positive floating values greater than 10;
—vectors with combinations of negative values and 0s;
—vectors with combinations of positive values and 0s;
—vectors with combinations of values (negative, positive and 0).

We used the same strategy to test the other function as well, because both functions use the same type of argument - the raw ECG signal vector. For the frequency parameter and for the gr flag we also made ISP (for the frequency we have the correct value of 125, then 0, negative number and value
greater than 125). For the gr flag we have 0, 1, negative value and value greater than 1. Each one of these checks represents a single test, which can be run separately.

In Matlab it is possible to define unit test suites. We combined all the tests in one file (test suite), and we ran the file once which ran all the tests automatically one by one.

By using the assertion functions from the matlab.unittest.qualifications.Assertable class, we compared the output HR and RR values with the expected ones and if the test passes, that means the assertion is true and the values from the test are the values we’ve been expecting to get.

To make test running easy, Matlab provides function runtests which runs all the tests in the current folder, gathers them into a test suite, runs the test suite and returns the results as a TestResult object. In our project, we have several test files: one with the test cases for signal vector values, one for testing the functions with different values for frequency, etc. Figure 2 shows the output from the first test file - preallocationTest. As the figure shows, the output is shown in the Command Window. In Matlab, if the test case passed, it doesn’t show any output. So, the output shown here is only by the test cases.
that failed. The output is presented in details, telling where and why the test case failed: function \( h3r \) can not provide results for signal contains only 0s, it does not work with negative values, combination of 0s and negative values, combination of 0s and positive values, etc.

Fig. 4. Performance testing of the \( \text{pan} \) \( \text{tompkin} \) function.

3.2 Performance testing

Performance testing is a non-functional testing which checks the behavior of the system when it is under significant load. In a case of performance testing the software system is evaluated from a user’s perspective, and is typically assessed in terms of throughput, stimulus response time, or both. Performance testing could be used to assess the level of system availability also [Vokolos and Weyuker 1998]. The goal of performance testing is to identify the performance bottleneck, to make comparison
of the performance, etc. Usually, the performance testing is made by using a benchmark - a program or workload designed to be representative of the typical software system usage [Pan 1999].

In the earlier versions of Matlab, in order to measure the code's performance, there was a testing framework, which included different performance measurement-oriented features. The newest version comes with a tool called Profiler, which automates the work. It provides details about the performance of a specific function: how much time did it take to execute the function, the percentage of the summed up time of executing the part of the program which that separate unit used, whether a line has been executed and if so, how many times, the full code coverage etc.

Figure 3 shows the results of the execution time of the \texttt{h3r} function. It gives information about the number of calls and total execution time for each children function call.
Figure 4 shows the executing time of the \texttt{pan_tompkin} function.

With the Profiler Tool, we can analyse each function call separately and see more detailed information about its executing, like it’s shown in Fig. 5. The results show that \texttt{pan_tompkin} function took most of the execution time (0.184s or 73.1% of the total execution time).

4. CONCLUSION

Testing of the scientific software must become a standard part of the development process. Not because we only want to implement the correct way of development process as specified in the literature, but also we should consider the fact that many of the scientific software programs are connected to people’s health and can be categorized as critical software programs. Many of the testing methods designed for commercial scientific software can be adapted to scientific software testing. Testing should be considered from different aspects also. For example, even if the scientific program code produces accurate results, problems with performance can exist.

In this paper we made unit and performance testing of a scientific software for calculating ECG-derived heart rate (HR) and respiratory rate (RR) designed to aid the triage process in the emergency medicine which is crucial for ranging the priority of the injured victims in mass casualty situations based on the severity of their condition. The testing was done in Matlab.

Multiple unit tests were created to test the functionality of the algorithm. Matlab provides a very user friendly testing interface. Most of the things are fully automated and only function parameters are required for testing. Unit testing and input space partitioning method helped us to find several function errors, especially with test cases with boundary values. For e.g. ECG signal with zeros, signal with negative values and signals with different lengths helped us to add exceptions in the code.

Performance testing helped us to check the execution times of the functions. We performed tests with different signal lengths in order to increase the data load. That was useful to think about the code optimization which is left as our future work.

REFERENCES

Paul Ammann and Jeff Offutt. 2016. Introduction to software testing. Cambridge University Press.


Software Metrics as Identifiers of Defect Occurrence Severity

GORAN MAUŠA, University of Rijeka
TIHANA GALINAC GRBAC, Juraj Dobrila University of Pula
LUCIJA BREZOČNIK, VILI PODGORELEC and MARJAN HERIČKO, University of Maribor

Successful prediction of defects at an early stage is one of the main goals of software quality assurance. Having an indicator of the severity of defect occurrence may bring further benefit to allocation of testing resources. This paper is a part of a project aimed at identifying the role of different software metrics in order to improve the software quality assurance activities. In a preliminary case study we analyzed the relationship between the software metrics and defects using fundamentally different approaches for feature selection. Our case study showed that some metrics do not indicate defect occurrence, several of them exhibit moderate level of correlation, and the choice of the appropriate metrics is biased by the choice of feature selection technique. The next step would be to integrate the finding from different approaches and various datasets to develop a hybrid method for a precise definition of software metrics and their threshold levels that are good indicators of defect occurrence.

1. INTRODUCTION

Software quality assurance is in need of finding indicators for early corrective activities in the software development life-cycle. Code smell is a term used to encompass the potential indicators of deeper problems in a software [Fowler 2018]. Unlike software defects, the code smells usually do not encompass the kind of bugs that prevents the software from functioning properly. Instead, they are used to describe technical incorrectness or other weaknesses in design that may deteriorate software development or increase the risk of future defects [Vidal et al. 2018]. Obviously, the code smells and software defects are interlinked and both pose a threat to software correctness, validity and performances.

The code smells may be divided into three main groups, depending on the granularity level of their analysis: application-level, class-level, and method-level smells [Fowler et al. 1999]. At each level, different software metrics may be extracted and the open research question is which of them are important for the detection of code smells. Existing metrics most often measure static code attributes like cohesion, coupling, complexity, encapsulation, inheritance, and size. These metrics have been widely used to build models for software defect prediction [Basili et al. 1996]. However, obtaining good prediction models requires the selection of appropriate software metrics and the definition and validation of their thresholds for a particular programming language and/or project type. Nowadays, we have two main branches of threshold derivation: statistical approaches and approaches based on machine-
learning techniques. The problem that occurs is that different code smell detection tools, as well as metrics tools/programs, give significantly different results, resulting in different perception of code smells [Paiva et al. 2017].

This paper is a continuous effort on analyzing code smells and their impact on software quality [Gradišnik et al. 2019b; Gradišnik et al. 2019a]. The aim of this paper is to address the variability of object-oriented software metrics’ importance for detecting the problematic code by investigating the relationship between faults and code smells, i.e. software components with technical debt. The relationship is analyzed on 5 subsequent releases of Eclipse JDT open source project written in Java by using four different approaches:

1. statistical approach based on Spearman’s correlation analysis,
2. feature selection algorithm based on optimization algorithm Binary Particle Swarm Optimization,
3. univariate classification algorithm based on Logistic regression, and
4. threshold derivation algorithm based on Bender method.

The structure of the paper is the following: a brief description of studies that motivated our approach is presented in section 2, the algorithms of the four different approaches is given in section 3, the results of the case study are shown in section 4, while the discussion along with threats to validity and the conclusion is given in sections 5 and 6.

2. BACKGROUND

A vast number of collected and stored attributes or features is nowadays presenting more disadvantages than advantages. One of the biggest reasons for that statement lies in the problem of building a suitable classifier based on the whole set of features, and related significant time complexity [Brezočnik et al. 2018]. To overcome such problems, researchers often employ some feature selection methods.

Too many independent variables can also have negative effects on model’s fault-proneness prediction, making the model more dependent on the data set currently in use and therefore less general [Briand et al. 2000]. Selecting the appropriate measures to be used in the model requires a strategy of minimizing the number of independent variables in the model. This paper investigates the usage of static code attributes as independent variables and univariate forward and backward step-wise selection principles in choosing the appropriate ones. By finding the subset of the most relevant measures, we might also gain deeper understanding of the relations between software metrics and defect occurrence. The model performances are evaluated using widely used performance measures such as accuracy, sensitivity and false alarm rate [Zhou and Leung 2006; Guo et al. 2004]. A defect prediction model should identify as many fault prone modules as possible while avoiding false alarms [Lessmann et al. 2008].

Some studies analyzed the relationship between software metrics and defects and found that certain metrics do exhibit strong association. An early study by Shatnawi et al. [Shatnawi 2010] and a recent study by Arar et al. [Arar and Ayan 2016] analyzed several object-oriented metrics like coupling between objects (‘CBO’), response for class (‘RFC’), weighted method per class (‘WMC’), depth of inheritance tree (‘DIT’) and number of children (‘NOC’). The results of their analysis showed that ‘CBO’, ‘WMC’ and ‘RFC’ metrics are the ones that have stronger association to software defects that the remaining ‘DIT’ and ‘NOC’ metrics. The replicated study by Arar et al. [Arar and Ayan 2016] included also other metrics like lack of cohesion between objects (‘LCOM’), maximum and average cyclomatic complexity (‘MAXCC’, ‘AVCC’) and lines of code (‘LOC’) and showed that these metrics may also prove to be significant for successful prediction models. However, the object-oriented programs can hold several types of software complexity [Seront et al. 2005] and hence, our project goal is to examine a larger number of software metrics aiming to find the appropriate Java code smell identification metrics.
3. SOFTWARE METRICS ANALYSIS
The goal to find relationship between static code attributes and defect occurrence in the program code. Four different techniques are described in the following subsections.

3.1 Correlation Analysis
Correlation analysis is a standard statistical approach to establish whether there exists a relationship between two variables. As the values within the variables are usually distributed normally, Pearson’s correlation is the standard technique. However, in software engineering this is rarely the case, so the Spearman’s correlation, a non-parametric variant, is used to express correlation [D’Agostino and Pearson 1973]. Spearman’s correlation coefficient ($r_s$) measures the power of a monotonic relationship between paired data in a sample $s$, that consists of $n$ observations, using the following expression [Myers and Well 1991]:

$$r_s = 1 - \frac{6 \sum_{i=1}^{n} d_i^2}{n^3 - n}$$  \hspace{1cm} (1)

Parameter $d_i$ presents the difference between the ranks of two variables at each observation $i$, yielding the values in range $-1 \leq r_s \leq 1$ with interpretation similar to Pearson’s correlation [Lehamn et al. 2005]. Positive and negative values indicate correlation, whereas values close to zero indicate absence of correlation. The more distant the value is from zero, the stronger the correlation is, and it is interpreted according to the five levels of correlation strength, as represented in Table I [Evans 1996].

<table>
<thead>
<tr>
<th>$r_s$</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.00-0.19</td>
<td>Very poor correlation</td>
</tr>
<tr>
<td>0.20-0.39</td>
<td>Low correlation</td>
</tr>
<tr>
<td>0.40-0.59</td>
<td>Moderate correlation</td>
</tr>
<tr>
<td>0.60-0.79</td>
<td>Strong correlation</td>
</tr>
<tr>
<td>0.80-1.00</td>
<td>Very strong correlation</td>
</tr>
</tbody>
</table>

3.2 Binary Particle Swarm Optimization and Feature Selection
Algorithm BPSO+C4.5 [Brezočnik, Lucija 2017] uses a Binary Particle Swarm Optimization algorithm (BPSO) [Kennedy 1997] as a basis and extends it with Feature Selection mechanism. Since particles in the BPSO+C4.5 algorithm are moving in a binary search space, their initialization is done with binary values 0 and 1. Each particle represents a feasible solution in an $n$-dimensional search space. For example, a particle 1001, presents 4-dimensional space where only the first and the last features are included.

The particles are moving through iterations following two extremes. First one is individual extreme $pBest_i$, which comprises the best-obtained position of each particle $i$ in the $n$ iterations $pBest_i = (pBest_{i1}, pBest_{i2}, \ldots, pBest_{in})$. The second extreme is called global extreme $gBest$ which comprises the best-found solution so far in the entire swarm $gBest = (gBest_{1}, gBest_{2}, \ldots, gBest_{n})$.

The movement of the $i$-th particle from old to new position in the search space is controlled by velocity vector $V_i = [v^1_i, v^2_i, \ldots, v^n_i]$ and by position vector $X_i = [x^1_i, x^2_i, \ldots, x^n_i]$. Velocity is calculated according to the equation 2 and consists of three main parts:

$$v_{id}^{new} = \omega \times v_{id}^{old} + c_1 r_1(pBest_{id}^{old} - x_{id}^{old}) + c_2 r_2(gBest_{id}^{old} - x_{id}^{old})$$  \hspace{1cm} (2)
where $\omega$ is the inertia weight, $c_1$ and $c_2$ are positive constants and $r_1$ and $r_2$ are two random functions in the range $[0,1]$. Inertia component $\omega \times v_{id}^{old}$ is responsible for controlling the velocity of each particle, cognitive component $c_1 r_1 (pBest_{id}^{old} - x_{id}^{old})$ or particle memory tends to direct particles to their personal best positions, and the social component $c_2 r_2 (gBest_{d}^{old} - x_{id}^{old})$ which tends to direct particles to the globally best position found so far in the entire swarm. After the velocity update, position adjustment is carried out with equation 3:

$$x_{id}^{new} = \begin{cases} 1, & \text{if } 1 + \frac{1}{e^{v_{id}^{new}}} > U(0,1) \\ 0, & \text{otherwise} \end{cases}$$

where $e$ is the base of the natural logarithm and $U(0,1)$ is the uniform distribution.

3.3 Univariate Logistic Regression

Logistic regression is a statistical modeling method used to estimate the probability of occurrence of an event, referred to as the dependent variable. Depending on the number of independent variables that are fitted to a logistic function, there are multivariate and univariate models, the later containing only one variable. Logistic regression is suitable for binomial dependent variables and hence it is an appropriate model for software defect prediction. In comparison to other related statistical techniques it is more flexible and robust [Hamadicharef et al. 2008; Tabachnick et al. 2007]. It does not assume normal distribution or equal variance within independent variables, nor linear relationship between the dependent and independent variable. These characteristics make logistic regression a commonly used classification algorithm.

The logistic regression models the probabilities of two different classes, ensuring the sum of probabilities equals 1 using the following equation [Hastie et al. 2009]:

$$P(X) = \frac{e^{\beta_0 + \beta_1 X}}{1 + e^{\beta_0 + \beta_1 X}}$$

where $\beta_0$ is the free coefficient and $\beta_1$ is the regression slope coefficients for dependent variable $X$. In case of software defect prediction, $X$ represents a software metric and $P(X)$ is the fault-proneness probability. The coefficients $\beta_0$ and $\beta_1$ define the curvature of the non-linear logistic regression output curve and they are estimated by the maximum likelihood procedure.

A likelihood ratio chi-square test is usually used to assess the statistical significance of each independent variable in the model. The null-hypothesis is: there is no relationship between the logistic regression model and the dependent variable, i.e. the true coefficients are zero. Finally, if the level of statistical significance $p$-value is below 0.05, the dependent variable is considered to be significant for prediction. In univariate logistic regression model, each software metric is estimated separately and this is another way to find the "appropriate indicators of software quality".

3.4 Bender Method

Bender method is a threshold derivation method based on the univariate logistic prediction model [Arar and Ayan 2016], graphically presented in Figure 1. The method itself consists of three phases: sampling the data according to the stratified 10-fold cross-validation, training the univariate logistic regression model, and computing the threshold level for all the metrics which are statistically significant for prediction in the logistic model. The fourth phase is the testing phase in which threshold-based defect prediction models are built and their effectiveness is evaluated in terms of the geometric mean (GM) accuracy measure. In these models, an instance is declared fault-prone should a metric level go beyond the calculated threshold.
The univariate logistic regression model described in previous subsection is used in the same way in this method. After performing the analysis of statistical significance, the free coefficient $\beta_0$ and the regression slope coefficient $\beta_1$ are passed along with the the base probability $p(0)$ to compute the threshold level. As shown in figure 1, the percentage of majority class is used as the base probability [Arar and Ayan 2016], thus taking into account the probability that a randomly selected instance belongs to the majority class. The value of an acceptable risk level, i.e. the threshold value for each metric $THR$ is computed using the following equation [Bender 1999]:

$$THR = \frac{1}{\beta_1} (\ln(\frac{p_0}{1-p_0}) - \beta_0),$$

The effectiveness of calculated threshold levels for each metric is evaluated in the last phase. A simple decision tree is built using the obtained threshold levels to classify the software modules from the testing dataset according to the following equation:

$$Y_i = \begin{cases} \text{faulty} & \text{if } X_i > THR \\ \text{non-faulty} & \text{if } X_i \leq THR \end{cases},$$

where $i$ represents the $i$–th software module in the testing dataset, $X_i$ represents the actual value of software metric and $Y_i$ represents the prediction output. After comparing the prediction results against the known values of defect proneness in the testing set, GM is calculated as:

$$GM = \sqrt{TPR \times TNR}$$

where true positive rate (TPR) and true negative rate (TNR) represent the accuracy of positive (faulty) and negative (non-faulty) class, respectively.

$$TPR = \frac{TP}{TP + FN}, \quad TNR = \frac{TN}{TN + FP}$$
4. RESULTS
This section presents preliminary results of a case study used to compare the four fundamentally different approaches which we used for scoring the importance of software metrics for defect prediction. Each of these four approaches has been used to perform some sort of feature selection, i.e. to find the important indicators of software quality. Only the BPSO method is designed to perform pure feature selection based on wrapper method, whereas other methods are used for different purposes. The aim of this comparison is to estimate their degree of overlapping and open new research questions about this issue.

4.1 Datasets
The datasets used in this case study are five consecutive releases of Eclipse JDT open source project (2.0, 2.1, 3.0, 3.1 and 3.2), systematically collected by the BuCo Analyzer tool [Mausa and Grbac 2016]. The datasets contain 50 different software metrics and the number of defects for each java source code file in the project. The full list of metrics and their descriptions may be found in [Mauša and Grbac 2017]. The software metrics that are computed describe static attributes like size, complexity, various object-oriented principles, design characteristics, programming style and more. The datasets are also available on-line for the whole research community1.

4.2 Comparison results
Table II presents full list of metrics from software defect prediction datasets and how well they scored in four different techniques used to analyze their importance for defect prediction. The correlation analysis computed the Spearman correlation coefficients (\(r_s\)), the BPSO algorithm returned the percentage of including a metric (Rate), Logistic regression returned the percentage of finding a metric significant for prediction of defects (Rate) and Bender Method yielded the geometric mean accuracy value of how successful a metric’s threshold is for finding defects (GM). There is no rule on how to interpret the inclusion rate given by BPSO and Logistic Regression, nor GM value of threshold values calculated by Bender method. In order to have a common baseline of comparison, the standard interpretation levels of correlation analysis, given in Table I, were used for all four approaches. This kind of interpretation seems understandable for interpreting the Rate given by BPSO and Logistic regression and it is in accordance with the interpretation of GM values obtained by Bender method. The interpretation is given in the right sub-column of each technique.

5. DISCUSSION
There are only a few metrics for which all techniques agreed about the level of their relationship with defects. For example, LCOM and cohesion (COH) exhibit low or very low relationship while efferent coupling (FOUT) and specialization index (SIX) exhibit low or very low relationship for all techniques besides the BPSO. On the other hand, a greater number of metrics have been found to have moderate or strong relationship, like: AVCC, HBUG, HEFF, UWCS, RFC, LMC, HVOL, EXT, TCC, NCO and CCOMHLTH. The disagreement between the techniques is present for metrics like INST, PACK, CBO, MI, CCML, NLOC, F_IN, R_R and HIER where correlation analysis and BPSO indicate lower level whilst Logistic regression and Bender method indicate stronger levels. The disagreement in the opposite direction is present for metrics like S_R, NSUB, where correlation analysis and BPSO indicate a stronger level of relationship than the other techniques. This means that the choice of the appropriate software metrics is biased by the choice of feature selection.

1http://www.seiplab.riteh.uniri.hr/?page_id=834
<table>
<thead>
<tr>
<th>Metrics</th>
<th>Correlation Analysis</th>
<th>BPSO Rate</th>
<th>Logistic Regression Rate</th>
<th>Bender Method Rate</th>
<th>GM</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>0.58</td>
<td>MODERATE</td>
<td>35%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>SLOC_P</td>
<td>0.58</td>
<td>MODERATE</td>
<td>55%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>SLOC_L</td>
<td>0.58</td>
<td>MODERATE</td>
<td>30%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>MVG</td>
<td>0.62</td>
<td>STRONG</td>
<td>60%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>BLOC</td>
<td>0.49</td>
<td>MODERATE</td>
<td>65%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>C_SLOC</td>
<td>0.58</td>
<td>MODERATE</td>
<td>50%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>CLOC</td>
<td>0.40</td>
<td>LOW</td>
<td>75%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>CWORD</td>
<td>0.51</td>
<td>MODERATE</td>
<td>70%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>HLOC</td>
<td>0.67</td>
<td>STRONG</td>
<td>70%</td>
<td>STRONG</td>
<td>4%</td>
</tr>
<tr>
<td>HCWORD</td>
<td>0.48</td>
<td>MODERATE</td>
<td>80%</td>
<td>STRONG</td>
<td>40%</td>
</tr>
<tr>
<td>No-Methods</td>
<td>0.50</td>
<td>MODERATE</td>
<td>55%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>AVCC</td>
<td>0.47</td>
<td>MODERATE</td>
<td>65%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>NOS</td>
<td>0.53</td>
<td>MODERATE</td>
<td>35%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>HBUG</td>
<td>0.53</td>
<td>MODERATE</td>
<td>70%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>HEFF</td>
<td>0.52</td>
<td>MODERATE</td>
<td>60%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>UWC</td>
<td>0.51</td>
<td>MODERATE</td>
<td>75%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>INST</td>
<td>0.28</td>
<td>LOW</td>
<td>30%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>PACK</td>
<td>0.13</td>
<td>VERY LOW</td>
<td>60%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>RFC</td>
<td>0.49</td>
<td>MODERATE</td>
<td>50%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>CBO</td>
<td>0.10</td>
<td>VERY LOW</td>
<td>50%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>MI</td>
<td>-0.20</td>
<td>VERY LOW</td>
<td>30%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>CCML</td>
<td>0.45</td>
<td>MODERATE</td>
<td>40%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>NLOC</td>
<td>0.52</td>
<td>MODERATE</td>
<td>25%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>F_IN</td>
<td>0.10</td>
<td>VERY LOW</td>
<td>55%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>DIT</td>
<td>-0.87</td>
<td>VERY STRONG</td>
<td>45%</td>
<td>MODERATE</td>
<td>23%</td>
</tr>
<tr>
<td>MINC</td>
<td>-0.49</td>
<td>MODERATE</td>
<td>60%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>S_R</td>
<td>-0.50</td>
<td>MODERATE</td>
<td>65%</td>
<td>STRONG</td>
<td>24%</td>
</tr>
<tr>
<td>R_R</td>
<td>0.24</td>
<td>LOW</td>
<td>50%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>COH</td>
<td>-0.29</td>
<td>LOW</td>
<td>45%</td>
<td>MODERATE</td>
<td>29%</td>
</tr>
<tr>
<td>LMC</td>
<td>0.54</td>
<td>MODERATE</td>
<td>55%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>LCOM2</td>
<td>0.42</td>
<td>MODERATE</td>
<td>40%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>MAXCC</td>
<td>0.54</td>
<td>MODERATE</td>
<td>35%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>HVOL</td>
<td>0.52</td>
<td>MODERATE</td>
<td>50%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>HIER</td>
<td>0.38</td>
<td>LOW</td>
<td>45%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>NQU</td>
<td>0.21</td>
<td>LOW</td>
<td>55%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>FOUT</td>
<td>0.10</td>
<td>VERY LOW</td>
<td>70%</td>
<td>STRONG</td>
<td>24%</td>
</tr>
<tr>
<td>SIX</td>
<td>-0.30</td>
<td>LOW</td>
<td>55%</td>
<td>MODERATE</td>
<td>27%</td>
</tr>
<tr>
<td>EXT</td>
<td>0.53</td>
<td>MODERATE</td>
<td>60%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>NSUP</td>
<td>0.19</td>
<td>VERY LOW</td>
<td>70%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>TCC</td>
<td>0.57</td>
<td>MODERATE</td>
<td>60%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>NSUB</td>
<td>-0.50</td>
<td>MODERATE</td>
<td>35%</td>
<td>LOW</td>
<td>36%</td>
</tr>
<tr>
<td>MPC</td>
<td>0.53</td>
<td>MODERATE</td>
<td>40%</td>
<td>LOW</td>
<td>100%</td>
</tr>
<tr>
<td>NCO</td>
<td>0.51</td>
<td>MODERATE</td>
<td>50%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>INTR</td>
<td>-0.07</td>
<td>VERY LOW</td>
<td>80%</td>
<td>STRONG</td>
<td>100%</td>
</tr>
<tr>
<td>CCOM</td>
<td>0.53</td>
<td>MODERATE</td>
<td>55%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>HLTH</td>
<td>0.53</td>
<td>MODERATE</td>
<td>45%</td>
<td>MODERATE</td>
<td>100%</td>
</tr>
<tr>
<td>MOD</td>
<td>0.10</td>
<td>VERY LOW</td>
<td>55%</td>
<td>MODERATE</td>
<td>97%</td>
</tr>
</tbody>
</table>
Further analysis of the obtained results have shown that the relationship of certain metrics and defects exhibits uniform level of strength within different datasets. The correlations analysis is a more strict technique, rarely indicating strong and never indicating very strong relationship. Hence, the moderate level of relationship indicated by the correlation analysis has a greater weight that the same level of relationship indicated by the other techniques. The only two metrics that have strong relationship according to the correlation analysis are McCabe complexity (MVG), header comments (HCLOC) and depth of inheritance tree (DIT). It is interesting to notice that HCLOC and DIT exhibit such relationship when analyzed by BPSO and correlation analysis, but opposite levels of strength when analyzed by Logistic regression or Bender method and the agreement of all four techniques is achieved only for MVG.

Unlike the correlation analysis, the uniform Logistic regression technique is a less strict one and it indicated very strong level of relationship for most metrics. The Bender method, which is based upon the Logistic regression, and BPSO are moderately strict techniques. The distribution of the levels of relationship found by these two techniques is closer to a normal distribution than for the other two techniques.

The validity of this small scale case study is strongly affected by the choice of data. This is a first preliminary study, so it is based only on a sample of data from five consecutive releases of an highly object-oriented open source project from Eclipse community. The datasets are chosen from big and complex software projects to resemble as much as possible the industrial projects to cope with construct validity. The external validity is clearly threatened and general conclusions cannot be drawn yet. The aim of this case study was more to open new research questions and motivate the research direction of future work. Projects from different background, like different communities, development methodologies or written in different programming language need to be included to obtain more general conclusions.

The four chosen techniques which were used to analyze the relationship between software metrics and defects are a threat to internal validity. There exists a number of other techniques of similar purpose, but these are chosen because correlation analysis and logistic regression are the well known and widely used ones, Bender method is a rarely used technique with completely different aim (finding threshold levels) and BPSO is a novel hybrid technique for feature selection. At this stage, the conclusions about the relationship levels between analyzed metrics and defects still lack a precise explanation, hence threatening the conclusion validity and the their causality is unknown and open to speculations. That is why we believe this project to be an important one, and giving precise answers to our research objectives may bring significant improvement in understanding the role of different software metrics and improving the software quality assurance activities.

6. CONCLUSION

The algorithms that implement the proposed methodology have been applied to 50 different software metrics that are present in the datasets. The moderate level of relationship is an indication enough that a metric may improve the classification of defective software modules. The metrics for which at least one technique found a stronger level of relationship is thus a potential indicator of code smell. However, this may be misleading since different techniques find different metrics to be important. Hence, it is important to continue this research direction and find stronger evidence to precisely defining software metrics and their threshold levels that are good indicators of code smell. If such metrics are to be found, it would be possible to pay more attention to them in the whole life cycle of software development and reduce the possibility of code smell turning into defects.
REFERENCES


Martin Fowler. 2018. Refactoring: improving the design of existing code. Addison-Wesley Professional.


The Technical Debt Management Tools Comparison

LUKA PAVLIČ and TILEN HLIŠ, University of Maribor

This paper focuses on the technical debt metaphor and its management practices. We present an overview of the currently available tools for managing technical debt. We also present leading tools, that enable technical debt measurement. We demonstrate the noticeable deficit in the number of tools that are intended and specialized for management activities only. Even existing tools do not support all technical debt management activities. This observation became a guide in the development of our own solution for managing technical debt. In this paper we also present our tool for managing technical debt (TD Tool).

1. INTRODUCTION

During the software development cycle, teams are confronted with changing requirements, short deadlines, and high-quality requirements. Combined, they lead to higher costs. Cost cuts are a common practice in all industries, software development is not an exception. Managed carefully, they can be even used to boost long-term software quality. In this context, the technical debt metaphor, rooted in financial world, describes a crucial challenge in the software development industry. In long run, debt causes problems. However, it can be employed in useful manner for a short time goal. This can be accomplished by managing technical debt carefully through the development cycle. The metaphor itself covers several aspects – including source code, IT architecture, design decisions, documentation, requirements, and testing as well. The area of technical debt management consists of several well defined and proven practices. Their purpose is to support the decision whether it is sensible to introduce some more debt to a project by monitoring its accumulated quantity. The debt management process can be facilitated using dedicated supporting tools. The specialized tools landscape is opening. Teams can choose between several types of tools that offer functionalities from assessing current debt quantity via measuring quality attributes to manual debt management tools.

To manage technical debt well, it is an imperative to use technical debt management tools. Where fast development is needed, we can accept some debt, which is later removed [Ambler Scott 2013]. First of all, teams need to be aware of its existence. Logical next steps include locating, assessing and describing existing or new technical debt items. They (technical debt items) are a basis for debt elimination in the long run, when teams decide to do so.

2. TECHNICAL DEBT MANAGEMENT PRACTICES AND CHALLENGES

Due to continuous changes in the market and the need for agility in the development of modern information solutions, it is crucial that development teams also manage technical debt. Development teams are confronted with changing demands, short-term, and high-quality requirements, which
makes the costs of implementing changes higher. To achieve desires for fast system development, developers often use shortcuts that provide a working prototype, which is not complete. Consequently, the emergence of technical debt is unavoidable. It is vital that development teams are aware of this and are focusing their time on the management of technical debt [Allman 2012]. Management consists of several activities and approaches that prevent the emergence of new technical debt while keeping the existing one below the critical limit.

2.1 Technical debt management activities
The purpose of management is to keep the accumulated debts under control. Aim of the technical debt management is not about abolished it entirely. Management of technical debt consists of a series of activities that prevent the creation of unwanted technical debt or is dealing with existing debt to keep it below the permissible limit. After reviewing the literature, we rely on the descriptions of the activities made in the systematic study of the technical debt (TD) and its management. The study defines eight activities, summarized in Table 1.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD identification</td>
<td>The purpose of the activity is the discovery of a technical debt arising from technical decisions within the software system. Operation is performed using various techniques, such as static source code analysis.</td>
</tr>
<tr>
<td>TD measurement</td>
<td>The activity quantifies the benefits and the cost of a known technical debt through evaluation techniques. It is also used for estimating the share of technical debt concerning the entire information solution.</td>
</tr>
<tr>
<td>TD prioritization</td>
<td>Ranks identified technical debt according to pre-defined rules. The allocation determines which the technical debt must be eliminated first and which can remain in later versions of information solutions.</td>
</tr>
<tr>
<td>TD prevention</td>
<td>Aims to prevent potential technical debt from being incurred.</td>
</tr>
<tr>
<td>TD monitoring</td>
<td>Activity watches the changes in the cost and benefit of unresolved technical debt over time.</td>
</tr>
<tr>
<td>TD repayment</td>
<td>Removes or resolves technical debt with techniques such as re-engineering or refactoring.</td>
</tr>
<tr>
<td>TD representation/</td>
<td>Provides ways to present and codify technical debt in a uniform approach to display it to specific roles in the development company.</td>
</tr>
<tr>
<td>documentation</td>
<td></td>
</tr>
<tr>
<td>TD communication</td>
<td>Makes identified technical debt visible to all the roles in a development company so that it can be discussed and further managed.</td>
</tr>
</tbody>
</table>

2.2 Technical debt management challenges
The management of technical debt involves several challenges. In the literature we can find the lack of a standard unit to indicate the amount of technical debt that would be measurable and generally
comparable. It is usually referred as the biggest challenge. During assessing the amount of debt, the amount of principal is estimated. The principal can be expressed in the form of working hours or in the form of several financial assets, needed to eliminate debt. Because this is a subjective assessment, large deviations can occur between the estimate and the actual investment. Due to difficulties in measuring the amount of debt development companies sometimes do not decide on its management.

The next management challenge is that debt is usually created and eliminated by various employees. Organizations find it difficult to determine the business value of the internal quality of the developed software. It is difficult to translate technical debt into economic consequences. The reviewed literature find that it is problematic to eliminate the existing debt and at the same time, prevent the emergence of a new one [Li et al. 2015].

2.3 Technical debt tools characteristics

Tools for managing technical debt should target both practitioners and researchers. The design of a tool should be guided by empirical evidence about developers’ practices so that the tool is tailored to the actual needs of the developers. Usage statistics should be recorded as early as possible, to allow focusing on improvements targeting the most popular tool features. A tool should be offered as an IDE plug-in instead of a standalone application, and it should require minimum installation and configuration effort. A tool should be tested in an industrial setting and should also be open-sourced as early as possible, even if it is not as mature and stable. A tool should be accompanied with documentation, tutorials, and code snippets demonstrating the use of its API.

3. LEADING TOOLS COMPARISON

For the successful management of technical debt, we need relevant information on individual debt. Data is also required for the proper removal planning. An essential step for a successful debt management is to record all debts and capture all necessary information. A good practice is the use of dedicated technical debt management tools.

In this section, we first introduce the landscape of currently available tools, according to the extensive literature overview. In our analysis, we have included tools for managing technical debt and, in addition, also leading tools which are intended for technical debt measurement.

With selected tools for managing technical debt, we rely on a study done in the form of a survey of how technical debt is managed. 15 large organizations were included in the study [Martini et al. 2018]. The questionnaire covered a wide area of technical debt management within the organization. We mainly focused on the answers to the question of which tools are used to track technical debt. Solutions were presented in the form of the word cloud, which shows the distribution of tools used among the respondents. Tools are mostly utilized for backlog, documentation, as a static analyzer, and as issue trackers. In conclusion, the following considerations on the tools were made [Martini et al. 2018]:

- Comments in the code cannot be considered for tracking technical debt.
- Documentation of technical debt increase awareness but it also has the highest overhead, therefore is not considered as a high level of tracking. Respondents answer that the main tools used for documentation of technical debt are Microsoft Excel or Word what we found as a not recommendable practice.
- Using the bug system for tracking technical debt does not increase the level of awareness, and it has a slightly higher overhead. Study infer that this is also not the best way of tracking technical debt.
- Backlogs increase tracking level and are also one with the least overhead. The study suggests using backlogs is one of the best practices at the moment to follow technical debt. The most used tools are Jira, Hansoft, and Excel.

After a comprehensive overview of the tools for managing technical debt that are mentioned in the literature and are considered as leading solutions, we focused on the following tools.

The **Jira software** is a product developed by Atlassian and is designed to record and track bugs and issues that occur during software development. Jira offers management of bugs and issues and also the project management functionality. In 2017, Jira was ranked first in the popularity of tools for bugs and issues management [According to management 2018]. In 2019, Gartner announced Jira as the leading solution in the field of enterprise agile planning tools in conjunction with AgileCraft [Gartner 2019]. The identified technical debt is only assessed by a priority and not by its quantity. The tool can be used as a trial version for one month [Atlassian 2019]. A significant disadvantage of Jira is the lack of support for technical debt measurement. The tool does not offer any quantitative input options to define the extent of the technical debt; therefore, monitoring is insufficient.

**Hansoft** is another tool for software development project management. Hansoft provides project management software for team collaboration and management in Agile and traditional products and services development. Native Windows, OS X, and Linux application with both on-premise and hosted option available. It also can be used as a plugin for Jira, or we can use their SDK for extensions [Perforce 2019]. The great advantage of software Hansoft is the possibility of free use for up to five users. It is mainly used in the development of video games and IoT. It can show recorded technical debt in the form of a list or with a graph which is manually set [Perforce 2019]. Hansoft, as Jira, has a lack of support for technical debt measurement. The tool offers that each issue can be assigned with status and severity.

Next, in a series of our selected tools is **Redmine**. It is open source and released under the terms of the GNU General Public License v2 (GPL). The tool is created with Ruby on a Rails framework and is suitable for individual use or smaller business groups. Redmine is one of the few tools that offer input option for quantitative debt assessment. We can specify the date by which the debt needs to be eliminated, and we can also add the number of hours that will be required to eliminate discovered technical debt. Furthermore, we can also mark debt to be partially removed, which is also reflected in the Gantt chart [Redmine 2019]. Currently, the biggest drawback of Redmine is the fact that it is not so well known and widespread.

The **DebtFlag** tool is designed to capture, track, and resolve technical debt in the software development. The tool is intended for use as a plugin in the integrated development environment. Developers can use a tool to capture technical debt and save the location of it. DebtFlag consists of two parts, a plug-in for the integrated development environment Eclipse, where the developer inputs the data of the identified debt, and the web application that takes care of the dynamic presentation of debts (Holvitie in Leppanen, 2013). DebtFlag offers out-of-the-box plugin for an integrated development environment. The weakness of the tool is small support for technical debt management activities, and that is limited to only one IDE.

**TD-Tracker Tool** is a tool that enables tabulation and management of technical debt characteristics. With the tool, we can create a catalog of technical debts from various stages of software development. TD-Tracker Tool helps developers in the decision-making process and lets you manage identified technical debts. It also allows you to connect to external tools. Debts can be
imported from external sources, or users enter them manually (Foganholi, et al., 2015). TD-Tracker Tool has a lack of sufficient technical debt monitoring.

**SonarQube**, formerly Sonar, is an open-source platform developed by SonarSource and is used to verify the quality of the code continuously. It is designed to performing automated checks by statically analyzing the code for detecting bugs, code smells, and security vulnerabilities in more than 20 programming languages. With these functionalities, we can also very effectively measure technical debt on various projects [SonarQube 2019]. Technical debt in SonarQube is shown as the calculation of the time needed to eliminate it. The calculation is based on the use of metrics. A detailed view of an individual debt displays critical files and suggests improvements. It also shows technical debt type, severity, and status. After calculating the technical debt, an “A” to “E” score is added to it, where “A” means almost no technical debt, and “E” means a critical value. The advantage of SonarQube is that it shows the time needed to eliminate the technical debt based on metrics. It also adds a rating for maintainability, bugs, vulnerabilities, and code smells. The tool has a lack of technical debt monitoring; if we want to review the progress of the elimination of debt, it is necessary to restart the calculation of metrics.

**Teamscale** analyzes the quality of the source code. With various static analyzes, it points to quality errors that can be quickly reviewed. Its primary purpose is to analyze code to identify specific maintainability constraints and avoid unexpected maintenance costs in the future. Teamscale's analyzes work on a wide variety of programming languages. Detecting clones, deep nesting, long methods, and files are included for all languages [CQSE 2019]. Additionally, many specialized checks are included, tailored to detect quality problems in specific languages. Tool core features are real-time feedback, dashboards, and integrated development environment integration. Supported IDE's are Eclipse, Visual Studio, IntelliJ IDEA, and NetBeans. TeamScala offers the ability to read the code directly from the source code repository. You can also link TeamScala to issue trackers, for example, with Jira [CQSE 2019]. Unfortunately, it does not report the approximate time of debt elimination like SonarQube. TeamScale is a useful tool when used in conjunction with others. It offers two months of a free trial.

**Ndepend** is a tool for measuring technical debt exclusively in the Visual Studio and is added as a plugin. After installation, we can access the dashboard where we select the project and analyze it. The functionality that the tool supports are analyzing, measuring, and monitoring technical debt. It also allows setting the errors priority. It supports only one programming language (C#). Therefore, it is more adjusted to this language. The tool displays errors in the project technical debt and time of debt elimination. Technical debt is also shown in individual categories. The scale of the debt is from “A” to “E”. The tool is suitable for a development team that develops exclusively in Visual Studio. A test and paid version are available [Ndepend 2019].

**Squore** supports various programming languages. The tool is cloud-based and available online. It offers functionalities for bugs and vulnerability overview, measurement of technical debt, and view for displaying files that have higher vulnerabilities. Squore is a platform with a lot of functionality and therefore is quite difficult to use. As with most other technical debt measurement tools, debt is shown in the form of time to eliminate it and critical categories [Squoring 2019]. The tool has a lack of technical debt monitoring; if we want to review the progress of the elimination of debt, it is necessary to restart the calculation of metrics.

The overview of the tools will be appended with the description of our own solution. Our tool (**TD Tool**) is intended for technical debt management. We developed a solution which includes an open
web platform and a plugin for the selected development environment. We tried to integrate as much of the functionality of the reviewed tools as possible and add those that we think are useful in managing the technical debt. A more detailed description of the tool follows after the comparison of the tools in the table 2 and table 3.

Table 2: Comparison of the tools, according to selected functionalities

<table>
<thead>
<tr>
<th>Functionality</th>
<th>Jira</th>
<th>Hansoft</th>
<th>Redmine</th>
<th>DebtFlag</th>
<th>TD-Tracker</th>
<th>SonarQube</th>
<th>Teamscale</th>
<th>Ndepend</th>
<th>Squre</th>
<th>TD Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project management</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Setting TD threshold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Manual entry of identified TD</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>The TD amount estimation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD calculation using metrics</td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD prioritization</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Determining who is responsible for the TD</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Managing TD status</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Graphical representation of the amount of TD</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
The Technical Debt Management Tools Comparison

Table 3: Comparison of the tools, according to supported activities

<table>
<thead>
<tr>
<th>Activities</th>
<th>Jira</th>
<th>Hansoft</th>
<th>Redmine</th>
<th>DebtFlag</th>
<th>Tool-Tracker</th>
<th>SonarQube</th>
<th>Teamscale</th>
<th>Ndepend</th>
<th>Square</th>
<th>TD Tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>TD identification</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD measurement</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD prioritization</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD prevention</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD monitoring</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD repayment</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD representation/documentation</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>TD communication</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

3.1 The TD Tool

An overview of the existing technical debt management and measurement tools has shown that there is a noticeable deficit in the number of tools that are intended only for management and not for measuring technical debt with metrics. However, those that are available do not support all technical debt management activities. This problem leads us to the development of our own solution, in which we are trying to capture all the technical debt management activities that are intended for its management.

The developed solution includes an open web platform and a plugin for the selected development environment. After reviewing the existing tools, we decided, which functionality should our tool offer. The online platform allows users to enter, manage, and edit projects. All of the tools examined are missing the functionality of determining the maximum amount of technical debt on a particular project and alerting the user when this limit is exceeded. We successfully implemented this functionality to our online platform. Our tool also supports the option for manually entering the identified technical debt, which can be edited later. The amount of technical debt is estimated for each debt individually. Through the management of debt status, we can prioritize all debts. A graphic representation of the amount of debt on an individual project is ensured for a more straightforward overview of debts. The plugin for the selected development environment offers the possibility to manually enter the identified technical debt from the development environment itself. For the development of our online platform, we used MEAN (Mongo-Express-Angular-Node.JS) stack, which allowed creating a simple RESTful API server also. It serves as an interface for communication with the database. The Angular platform, in conjunction with Angular material components, made it possible for us to build a modern user interface. Login and registration are handled by the Auth0 authentication system. After login, we are redirected to the control panel with two lists, one with all the projects and one with all the debts which are linked to the logged in user. We must click on a particular project or debt to see details about it. We also have options for editing
and deleting of the opened project or debt. The project can only be removed when we remove all the
debs that belong to it.

On the left sidebar, we find the option for adding a new project, which is enabled only to the
admin users. When the user clicks on the button to add a new project, an input form in the form of a
dialogue is opened. The user must enter the name of the project and add members who will
participate in the project by their emails. Then it is necessary to choose how to assess the debt in the
newly created project. We can choose between an estimate in value of money or an assessment with
effort, which is reflected with the number of working days. Finally, the upper limit of the allowed
debt should be set for the project. The entered value serves us as a threshold. A warning pops up
when the limit is exceeded.

When the user pressed on the desired project, its details are displayed, and the list of all debts is
replaced with debts that belong to the chosen project. Each debt is presented with Angular Material
card component. With a press on an individual card, details of debt are shown. The current amount
of debt on the project is shown using charts. The main user interface is demonstrated in figure 1.

![Figure 1: TD Tool main user interface – grouping TD items, based on projects; we can see TD accumulation and upper TD limit](image)

The main functionality of the developed solution is the input form for entering the technical debt
item. The user can first choose between the simple and advanced view. For simple entry, the cause
that caused the debt should be selected; the user can choose between technical cause and an
organizational cause. In the advanced view, the user must add type and subtype of technical debt.
Other input fields require name, a description which part of the system is affected by debt and an
estimate of debt estimate in value of money or an assessment with effort, which is reflected with the
number of working days. We also must add criticality and status of the debt. Every debt has
a particular person who is responsible for its elimination.

To complement the online platform and that the developed solution meets the useful
characteristics of the technical debt management tool, we developed a plugin for an integrated
development environment. Currently supported IDEs are IntelliJ and Visual Studio.
The plugin aims to complete the online platform and to enable adding of identified debt within the development environment. Debt is then available on the online platform and to its users. The visual part of the plugin includes a shortcut in the main menu of the development environment, where the input field to entry technical debt can be opened.

4. CONCLUSION

In this paper, we discussed the concept of the technical debt metaphor. We presented the activities and approaches for managing technical debt, and our observations. The collected activities allowed us to compare the reviewed tools with each other. Overview of existing tools has shown a significant increase in a number of tools that are involved in technical debt, but our review only covers tools that we think are currently among the leaders in this field or have shown great potential for proper management of technical debt. With an overview, we have discovered an area that is poorly supported by tools or the existing ones do not cover a lot of activities that are necessary for the successful management of technical debt. That has become a guide in the development of our solution for managing technical debt. The first step in the development was the production of an entry form of the identified debt item. After the overview of the literature and existing solutions, we have put together a simple and advanced form for entering debt item. We have selected the appropriate functionalities from the current tools and supplemented them with our own so that we could develop the most comprehensive management tool. Development continued with the search for suitable technologies for the development of an online platform and an expansion module for an integrated development environment. We believe that we have created so far, a practical and useful alternative to existing technical debt management tools. The development is, however, ongoing.

REFERENCES

Recent Trends in Software Testing – A Case Study with Google Calendar

BOJAN POPOV, BOJANA KOTESKA and ANASTAS MISHEV, Ss. Cyril and Methodius University

In this paper we make an overview of the software testing trends and as a case study we perform testing of the Google Calendar service. We present some of the latest testing techniques, frameworks and tools used for commercial software. Finally, we perform black-box automated testing of the Google Calendar component by applying several different testing technologies and frameworks. We use JUnit, Selenium and Mockito frameworks to create 22 tests to perform interface and functionality testing.

1. INTRODUCTION

Testing is one of the most important and crucial parts in developing any kind of software [Ammann and Offutt 2016]. Approximately half the budget spent on the development of the software projects is spent on software testing [Harman et al. 2015]. To create quality software (reliable, secure, etc.) means to test the software from different aspects and to create optimal combination of different coverage criteria because of the deadlines and system size [Rojas et al. 2015]. The purpose of the test team is to provide effectively and efficiently accurate and useful testing services and quality information about the project [Black 2016].

One of the common software testing categorization is: black box testing, white box testing and gray box (combination of black box and white box) [Jan et al. 2016]. If the software testers/developers have the software source code then they can use both black box and white box testing, but if they do not have the internal source code, then, only a black box testing is possible. Usually, black box testing, also called a functional testing, is performed by using the software documentation and design specification [Nidhra and Dondeti 2012].

From another point of view, the testing can be divided into automated and manual testing. In a case of manual testing, the software testers have the role of end users and they are responsible for checking the behavior of software by following a test plan and a set of test cases. Automated testing is performed by using a specific testing software framework that executes the tests and compares the actual outputs with the expected outputs [Garousi and Mäntylä 2016].

Software testing includes different level of testing such as unit, integration, system, and acceptance testing. Unit and integration testing focus on individual modules, while system and acceptance testing focus on overall behavior of the system [Dhir and Kumar 2019].

This work is supported by the Faculty of Computer Science and Engineering, Skopje, North Macedonia.
Bojan Popov, Bojana Koteska, Anastas Mishev, FCSE, Rugjer Boshkovikj 16, P.O. Box 393 1000, Skopje; email: bojan.popov@students.finki.ukim.mk; bojana.koteska@finki.ukim.mk; anastas.mishev@finki.ukim.mk.

Copyright © 2019 for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).
In this paper we make an overview of the current trends in software testing, including methods, practices and frameworks. As a case study we perform black-box testing of the Google Calendar service [Google 2006] by using different testing technologies and frameworks (JUnit, Selenium and Mockito).

The paper is organized as follows. Section 2 gives an overview of the current trends in software testing. Section 3 provides a comprehensive explanation of the testing methodology, definition of test cases, test generation and design the results. The final Section 4 concludes the paper.

2. RECENT TRENDS IN SOFTWARE TESTING

Agile testing is a trend in software testing which follows the rules of agile development policy. It considers software improvement as a critical part like a client in the testing process. In agile development the code is written during each iteration and the testing is done after each iteration. In [Dhir and Kumar 2019], the authors propose a model for automated agile testing and as an experiment they perform testing by using the Selenium tool [Jason Huggins 2004] which is an automating web application testing framework. Selenium is probably the most widely-used open source solution for testing web applications [Gundecha 2012]. The Selenium WebDriver provides opportunity to create custom automation framework that could reduce development time, increase the return of investments and to minimize the risks [Vila et al. 2017]. Another tool interesting for the Agile market is Testsigma [inc 2019]. It is one of the best automation tools which is AI-driven and automates complex tests using simple English and no programming.

Running unit tests in parallel and distributed environments can significantly speed up the time required for test execution. A tool for automatic execution in distributed environments CUT (Cloud Unit Testing) is presented in [Gambi et al. 2017]. Based on the set of unit tests, this program allocates computational resources such as virtual machines or containers and schedules the execution of the unit tests over them. The tool is implemented in Java and it monitors its progress directly inside the JUnit [The JUnit Team 2017] test automation framework which from the developers’ point of view the execution of the tests looks same like they are executing locally. JUnit platform is used as a foundation for launching tests on the Java Virtual Machine (JVM).

Automated program repair (APR) techniques became very popular in the recent years [Wang et al. 2019]. They have shown to be promising in increasing the effectiveness of automated debugging [Adamsen et al. 2017]. The general idea of these techniques is to provide automatic software repair and to produce fix of the programs which needs to be validated by software testers. The benefit of these techniques is that they fix the software and they decrease the effort to identify and correct faults [Gazzola et al. 2017].

The elimination of unnecessary test cases or selection of specific test cases can significantly reduce the time for testing. The goal of the test suite minimization is to distinguish repetitive experiments and to eliminate the redundant test cases. Test case selection decreases the number of test cases to be executed by recognizing the important test cases (test cases related to the latest changes of the software). Lately, test case prioritization is becoming trend in software testing the testing is performed by an early optimization based on proffered properties. It means that the test cases that are considered as highly significant will be executed first and will provide feedback to the testers earlier [Khatibsyarbini et al. 2018]. In [Azizi and Do 2018], the authors propose an item-based collaborative filtering recommending system that uses user interaction data and application change history information to develop a test case prioritization technique. A multi-objective optimisation technique is used to analyse the trade-off between the code coverage and execution time of the test suite written by programmers who are experts in testing [Turner et al. 2016].

Testing of an object-oriented code can be difficult because of the dependency objects. When dependency objects occur, then software testers can mock objects and simulate the complex dependency.
Mocking is a common approach in unit testing which isolates a class from its dependencies by replacing a dependency class instead of the original one. Authors in [Arcuri et al. 2017] create mock objects using the popular Mockito framework [Faber 2008]. Mockito framework is used for effective unit testing of JAVA applications. It is used to mock interfaces so that a dummy functionality can be added to a mock interface. It can simulate a method call and return result as needed. It is a very powerful tool for mocking database objects, and some functionalities that require time and acceptance from other users of the system. Mockito is among the top 10 most used Java libraries hosted on GitHub [Arcuri et al. 2017].

3. TESTING OF THE GOOGLE CALENDAR SERVICE

Different testing methods have been found in the literature for testing Google services. For example, the Google Map search application was tested by performing an improved random testing using some predefined values also used for verifying specific properties. The framework used for testing was composed of several testers which control and monitor the test execution [Salva and Laurencot 2009]. Brown et al. [Brown et al. 2018] propose a metamorphic testing strategy to test the Google Maps mobile app navigation. They tested its web service API and its graphical user interface and detected several real-life bugs in the Google Maps app.

In [Carlini et al. 2012], the authors analyzed 100 Google Chrome extensions, including the 50 most popular extensions to determine whether Chrome’s security mechanisms successfully prevent or mitigate extension vulnerabilities. They find that 40 extensions contain at least one type of vulnerability. The testing was performed by using a three-step security review process: black box testing, source code analysis and holistic testing.

3.1 Definition of Test Cases

Google Calendar provide multiple options: to create public events and share with other people; to create private events; to create default calendars or to add calendars created by other users; to delete events; to change date, time and color of the specific event, etc. We test the basic functionalities and some of the additional functionalities that Google Calendar has. The purpose of the testing is to find the suitable testing frameworks and to check if both the interface and functionalities of Google Calendar service work properly.

The following test cases were defined as a part of the interface testing:

(1) Counting events in a specific calendar view;
(2) Change the view of the calendar and count the days that are displayed depending on the view;
(3) Deactivate and activate additional calendars (for example: display of holidays - religious, national);
(4) Click the back and forward buttons to change day/week/month/year.

The functional testing covers the following test scenarios:

(1) Create an event;
(2) Search events;
(3) Delete event;
(4) User login;
(5) User logout;
(6) Edit event (change time, color, date, etc.)
3.2 Testing Frameworks

To perform the testing of the Google Calendar we use three different testing frameworks. All tests are written in Java 8, JavaSE-1.8 [Oracle 2014].

The core framework we use for testing is JUnit. Next, we use Selenium to test the web pages and the last one is Mockito, which simulates responses of some Google Calendar functionalities. The capabilities of these frameworks are described in Section 2.

3.2.1 JUnit. Testing of the Google Calendar service is made by using JUnit5 which requires minimum Java version 8. All methods annotated with @Test are considered as tests in JUnit.

Figure 1 represents a simple test written in JUnit 5. The purpose of this test is to check the sign-in button click on the Google home page (www.google.com). After that, a new page should be loaded which provides user login.

```java
@Test
dpublic void testaa() throws InterruptedException {
    String[] return login string split = c.clickLogin().split("\n");
    assertTrue((return login string split[0] + " " + return login string split[1])
        .equals("Sign in with your Google Account"));
}
```

Fig. 1. JUnit test

3.2.2 Selenium. A Selenium WebDriver API and Firefox web driver were used to perform testing of the Google Calendar web pages. Selenium WebDriver object provides access to an HTML page. Some of the basic functions of Selenium can be seen in Figure 2. The sleep method allows specific waiting time for the page loading. The web driver object allows users to access the HTML page web elements, but it is important to specify the XPath or ID of the web element.

```java
dpublic String clicklogin() throws InterruptedException {
    Thread.sleep(1500);
    driver.findElement(By.id("gb_70")).click();
    Thread.sleep(1500);
    return driver.findElement(By.xpath("/div[@jsname="tJHJj"]"))\getText().trim();
}
```

Fig. 2. Using Selenium WebDriver to find specific HTML web element

3.2.3 Mockito. Figure 3 is an example of how Mockito works with JUnit. A Dao class (DaoCalendar) is mocked and the mocked object is passed to the Dao service class (DaoServiceCalendar).

3.3 Test Creation and Execution

A total of 22 tests were defined and each functionality was tested separately. We organized the tests so that the login test is first. Second, we test the calendar functionalities, and log out process is tested last. The goal of the first login test is to click the login button on the Google home page, then to enter the email address and password and finally to click the sign-in button. These tests are shown in Fig. 4.

The first test (testaa) tests if the page loads correctly after clicking the sign-in button.
Fig. 3. Mockito usage in JUnit test

```java
@Test
public void testaa() throws InterruptedException {
    dao = Mockito.mock(DAOServiceCalendar.class);
    DAOServiceCalendar daoService = new DAOServiceCalendar(dao);
    DAOServiceCalendar spy = Mockito.spy(daoService);
    Mockito.doNothing().when(spy).addCalendar(Mockito.isA(String.class));
    Mockito.when(dao.returnNumberOfCalendars()).thenReturn(6);
    dao.addCalendar(somebody.getEmail.com);
    assertEquals(6, dao.returnNumberOfCalendars());
}
```

Fig. 4. Login tests

```java
@Test
d public void testab() throws InterruptedException {
    String email = "your email here!
    String string = c.enterEmailAddress(email);
    String[] return_password_string_split = string.split(\n);
    assertEquals((return_password_string_split[0] + " + return_password_string_split[1]).equals("Welcome + email));
}
```

```java
@Test
public void testac() throws InterruptedException {
    String password = "your password here!";
    assertEquals(c, enterPassword(password).isEnabled());
}
```

Fig. 5. Clicking login button

The function `clickLogin` from the test `testaa` is shown in Fig. 5. Three seconds are given for loading (1500 ms for loading the Google home page and extra 1500 ms after clicking the sign-in button), from which we take the text that gives feedback of the entered sign-in page.

Figure 6 provides details about `enterEmailAddress` (testab) and `enterPassword` (testac) functions.

The test `testab` first enters the email address by calling the `enterEmailAddress` function which clicks the "next" button and gets the text for the password form.
The test testac calls the method enterPassword which enters the password, clicks the "sign-in" button and gets the text for successful login.

After the login, the URL is changed to the URL of Google Calendar, from where the calendar functionality testing starts.

To test change of a view, we created parameterized tests that provide testing with multiple inputs. There are several options for each view and each of these options was tested separately. The simulation of button clicks and the click responses were also considered in multiple tests and their goal was to detect a page change.

In order to test the creation of an event, the number of existing events in the calendar was checked, and after a new event was created, this number was checked again. The tests for this part are shown in Fig. 7.

The first test testbf gets the number of existing events before creating a new event. The goal in this test is to click on the calendar form for creating events and click on more details. Then, the second test testbg enters the event parameters (date, start time, end time, name) and it checks the number of notifications about the new event. If everything is performed correctly, the result should be 1.

The last test testbh enters the additional event parameters such as color, it clicks the "save" button and checks the number of events again.

The delete event functionality is tested by using the search bar provided by Google Calendar. Fig. 8 shows the test for deleting an event. It counts the number of events before and after the event is deleted. The removeEvent function first searches the event, gets all the events with the searched name, clicks the first event and then deletes the event by clicking the "delete event" button.

Furthermore, there are functionalities in Google Calendar that wait for a certain response and these parts were tested with Mockito framework. We simulated that a certain response was sent, but in the meantime, we tested if the button was clickable.

The test file was written in a way so that the tests were defined as simple method calls and JUnit methods (assertTrue or assertFalse) were used for the evaluation of the correctness of the results.

```
* function that enters the given email into the web page field for email[]
public String enterEmailAddress(String email) throws InterruptedException {
    driver.findElement(By.id("identifierId")).sendKeys(email);
    Thread.sleep(1500);
    driver.findElement(By.id("identifierNext")).click();
    Thread.sleep(1500);
    return driver.findElement(By.xpath("//div[@jsname="tJHi"]")).getText();
}

* function that enters the password in the web page field for password[]
public WebElement enterPassword(String password) throws InterruptedException{
    driver.findElement(By.xpath("//input[@jsname="YPqjb"]")).sendKeys(password);
    Thread.sleep(1500);
    driver.findElement(By.id("passwordNext")).click();
    Thread.sleep(1500);
    return driver.findElement(By.xpath("//a[@title="Google Account: Bojan Popov \n" + "(bojanpopovred@gmail.com)\n"]"));
}
```

Fig. 6. Entering email and password functions
The test class contains a service class which is responsible for the button clicks, for the searching through the web page elements, for creating new events, deleting events, etc.

There is also one more service class which is responsible for changing the dates. Since each month has a specific number of days we used the Java Calendar class which returns the number of days in a non leap year.

Table I shows the results of the executed tests. As it is shown, the only test that failed is "Testing calendar days after changing a view". Errors occurred in such occasions, mostly in the part of deleting an already created event. Sometimes the event was not deleted from the calendar, but when that test was executed no failure was detected. The conclusion after a couple of tests executed was that Google Calendar does not load fast and some of the page elements are not fully loaded at the time of the test execution which was the case with the failed test also.

<table>
<thead>
<tr>
<th>Tests</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log in</td>
<td>pass</td>
</tr>
<tr>
<td>Testing calendar days after change of views</td>
<td>failure</td>
</tr>
<tr>
<td>Creating an event</td>
<td>pass</td>
</tr>
<tr>
<td>Editing an event</td>
<td>pass</td>
</tr>
<tr>
<td>Deleting an event</td>
<td>pass</td>
</tr>
<tr>
<td>Log out</td>
<td>pass</td>
</tr>
</tbody>
</table>
It is important to mention that Google constantly releases new updates of Google Calendar with changed interface. It means that HTML elements, which are important in Selenium, have different IDs and class names. The interface changes caused some of the tests to fail and we had to change them multiple times. Table II shows the execution results after the Google calendar update.

<table>
<thead>
<tr>
<th>Tests</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Log in</td>
<td>pass</td>
</tr>
<tr>
<td>Changing view</td>
<td>error</td>
</tr>
<tr>
<td>Testing calendar days after change of views</td>
<td>failure</td>
</tr>
<tr>
<td>Creating an event</td>
<td>pass</td>
</tr>
<tr>
<td>Editing an event</td>
<td>pass</td>
</tr>
<tr>
<td>Deleting an event</td>
<td>pass</td>
</tr>
<tr>
<td>Log out</td>
<td>pass</td>
</tr>
</tbody>
</table>

4. CONCLUSION

In this paper we provided an overview of some of the new technologies and practices in software testing and we applied them in practice by making black-box testing of the Google Calendar component. The testing was made by using some of the popular testing frameworks such as JUnit, Selenium and Mockito. A total of 22 tests were created to test the interface and the functionalities of the Google Calendar.

In conclusion, we believe that good software testing requires a lot of practice and constant monitoring of world trends in the field of software engineering. Testers should make a test strategy and find a balance between code coverage and time required for testing. There are many useful testing tools that can automate and ease the process of testing. Some of them do not even require programming skills.

From the testing process of the Google Calendar service, we learned that an integration of more testing frameworks and test automation can be done fast and with small effort, but it requires field knowledge. JUnit provides user friendly way to automate the test execution by creating test suites. The challenging task is to define the test scenarios and to find which tools and techniques are more suitable.

REFERENCES


Paul Ammann and Jeff Offutt. 2016. Introduction to software testing. Cambridge University Press.


Recent Trends in Software Testing – A Case Study with Google Calendar


Detecting Source Code Similarity Using Compression

IVAN PRIBELA, GORDANA RAKIĆ and ZORAN BUDIMAC, University of Novi Sad

Different forms of plagiarism make fair assessment of student assignments more difficult. Source code plagiarisms pose a significant challenge especially for automated assessment systems aimed for students’ programming solutions. Different automated assessment systems employ different text or source code similarity detection tools, and all of these tools have their advantages and disadvantages. In this paper we revitalize the idea of similarity detection based on string complexity and compression. We slightly adapt an existing, third party, approach, implement it and evaluate its potential on synthetically generated cases and on a small set of real student solutions. On synthetic cases we showed that average deviation (in absolute values) from the expected similarity is less than 1% (0.94%). On the real life examples of student programming solutions we compare our results with those of two established tools. The average difference is around 18.1% and 11.6%, while the average difference between those two tools is 10.8%. However, the results of all three tools follow the same trend. Finally, a deviation in some extent is expected as observed tools apply different approaches that are sensitive to other factors of similarities. Gained results additionally demonstrate open challenges in the field.

1. INTRODUCTION

Automatic assessment systems encounter significant problems when assessing students’ solutions to programming assignments. Main difficulties arise from lack of academic discipline among the students. While the students are expected to submit their original solutions, sometimes this is not the case. The assignments that they are expected to solve are usually very similar, and there is a restricted set of varieties among the correct solutions. Therefore, plagiarism detection is always an open question in the assessment of students’ work.

Multiple tools and approaches are available for plagiarism detection. They are based on source code similarity detection algorithms or code clone detection tools. Available approaches rely on lexical, syntactic, structural, or semantic information about the programming solution. Thus, algorithms analyze character streams, token streams, controls structures or dependencies in the observed code. Usually, they are applied to the source code or some of its intermediate representations in the form of an intermediate language, a tree or a graph. All of these similarity analysis approaches have their advantages and disadvantages. Their sensitivity to different categories of similarities and precision varies and therefore their applicability and usefulness depend on objectives of the analysis [Novak et al. 2019].

Although high similarity between two projects does not mean that one is plagiarized from the other, it is often a good indicator. The students that are submitting work of other students usually make...
only small superficial adjustments. They often do not change the structure of the code but only rename variables and functions or reorder statements that they know will not change the program correctness.

In this paper we describe an approach to similarity analysis based on string complexity and compression. We have chosen this method because the initial study presented in [Chen et al. 2004] showed promising results and a fresh perspective on settled practices, but the system is not available anymore for use or extension.

Other approaches to measuring similarity and detecting plagiarism were concentrated around feature comparison and program structure matching. Early attempts are usually based on computing different software metrics and comparing the results. More recent tools usually use a form of greedy string tiling to compare two streams of tokens instead a summary indicator. Both types of approaches have their strengths and weaknesses with attribute counting systems performing better in the case of very small changes, and structure comparison systems exhibiting better performance when only a part of the code is copied.

In section 2, we describe the background and motivation for our work, that lays in the previous third-party solution [Chen et al. 2004] that introduced this approach and showed perspective but is not available anymore. Description of our approach and our contribution to the previous results can be found in the section 3. Results of our contribution, demonstrated on both artificial cases and real-life student solutions, are described in the section 4, followed by conclusions and the future work in section 5.

2. BACKGROUND AND RELATED WORK

Kolmogorov complexity $K(s)$ of a string $s$ [Zvonkin and Levin 1970] is the length of the shortest computer program that with no given input produces the string $s$ as the output. It can be considered as a measure of the amount of information contained in the string $s$.

The conditional Kolmogorov complexity $K(s|t)$ of two strings $s$ and $t$ [Zvonkin and Levin 1970] is the length of the shortest computer program that produces the string $s$ as the output with the string $t$ given as an input. It can be viewed as a measure of the amount of information contained in the string $s$ that is not present in the string $t$.

The chain Kolmogorov complexity $K(s, t)$ of two strings $s$ and $t$ [Zvonkin and Levin 1970] is the length of the shortest computer program that produces the string $s$ followed by the string $t$ with the information on how to separate them.

The chain rule for Kolmogorov complexity [Li et al. 2008] states that $K(s, t) \leq K(s) + K(t|s) + O(\log(K(s, t)))$ or $K(s, t) \approx K(s) + K(t|s)$. In other words, the length of the shortest computer program that produces the string $s$ followed by the string $t$ is approximately the same as the length of the shortest computer program that produces the string $s$ plus the length of the shortest computer program that produces the string $t$ having the string $s$ as input.

The algorithmic mutual information for Kolmogorov complexity $I_K(s, t)$ is defined as $I_K(s, t) = K(s) - K(s|t)$ and represents the amount of information about the string $s$ that is contained in the string $t$. This quantity is symmetric up to a logarithmic factor as shown in [Gács 1974] which gives us $I_K(s, t) \approx I_K(t, s)$.

Having all this in mind we get $K(s, t) \approx K(s) + K(t|s) \approx K(t) + K(s|t) \approx K(t, s)$.

If we take the absolute amount of mutual information in the strings $s$ and $t$ and divide it by total amount of information in both strings as in formula 1 we get a normalized measure of similarity between the strings $s$ and $t$.

$$sim(s, t) = \frac{I_K(s, t)}{K(s, t)}$$
If the strings \( s \) and \( t \) are identical, this will be equal to one, if the two string are completely unrelated, the value will be zero. However \( \text{sim} \) function is not a properly defined metric function. On the other hand, the function \( d_{\text{sim}}(s, t) = 1 - \text{sim}(s, t) \) satisfies all the necessary conditions for a metric up to a logarithmic error as shown in [Li et al. 2001].

Unfortunately, Kolmogorov complexity is not computable. It is not possible to compute even the lower bound [Chaitin 1974]. However, a reasonable upper bound can be expressed as the length of the compressed representation of the string as produced by a reasonable compression algorithm. This is the approach taken in a tool called SID [Chen et al. 2004] that employs algorithmic complexity to check for similarities in source code. As approximation for this value, a special compression algorithm called TokenCompress is used. However, this tool is no longer available for usage, which is one of the motivating factors behind our work.

3. DESCRIPTION OF THE APPROACH

Our tool is also based on measuring Kolmogorov complexity, or more accurately approximating it using compression similarly to the work done in [Chen et al. 2004].

A compressed version of the string can be viewed as a program written in a language executed by the corresponding decompressor. Interpreted by the decompressor, the output of such program is the original string. Also such program tends to be the shortest as the main goal of a compression algorithm is to produce the shortest possible sequence that decompresses into the original input.

As the original compression algorithm described in [Chen et al. 2004] is not available, we have chosen GNU gzip library to approximate \( K(s) \). This library uses Lempel-Ziv coding (LZ77) [Ziv and Lempel 1977] and Huffman coding [Huffman 1952], and is considered the industry standard for general compression with a good balance of time and compression ratio. We define \( |s| \) as the length in bytes of the compressed form of the string \( s \) when compressed using the mentioned library. As stated earlier, the value \( |s| \) is a reasonable approximation for \( K(s) \). Similarly, \( K(s, t) \) is approximated by \( |st| \) and \( K(s|t) \) is approximated using the formula \( K(s|t) \approx K(t, s) - K(t) \) based on the chain rule for Kolmogorov complexity.

Having this in mind, the final similarity is calculated using the formula 2.

\[
\text{sim}(s, t) = \frac{|s| + |t| - |ts|}{|st|} \quad (2)
\]

In order to calculate the percentage of the common information we have to further interpret this similarity metric, as was done in [Chen et al. 2004]. If we assume that the two strings \( s \) and \( t \) are approximately the same length as is common among student solutions, then \( \text{sim}(s, t) = \frac{p - 1}{2} \) where \( p \) is the percentage of the information common in both strings [Chen et al. 2002]. This gives us the formula 3 to calculate \( p \).

\[
p = \frac{2\text{sim}(s, t)}{1 + \text{sim}(s, t)} \quad (3)
\]

However, when students try to hide similarities, one of the methods they employ is adding redundant or unused code thus increasing the total length of the copy. In such cases, when compared with each other, the copy and the original can have considerable length difference.

To combat this we have proposed two new measures in addition to the one already described in [Chen et al. 2004]. These measures represent the percentage of string \( s \) that is present in \( t \) and vice versa, and are given with formulas 4 and 5.
\begin{align*}
\text{sim}_s(s, t) &= \frac{|s| + |t| - |ts|}{|s|} \quad (4) \\
\text{sim}_t(s, t) &= \frac{|s| + |t| - |ts|}{|t|} \quad (5)
\end{align*}

These two measures are not always symmetrical. For example, if we have a string $s$ representing the source code form one student solution, and string $t$ generated from $s$ by adding two times more unused code, then $\text{sim}(s, t)$ will not give a strong indication about possible plagiarism as the percentage will be around 50%. However, $\text{sim}_s(s, t)$ will give the percentage around 100% immediately stating that one string is present in the other in its entirety. Similar situation can be observed when a part of one solution is copied from the other without adding or modifying it much.

4. RESULTS

To explore the viability of our proposed approach, we tested the performance of our tool both in isolation and in comparison with two other similarity detection tools.

4.1 Synthetic cases

In the first step we tested our tool using synthetic benchmarks. We have performed 22 separate tests by supplying two randomly generated strings as input to our tool and recording the calculated similarities. Each test was repeated 1000 times and the average similarity was recorded. In each test we generated three sequences of random data of predefined lengths for that test: unique content for the first input string, unique content for the second input string and content common for both strings. The two inputs supplied to the tool were generated by combining the content from the appropriate random sequences.

The results of the tests are given in Table I. The value $\text{unq}(s)$ represents the total amount of data unique to the string $s$, while $\text{unq}(t)$ represent the same for the string $t$. The value $\text{cmn}(s, t)$ represents the total amount of data common in both strings. The next two groups of values represent the expected and calculated percentages of similarity, while the last three columns contain percentage of difference between expected and calculated values. The value $p$ represent the percentage of data common to both strings. The value $p_s$ represent the percentage of data in the string $s$ that is also present in the string $t$. The value $p_t$ represent the percentage of data in the string $t$ that is also present in the string $s$.

As can be seen from the data, the similarity calculated by our tool is very close to the real values. More precisely, values of all three functions differ from the expected value within an interval between 0.04% and 2.69%. It can be observed that the calculated values differ from the expected ones on average 0.94% for both strings, 1.1% for the amount of data $s$ has in common with $t$, and 0.85% for the amount of data $t$ has in common with $s$, while standard deviation of differences is 0.59%, 0.69% and 0.57% respectively. The retrieved values are consistently slightly lower because of the imperfections of the compression algorithm used to approximate the Kolmogorov complexity. With a compression algorithm closer to the perfect compression these values would probably be closer to the expected.

4.2 Real world examples

In the second step we have compared the results from our tool with SIM [Grune 2006] and JPlag [Prechelt et al. 2002]. SIM tests lexical similarity in natural language texts and in programs written in various programming languages. Its main purpose is to detect duplicated code in large software projects and to detect plagiarism between different software projects. JPlag is a system that finds similarities among multiple sets of source code files. It is aimed at detecting software plagiarism and as such was used by expert witnesses in several intellectual property cases.
As sample data, we have selected a few student solutions to practical exercises. All solutions belong to a single group of second year students working independently on the same assignment. The students were not sharing code among them. Along with student solutions one produced by us was added. We have randomly chosen one of the student solutions and changed all variable names without changing the rest of the code. This was done to illustrate a common way students try to mask plagiarism.

The results of this test are illustrated by charts in figures 1 and 2. From the starting 14 solutions, all possible pairs were generated. Figure 1 shows 182 points in total representing each pair twice, once for each component in the pair. For each of these components, the figure shows the percentage of the code that is also present in the other component of the pair, as reported by all three tools. Figure 2 contains the similarity values reported for the pair as a whole. As SIM does not report this measure, it is not included in the graph on this figure.

The results from our tool are comparable with the results from SIM and JPlag. As can be seen from the charts, percentages are roughly consistent within a margin, both for a single solution and for the whole pair.

There are a few instances in which each of the tools give a slightly different result from the others. The discrepancies between our tool and each of the other two are roughly the same as differences between them and all three tools are showing the same trend. Anyhow, in no situation one of the tools gives a strong indication of similarity while the other two are indicating the opposite.

However, our tool usually gives a slightly higher percentage of similarity. This is rooted in two fundamental reasons. Firstly, both SIM and JPlag have a concept of a minimum run length. This parameter dictates how many consecutive matches are needed to consider a part of the program code a potential copy. Our tool does not have such restriction stated explicitly. However, implicitly as a feature of standard compression algorithms, a match that is too small is disregarded on the basis that its compressed representation is longer than the uncompressed.
Secondly, our tool is basing the comparison on the level of characters, while SIM and JPlag are comparing token streams. This results in our tool detecting finer similarities that do not span whole tokens, like for example, between Integer, LongInteger and ShortInteger or by capturing a part of a token in front or behind the group detected by other tools.

This does not mean that either of the approaches is better that the other. It depends on the particular case and warrants further investigation.

5. CONCLUSIONS AND FUTURE WORK

In the area of automatic assessment of students’ programming solutions, one of the open challenges is plagiarism detection. While there are numerous tools for source code similarity analysis and for code clone detection, there is still no tool fully suitable to be used for these purposes. Some of the factors that affect the usefulness of these tools are the applied approach and the used similarity algorithm. In this paper we explore possibilities of employing string complexity and compression in source code similarity measurement. We evaluate gained results on synthetic examples in comparison with the expected values based on the background theory, as well as on real-life student solutions in comparison with alternative tools.

In comparison with the expected values, we gain slightly lower similarity values. The cause of this is the imperfection of the chosen compression algorithm, and the fact that it implements lossless compression which considers the equality of string sequences and not actually their similarity. Hence, in
the future work we can experiment with alternative algorithms that are lossy and sacrifice a little accuracy to, for example, better capture attempts at renaming variables.

Results obtained from our tool, when compared to the similarity values acquired from competitive tools, show the same general trend, although our calculated similarity values tend to be slightly higher. The differences are caused by different observation unit employed (character- versus token-based observation). Experimentation in this direction may bring higher precision, but also lower detection capabilities.

Also, one of the important features in similarity analysis is the consideration of the base code, as student assignments usually include some predefined elements for the future solution. This is another significant direction for future work.

REFERENCES
Process Evaluation and Improvement: A Case Study of The Loan Approval Process

MAJA PUŠNIK, KATJA KOUS, ANDREJ GODEC and BOŠTJAN ŠUMAK, University of Maribor,

Quality assurance in software development is one of the key processes in any organization, where Information Technology systems impact the realization of business processes significantly. For digitalization and informatization of business processes, organizations need to have a clear notion of their processes, which can be achieved by focusing on the evaluation, optimization, and continuous improvement of business processes, in addition to supporting software quality. The research addresses the problems of the financial business sector. More specifically, the frequently performed loan approval process and its existing information solution support. Reports have been made about existing risks, unexpected fallouts, resources wasted in the loan approval process and lack of quality information support. Therefore, the research focuses primarily on the possibilities to optimize the business process by analyzing and evaluating the process activities (measuring the quality of existing software, introducing new information system support and risk management solutions, as well as identification of optimization potentials, where possible). The current state is modeled, simulated, and evaluated (according to a literature review and interview results). Finally, the possibilities of optimization are proposed, and the potential effect on the quality of the loan approval process.

1. INTRODUCTION

Measuring the quality of processes and their supporting software, as well as the constant process of optimization, are important elements of every business organization’s success, keeping its processes free of waste, optimizing time and cost, as well as achieving optimal values of different Key Performance parameters or Indicators (known as KPIs). Since businesses rely on Information Solution (IS) support, it is important to provide qualitative solutions that fit the user’s expectations, are well accepted and provide a positive user experience. In addition, management and quality assurance of processes is a growing issue in companies that rely on several (often unconnected) IS, and the modeling approach affects the understanding of the IS role and supports quality assessment of IS based process. In addition to the growth of users and their demands, there is also an escalation of providers producing several challengers among Information Technology (IT) companies and their processes. Growing numbers (of devices, solutions as well as users), result in several unconnected solutions, causing complex integration of products, introducing chaos, uncertainty and risk in users’ (digital) lives.

Due to the importance and actuality of the subject, even more so in the past years since the economy suffered, efforts are dedicated to managing and anticipating risks. The importance of measuring several quality aspects (quality of business processes, based on IS), is also increasing. Reportedly, expected risks can be detected in a timely manner, and managed only by a systematic approach. Even more so in the present day, banks carry out various risk management activities and processes before decisions are being made (loans are issued, for example) performed manually, or...
supported by software solutions. As part of risk management, banking institutions reportedly use several approaches, some of them automated, however, not all, and there is potential for improvements. Information support includes different methods, most of them from the field of Operational Research, such as stochastic processes for determining risks, prices, guarantees, and delays’ evaluation (Duffie 2005). Among the used methods (as reported in academic research and interviews) are also linear discriminant analysis, logit analysis, logistic regression, classification, and regression trees (Vojtek and Kocenda 2006).

With the increase of Information Technology (hereinafter, IT) support and possibilities electronic banking provides, the majority of business processes are being redesigned and re-planned, with the inclusion of greater informatization and automation (Dias et al. 2019). However, those efforts vary among banks, as well as countries. Therefore, the aim of our research is an analysis of the quality of processes focused on (general) existing IT support (through literature review and interviews), resulting in a process optimization proposal. As part of the research, the business process is suggested to be transformed from the AS-IS model to the renewed TO-BE model. Therefore, due to lack of access to software solutions, the quality of listed information support will be evaluated on the process level.

The paper is organized into five sections. Definitions of business process optimization and different optimization approaches are presented in Section 2, while the preliminary research is described in Section 3. Section 4 presents the case study of the optimization of the loan approval process, including a general description of the process, as well as the simulation of the loan approval process with different optimization approaches. The last, Section 5, includes a summary of results and the conclusion, followed by listed literature.

2. BUSINESS PROCESS OPTIMIZATION

2.1 Definitions

A business process is defined as a collective set of tasks that, when connected and sequenced properly, perform a business operation (Vergidis, Dhish, and Tiwari 2012). The aim of a business process is to perform a business operation, i.e., any service-related operation that produces value to the organization. The design and management of business processes is a key factor for the operation of the organization. By focusing on the optimization and continuous improvement of business processes, organizations can improve their efficiency and quality, reduce cost, and enable adaptations to change requirements (Vergidis, Dhish, and Tiwari 2012). Therefore, business process optimization is a technique to help organizations improve their efficiency by improving their processes.

2.2 Process optimization approaches

Primarily, there are several known approaches to process optimization, respectively, transformation of a process (Pušnik et al. 2018), including Modeling, Simulation, KPI identification, the 5 why method, revision, Waste identification, the root cause analysis, Voice of a customer, process success measuring, value flow mapping, IS/IS-NOT method, critical path method, risk analysis method, a 5S work environment organization, and others. A preliminary survey, (Pušnik, Welzer Družovec, and Šumak 2019) conducted in 2019, ranked the mentioned methods from most to least useful. As shown in Fig.1, Modeling was recognized as the best possible approach, followed by Simulation and KPI identification. Voice of customer, Ishikawa diagram, and process success measuring were identified as the least useful optimization approaches.
Similar findings are also presented in an extended research (Čuček 2018), where a supplementary survey about optimization methods use was performed among IT companies, and coincides with similar research results in (Pušnik, Welzer Družovec, and Šumak 2019). The results in Table 1 show that Modeling, Simulation, and KPI identification were identified as the most useful approach/method for process optimization, followed by other approaches. All listed approaches/methods are described in the following subsection and used in the case study presented in Section 4.

Table 1: The list of methods and approaches, ordered according to the evaluation of their usefulness (Pušnik, Welzer Družovec, and Šumak 2019)

<table>
<thead>
<tr>
<th>Method/approach</th>
<th>1 – very useless</th>
<th>2 – useless</th>
<th>3 – neutral</th>
<th>4 – useful</th>
<th>5 – very useful</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeling</td>
<td>0%</td>
<td>0%</td>
<td>0%</td>
<td>42%</td>
<td>58%</td>
</tr>
<tr>
<td>KPI identification</td>
<td>0%</td>
<td>0%</td>
<td>4%</td>
<td>50%</td>
<td>46%</td>
</tr>
<tr>
<td>Simulation</td>
<td>4%</td>
<td>8%</td>
<td>21%</td>
<td>33%</td>
<td>33%</td>
</tr>
<tr>
<td>Ishikawa diagram</td>
<td>0%</td>
<td>17%</td>
<td>38%</td>
<td>29%</td>
<td>17%</td>
</tr>
</tbody>
</table>

**Process Modeling**

Process modeling is a method for representation of processes in a process model. This is usually done through different graphing techniques, such as the flowchart, IDEF, Unified Modeling Language (UML), Petri nets, Business process models based on mathematical or algorithmic models, etc. (Kostas, Tiwari, and Majeed 2008). The de-facto Standard for business processes diagrams is the Business Process Modeling and Notation (BPMN) developed by the Object Management Group. BPMN is a graphical representation for specifying business processes in a business process model based on a flowcharting technique. The primary goal of BPMN is to provide a notation that is readily understandable by different business roles (Object Management Group 2019). Therefore, from the business analysts that create the initial drafts of the processes to the technical developers
responsible for implementing the technology that will perform those processes, and also, to the business people who will manage and monitor those processes (Object Management Group 2019).

**Process Simulation**

Simulation is the process of creating and analyzing a digital prototype of a physical model to predict its performance in the real world. Simulation modeling is used to help designers and engineers understand whether and under what conditions a process could fail and what loads it can withstand. The act of simulating a complex system needs the support of an IT solution, such as Signavio, Enterprise Architect, or others. (Pušnik, Welzer Družovec, and Šumak 2019)

**KPI identification**

KPI identification is the most known method in measuring business process quality (Pušnik, Welzer Družovec, and Šumak 2019). KPIs are performance metrics that measure specific goals for businesses’ processes. Based on the identified risks and possible problems in the process, indicators are defined to help measure the success or effectiveness of the process (usually numerical values, such as time, cost, profit, number of complaints, number of rejections, etc.). Each KPI includes a definition, how is it measured, and when is it successful. KPIs express how to increase efficiency, representing a multitude of measurements that focus on the aspect of organizational performance that is most critical to the current and continued success of the business process (Pušnik, Welzer Družovec, and Šumak 2019).

**Ishikawa diagram**

Ishikawa diagrams were popularized in the 1960s by Kaoru Ishikawa, who pioneered quality management processes in the Kawasaki shipyards, and, in the process, became one of the founding fathers of modern management. The Ishikawa diagram (also known as fishbone diagrams, herringbone diagrams, cause-and-effect diagrams, or Fishikawa) is a tool used for identifying and presenting all possible causes of a particular problem in graphical format systematically. Usually, it can be made using the following steps:

1. Identify the problem
2. Work out the major factors involved (8M):
   - **People or Man power** – causes, caused by people
   - **Methods** - causes caused by rules, regulation, legislation or standards
   - **Machines** – causes, caused by equipment such as machinery, computers, tolls
   - **Materials** – causes, caused by a defect or material properties
   - **Measurements** – causes, caused by improper or poorly chosen measurement
   - **Environment (Mother nature)** – causes, caused by the environment - temperature, humidity or the culture
   - **Management** – causes, caused by improper management
   - **Maintenance** – causes, caused by improper maintenance
3. Identify possible causes, and
4. Analyse the diagram.

Despite the lower grade of the technique when performing the survey in Table 1, the Ishikawa diagram was, nevertheless, chosen due to its graphical presentation and clarity of potential problems. Fig. 2 illustrates the general structure of the Ishikawa diagram.
3. PRELIMINARY RESEARCH: INTERVIEW

The preliminary research was carried out with an interview and survey targeting banks and financial institutions. Most banks declined the received collaboration request. Therefore, the difficulty of gaining data was evident. Due to the very poor responsiveness, the interview was conducted with two employees from one bank (one in person, one through e-mails).

The summary of questions and answers is presented in Table 2 (both interviewees’ answers were the same). To point out question 3 (Q3), the inquiry was focused on which activities are computerized in the bank (e.g., informative calculation of loan, calculation of the interest rate and repayment period, financial verification of the client, deciding on the suitability of the loan applicant, digital signature of the contract and other). The results indicated that, despite some computerized activities, not all are completely automated, and there is a lot of human effort necessary. The results show that the loan approval process is partly automated; however, no optimization methods are used for evaluating or approving loans.

Based on the received information, the problem of non-optimized activities was recognized in the loan approval process. To investigate potential possibilities to create change, general examples of the loan approval process are presented in the following sections, in addition to a conducted case study.

Based on the identified setbacks of banks’ loan approval processes, a case study investigating the process’ possibilities was performed in the following Section.
Table 2: Interview structured questions.

<table>
<thead>
<tr>
<th>ID</th>
<th>Question</th>
<th>Answer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Does most of your work require the use of personal computers?</td>
<td>YES</td>
</tr>
<tr>
<td>Q2</td>
<td>Do you expect increased use of computers in the future?</td>
<td>YES</td>
</tr>
<tr>
<td>Q3</td>
<td>Are most of your activities computerized (loan approval process among them)?</td>
<td>SEVERAL, but not ALL and never COMPLETELY</td>
</tr>
<tr>
<td>Q4</td>
<td>How many loan applications are received via e-bank daily?</td>
<td>0-20</td>
</tr>
<tr>
<td>Q5</td>
<td>Are there more physical or electronic credit claims?</td>
<td>PHYSICAL</td>
</tr>
<tr>
<td>Q6</td>
<td>Does your bank offer customers mobile banking services?</td>
<td>YES</td>
</tr>
<tr>
<td>Q7</td>
<td>Can a customer apply for a loan on a mobile application?</td>
<td>NO</td>
</tr>
<tr>
<td>Q8</td>
<td>Does your bank have a business process for granting loans?</td>
<td>YES</td>
</tr>
<tr>
<td>Q9</td>
<td>Is there room to improve the existing business process of granting loans?</td>
<td>YES</td>
</tr>
<tr>
<td>Q10</td>
<td>Is the loan approval process automated?</td>
<td>PARTLY</td>
</tr>
<tr>
<td>Q11</td>
<td>What optimization methods do you use when approving loans?</td>
<td>NONE</td>
</tr>
<tr>
<td>Q12</td>
<td>Do analysts solve problems related to operating or optimization methods without IT support?</td>
<td>YES</td>
</tr>
</tbody>
</table>

4. CASE STUDY: OPTIMIZATION OF THE LOAN APPROVAL PROCESS

4.1 The business process of loan approval

The business process of loan approval is an integral part of a bank or institution. Each business process also includes a certain level of risk associated with credit scoring. Based on a literature review, the bank distinguishes between two groups of credit: Approval and rejection of a loan, which involves a risk assessment process. Before the loan granting, banks verify the individual user who would like to take out a loan (verification in Slovenia is carried out using the SISBON information system (Banka Slovenije 2019)). SISBON aims to exchange and process clients’ personal data between banks. The SISBON information system collects and processes data that relate to the actual and potential indebtedness and the fulfillment of contractual obligations of customers. Its mission is to manage the credit risk of banks, savings banks, and other creditors, to ensure responsible lending and avoid excessive indebtedness of individuals (Banka Slovenije 2019).

Once a loan is approved by a bank, an agreement between the bank and a customer is set, under terms which vary among banks, as well as countries. Credit risk is the possibility that money will not be returned, resulting in a financial loss. Credit risk from the point of view of a banking institution is the risk that the claim will not be settled within the specified time limits or under certain conditions by the debtor. There are several different types of credit risk, which are presented below (Anson et al. 2004):
(1) Default risk (PD): The primary type of risk based on the probability that the borrower will not be able to repay his claims. The risk of default by the borrower is determined primarily by its creditworthiness and the duration of the credit relationship. In order to calculate this risk, the bank needs an appropriate system for obtaining information and regular monitoring of the client. By extending the credit period, the probability of PD by the borrower increases.

(2) Loss Given Default (LGD): The bank will not be able to recover the debt, or that it will not be possible to repay losses from the sale of insurance instruments. In order to approve the loan, the bank requires adequate insurance with redemption, which is secured against possible default of the loan. Recovery of debt, or repayment in the case of the unpaid loan, depends on the quality of the insurance, which requires adequate monitoring of its liquidity.

(3) Exposure risk (EAD): Is caused by uncertainty in repayment. The Bank's exposure is divided into two types: Maximum and expected. The expected exposure is the expected loss of the bank in the event that the debtor fails to settle its obligations, while the maximum exposure is the maximum amount lost by the bank in the event of default by the debtor.

Different KPI’s were identified based on the risks (Table 3). The listed risks are a part of the “Evaluation of the credit loaner” activity in Fig. 3. In addition to risk evaluation, there are also KPIs, that are critical to the current and future success of the loan approval business process.

In our case study, most of the KPI’s cannot be simulated, nor did we receive sufficient information from the banks. However, basic time and cost constraints can, nevertheless, be included. Regarding the time constraint, we overviewed some possible wastes as well, connected mainly to excessive waiting (for documentation, approval or other non-optimized respectively non-automated activities).

<table>
<thead>
<tr>
<th>KPI</th>
<th>Name/Sim. appropriate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>KPI1</td>
<td>Level of difficulty to approve a loan/NO</td>
<td>Included in all activities, performed by the bank. KPI is focused on the speed of how quickly the bank succeeds in checking the conditions for granting a loan.</td>
</tr>
<tr>
<td>KPI2</td>
<td>Number of creditworthy clients/NO</td>
<td>Set to the “Evaluation of the credit loaner” activity. KPI represents the number of clients classified according to their creditworthiness. An indicator connected to the creditworthiness of a client is the actual percentage of clients who paid their credits successfully.</td>
</tr>
<tr>
<td>KPI3</td>
<td>Number of contracts/NO</td>
<td>An indicator of success representing how many contracts were set, which enables measuring the quality of the services to the process.</td>
</tr>
<tr>
<td>KPI4</td>
<td>Costs/YES</td>
<td>Process costs are built into the Signavio online environment to measure the costs within the process of granting credits.</td>
</tr>
<tr>
<td>KPI5</td>
<td>Time consumed/YES</td>
<td>Built into the Signavio web environment, enabling measuring of the time spent from loan request to loan approval. The tool allows you to measure the time in the credit approval process.</td>
</tr>
</tbody>
</table>
4.2 AS-IS model

In Fig. 3, a generalized credit approval model is presented, modeled using the BPMN notation in the Signavio web tool. The process contains two main roles: The bank and the customer/potential loaner. The process begins with the activity of selecting a client's bank, then the bank presents its information on loans, on the basis of which the customer decides to apply for a loan. When the bank accepts the application, it begins checking the data based on the documentation provided by the customer, and through the SISBON information system. After verifying the results and credit assessment, the bank prepares a Contract. The client signs the Contract. The signed Contract becomes valid, and the bank transfers funds to the client's account. The transfer of funds ends the process. The model and simulation of the process are presented in Fig. 3.

![Simplified BPMN credit approval process.](image)

The process analysis was carried out using the Signavio online tool. Through the web tool, we entered the information costs on the activities, or the number of costs that a given activity has. Information was obtained with the help of literature (Dermine 2014; Smartbizloans 2017) and the answers of interviews. Based on the interviews, some possible delays are presented on the Ishikawa diagram in Fig. 4. Although the mentioned diagram was not chosen as the most appropriate approach when optimizing a process, we nevertheless, included an example focusing on weak points of the process and possible sources of problems that need to be addressed in order to optimize (in our case simulate) the correct process activities.
To simulate the process with several instances, we observed the queue theory, a study of waiting lines in order to predict the length and waiting time (Howl 1966). The queueing theory is a mathematical study of waiting lines, or queues (Duffie 2005). A queueing model is constructed so that queue lengths and waiting time can be predicted (Duffie 2005). The queueing theory is generally considered a branch of operations research, because the results are often used when making business decisions about the resources needed to provide a service. The analysis of the theory has shown that an increase in the number of employees from a cost point of view is impractical, which led to an assessment of the reduction of process times in order to achieve the goal within the limits of satisfactory intervals. Therefore, one of the primary aims of the TO-BE model is time reduction.

4.3 TO-BE model

The TO-BE model is a proposition of the AS-IS model optimization, indicating the activities that need additional support (through automation or human interaction) to improve KPIs (especially time-related indicators). A business process transformation can be carried out in several ways. Some simple solutions (digitalization, for example), are already implemented within most banks; the renovation can also be carried out using methods based on the design of the product. The key factors are influenced by the restructuring of the business process: Time, quality, costs, flexibility. Only time and cost were evaluated through simulation.

The TO-BE model has to improve at least one of the factors (Schoenberg 2013). Using the heuristics of the implementation of the process, we can renew the process by removing tasks (deleting unnecessary activities), job scheduling (task aggregation), and parallelism. In order to redesign the business process, key variables of business process renewal must be considered, since an acceptable relationship between maintaining the integrity of the process and functionality should be balanced.
The activities that were proposed during the renovation are presented in Fig. 5. Activities with no added value, such as printing a credit agreement or handwritten signature, are replaced with informatization where possible. The changes in the model are shown with a different color of activities. Due to several non-automated activities, the simulation of the AS-IS process produced several bottlenecks, omitted in the TO-BE process. The comparison of the bottlenecks is presented in Table 4.

Table 4. The ratio of resource consumption during indicating bottlenecks

<table>
<thead>
<tr>
<th>Activity</th>
<th>AS-IS</th>
<th>TO-BE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Choosing the bank</td>
<td>39</td>
<td>6</td>
</tr>
<tr>
<td>Obtaining information about loans</td>
<td>0</td>
<td>28</td>
</tr>
<tr>
<td>Loan type selection</td>
<td>4</td>
<td>40</td>
</tr>
<tr>
<td>Informative calculation</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>Loan review</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Application for loan</td>
<td>48</td>
<td>3</td>
</tr>
<tr>
<td>Documentation preparation</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Documentation validation</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Evaluation of the credit loaner</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Loan insurance</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Contract preparation</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>Contract review/ Contract signature</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Transfer to account</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Not all bottlenecks were solved, and some new ones accumulated due to the rearrangement. However, the sum of all bottlenecks has, nevertheless, decreased (Table 4). Table 5 presents the improvements of the process renewal; the costs have not changed significantly, however, the time aspect is significantly improved due to the informatization in activities such as Information calculation, Evaluation of the credit loaner and Loan insurance decision.
The optimization technology has enabled modeling, partial integration of success indicators, simulation of the business process, and business process comparison between AS-IS and TO-BE conditions. The TO-BE process is less time-consuming and more efficient. After the analysis, we concluded that IT solutions could improve the business process characteristics. It is sensible to include additional risk management in the process of approval, as well as evaluate the quality of software solutions.

Despite relatively affirmative results, there are several risks to the validity of the research. An interview was conducted at one bank with two people, and the survey of optimization approaches and simulation was run within IT companies and not banks. Therefore, the results are challenging to be generalized, with its answers, to other banks in Slovenia.

### 5. CONCLUSION

The loan approval process is one of the more sensitive areas of banking and the Finance sector. Due to the pursuit to create efficient processes and achieve an optimized resource consumption regardless of a domain, the search for a qualitative product application is becoming vital. Measuring quality of software can, additionally, be extended to measuring the quality of processes, assuming that the process model is a prerequisite for the real process and the process execution relies on IS. In addition to economic and mathematical knowledge, appropriate IT solution support needs to be applied, including information security, transfer of business processes into digital form with the purpose of measuring and evaluating with the help of computer programs, application of mathematical methods to activities in the process, replacement of manual operations with new ways of electronic business, and introduction of e-banking among the widest possible target audience.

As a proposal for optimization, using advanced IT support can optimize the credit approval process to some extent. However, traditional systems like banks are less prone to change, and use few of the available possibilities. Within the research, some potentially replaceable activities were highlighted where IT could be included. To support defined ideas, a survey was made among employees from IT companies evaluating different optimization approaches (Čuček 2018)(Pušnik, Welzer Družovec, and Šumak 2019). The survey included an evaluation of several optimization methods. However, modeling, simulation, KPI definition, and Ishikawa were chosen as the most favorable ones. They were later used within the case study.

Within the paper, we evaluated the high level loan approval process, and presented some possible methods and approaches to optimization. Supporting IT solutions, as well as additional methods, can play an important part in improving every business process, as long as it is acknowledged that improvements can be made and would benefit end-users, as well as employees. Since the paper presents an initial acquaintance with one of many processes (from the Information Solution support point of view), the future work will expand the analysis and comparison of a further set of processes. Future work will include an expansion of the survey, respectively, interviews, covering other non-IT-domains in smart cities, comparing the optimization level of different business processes, and evaluating motivation to change and improve processes within different domains, supported by digitalization and informatization. Among the process optimization approaches presented in Section 2, the use of operational research for smart(er) decision-making within IT solutions was often

<table>
<thead>
<tr>
<th></th>
<th>Costs</th>
<th>Total cycle time</th>
<th>Resource consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>AS-IS</td>
<td>4000 €</td>
<td>4d 15 h</td>
<td>1d 16 h</td>
</tr>
<tr>
<td>TO-BE</td>
<td>3950 €</td>
<td>3d 15 h</td>
<td>1d 16 h</td>
</tr>
</tbody>
</table>
mentioned, which will be the subject of future research and, therefore, included among the opportunities for process optimization and process quality growth.

Acknowledgments
The authors acknowledge the financial support from the Slovenian Research Agency (Research Core Funding No. P2-0057).

REFERENCES
An Insight on Standardized Patterns in Model-Driven Software Development

PETAR RAJKOVIĆ, IVAN PETKOVIĆ, ALEKSANDAR MILENKOVIĆ and DRAGAN JANKOVIĆ,
University of Niš

The area of model-driven software development covers many different topics starting from process and organization and then via domain modeling and tool architecture to application platform development. For this reason, it is important to impose a standardized set of recommendations, or patterns, that could be used both for new or ongoing projects. For this research, we focused on “fixed budget shopping basket” organizational pattern and the implementation of metamodel followed by ignoring concrete syntax and separation of generated and non-generated code. The main aim of this paper is to present a general overview and give insight from our development team on the usage of standardization and patterns in model-driven software development.

1. INTRODUCTION AND MOTIVATION

Model-driven software development (MDSD) is the area of software engineering that is continuously developing in the last three decades. There are many different approaches defined and many successful projects came out from MDSD biosphere [Mohagheghi et al. 2013] [Brambilla et al. 2017]. Nevertheless, it is still easy to find many sources that not support MDSD as the paradigm, but strongly opposing it [Snee 2007] [Osis 2018]. One of the main reasons, that is often pointed out, is the lack of overall standardization in the area. This fact also leads to the situations that the significant number of project managers tries to avoid MDSD in longer projects since they cannot see the clear benefits from the first sight [Klien and Ludin, 2019] [Vijayasarathy and Butler, 2015]. On the other hand, it is not easy to exactly measure the effect of many MDSD standard approaches [Christensen and Ellingsen, 2016] [Bolender et al. 2017], since when they got applied in significantly different environments they end up with different results. Both these two facts lead to the situation that MDSD still is not on the position in software development that, in our opinion, it should be.

The development team from our Laboratory of Medical Informatics advocates MDSD for the last 15 years stating that many parts of the projects can be finished faster and more reliably [Rajkovic et al. 2015] [Rajkovic et al 2017]. Primarily, we develop medical information systems based on the OpenEHR metamodel and with a set of data modeling and data generation tools following the standards given initially by Voelter [Voelter 2004]. Beside for medical information systems, we successfully used MDSD as a basic approach in manufacturing execution system development projects [Aleksic et al. 2017]. Usage of modeling and generation tools was proven successful and...
made our development process faster, testing process more standardized and help us in an overall bug rate reduction [Rajkovic et al 2015].

After several successful implementation and upgrade projects based on model-driven engineering, we wanted to contribute to the MDSD community by supporting the standardization processes through this, rather technical, paper. Since MDSD is mostly presented through the papers that explain and evaluate specific approaches and coding techniques, with a little or no attention to identify and promote certain standards. Thus, an important segment of the paradigm that we wanted to present here are standardized parts in MDSD and enrich the initial definition with our insights.

One of the most important sources for existing standardization in MDSD is the original work [Voelter 2004]. Claiming [Alexander 1977] and [Gamma 1995] as the motivational starting point and influenced by [Bettin 2002] and [Bettin 2003], the author tried to formalize the systematics in MDSD processes. The ideas shown in the mentioned work are further developed, and elaborated in [Buschmann 2007], [Possatto et al. 2015] and [Syriani et al. 2018].

Since the area of MDSD became wide and voluminous it was necessary to identify the most important sub-concepts and to join the most important coding techniques, processes, routines and organizational sets of recommendation into logical groups. According to [Voelter 2004] [Strembeck 2009], they are structured and organized in four main groups - domain modeling, process and organization, tool architecture and application platform development. Since these groups are different by its content and focus, the common name must be chosen to describe any of the items for any of the groups. The name pattern itself was chosen in [Voelter 2004] after Alexander’s [Alexander 1977] discussion on patterns – “pattern represents our current best guess as to what arrangement of the physical environment will work to solve the problem presented. The empirical questions center on the problem—does it occur and is it felt in the way we describe it?—and the solution—does the arrangement we propose solve the problem?” and “the patterns are still hypotheses, all 253 of them—and are, therefore, all tentative, all free to evolve under the impact of new experience and observation”. The usage of the term pattern could seem unappropriated due to its formal definition, and even though the work [Alexander 1977] is from the area of civil engineering and architecture, it is adopted as the common name for all the MDSM standardized processes, software patterns, and organizational routines.

Starting from the domain modeling step we can say that this area is well covered by many different tools and approaches such are architectural-centric metamodel and formal metamodel. Our experience with data modeling is based on the and usage of locally developed modeling tools together with standardized metamodels. It gives us the results as presented in [Rajkovic et al. 2015].

The main challenge we wanted to point attention here is from the area of process and organization. The pattern is called fixed budget shopping basket and it should give the recommendations for project managers how to reach the goal in a given time with an assigned workforce. Since our experience says that the projects are often understaffed and everchanging (from the point of view of the requirements collection), the existence of any software tool that can speed up the development process is more than desired. For this reason, using the most effective way to generate code from the model is the next major area where standardization can help. Our research showed that the generator tool can reduce the time needed for development up to two thirds [Rajkovic et al. 2015]. To us was essential to follow the pattern of the implementation of the metamodel when comes to the tool architecture. In the application platform development area, we tend to have an adaptable system that can easily switch to the different programming environment and to generate a series of different software component. From this area we identified ignoring concrete syntax and the separation of generated and non-generated code as the most important patterns we used to follow.
In this paper, we will present mentioned four (fixed budget shopping basket, implementing a metamodel, ignore concrete syntax and separation of generated and non-generated code), at least for us, the most influential patterns and give the general overview and our internal insights.

2. FIXED BUDGET SHOPPING BASKET

Starting from process and organization, the one common problem in the MDSD engineering process is work organization in a way to develop a product aligning the requests from the customer in a given amount of time. From the product manager’s point of view given amount of time multiplied by the cost per hour makes a nightmare called fixed budget. In that light, the pattern imposing application of the iterative development to ensure proper result on time is named as a fixed budget shopping basket. The initial definition presented in [Voelter 2004] and further developed in [Serrador et al 2015] and [Gregory et al. 2016] contains the following rules:

- Use timeboxed iterations that are shorter than six weeks, validated by users/customers.
- Produce shippable code at least every three months.
- Ideally, deploy into production every three months to get “live” feedback.
- For the development of new business applications, “go live” within nine months, don’t risk losing the team (mother) or the application (baby).

Developing large scale projects is generally risk itself. When projects must last at least half of a year, there is a significant chance that problems either on the side of the customer or on the side of the engineering causes the delay. For this reason, project execution in pre-defined timeboxed iterations and validation sessions with customers is required. From our experience, the period should be shorter than six weeks, and usually weekly or bi-weekly meetings proved to be more convenient. Within the period of six weeks, the customer can easily lose the focus and start thinking in the directions different than specified in stakeholder requests. In [Rajkovic et al. 2013] we published the results based on our experience with different level of interaction with the customers. We had three focus clinics in our project, and, due to the differences in their organization, we established different communication routines. The best results were with the development of an information system for cardiological clinic beside the volume of the required work significantly exceeded one needed for other two clinics. Proper communication helped in later full software acceptance. In other two clinics we had only few meetings for over a year, and the level of common understanding was significantly lower, and the project took 50% longer to finish where not all the modules got accepted.

Better communication when developing an information system for cardiological clinic helped in the development process itself, especially in earlier bug discovery. Going in this line, production of shippable code also should be in iterations that lasts two to three weeks. Period of three months is longer than most of the end-users can stay active. More frequent deployments in the test environment will keep the customer interested and keep for the project.

On the other hand, deployments to the production environment should be less frequent than deployment to test suite. Regarding the deployment plan for the production server, it seems that it is more convenient not to have strict time defined deployment slots, but rather after a specific set of functionalities realization.

This approach could also be improved by iterative multi-track development. Depending on several team members the team can be divided, and the complete process could be split into several tracks and proper development approach and methodology can be applied. From the technical point of view, this approach seems optimistic, but in real development, many other problems can appear that can prevent the customer from getting an active system in desired time for a fixed budget. For example, during the iterations, the customer will easily get to the idea that something, in addition, should be
developed. Even when the customer accepts to sign the change request, the negotiation itself will take additional time which can delay the overall project.

This approach is also useful when it comes to the deployment of the developed information system in a medical institution. Using the interactive multi-track development approach we managed to reduce the time needed to deploy and customize software in Serbian medium-size primary care medical facilities by 50%. Initially we needed 60 days full of stress, but eventually we ended up with a less than one month, where number of support calls got reduced to about 30% [Rajkovic et al. 2015].

It is even harder to plan projects that should last longer. As longer the project timeline is, there are significantly more chances that the project will not be delivered on time and within budget. Eventually, the key to success in these types of projects is a mutual understanding of both sides and continual communication.

3. IMPLEMENTING A METAMODEL

Using models to improve software development is the leading idea of MDSD. The process should start with defining or adopting a metamodel, then creating specific models and based on them generating a set of various software components. This is an excellent idea and a streamlined process that looks logical and straightforward. Unfortunately, there are many technical problems hidden behind this process.

The first point to ensure is validation. Specific models must be validated against the metamodel and, at the time of software generation, generated components must be checked for consistency against the specific model. Thus, the definition from [Voelter 2004] [Wachsmuth, 2007] [Mens et al 2016] states: “Implement the meta-model in some tool that can read a model and check it against the metamodel. This check needs to include everything including declared constraints. Make sure the model is only transformed if the model has been validated against the metamodel”. The definition for this pattern gives the only direction on “what”, but not “how” the process should be done.

Once agreed with this, the next point is – “how to perform the validation”. Should this be done manually, or through some of the available pieces of software or through some of the homebrewed generator tools? Manual validation is, in most cases, out of the scope. Models tend to be complex and manual check is usually time-consuming task with a high probability of errors. Next choice is to use existing model transformation pieces of software. In many cases, the organizations that maintain metamodels for some domains, also offer some set of validation tools. These tools could be efficiently used for specific model validation, but the situation with data generation tools is a bit different. The level of customization for a specific project could be significantly higher than support, so developers should choose between:

- Using the existing tool as much as possible and then perform manual coding for the missing parts
- Extend the existing tool to suit most of the needs and then perform manual coding for the missing parts
- Develop specific code generation tools/engines that would be able to generate specifically required components and then perform manual coding for the missing parts

In all three cases, the common part is “perform manual coding for the missing parts”. Analysis of this segment is crucial to choose the strategy. Since the main aim of MDSD is to reduce the amount of the specifically typed manual coding, this is the key part in the whole MDSD, how we see it. Once the correct choice is done here, the implication for the project will be higher. Unfortunately, developing code generation frameworks/tools will take some time and can slow the initial development of all the “items from the shopping basket”.
From our experience, it is worth to invest time in specific tool development especially when the team is focused on specific domain development. If a team is doing one spin-off project for a specific domain, then is questionable if this kind of development is needed at all. Using the existing tools is useful in most of the cases when specific models should be validated against the domain model. For large scale organization like OpenEHR or ISA-95, this looks prominent, but for some other organizations, the quality of offered tools could be questioned [Rajkovic et al. 2014].

We have experience with both approach and depending on a step in the software cycle, the usage of one or another has own pros and contras. The standard generation tools are usually better fitted for one category of generated components, but for other cases, custom-built tools have own advantage. In [Rajkovic et al. 2015] we compared our results for component generation. For example, standard tools had a slight advantage when generating classes and standard logic for the components such are Windows forms, but our custom build generator tools gave better results when some specific parts should be generated such are custom logic generation, automatic configuration files creation and automatic test case builds. Our experience is that custom build MDSD suite can reduce development time in some steps up to 50%.

Since project managers need to balance the shopping basket, the decision is mostly on them. We just need to point out that our view is maybe biased by the fact that the domains of our projects (medical information systems and manufacturing execution systems) came from highly regulated areas and some team members are continuously involved in development for last 15 years.

4. IGNORE CONCRETE SYNTAX

Ignoring concrete syntax during code generation as longest as possible is a pattern which requirements are often neglected during the development of code generation tools (Fig. 1.). Again, looking at the whole development process as “the shopping basket” with limited resources, the project managers strived to bring the results sooner as possible and thus tend to reduce all of the unnecessary developments and then got some more time for “more important parts” of the project. This greedy approach can bring benefits in cases when the organization works with a single technology and has no plans to change it in longer terms, or when the generated components are relatively simple and require no further updates after generation.

The directive from [Voelter 2004] [Paige 2016] defines the process around model transformation in the exact following three phases:

- *convert the input model into some in-memory representation of the metamodel (typically an object structure)*,
- *then transforms the input model to the output model (still as an object structure)*
- *transform the target model to a concrete syntax*.

![Fig. 1. Steps in ignore concrete syntax pattern (adopted from [Voelter 2004])](image)

This pattern requires an implementation of the parser for the concrete syntax of the application model, then transformer that converts input (meta) model to specific model, and eventually the “unparser” or “petty printer” that will eventually convert the model to the concrete syntax.
Usual implementations tend to skip the second step and to implement direct transformation from the initial model to the textual output. This will reduce the system’s flexibility but speed up the development to approximately half of the required time.

Initially, we tend to go for the mentioned greedy approach, especially because all our models were from the same category and internal transformations will bring no significant, by our opinion at the time, benefits. Eventually, we switch to the full three-stage model and successful implementation is later used for manufacturing execution systems [Aleksic et al 2017].

5. SEPARATION OF GENERATED AND NON-GENERATED CODE

When comes to the generated parts of the code it is important to clearly define how the generated code should be maintained. Generated code should be handled in such a way that will not interrupt manually written code. The most common problem is the fact that code generators will completely regenerate its parts of the code. Any change in manual code will be then lost and such changes are hard to track and revert if needed. Two most common approaches are the separation of generated and non-generated code and so-called forced pre and post code. The definition from [Voelter 2004] states:

*Keep generated and non-generated code in separate files. Never modify generated code. Design an architecture that clearly defined which artifacts are generated, and which are not. Use suitable design approaches to “join” generated and non-generated code. Interfaces, as well as design patterns such as factory, strategy, bridge, or template method, are good starting points.*

The code generation itself is not one simple process, especially because of different possible relations between generated and non-generated code. Depending on the relation between the generated and non-generated code different types of relations should be considered. In [Voelter 2004] and [Torres, et al. 2017] five base cases are examined (Fig. 2.):

- a) Generated code calls non-generated code
- b) The non-generated code calls generated code
- c) The non-generated code calls generated code through the adapter class
- d) Generated classes are sub-classes of non-generated classes
- e) Usage of template method pattern

![Fig. 2. Five major relations between generated and non-generated code (adopted from [Voelter 2004])](image)

When generated components call the non-generated library, case a is the safest environment for the code execution. The generated code relies on already running and validated functionality which means that any new bug that could appear, should appear in the generated code. Testing, in this case, can then be reduced only to test the newly created library. We assume this approach as the easiest and the most convenient for implementation. It is used when the array of components...
sharing the same existing functionality should be automatically generated by the model, and where no additional dependencies are introduced.

Case b is the situation when an existing non-generated code should call the functionality from the generated code. This approach is used when some options need to be introduced into the existing functionality and it usually requires some intervention on the existing code. This intervention usually consists of introducing abstract classes or extracting specific methods and make them virtual. Then, the base functionality calls, by the default its initial behavior, while newly generated code contains the implementation of abstract or virtual methods.

The extension of this approach is the case c when the link between non-generated and generated code is an adapter class. An adapter class is built on the way that is partly generated, since the methods needed to interact with the non-generated code came from the existing library, and the adaptation methods are generated. We used this approach to handle situations where different plug-ins or completely interchanged functionality are developed. For example, the classes that support connection to different data sources are implemented this way [Rajkovic et al. 2015].

The case d is basically the extension of the case a. Generated code calls non-generated, but generated class is defined as a sub-class. This approach is used when generated code shares many functionalities with the base code, but some minor changes are applicable from case to case. We used them for different validators when validation against the same set of values could be altered to show some different aspect – i.e. in the description of neurological statuses.

The last of the presented cases is the case when the template method pattern is used during the generation. Base class (non-generated code) contains one template method (which can even be virtual) and few virtual (or abstract) and non-virtual methods. Template method consists of the sequence of calls which is strictly defined. The generated code contains classes that derive a base class and implements or overrides only requested method. In later usage, the proper instantiation leads to the execution of specific code. Within our system, we used this approach when developing generator tools. Since we use the same model to develop multiple different classes of the components, we used this approach to speed the implementation up and to make the process more effective.

6. CONCLUSION

MDSD is generally envisioned as a paradigm that should make easier to handle all the major steps of the information systems life cycle. Considering all levels of complexity, the time needed for system development is, often, much longer than it is envisioned by project management. Colloquially, the system will not fit in its shopping basket. Making the software project on the way to fit in budget and time constraints was one of the requests that drive MDSD evolution. Evolution itself went through many steps and it is still active. The new generation of MDSD frameworks offer much more than even five years ago and, in some area, such as an object-relational model representation of databases, they are considered as a standard.

On the organizational side of the process, we decided to present a fixed budget shopping basket as one of the hardest to achieve goals in project management. Along with basic recommendation, we present our findings that are in favor of the general opinion, but with a higher focus on communication with customers.

Implementing a metamodel is considerably core of the MDSD. Generating code based on the model and having in mind potential compromises on the way from system flexibility to fastest deployment is discussed here and enriched by the examples from our development projects. Two additional patterns important for the code generation itself is mentioned too – ignoring concrete syntax and separation of generated and non-generated code.

Since MDSD is a considerably wide area, starting from process organization and ending with code generation, one can tell that it lacks specific focus and that is out of standardization. With this
paper, we wanted to present major parts of the MDSD environment with belonging standardized sets of recommendation (or patterns) followed by our insight based on the decade and a half long experience of our development team.

REFERENCES


DOI:10.1007/s00170-017-0286-1. (M22, IF 2.748) (https://link.springer.com/article/10.1007/s00170-017-0286-1)

Erich Gamma 1995, Design patterns: elements of reusable object-oriented software. Pearson Education India


Approach to Selecting an Appropriate Javascript Charting Library for Graphically Rich Single Page Javascript Applications

ALEN RAJŠP, GREGOR JOŠT, VIKTOR TANESKI, SAŠA KUHAR, LUKA PAVLIČ, University of Maribor

React is one of the three most popular libraries for the development of single page applications. It also enables the implementation of reusable user interface interfaces. Additional challenges and limitations in development are the use of external libraries and frameworks. The paper presents a selection method for selecting an appropriate JavaScript charting library. The steps (identification, licensing elimination, functionalities' elimination, complexity elimination and prototyping) of the method are presented briefly, and demonstrated on a use of the method for the needs of a React Single Page Application that was focused heavily on graphically intense visualizations. The selected library (C3.js) is presented, and some cases of use are shown, with accompanying screenshots that demonstrate the complexity of implemented visualizations.

1. INTRODUCTION

The web has become a de-facto platform of user-friendly user interfaces. Web applications are no longer filled with inconvenient flashing webpages that, whenever a user clicks on a button, the whole website needs to be served to the user client. The concepts that were once meant for websites are no longer viable for user-friendly web applications. Instead, web applications became exactly what they are called - applications. They operate within the web browser environment and communicate independently with the backend system. Technological solutions, such as JavaScript, REST interfaces and JSON format, are the basic components of such applications, but, for a fast and relatively sustainable development of web applications, this set is inadequate. Without the use of specialized libraries (e.g. React.js) or frameworks (e.g. Angular), we can no longer deal with the development of single page applications. Libraries and frameworks provide developers with the means to develop single page applications. At the same time, they also bring additional challenges, such as the coexistence of "classical" HTML or JavaScript content and components in a wider application, built on third-party libraries and frameworks.

We had to develop a single page application for an external client. One of the main functionalities of the application was the appropriate visualization of data that the application receives from the backend system. Data visualization was achieved with the help of a third party library, which had to be adapted to the host - the React library. The article presents our selection process of JavaScript charting libraries, and experience with the selected JavaScript charting library C3.js.
1.1 React.js library

React.js is a JavaScript library for user interface development, developed by Facebook. Library was first used in 2011 on the social network Facebook for the news feed functionality [Education Ecosystem 2018], after which it was also used on the social network Instagram in 2012. The library became open source in 2013. The library soon became very popular for developing user interfaces among web developers [Williams 2019]. A survey on Stack Overflow [Stackoverflow 2019] for 2019 shows that the React library is at the top of the most wanted web frameworks, and the second most used web framework (after jQuery). React library is used for the View in traditional Model-View-Controller (MVC) architecture. It is based on components where each component represents a standalone module that renders a predefined structure. Modules can be reused and nested.

Unlike other JavaScript libraries or frameworks, React does not work directly with the Document Object Model (DOM). DOM is an interface that allows scripts to access and update the content, structure, and style of a document dynamically. Because DOM manipulation is ineffective, React uses virtual DOM (VDOM) [Facebook Inc.], where a virtual presentation of a user interface is saved in the memory. When components are updated, so is VDOM, and React synchronizes VDOM continuously with the DOM structure. It is important to note that only parts where changes occurred are re-rendered, and the rest stay unchanged. This allows React only minimum DOM manipulations, which allows VDOM manipulations to remain fast and efficient [Abe; Hamedani 2018].

The main feature of the React library is the non-segregation of the application’s logic from its presentation to various files. For this purpose, the JSX extension is used, which represents a syntactic extension of the programming language JavaScript. It is designed to define elements.

React is not a framework, but a library. Therefore, for complex applications, additional libraries are needed, e.g. for routing between individual pages, obtaining data, managing the situation throughout the application, visualizing data, etc. Selection of complementary libraries to use with React presents a special challenge.

2. SELECTION METHOD OF JAVASCRIPT VISUALIZATION LIBRARIES

2.1 Pretext

The practical solution developed enables complex visualizations of linked data. The visualization of data was the core functionality of this application. Because JavaScript has been gaining in popularity, so have third party charting libraries and libraries for data visualizations. For our needs, the understanding of the operation and supported functionalities in popular visualization JavaScript libraries was crucial. It was necessary to select what would best suit the requirements of the client.

One of the crucial factors for selection were: (1) The right set of graphs for display of data, (2) The amount of data shown on charts, (3) Display of graphs on different devices (responsiveness), (4) Browser support, (5) Ease of customization.

2.2 Requirements

The developed application was to be used on bigger screens (computers and tablets, not mobile phones), and had to allow visualization of larger amount of data. The user interface visual requirements were provided from the external client as an Adobe XD rendered wireframe prototype. One of the prototype images is shown in a screenshot below in Figure 1. The actual text content is omitted due to copyright.
As such, the implemented application had to be visually as similar as possible to the provided screens in the visual prototype. In addition to rather standard types of graphs (column, row, line, stacked), some heavily specialized types of graphs were also needed that had to be modified heavily e.g. overlapping row charts (Figure 2), stylized gauge charts (Figure 3), stacked column charts (Figure 4), column charts with overlay region (Figure 5), bar charts combined with stacked bar charts below (Figure 6).
2.3 Selection method

Multiple formal methods for software selection already exist, as presented in [Jadhav and Sonar 2009]. Their systematic literature review proposes a generic [Jadhav and Sonar 2009] seven step methodology at selecting software. The steps roughly translate to (1) Determining the need for purchasing the software, and preliminary research of availability on the market, (2) Shortlisting of candidates, (3) Eliminating most candidates based on features, or which do not work with existing systems, (4) Evaluation techniques to evaluate remaining packages and scoring, (5) Trialing top software packages, pilot testing the candidates, (6) Negotiating a contract, (7) Purchase and implementation.

Since JavaScript software packages contain hundreds of dependencies, such rigorous process needs to be updated. We wanted to ensure that the system could be updated in the future, and maintain the possibility of introduction of new functionalities. The selection process goal was to select an appropriate visualization library. In our selection criteria, it was important that the library enabled implementation of the required graph types, was open source, free to use in commercial applications, well documented, regularly updated, as simple as possible to use, and to allow for adjustments.

As shown in Figure 7, we created the following selection process for selection of JavaScript libraries. In comparison with the previously presented formal selection methodology, we eliminated the steps based with contract negotiation, since they are not as applicable with JavaScript dependencies.

**Libraries’ overview**

Identified charts are shown below in Table 1. The chosen library **C3.js** is marked by a trophy icon ( Trophy ) and libraries selected for prototyping phase by a check icon ( ✔). Regarding the identification
criteria at each step, we’ve taken inspiration from the publicly posted comparison available on Wikipedia [Wikipedia 2019].

Table 1. Overview of identified Javascript charts

<table>
<thead>
<tr>
<th>Selected Prototyping Library</th>
<th>GitHub stars (11. 2018)</th>
<th>License</th>
<th>Implementation complexity</th>
<th>Customizability</th>
<th>Missing graph types</th>
<th>Missing functionalities</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart.JS [Chart.js]</td>
<td>40361</td>
<td>MIT [Opensource.org]</td>
<td>low</td>
<td>low</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Chartist.JS [Chartist]</td>
<td>11231</td>
<td>MIT</td>
<td>low</td>
<td>low</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>C3.js [C3.js]</td>
<td>8089</td>
<td>MIT</td>
<td>low</td>
<td>medium</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D3.js [Bostock]</td>
<td>80246</td>
<td>BSD-3 [Open Source Initiative]</td>
<td>high</td>
<td>high</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>✔ Recharts [Recharts]</td>
<td>10318</td>
<td>MIT</td>
<td>low</td>
<td>medium</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>✔ Plotly.js [Plotly.js]</td>
<td>9282</td>
<td>MIT</td>
<td>low</td>
<td>medium</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>✔ NVD3.js [NVD3]</td>
<td>6730</td>
<td>Apache 2.0 [ASF]</td>
<td>low</td>
<td>medium</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>✔ Victory [Formidable Labs]</td>
<td>6548</td>
<td>MIT</td>
<td>low</td>
<td>low</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>✔ Vx [Shoff]</td>
<td>5311</td>
<td>MIT</td>
<td>medium</td>
<td>high</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>✔ React-Vis [Uber Open Source]</td>
<td>4696</td>
<td>MIT</td>
<td>low</td>
<td>low</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>✔ Britecharts [Eventbrite]</td>
<td>3184</td>
<td>Apache 2.0</td>
<td>low</td>
<td>low</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>✔ Incubator echarts [Apache]</td>
<td>31145</td>
<td>Apache 2.0</td>
<td>medium</td>
<td>medium</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>✔ AnyChart [AnyChart]</td>
<td>188</td>
<td>proprietary</td>
<td>Libraries that are not free for commercial use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✔ Amcharts [Amcharts]</td>
<td>373</td>
<td>proprietary</td>
<td>Libraries that are not free for commercial use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✔ CanvasJS [Fenopix]</td>
<td>9</td>
<td>proprietary</td>
<td>Libraries that are not free for commercial use</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>✔ Google Charts [Google]</td>
<td>763</td>
<td>proprietary</td>
<td>Libraries that are not free for commercial use</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Identification

The first phase is identification of candidate libraries. This is best done by researching published libraries and dependencies on source code repositories (e.g. GitHub) and package manager websites (e.g. NpmJs [NPM Inc]). These sites are beneficial because they provide transparent statistics about the libraries. We can see how many times specific libraries have been downloaded, how much and when they have received updates, how many developers use them, etc. We used the code repository GitHub [Microsoft] for identification of charting libraries by querying the following phrases “Charts”
and “Graphs” independently; the search was conducted in November 2018. We filtered the results by programming language to JavaScript and selected the 16 most popular (most stars on GitHub).

**License elimination**
In license elimination we eliminated all the libraries where licensing is inadequate for specific projects. It is important to identify the licensing of each identified library in order not to waste resources with invalid libraries. In our case, this meant eliminating libraries where the whole source code was not publicly available, or that their use wasn’t permitted or free in commercial applications. Of the 16 libraries identified 50% (8) used MIT license, 19% (3) used Apache 2.0 license, 6% (1) used BSD-3 license, and 25% (4) used proprietary licensing and were eliminated from this phase.

**Functionalities’ elimination**
In this phase, we identified the needed functionalities of our project and the functionalities of the selected libraries, and compared them. The source for identification of functionalities is, in most cases, the official documentation. In our case, we identified functionalities of the initial selection with the supported types of graphs. We identified 13 basic types of graphs that appeared in libraries (underlined are those that were required in our implementation): Gauge, column (normal, stacked), line (normal, stacked), radar, donut, pie, scattered, funnel, gantt, quartile, polar, and bubble. From the requirements, we also identified 5 specific functionalities that we needed: Overlapping regions, icons on graphs, combination of graphs and vertical / horizontal lines with added text. Thus, we selected 4 potentially suitable libraries (C3.js, D3.js, Recharts and Incubator echarts).

**Complexity elimination**
In this phase, prerequisites are analyzed (mostly on what other libraries does the library depend) for using a library and difficulty of implementation. Complexity of implementation was measured by the quality of documentation and length of code needed for simple use cases.

In complexity elimination we eliminated the D3.js library from the remaining four libraries, due to severely higher implementation complexity than in the remaining cases. This was because D3.js is a visualization library, not a charting one. This means that to create charts in D3.js one would have to implement each, even the most basic building block, by himself. This means that D3.js does not come with any of the prebuilt charts that could be modified further. C3.js, on the contrary, is built on top of D3.js, which means that basic graphs are already built, but transformations possible on D3.js are also possible in C3.js.

**Prototyping**
In the prototyping phase we prototyped the top libraries by implementing the same cases in all of them, to select the most suitable library from the working prototypes.

In our project, the purpose of the prototyping phase was the implementation of more complex types of graphs on static web pages. An example of an attempt to draw an overlapping column graph is shown in Figure 8. In the other cases, the C3.js library also made it possible to achieve the greatest approximation to the required appearance, depending on the requirements. That is why we selected C3.js as the most suitable for our implementation.
3. VISUALIZING DATA WITH THE C3.JS LIBRARY

C3.js [C3.js] is a graphical drawing library that is licensed under MIT [Opensource.org]. This allows us to use it without any compensation in commercial projects. It is based on the library D3.js [Bostock]. The library was first published in 2014, and has already received 104 issues so far, indicating that it is topical and up-to-date. Graphs like the D3.js library are drawn in the SVG format. To use, we need to include its stylish template (CSS), the D3.js library, and the JavaScript library file.

The library divides different data types according to their use in graphs. It divides them into data, axes, legends, grids, overlay regions, each of the data types represents an object that is included in the root JSON object. Existing graphs can also be updated without creating a new instance of the graph. Implemented graph visualizations are shown in Figures 9 and 10.

![Implemented visualizations on “Daily summary”](image_url)
3.1 Graph adjustments

The basic graphs had to be modified so that they would meet the requirements visually and functionally. The adaptations concerned both style and functionality. Stylistic modifications were achieved by modification of stylesheets (CSS). The gauge graph (which is not a part of the prebuilt graphs in C3.js) required (Figure 11) to fill a part of the pie chart with a transparent color (2), which was done by inserting virtual data with the same color as the background, then, using the CSS transformation, the graph was rotated (3), and the words had to be rotated in the opposite direction in order to maintain normal orientation (4).

In places where CSS customizations were not sufficient, we also used the methods (e.g. init - on initialization, onrendered - upon drawing the graph, onmouseover - when the cursor enters the graph of the graph, onmouseout - on leaving the cursor from the area of the graph, onresresize - while changing the size of the browser window) of the C3.js library that are provided for such specialized cases. Use of C3.js methods for transformations is much slower than use of CSS transformations, but also allows use the full use of D3.js library methods, so they were only used where CSS transformations were inadequate tools for the required results.
4. CONCLUSION

Our method for selecting the JavaScript visualization library was deemed appropriate, and its output – the selected library C3.js, was successful in implementing all the required visualizations. The selection process ensured that the selected library was compliant with all the functional and nonfunctional demands, was versatile and successful with implementing the required charts. We deem the process used in this special case to be useful also for selection of other JavaScript libraries, since it was lightweight, domain independent and simple to follow. The exact boundaries on which libraries to eliminate on each step were not mentioned, because they are specific for each project and should be selected accordingly.

REFERENCES

Opensource.org. The MIT License | Open Source Initiative. https://opensource.org/licenses/MIT.
Complexity Analysis of Decentralized Application Development Using Integration Tools

PATRIK REK, BLAŽ PODGORELEC, and MUHAMED TURKANOVIČ, University of Maribor

Decentralized applications development on the Ethereum platform is becoming very popular in the last few years. However, it requires developers to have knowledge and skills to integrate large numbers of components, such as smart contracts programming, IPFS decentralized storage, RPC calls or Truffle for smart contracts management and various JavaScript libraries (e.g., Web3.js, TruffleContract, JS-IPFS). This makes the development process very complex and difficult. At the same time, the developer has multiple front-end frameworks available, which however lack the ability to easily integrate the majority of previously mentioned components. To solve this problem, there are integration tools which address above issues and are intended to support a comprehensive development of decentralized applications (e.g., Drizzle, Vortex, Web3-React). The paper focuses on these integration tools and analyses the code complexity of decentralized application development using such tools. The analysis of code complexity was performed using multiple code complexity metric assessment methods.

1. INTRODUCTION

Due to the current stage of the Decentralized Applications and according to Gartner’s 2018 Hype Cycle for Blockchain Technologies [Furlonger and Kandaswamy 2018], there is still a lack of studies and tools with which the development and understanding of such applications could be easier. In order to achieve the level of productivity, the development of decentralized applications needs to be well studied, simplified, and documented. The decentralized application development process is currently marked as complicated because of multiple factors (e.g., lack of tools, good practices, and documentation), which are tightly related mainly with the purpose to facilitate the development process. In this article, we focus on the issues of the development process of the decentralized application, based on the Ethereum platform, as it is the most used platform for the development of such applications with the largest number of active developers among other platforms [StateOfTheDApps 2019].

Decentralized applications developer must be acquainted with the Solidity programming language, with which smart contracts on the Ethereum platform are developed. Smart contracts also represent the fundamental building blocks of decentralized applications [Modi 2018]. If we look from the traditional web application development perspective, smart contracts can be considered as a back-end logic of decentralized applications. Therefore, it is also necessary to know how to connect the aforementioned smart contracts within a blockchain network (e.g., permissioned or permissionless) with the end user in a user-friendly way, i.e., with decentralized applications. [Antonopoulos and Wood]
connection between smart contract and the decentralized application front-end, which is usually implemented with the use of different frameworks (e.g. Angular, ReactJS, Vue.js) can be done with the help of the Web3 library or with the use of dedicated frameworks (e.g. Truffle, Embark). In case that the decentralized application includes any additional large data files, there is also an decentralized option for a file storage, i.e. a decentralized storage technology as IPFS, Swarm or StorJ. [Google 2018; Facebook Inc. 2019; Evan You 2018; Consensys 2018; Embark 2019; Protocol Labs 2018b; Ethersphere 2018; Storj Labs 2018]

The whole development process of decentralized applications is complex due to the related and various programming languages, tools, frameworks, and new architectural concepts, which will be described in more detail in the following chapters. The aim of the paper is to present integration tools that simplify comprehensive decentralized applications development and compare the process of development with and without such integration tools. We limit ourselves to the development process of the front-end building blocks of decentralized applications. To better assess the meaningfulness of using such integration tools, we have also performed a code complexity analysis using several metric assessment methods, thus concluding with the results of these.

2. PRELIMINARIES

In this section, we present a short overview of concepts of technologies used within the development of decentralized applications.

2.1 Blockchain technology

The blockchain is a sequence of connected data blocks that contain a complete list of transactions sent to a blockchain network. Each block consists of a head and body. Inside the head, there is the information about the version of the block, tree root hash value, timestamp, number of bits, nonce and hash value of the block are written, while body consists from the transactions and the transaction counter [Zheng et al. 2017]. The blockchain network usually consists of multiple nodes, which have equal privileges, and each of them possesses the same ledger with all blocks and transactions performed through the network [Zheng et al. 2017]. To achieve consistency among all nodes, they need to follow the network protocol, part of which is also the consensus mechanism (e.g., proof of Work, proof of stake, proof of authority) [Bach et al. 2018].

2.2 Smart contracts

A smart contract is a computer program that is stored and implemented on a blockchain network (e.g., Ethereum). Smart contracts represent the business logic of decentralized applications and are programmed arbitrarily. They are instantiated on the blockchain network within the special "create smart contract" transaction. Once this transaction is included in the valid block, each account inside the blockchain network can use its specified functions. These functions are performed by sending the transactions to the address of the smart contract, and it is crucial to notice that all nodes of the blockchain network with the purpose of validation of this transaction perform the same action inside own virtual machines (e.g. Ethereum Virtual Machine). After that, the transaction results are irrevocably recorded in the blockchain [Aldweesh et al. 2018].

2.3 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) is a stack-based virtual machine that executes smart contracts. EVM executes the smart contracts based on the byte code and consumes a certain amount of gas for its execution. The gas is determined by cryptocurrency - Ether, which is necessary for performing operations on the blockchain network. The amount of gas required to perform individual operations is
predetermined in terms of the amount of memory, network activity, and processor work. The function which is defined by a smart contract and it is called within a transaction sent on the blockchain network is executed inside EVM on all nodes of the blockchain network. [Wohrer and Zdun 2018; Aldweesh et al. 2018]

2.4 Distributed data storage

Data can be stored inside the persistent storage of a smart contract, but because of the higher processor complexity and, on the other hand, the higher costs incurred by using the storage space of each network node, it is not suitable for smart contracts to store large amounts of data or even files inside it. For this purpose, the distributed and decentralized storage systems (e.g., IPFS, Swarm, StoreJ) can be used, which can operate independently of the blockchain or they can be used interchangeably with the blockchain network [Zheng et al. 2018].

2.5 Decentralized applications

Decentralized applications are applications that store all data about the performed operations in a distributed ledger (DL). Their business logic does not rely on any central entity but is unlike the traditional applications defined in smart contracts, which are part of DL. Those smart contracts are later (when the function is called) executed by every node of the distributed and decentralized blockchain network. Decentralized applications are usually open source and have a front-end part for interacting with the user, while the cryptographically signed data is processed within the blockchain network [Furlonger and Kandaswamy 2018]. The high-level decentralized application architecture which consists of four fundamental building blocks is shown in Figure 1, and these building blocks are executed in two different environments (local and network). The user interacts with the decentralized application through the usage of BC-CLIENT building block. BC-CLIENT store user credentials (cryptographic keys) with which the transactions are cryptographically signed, and provides the connection to the
blockchain network. Decentralized application front-end building block communicates with smart con-
tracts deployed on the blockchain network and provides to the end user the user-friendly usage of
decentralized application. If there is a need for additional data storage the decentralized application
front-end also communicates with a distributed storage network. Both of the building blocks (decen-
tralized application front-end and BC- CLIENT) mentioned above are executed in the local user en-
vironment - usually inside the browser. Blockchain and distributed storage networks are executed in
a remote network environment [Pustišek and Kos 2018]. Purpose of both of these building blocks is
described in Section 2.

3. DECENTRALIZED APPLICATION DEVELOPMENT

3.1 JavaScript frameworks
In order to connect the logic with the presentation level of web applications, there are many frame-
works, including React, Angular, and Vue.js [Facebook Inc. 2019; Google 2018; Evan You 2018].

3.1.1 React. React is a JavaScript library for the design and construction of user interfaces, which
is based on various components. The components are encapsulated and have their own state, which by
combining several different components allows the development of more complex user interfaces. The
logic of the components is written with the JavaScript programming language, while the presentation
level is written in XML-like syntax called JSX. An application based on the React framework can
be run on the server-side as well as on the client-side. Because of the JavaScript-based components,
they are flexible and handle the state that is sent to the server or, in our case, into the blockchain
network. The more advanced React developers also use Redux, which is the advanced application state
management tool. Without the use of Redux, it is necessary to store the application state for each
component, which in the case of blockchain means reconnecting with the network on each interaction
of the user with the application [Facebook Inc. 2019; Abramov and Redux 2019].

3.1.2 Angular. Angular framework is similar to React, but it is based on a strongly typed pro-
gramming language - TypeScript. It enables the easier development of web applications. Angular uses
declarative templates and real-time debugging. The platform itself also allows building applications
that will be used on different devices and platforms. For templates, the enriched HTML language is
used. Similarly as React, Angular also encourages the development of modules and components, di-
vided into smaller portions of the user interface, since it is possible to achieve easier collaboration
between developers with this approach to development [Google 2018].

3.1.3 Vue. Vue is a progressive framework for building user interfaces. It is, like React, based on
the JavaScript programming language. Vue is designed to be used incrementally, which means it does
not need to be used exclusively, but can be used in conjunction with other frameworks and is only used
partially. The framework is focused solely on the presentation level, which makes it easy to integrate
with other libraries and projects. Despite its simplicity, it is possible to run sophisticated web applica-
tions in combination with modern tools and libraries. In Vue, the template is written in an enriched
HTML language, just like in Angular. When building applications using the Vue framework, it is also
recommended that the entire project be separated into individual components [Evan You 2018].

3.2 Support technologies
There are several tools and JavaScript libraries that simplify development of decentralized applica-
tions, which will be described in this section.
3.2.1 **Web3.js.** A dedicated JavaScript library is required to connect a decentralized application to the blockchain network. One of these libraries, which is widely used, is web3.js. This is a collection of libraries that allow interaction with local or remote Ethereum node using HTTP, WebSocket or IPC (inter-process communication) connection. The library collection is divided into three major parts [Ethereum 2016]:

1. **web3-eth,** representing an access point to the Ethereum blockchain and smart contracts,
2. **web3-shh,** which represents a way to connect to the Whisper protocol for peer-to-peer communication,
3. **web3-utils,** which contains useful features for developers of decentralized applications (conversion between different types of encodings, validation of records ...).

With the help of the web3.js library, it is also possible to subscribe to events, which are triggered upon successful completion of a state change in smart contracts. An important fact is that web3.js and its functions are executed synchronously, which adds an additional challenge to the development of decentralized applications, since the above-described frameworks operate asynchronously, so it is necessary to manually implement asynchronous operations in cooperation with the web3.js library [Ethereum 2016].

3.2.2 **Ethereum development frameworks.** There are several frameworks that help full-stack Ethereum developers in decentralized applications development.

**Truffle.** One of such tools is called Truffle. It is a development and testing framework for the Ethereum blockchain. Truffle provides tools to compile, deploy and migrate smart contracts to your network, automatically test smart contracts, manage networks and load smart contracts on any number of public and private networks, an interactive console for direct communication with smart contracts, and a tool for easy continuous integration of smart contracts. Using Truffle to deploy smart contracts, we can use the JavaScript truffle-contract library to interact with them within decentralized applications, which makes it easier for developers to initialize and use smart contracts. Thus, it is no longer necessary to manually obtain an application binary interface, as Truffle automatically takes care for this. This makes the development cycle a bit shorter and easier because the use of functions is more similar to the standards used by JavaScript [Consensys 2018].

**Embark.** Embark, like Truffle, is a development framework for building decentralized applications and deploying on distributed networks. It connects to the Ethereum blockchain and IPFS distributed storage. It provides automatic deployment of smart contracts to the blockchain network, testing, distribution of decentralized applications to the distributed IPFS file system and peer-to-peer messaging. It also includes the Cockpit application, which represents an interface for managing smart contracts and blockchain networks. It also comes with the JavaScript library EmbarkJS, which brings similar functionality as the truffle-contract library in the case of the Truffle environment. It allows connection of a decentralized application with a blockchain network, sending transactions and calls, subscribing to events, usage of distributed IPFS file system, and working with user accounts [Embark 2019].

3.2.3 **JS-IPFS.** IPFS is a distributed file system that aims to connect all computers with the same file system. It is a versioned file system that manages files in different places and also tracks their variations. The system is similar to the operation of the Bittorrent protocol [Protocol Labs 2018b].

Connection to IPFS can be established using HTTP calls to one of the nodes or, in the case of decentralized applications, using the js-ipfs JavaScript library, which allows connection to the IPFS node and adding or getting files. The js-ipfs library needs to be initialized and connected to local or remote node after installation. Once the connection is established, the developer gets access to the application
programming interface for the IPFS file storage. It can listen to various events that are triggered by faults or interruptions. At the same time, it can add files or other data to the IPFS network, where it receives their hash value as a return, and the files can also be read from the network by entering their hash value. All the functions of the js-ipfs library are asynchronous [Protocol Labs 2018a].

3.3 Integration tools for a comprehensive development

Due to the specifics of decentralized application development and its many frameworks and libraries, the development process is relatively complicated. There are various tools that aim to simplify and unify the development process of comprehensive decentralized applications. Examples of such tools are Drizzle, Vortex and Web3-React. In the following sections, we will describe these tools and compare them with each other according to the functionalities they provide.

3.3.1 Drizzle. Drizzle is a collection of libraries designed to develop the front-end of decentralized applications. The goal of the tool is to make the development of these easier and more predictable. The core of the Drizzle is based on the Redux store. Drizzle ensures the synchronization of smart contract data and transaction data. It provides responsive data obtained from smart contracts, including balances, events, and transactions. These can be used by the developer in a similar way to traditional functions and methods in JavaScript applications. Drizzle also boasts declarativity, where the developer determines what information he needs and thus does not use the computational power for unnecessary data [ConsenSys 2018].

Since Drizzle is developed by ConsenSys within the Truffle Suite, it is desirable to use it in cooperation with Truffle to work properly. Drizzle is installed using the npm package manager and can be used in combination with any Redux-enabled framework. Therefore, the most appropriate framework to use with Drizzle is React, for which Drizzle developers have also developed components that simplify the development of decentralized applications. Drizzle is compatible with version 1.0 of web3 and WebSocket protocol [ConsenSys 2018].

React components for Drizzle include (1) a component that checks blockchain and smart contracts connection status and displays specific views for loading and failure status, (2) a component for making smart contract calls and displaying returned data and (3) a component that generates an input form from smart contract methods. With these components and using the entire React framework, the development of decentralized applications is accelerated [ConsenSys 2018].

3.3.2 Vortex. Vortex is a Redux store that deals with transactions, smart contracts, events, accounts, method calls, web3 status, IPFS file retrieval, etc. It is mainly used with the React framework, since it allows better responsiveness without refreshing and makes fewer web3 requests for even better results. It can also be used to read files from a distributed IPFS file system and to cache them in the Redux store [horyus 2019].

It provides fast loading of all smart contract instances within Redux store. It stores information about all user actions in the cache and tracks the user's transactions and the balance of the Ethereum account. All data from smart contracts is consistently obtained from the blockchain and saved into the cache. When the event is emitted on a smart contract, it is automatically retrieved. The development of decentralized applications is thus possible in a more uniform way [horyus 2019].

Vortex can work in conjunction with Embark or Truffle. Installation is possible using the npm package manager. Features or components for React can then be used. The current weakness of the Vortex tool is that the current version is already outdated because it does not fit with some of the more recent libraries it refers to. A new version, called ethvtx, is being developed, but it has not yet been developed to the same extent as the first version of the Vortex tool [horyus 2019].
3.3.3 Web3-react. Web3-react is a simple framework for developing decentralized applications of the Ethereum platform using the React library. It supports most of the commonly used tools that establish the web3 connection to the blockchain network (e.g., MetaMask, Infura, Portis). It is developer friendly because it establishes the web3.js instance and manages the settings. It does not offer as much functionality as the previously described Drizzle and Vortex, but provides a basis for the development of advanced functionality that manages every aspect of the decentralized application. It solves the asynchronism issue, which, instead, the developer would have to implement himself by creating promises for all existing Web3.js functions [horyus 2019].

3.3.4 Comparison. An exact comparison of functionalities of the described tools is displayed in the table I.

<table>
<thead>
<tr>
<th></th>
<th>Drizzle</th>
<th>Vortex</th>
<th>Web3-react</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connection establishment</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Smart contracts</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Making transactions</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Events subscription</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>IPFS connection</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Asynchronous operation</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Redux store</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Limitations</td>
<td>Truffle &amp; Redux</td>
<td>React</td>
<td>React</td>
</tr>
<tr>
<td>Development state</td>
<td>Production ready</td>
<td>Production ready, but it does not comply with the latest versions. New version is still in development.</td>
<td>Production ready</td>
</tr>
</tbody>
</table>

4. COMPLEXITY

The complexity of the software is the measurement of the software code quality. It is considered as the amount of hardware capabilities required to interact with the software. For the computer, the complexity can be described as the amount of time and memory needed to perform computational operations. For the developer, the complexity is defined as complexity of performing tasks such as programming, debugging and testing. A high level of complexity in an individual part of the software (function, object, class, etc.) is considered bad. The complexity measurements must be carried out using certain metrics that address different software attributes [Debbarma et al. 2012; Bhatia and Malhotra 2014].

There are many metrics for measuring the complexity of the software. Number of lines of code (LOC) is the oldest and very often used. This counts each line of program code, including comments and blank lines, while Effective Lines of Code (ELOC) ignores comments and blank lines. The advantages of this metric are in its simplicity and general acceptance, with the disadvantage being that it is not necessary that a smaller number of code lines mean a higher quality code [Pawade et al. 2016].

The ABC metric consists of three components, namely Assignment, Branch and Condition. Assignments represent the number of variables, branches represent the number of function calls, and the conditions are the number of conditional statements in the program code. Components represent components of the ABC vector, whose length is the value of the ABC code. Higher value means greater complexity [Pawade et al. 2016].
Patrik Rek et al.

Halstead’s metrics (HSS) extend the LOC metric to treat the program as a set of operators and operands. They define multiple software complexity attributes [Halstead and H. 1977]:

—number of distinct operators: $n_1$
—number of distinct operands: $n_2$
—total number of operators: $N_1$
—total number of operands: $N_2$
—program length - total number of operators and operands in program code: $N = N_1 + N_2$
—program vocabulary - total number of distinct operators and operands: $n = n_1 + n_2$
—program volume - size of the entire program, calculated according to the following equation: $V = N \times \log_2 n$
—degree of difficulty - the difficulty of writing or understanding the program, calculated by the equation: $D = \frac{n_1^2}{n_2^2} \times \frac{N_2}{n_2}$
—effort - the actual time of programming according to the equation: $E = V \times D$
—time required to develop in seconds: $T = \frac{E}{18}$
—number of bugs in the software: $B = \frac{V}{3000}$

Based on the above attributes, the complexity of the software can be determined according to Halstead principles [Debbarma et al. 2012; Pawade et al. 2016].

The metrics described so far have been quantitative, while the following metrics address the control flow. This is analysed according to the program flowchart. One such metric is called cyclomatic complexity (MCC) and is defined by the equation $CC = e - n + 2$, where $e$ is the number of edges and $n$ is the number of nodes in the graph. Cyclomatic complexity can more easily be calculated using the equation $CC = \Pi - s + 2$, where $\Pi$ is the number of decision statements and $s$ the number of endpoints. The decision statements are all if, while, for, case, catch and other dependency items [Pawade et al. 2016].

For object-oriented programming languages, which include JavaScript, there are specific metrics, and one such is the Weighted Methods per Class metric (WMC). In WMC, we sum complexities of all methods, calculated from the selected metric for computing complexity, such as, for example, cyclomatic complexity [Beranič 2018; Nuñez-Varela et al. 2017].

5. COMPLEXITY ANALYSIS

We have developed an example decentralized chat application in two ways. In both ways, we have manually written the source code, while using a private Ethereum blockchain network. While developing the decentralized application, we used the Truffle tool for smart contracts’ management and the React frontend library. In the first case, we have not used any comprehensive decentralized application development tools, whereas in second we have used Vortex. We have selected Vortex due to its highest number of functionalities, as displayed in table I. Both decentralized applications used the same smart contract and had the same functionalities. Users would register to the application where they can see all the users and conduct chat conversations with them. All the messages are stored on the IPFS platform and noted in the blockchain network. Users interact with decentralized applications using transactions to smart contracts. Each chat with another user is separate smart contract.

To calculate the complexity of the program code we used the previously described metrics, namely LOC, ABC, Halstead Metrics, MCC and WMC. Since smart contracts are the same in both examples, we have excluded them from the research. Calculations for both examples are displayed in Table II, where $N$ represents the program length, $n$ is the program vocabulary, $V$ the volume of the program,
D is the difficulty level, E is the effort, T is time to develop in seconds and B number of bugs in the program. LOC, ABC, MCC and WMC are calculations by each metric respectively. As we can see, the method which uses Vortex is only better in difficulty level, MCC and WMC metric of software complexity. According to other metrics, the example without use of comprehensive decentralized application development tool is less complex than the one using Vortex.

<table>
<thead>
<tr>
<th></th>
<th>Without integration tool</th>
<th>Using Vortex</th>
<th>% change</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC</td>
<td>352</td>
<td>465</td>
<td>+ 32.1</td>
</tr>
<tr>
<td>ABC</td>
<td>132.1</td>
<td>150.9</td>
<td>+ 14.2</td>
</tr>
<tr>
<td>N</td>
<td>1314</td>
<td>1469</td>
<td>+ 11.8</td>
</tr>
<tr>
<td>n</td>
<td>146</td>
<td>154</td>
<td>+ 5.5</td>
</tr>
<tr>
<td>V</td>
<td>9447.4</td>
<td>10674.9</td>
<td>+ 13.0</td>
</tr>
<tr>
<td>D</td>
<td>38.3</td>
<td>38.0</td>
<td>- 0.8</td>
</tr>
<tr>
<td>E</td>
<td>361904.8</td>
<td>405182.4</td>
<td>+ 12.0</td>
</tr>
<tr>
<td>T</td>
<td>20105.8</td>
<td>22510.1</td>
<td>+ 12.0</td>
</tr>
<tr>
<td>B</td>
<td>3.1</td>
<td>3.6</td>
<td>+ 16.1</td>
</tr>
<tr>
<td>MCC</td>
<td>3.8</td>
<td>2.5</td>
<td>- 34.2</td>
</tr>
<tr>
<td>WMC</td>
<td>19</td>
<td>15</td>
<td>- 21.1</td>
</tr>
</tbody>
</table>

6. DISCUSSION

Metrics which measure program code size (i.e. LOC, ABC and Halstead metrics) show better results for example without using integration tools, which is due to the higher number of components used in example using Vortex tool, since the tool provides component for each scenario. However, the structural complexity has decreased, as noted using MCC and WMC metrics in Table II.

7. CONCLUSION

Using Vortex tool, there is a decrease in structural complexity of decentralized application, while code size has increased. We think that existing software complexity metrics are not appropriate for web applications, since they are specific and more components do not always mean more complex software, since components are intended to diversify application and split it into smaller independent parts which make further development easier.

With the use of comprehensive integration tools for decentralized applications development such as Vortex or Drizzle, developers obtain unified access to each functionality and do not have to focus on lower level of decentralized application development. These tools are made with the goal to change developer’s focus from decentralized technologies to functionalities.

REFERENCES


Fitness Functions and Transformations in an Automated Process

NATAŠA SUKUR, DONI PRACNER and ZORAN BUDIMAC, University of Novi Sad

An important aspect of program maintenance is the understanding of the original code. One option is to automate the process with code transformations as much as possible by using fitness functions to evaluate the improvements made. The process presented focuses on low-level code and relies on FermaT, a program transformation system based on the WSL language. It has been used in software evolution applications, mainly for legacy systems and conversions of low-level code into high-level structures. This paper presents experiments with different fitness functions and compares their influence on the end result. The main focus is on the number of tried and applied transformations.

1. INTRODUCTION

In modern environments software is almost constantly in a state of change, adapting to the new needs. There is a strong need for tools that can help in various stages of maintenance. One of the problems is that even just understanding the original logic of the code can be hard. The experience has shown that understanding a piece of code one has written can often be cumbersome, let alone when the work is of someone else. Not understanding the original code and its functionalities correctly can lead to creating new errors in software rather than its improvements.

Software maintenance is a very important part of the software life cycle. It is the longest phase of the life cycle and it can be used for perfective, adaptive and corrective purpose, depending on the needs. When maintaining the software, it is very important to keep the quality of the software at the same level or to improve it. The perseverance in quality preservation is important for providing a long life for the software at hand. The deadlines are short and the demands for changes are high, especially compared to the early days of computer science and technology. By introducing new functionalities and correcting the existing ones, there is a lot of room for creating errors.

Software evolution and reengineering are another important aspect of the software lifetime. It is almost inevitable that software will have to adapt and evolve from the functionalities, environment and scale point of view over time. If the software has had many changes or if there is a need for significant changes, it is often necessary to completely reengineer it. Again, without understanding the functionalities of the software, reengineering cannot be done properly. Software evolution is repeated reengineering working towards creating a better system.
If these perfective processes are done by hand and rely purely on the expertise and experience of the engineer, as well as his understanding of the problem, there is still a possibility for creating errors. The ideal scenario would be having a helper tool that would provide automation or at least semi-automation of software evolution.

This paper presents experiments with an automated source code transformation process. The foundation of this research is based on FermaT program transformation system and mjc2wsl, a tool that translates MicroJava bytecode to WSL (Wide Spectrum Language). Upon translation, the resulting WSL code is transformed by FermaT. The transformation process in this research is automated by another tool, from the translated low-level program to a higher-level and more understandable code. This approach uses the hill climbing algorithm [Russell and Norvig 2016], a search algorithm which moves "uphill", which means that it constantly tries to move towards the increasing value until it reaches a peak. In this case, the peak would be the best program possible, meaning that no further transformation could lead to a better program. In order to determine which values are better, this process is guided by a fitness function, a concept which originates from evolutionary computing. The ideas of evolutionary computing come from the natural process of evolution, where the qualities of an individual determine its chances of survival. The same could be applied to the solving of the computing problems: a fitness function should point to the most suitable candidate solutions which seem to have the best chances for solving the problem at hand. In this case, a "better" program is the one which is simpler and more understandable. The quality of the candidate solutions usually reflects in the values of metrics, where the better ones tend to be lower. That is the reason why the built-in WSL metrics were the obvious first candidates for fitness functions.

In our previous work [Sukur and Pracner 2018], the main question we tried to answer was whether using a certain metric as a fitness function would lead to the best improvements of the said metric in the end result. Our assumption was that will not be the case for all metrics. The experiments performed confirmed our expectations, since the best results for a metric were not always accomplished by the same metric as a fitness function. Also, there was no single fitness function which would give the best results by far. The best fitness functions were metrics whose initial values were quite high and whose values changed frequently.

In this paper, we are trying to analyze another aspect of using different metrics as fitness functions. The main observations in this paper will be in regard to comparison of the efficiency of the process when using different fitness functions and the differences in the number of transformation that the process tries and succeeds to apply. The final, transformed code is often very similar when compared across all variants of the process. However, there are noticeable differences in the number of tried and applied transformations.

The rest of the paper is organized as follows: related work is given in Section 2, the automated transformation process and experiment setups are explained in Section 3, the results of the experiments and their discussion are given in Section 4, finally the conclusions and options for future work are given in Section 5.

2. FOUNDATIONS AND RELATED WORK

Formal methods are important for the overall software reliability and are often used in various stages of its life cycle. They can be applied to software artifacts such as specifications, models or source code. Formal methods are used in different fields of software engineering, such as specification or development, but also for the purpose of software reengineering. Software reengineering consists of reverse engineering, functional restructuring and forward engineering. It relies on numerous formal methods, such as assertional methods, temporal logic, process algebra and automata throughout forward engi-
neering and functional restructuring. In reverse engineering, it sometimes relies on formal methods for activities such as formal specification and verification of existing systems, as well as introduction of new functionalities. It is, however, still debatable whether formal methods should be used in system development and to what extent. On one hand, formal methods are very important for the reliability of the systems and for the quality of the entire development process. In other opinion, they are not 100% reliable in (re)engineering processes and using them can be very costly. What is obvious is that there is, beyond any doubt, a strong need for a reliable methodology which would take care of the quality of processes throughout the entire life cycle [Yang and Ward 2003].

Formal methods are very versatile and there is not a single formal method suitable for all purposes. There are different quality criteria of formal methods, some of which are supporting automated tools for development, reliability, concurrency and existence of a proof system. Although many formal methods have their advantages and disadvantages in regard to these criteria, the conclusion by [Yang and Ward 2003] is that formal methods should be chosen depending on the nature of the problem at hand. Depending on the scale and nature of the problem, one should choose a suitable formal method, for example, Z in the case of large industrial applications [ISOZ 2002]; different process algebras for reasoning about concurrency and communication; and net-based formalisms for visual representation. However, formal methods are not so frequently used in reverse engineering. WSL [Ward 2013] is a language for reverse engineering of sequential systems, based on formal methods. The main idea of the entire WSL/FermaT system is strongly based around reverse and forward engineering.

Formal methods can also be suitable for code transformations. A program transformation is an operation which, once applied to a program, produces a program with the same external behavior [Ward 1989]. The idea around code transformation is to achieve cost reduction – for example, improvements regarding performance, memory usage or even portability. Code transformations are not only useful for evolution of existing software, but also in the development phases of new software.

FermaT is one such system that offers program transformations. It is based around the WSL language, which stands for *wide spectrum language*, meaning that it contains both abstract mathematical specifications and low-level programming constructs. WSL contains standard language functionalities, such as commands and structures. Apart from that, another important aspect of WSL is MetaWSL, a set of operations that work on WSL programs themselves, as the name suggests. One of the main purposes of MetaWSL is its role in transformations. Program transformations are a part of the system and their correctness can be automatically checked. Transformations can be used for creating programs from specifications, performing reverse engineering of programs and getting specifications, as well as analyzing properties of a program. WSL was shown as very useful in various restructuring activities [Yang and Ward 2003], including industrial projects, where the aim was to convert legacy assembly code to human understandable and maintainable C and COBOL [Ward 1999, Ward 2004, Ward et al. 2004, Ward 2013]. Another tool that was made for assembly translation, with a slightly different focus is *asm2wsl* [Pracner and Budimac 2011].

The experiments in this paper use the hill climbing approach and rely on fitness functions for automated reengineering. There have already been some attempts to answer the question whether hill climbing is adequate and optimal approach for automatic program repair in [Arcuri and Yao 2008]. However, the conclusions did not show a lot of optimism for this approach, due to hill climbing tendency towards local optimums. Fitness functions have also been used for code improvement. Extensive research on automated software repair using fitness functions has been done, firstly focusing on C programs [Forrest et al. 2009] and assembly programs [Schulte et al. 2010], which resulted in applicability to any kind of code in general [Le Goues et al. 2012]. In this paper, we also tried to show that using different fitness function can lead to change in results and that these functions should be selected based on properties of the problem at hand. The research which focuses on the automated bug
detection [Fast et al. 2010; de Souza et al. 2018] also speaks in that favor and tries to give directions for designing these fitness functions in order to get the best results.

Rascal is a domain specific language (DSL) for metaprogramming, which can be used for static analysis, program transformation and implementation of other DSLs [Klint et al. 2011]. It has been used, inter alia, on C, Java and PHP [Hills and Klint 2014].

SmaCC (Smalltalk Compiler-Compiler) is a parser generator, successfully used to write custom refactoring and transformation tools for languages such as Java, C#, and Delphi [Brant and Roberts 2009]. The purpose of these tools varies from small scale refactorings to large scale migration projects. SmaCC was also ported to Pharo, which made usage of Moose analyzers and other Pharo-based software available [Brant et al. 2017; Ducasse et al. 2000].

3. AUTOMATED TRANSFORMATION PROCESS

The automated code transformation is done by a tool [Pracner and Budimac 2017] which uses a hill climbing algorithm, where the progress relies on the results of a fitness function. Simply explained, the process tries to apply code transformations as long as it is improving the structure of the input program, and the fitness function determines whether the program is improved after a transformation is applied. The fitness function can be some numeric value which indicates the complexity of the program (various software metrics). The program which has better fitness is usually the one whose complexity is less than of the original. The hill climbing script which was used in this research tries to apply one transformation at a time. However, if there is no improvement, the script attempts to achieve it by combining two transformations. The process is finished once it has reached the program of the highest quality possible. The original automated script used the structure metric as fitness. It is a custom WSL metrics which gives a weighted sum of the structures in the program.

The process records all intermediate steps which have resulted in some improvement in separate files, which gives more insight into the details. The process is also fully recorded in logs, which means that the order of transformations that were tried and successfully applied is also available for further inspection.

Code transformation is done by FermaT. However, since FermaT transforms only code written in WSL, it is necessary to translate the original source code to WSL by corresponding translation tools. The reduction of size and complexity of the outputs is not highly important for these tools, since the transformation part of the process takes care of that. The translation is also done in such a manner so that the low-level structures and operations retain the same level of abstraction. The translator tool for MicroJava, a subset of Java programming language [Mössenböck 2018], is mjc2wsl [Pracner and Budimac 2017]. The tool does not work with the code written in MicroJava, but rather with the compiled bytecode obtained from the original high-level source code.

In this paper, the hill climbing process with a number of different fitness functions was tested on a set of MicroJava programs called alpha-mj [Pracner 2019]. This set of code samples was carefully created with the idea to cover different properties of code and the virtual machine – recursion, in/out operations, some erroneous situations (division by zero) and similar. Previous experiments have shown that changing the fitness function can influence the end results of the process. Some common properties of the fitness functions that led to best code improvements were usually software metrics which had high starting values and which had tendencies to change their values easily when transformations were applied. Changing values easily meant that the process could make a lot of progress by trying and successfully applying many transformations, whereas the process was significantly longer and less successful when these properties were not present (i.e, McCabe’s cyclomatic complexity) [Sukur and Pracner 2018].
Most of the tested fitness functions were basic metrics built-in to WSL [Yang and Ward 2003]. These include McCabe’s cyclomatic complexity, marked as fit-mccabe; the number of statements (fit-stat); the size of the abstract syntax tree (fit-size); control flow and data flow – the number of variable accesses and updates, combined with the number of procedure calls and branches (fit-cfdf); structure metric – a custom WSL metric for representing the complexity of program structures, gives different weights to various types of structures (fit-struct).

As an initial experiment into more complex fitness functions, two combinations of “simple” metrics were also used. One tries to combine many different aspects of the program in a sequence. It is named fit-o1 and evaluates the new program as better if it has, in order, less actions, calls, McCabe’s complexity, statements or expressions. The other one, fit-o2, is a simple expansion of the originally used metric. It first compares the number of calls, and then the structure metric.

Additionally, another fitness function (fit-max) was used to see the maximal number of transformations tried. Basically it is a function that always returns the same result for any program, therefore the hill climbing process can never advance and will just go through all of the possible transformations.

We also define several groups of these fitness functions, based on the results that will be discussed later. The two complex functions, fit-o1 and fit-o2, are in group o-fit; functions fit-size, fit-struct and fit-stat are in group s-fit; and the union of these two groups is named so-fit.

4. EXPERIMENT RESULTS

The main focus of this paper is the comparison of efficiency of the process with different fitness functions, mainly how many transformations were tried and how many were applied to achieve the end result. However, the fitness functions can and will lead to different final programs and these need to be compared quality-wise first to have a useful comparison of efficiency. The analysis here will focus on the structure metric of the end results, since most of the other metrics have very similar relative values. It was chosen since it gives a good approximation of the abstraction level of the program. The improvements of WSL programs from their original low-level version to the final, more abstract version are expressed as the percentage of improvement, to overcome the differences in program sizes.

It is obvious from the definition of fit-max that it will lead to no improvements in the programs, so it will not be discussed in this context further. The least improvements in programs is always with fit-mccabe, usually by a significant margin compared to the other fitness functions. The main reason for this is that the values of McCabe’s cyclomatic complexity are not easily changed by a transformation, and therefore there are very few that will lead to forward steps in the hill climbing process. This is not the case with the other fitness functions tested. The differences between end result metrics for the other fitness functions were relatively small, as shown in Table I. The so-fit group of fitness functions have nearly identical metrics, with a difference from the best being often 0, and rarely, on same samples up to 5 or 6 percentage points. Following them is the fit-cfdf function, which was consistently somewhat lower in its results, but not by a significant amount, mostly just a few percentage points, sometimes up to 8.

| Table I. Differences of fitness functions from best results for alpha-mj |
|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|-----------------|
| fit-mccabe       | fit-cfdf        | fit-o1          | fit-o2          | fit-size        | fit-stat        | fit-struct      |
| avg              | 81.75           | 3               | 0.5             | 0.44            | 0.25            | 0.56            | 0.06            |
| stdevp           | 11.22           | 1.9             | 1.46            | 1.22            | 0.75            | 0.93            | 0.24            |
| min              | 53              | 1               | 0               | 0               | 0               | 0               | 0               |
| max              | 90              | 8               | 6               | 5               | 3               | 3               | 1               |

*Differences are expressed in percentage points; less is better*
One of the aspects that is important is the length of the process itself. The number of transformations tried in the whole process is very good for this purpose since these numbers are hardware independent and are not influenced by any other processes that might be running on the same machine. Figure 1 shows all of the tested fitness functions with a log scale of the transformations tried. The highest numbers are always with fit-max, as they should be, since this is a “fake” fitness function that never leads to improvements, and was meant to get an idea of how long the process can be. The highest number of transformations on these samples was more than 3 million, while the average values exceeded half a million. Next is fit-mccabe, which is always significantly worse than the other “real” fitness functions, generally almost as bad as fit-max. The rest of the functions have much more intertwined results, with a lot of variations between samples. Table II shows the ratio of transformations tried compared to fit-max per fitness function. In this aspect the s-fit group was the best, with overall very similar results among them. Following them are the two o-fit functions, with similar averages, but higher deviations and significantly higher maximums. Finally fit-cfdf had a slightly higher average, but low deviation numbers and a better maximum than the o-fit functions.

In general, functions that have much worse end results are also the ones with significantly more transformations tried. This also holds for fit-cfdf, having somewhat higher averages than the functions in the so-fit group. The reason for this behaviour is that a successful transformation will reduce the size of the program, and in turn reduce the number of places where the following transformations can be tested. This means that, in a general case, a less successful transformations process naturally leads to longer search times.

![Fig. 1. Number of transformations tried, per fitness function, on alpha-mj](image_url)

In the comparison of the number of transformations that were applied during the process fit-max and fit-mccabe will not be included, since one results in no changes, and the other shows very little
Table II. Ratio of transformations tried compared to fit-max

<table>
<thead>
<tr>
<th></th>
<th>fit-mccabe</th>
<th>fit-cfdf</th>
<th>fit-o1</th>
<th>fit-o2</th>
<th>fit-size</th>
<th>fit-stat</th>
<th>fit-struct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>avg</strong></td>
<td>69.86</td>
<td>9.84</td>
<td>7.65</td>
<td>8.10</td>
<td>7.80</td>
<td>7.20</td>
<td>7.22</td>
</tr>
<tr>
<td><strong>stddev</strong></td>
<td>12.17</td>
<td>3.68</td>
<td>6.05</td>
<td>5.19</td>
<td>3.62</td>
<td>4.31</td>
<td>3.08</td>
</tr>
<tr>
<td><strong>min</strong></td>
<td>54.36</td>
<td>3.25</td>
<td>2.32</td>
<td>2.73</td>
<td>2.20</td>
<td>2.23</td>
<td>2.25</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>99.45</td>
<td>17.15</td>
<td>28.90</td>
<td>25.00</td>
<td>13.50</td>
<td>15.77</td>
<td>13.27</td>
</tr>
</tbody>
</table>

All numbers are percentage of fit-max

Improvements compared to the others. Figure 2 shows the variations of the fitness functions using a log scale. The lowest number of applied transformations was almost always with fit-cfdf, but this fitness function also gave slightly worse end results. Functions fit-o1 and fit-o2 mostly had the highest values, on average over 60% more transformations. On average fit-struct had about 30% more transformations than the minimum, but would sometimes match the o-fit group. Function fit-size was mostly 10% above the minimum. Whereas, fit-stat was sometimes the minimum, and on average just 4% above it. The conclusion is therefore that using the number of statements as a guide will, for most cases, lead to a process with the lowest number of applied transformations for the same end result.

![Fig. 2. Number of transformations applied, per fitness function, on alpha-mj](image)

The percentage of transformations that were applied from all of those that were tested is in most cases around 1%, but can be as low as 0.05%, or almost 5% in some cases (excluding fit-mccabe, which was on average 0.0019%). Again the groupings are similar as in previous considerations. Function fit-cfdf is a bit lower than others (around 0.5%, 1.63 at most), the o-fit group is around 1.5%, with the highest values being almost 5%, while s-fit are around 1% and maximums of around 3%.
5. CONCLUSIONS AND FUTURE WORK

FermaT and WSL can be successfully used for code transformation from the low-level to a higher level of abstraction. In this approach, the entire process is automated by using a hill climbing algorithm which relies on a fitness function. The fitness function is a means which is used to evaluate the results of the applied transformation and help deduce whether applying a transformation leads to program improvement. The process tries to improve the input program as long as it is possible, that is, as long as it can generate better versions of it by applying transformations. In this research, different fitness functions were used, from the obvious candidates – built-in metrics for WSL, to a few more complex ones, which consist of different combinations of the simple metrics.

This paper presents an analysis of these alternative fitness functions that can be used in the process and how they influence the end results, and especially how they change number of transformations tried and the number of transformations that were applied in the process. They were all run on a set of MicroJava programs called alpha-mj. A special fitness function, fit-max, which never advances the process was used to estimate the maximum number of transformations tried.

In terms of metrics, the fitness functions in the so-fit group resulted in very little differences, and generally gave the same end results. The general conclusion is that functions that have relatively high numbers which are more prone to change are better as a fitness function (as already analyzed in [Sukur and Pracner 2018]).

When comparing the number of transformations tried, there is a strong trend that better end results actually lead to fewer transformations tried. This is an inherent property of the process itself – the “better” programs are in general shorter, and therefore have less possible places for transformations to be applied at, which reduces the search space.

The number of applied transformations in the process was almost always lowest with fit-cfdf, but this function gave slightly worse end results. From the so-fit group, fit-stat had the lowest average number of transformations applied. This means that it tends to make larger steps forwards with individual transformations, but it is still inconclusive whether this is inherent to this fitness function, or is it a combination of the order of transformations tried and the sample set that was used. The number of applied transformations in proportion to the number of transformations tried was in general around 1% for all fitness functions (except fit-mccabe, which was much lower).

Overall, these results give more insights into the behaviour of the process, and how it might be improved, both in terms of end results, but also in terms of its length. It is not likely that there is an universal best fitness function that will lead to best results on any input program, as this is a case of the no free lunch theorem [Wolpert and Macready 1997]. However, recommendations can be made about these specific types of low-level programs, and there are trends to be observed.

There is a lot of room for improvements and additional analyses. The functions in the so-fit group have on average a very similar number of transformations tried, just like their end results are very similar. However, there is some per-sample variation that should be inspected in more detail, which could lead to more insights into the process and how it can be improved.

The knowledge of which fitness function leads to the lowest number of applied transformations for similar end results could be very interesting for general recommendations on what transformations should be applied both in manual and in automated scenarios.

There is a need for further experiments with more complex fitness functions. This paper used only two of these, one with a large number of parameters checked, and another that was a simple expansion of one of the basic ones. These initial results should be expanded with a larger set of functions created in a systematic approach. A deeper analysis of individual successful end results should also be used when combining the metrics into more complex ones.
The main drawback of the hill climbing algorithm is its tendency towards local optimums, which can hinder the process. Future versions could try to use one of the classical solutions, in which the process is started from multiple points and the final results are compared. It could also be replaced altogether with an algorithm which offers better results.

REFERENCES


Martin Ward. 2013. Assembler restructuring in FermaT. In SCAM. IEEE, 147–156. DOI: [http://dx.doi.org/10.1109/SCAM.2013.6648196](http://dx.doi.org/10.1109/SCAM.2013.6648196)


David H. Wolpert and William G. Macready. 1997. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation* 1, 1 (April 1997), 67–82. DOI: [http://dx.doi.org/10.1109/4235.585893](http://dx.doi.org/10.1109/4235.585893)

Software Business: A Short History and Trends for the Future

KATARIINA YRJÖNKOSKI, Tampere University,
HANNU JAAKKOLA, University of Lapland,
KARI SYSTÄ, Tampere University,
TOMMI MIKKONEN, University of Helsinki,
JAAK HENNO, Tallinn University of Technology

During its 70 years of existence, the software business has been following an evolution curve that can be considered typical for several fields of industrial businesses. Technological breakthroughs and innovations are typically seen as enablers for business evolution in the domain of technology and innovation management. Software, data collection, and data analysis represent a greater and greater part of the value of products and services, and today, their role is also becoming essential in more traditional fields. This, however, requires business and technology competences that traditional industries do not have. The transformation also enables new ways of doing business and opens the field for new kinds of players. Together, all this leads to transformation and new possibilities for the software industry. In this paper we study the overall trajectory of the software business, and then offer some viewpoints on the change in different elements of business models.

1. INTRODUCTION

Software development and the software business have evolved over the past decades. There have been major changes on just about every level of operations. The most well-known change concerning development work is from linear development models like waterfall to agile and lean software development and further to Continuous Deployment [Royce 1970; Olsson et al. 2012]. This change is driven by a better understanding of the difficulties of setting detailed requirements for abstract artifacts like software in advance.

This is not, however, the only change driving the changes in software development and the software business. In this paper we explore technology-related breakthroughs that have disrupted software. Then we take a look at the concept of the business model in the domain of technology management. Finally, we describe how the value creation and selected components of business models have changed. The changes are discussed from different points of view – client, developer, and supplier. The business model is seen widely in the sense of “a way of doing business” – combining these views as its components. This paper proceeds as follows:

1. Defining the industry evolution. The evolution of the industry is defined and divided into three phases based on technology breakthroughs.
2. Reviewing business models in the software business. We take a brief look at general elements of business models in the context of the software business.
3. Reflecting the elements of business models in the industry evolution.
Finally, we discuss the changes of the industry evolution from different viewpoints. We have divided the history of the software business into phases. The transitions between the phases are driven by the technology-related evolution:

1. Hardware-driven ecosystems. Software was developed for specific hardware. Even though some high-level programming languages were used and development of portable and manufacturer-specific hardware would have been possible, the ecosystems were formed around manufacturers. Applications were usually written by the manufactures or their partners.

2. Open eco-systems driven by open interfaces. Largely driven by PCs, open interfaces to hardware and software systems started to appear. This led to the appearance of independent software vendors that conducted their business independently from hardware vendors.

3. Diversification of software business models. The Internet together with the pervasiveness of computing capacity has diversified software business models. The consumers have become end users and sometimes also customers of the software business. This development has been enabled by three technical drivers: namely, affordable devices, the Internet as a delivery mechanism, and virtualization with cloud computing. These enablers are heavily interlinked and have an effect on each other.

The evolution of the software business has been enabled by the above technological development steps and affected by related trends. In this paper we review this evolution of the software industry, the kind of transformation there has been and analyze the key trends behind the transformation from one business model to another. The goal is to understand the new developments in the business domain in the light of the core competencies of software companies as well as their customers.

The rest of the paper is organized as follows: in Section 2, we review the trajectory of the software industry and describe the biggest transitions within the business. In Section 3 we take a look at the concept of business models within the software industry to answer the question “What has happened?” In Section 4 we focus on observing the change in business model elements from different viewpoints; this answers the question “Why have the changes happened?” Then, in Section 5, we predict some trends for the future – to answer the question “Where are we going?” - and in Section 6, we provide a summarizing discussion. Finally, towards the end of the paper, we draw some final conclusions in Section 7.

2. EVOLUTION OF THE SOFTWARE INDUSTRY - A HISTORICAL PERSPECTIVE

2.1 The Era of Hardware Driven Software

The history of modern computers has its roots in the middle of the 1940s. Computer equipment – hardware and peripheral devices – has always played the role of enabler for software engineers. During the first decades, the development of software was directed by low performance power and by the modest memory capacity of computers, as well as the poor technology of mass memories. At that time, software engineers had to adapt their work to the demands of the equipment; part of memory management (segmentation) had to be included in the code and code optimization had to be executed to guarantee its better performance. From the software development point of view, we can separate two eras. From the 1940s to the 1970s, software development was done in accordance with the conditions of the device – software was adapted in the equipment (hardware as an enabler). Since then, the situation has been accelerating in the opposite direction thanks to lowering hardware prices and new kinds of scalable infrastructures (cluster, cloud).

Likewise, the use culture of computers has changed significantly in the course of the decades. The era from the 1940s to the 1950s can be described as the era of computer laboratory use, in which computers were used mainly for technical calculations by a few research-intensive organizations. The era of closed shop mainframe computers for common use started in the 1950s; computers were separated from their users and also from software developers. Computers were rare and expensive; only large organizations had an opportunity to acquire them. The period of commonly available computing started in the 1960s;
first in big companies and spreading to smaller companies gradually in the form of minicomputers. Time-sharing operating systems in the 1970s opened the “closed shops” up to users and software developers (operating system as an enabler). Computers could be used via terminals for direct access to the applications for users and computer resources for application developers. This era could be called the era of centralized distribution, because the computers were still located in closed machine halls and the terminals were connected to them with fixed cable connections. The first commercial personal computers were adapted for business use at the end of the 1970s and early 1980s. This can be considered as the birth of the era of distributed computing resources, which is still going on. This progress first started the era of unmanaged distribution: computing power was available at a low price and allowed the satisfaction of individual needs. Networking (first Ethernet, then a variety of other network technologies) of computers returned the culture back to its roots – the era of managed distribution started in the early 1990s and still continues in its enriched form called the era of cloud technologies.

The old prediction made by Boehm in 1976 indicates the liberation of information systems from hardware dominance (Figure 1). The original figure (left side) is continued by the authors to cover the period from 2000 to 2020. The aim is to point out the constantly falling importance of device costs in information systems development and the transfer of costs to maintenance and especially to the related, use-time services. These aspects will be discussed later in this paper.

2.2 The Era of Open Ecosystems Driven by Open Interfaces

**Liberation of Software Development.** The first microprocessors had been launched in the late 1960s. Personal computers brought computing power available at a low price to satisfy the individual needs of computing. Since the late 1970s, the microcomputer evolution has dramatically increased the quantity of software produced and the number of companies involved in software development. These changes together kicked off the software revolution: software started to appear everywhere, and can nowadays be found not only in personal computers but in homes, domestic appliances, public services, offices, and cars. [Fayad et al. 2000]

Until then, the synchronous development of hardware and software gave few opportunities for software developers to iterate over the available features – as everything was basically controlled by hardware. Until then, even software written in high-level programming languages tended to depend on hardware
since the evolution of and interface to the new hardware features was controlled. The development platforms were also provided by the manufacturers – the ecosystem was more or less under the tight control of the computer manufacturers. Due to the need for hardware-specific optimizations, developers had to have a deep understanding of the underlying hardware and thus had to specialize in certain hardware-driven ecosystems. In addition, applications became optimized for specific hardware because of these optimizations and the lack of system libraries that would be the same for all types of hardware.

The key invention that liberated software developers from the dominance of hardware was standard interfaces, offered by an operating system that acted as an abstraction layer between a piece of hardware and the software it runs. Furthermore, while some operating systems are hardware-specific even today, operating systems such as Unix and Linux can be run on various hardware platforms.

The ability to write a program against a well-defined abstraction – given in the form of a stable interface – meant that the same approach could also be applied to contexts other than hardware only. Moreover, software systems could be partitioned so that some parts would come from one vendor, and others from someone else, based on open interfaces that would act as a contract between the subsystems. This progress was called by Barry Boehm [Boehm 2006; 2006a] in his ICSE 2006 keynote presentation the transfer towards “complex systems of systems”. It points out the situation in which interfaces and collaboration between software assets become dominant factors in software development (interface as an enabler).

The emergence of open interfaces had many consequences. To begin with, the size of a software system was no longer limited by the size of the team that could be employed to build it, but libraries and other standard subsystems such as databases could be used off-the-shelf. Furthermore, the development cycle of new software features was no longer entangled with that of hardware, meaning that it was possible to start developing software at a pace that was better suited to meet the needs of the end user. This meant that hardware and its requirements no longer dictated software requirements, but rather the focus was placed on end-user requirements and needs.

While open interfaces have become an important enabler for business, they have also become a business tool for the companies that define the interfaces. With a public interface, the developer community can be divided into externals that can only use the public open interface and an inner circle that use functionalities beyond the public interface. Furthermore, the implementers of a public interface may be subject to specific licensing conditions.

From projects to product business When open software ecosystems appeared, the business started to be organized mainly according to two different business paradigms: the software project based business model and software product based business model. Consequently, software was mainly developed as unique systems for each organization, and very little standardization existed. In the ISO/IEC 12207 Systems and software engineering – Software life cycle processes standard, “project” is defined as “an endeavor with defined start and finish dates undertaken to create a product or service in accordance with specified resources and requirements” [ISO/IEC 2008]. Project business in general is the part of business that relates directly or indirectly to projects, with the purpose of achieving the objectives of a firm or several firms [Artto & Wikström 2005]. The end result for the customer is tailor-made software that is individually installed on a dedicated server. In the project model, pricing is typically based on total work, and that also limits selling this kind of software to Business-to-Consumer markets.

When the industry started to grow and the consumer market opened up, suppliers noticed that once software had been built, stabilizing and re-selling it could lead to significant economically benefits. Software is built both for increased selling volumes and for better profit margins. These observations led to the birth of the software product business: software is delivered in a similar form to every customer. Both the software and all the marketing, delivery, materials, support, and services are productized. The term “Commercial off-the-shelf” (COTS) is also used. According to Wikipedia, this means the purchase of packaged solutions that can be bought “as is” and then adapted to satisfy the needs of the purchasing organization, rather than the commissioning of custom-made solutions. When a software company
expands its platform outside organizational platforms, the model may be called the software ecosystem approach. [Bosch 2009]. While the two models, the project- and product-based business, are seemingly at the different ends of the spectrum of the software business, they surprisingly share many similarities in the actual software development itself. Today, both are done in an agile, customer-driven development, where either a direct customer or an imaginary customer played by, e.g., a product manager, feeds user requirements into the development process. Thus, there is a clear feedback loop from end users to the development, no matter what the business model is – the difference is often only in the monetization model.

2.3 The Era of Diversification in the Software Business

The technical drivers of the diversification discussed in Section 1 were affordable devices, the Internet as the delivery mechanism, and virtualization in the cloud. The consequences of these drivers will be discussed below.

Affordable devices Most office desks have PCs or laptops, most homes have PCs and game consoles. The majority of consumers in developed countries carry mobile devices like tablets and smart phones. These devices are used for both professional and private purposes. This means that the software is not only sold to professionals but also to consumers and business users who want to also use the application with mobile devices. This has caused changes to marketing and pricing but also imposes additional requirements for aspects like security and liquid multi-device experiences [Artto & Wikström 2005; Taivalsaari et al. 2014]. The appearance of mobile devices opened the market to low-cost mobile applications at a very low price – or the initial purchasing price may even be zero with the developer earning a living from alternative sources like embedded advertisement and in-application purchasing.

Internet as a delivery mechanism First, the Internet enabled the delivery of software without hardware media, but the applications were still installed on computers at customer premises. This made the delivery of software and updates faster and cheaper. When the speed, availability, and reliability of the software increased, many companies started to offer the applications primarily over the Internet. Thus, in addition to traditional customized or "commercial off-the-shelf" software, the software business began to branch into "Software-as-a-service (SaaS)". This freed customer from hosting the application, but at the same time gave more control to the software vendor.

Virtualization and the cloud With these technologies, computing capacity can be sold as a utility and charged according to usage. This has improved the business of SaaS providers since the hardware capacity and costs became easier to manage. Virtualization is a key enabler of the cost-benefit of the SaaS model described above. It also amplified the challenges related to multi-tenancy, and solving these challenges has increased the engineering costs. The services offered to several customers are provided with the same computing resources and implemented with shared components. However, different customers need separate customizations. SaaS and multi-tenancy cause totally new, different requirements and affect every layer in the architecture. [Bezemer & Zaidman 2010]

The data of different customers need to be separated. Still, the use of SaaS with virtual clouds raises concerns and, among IT executives, security threats are the most dominating factor in risk perception.

Furthermore, new stakeholders – providers of computing capacity – have appeared. This is also called utility computing; such an approach means providing computational resources, with their provisioning based on actual use. This lowers the barriers to using a new system, as the initial costs are low or even non-existent if operational costs are ignored.

3. BUSINESS MODELS IN THE CONTEXT OF THE SOFTWARE BUSINESS

A look at previous literature shows that scholars overall do not have a common definition for business models. The challenge in finding a common definition is that business model related research is developed typically in silos, according to the researcher's interest [Zott et al. 2011]. Although the concept of business
model has been described in many different words in the literature, all definitions usually cover the following two areas: “what does the customer get?” (value creation process) and “how do we get the money?” (converting the market opportunities into revenue). Some researchers use the term “business model” very simply:

- A business model describes how a company makes money and how it specifies its positioning in the value chain [Rappa 2001].
- Business model is an architecture for the product, service, and information flows, including a description of the various business actors and their roles, and a description of the potential benefits for the various business actors as well as a description of the sources of revenue. [Timmers 1998; Amit & Zott 2000]

Timmers [1998] considers a business model to be more of an industry level concept - or at least he does not limit it to the scope of one company. In the field of software business research, there is also a lack of rigorous previous definitions of a business model [Rajala et al. 2003]. They identified the need for a software industry specific framework for analyzing business models. The framework consists of four elements: product, revenue logic, distribution model, and services and their implementation. They consider that the business model refers to a single company, and that it describes only a single product at a time.

According to the literature review of Zott & Amit [2011] the concept of business model has been employed mainly in trying to explain some of the following three phenomena: e-business, strategic issues, or technology and innovation management. The business model is applied and described in different ways in these three domains. Literature focusing on e-business has been interested only in cases where a company is engaged in Internet-based way of doing business; the concept of a business model consists of a value proposition, revenue model, and network and relationships together. In literature focusing on strategies, business models are seen as a strategy concept and the most interesting factor is the firm’s activities; these studies describe a business model as a notion of activities or activity systems. In technology management literature, business models are seen mainly as a mechanism to transfer technology and technological innovations into commercialized products; an important role of a business model is to release the potential embedded in a technology into market outcomes.

Technological development steps can trigger changes in a company’s business model, enable the new ones and force companies to seek them [Calia et al. 2007]. Existing revenue streams may decrease before the new ones begin to start bringing in cash, so a strong financial position is needed. A company needs new competences: new personnel have to come in and the existing personnel need to be re-trained/re-skilled. In software products, new architectures and modifications in existing software appear, which usually increases the complexity of the software. The whole field, the players, and their roles may change, which causes a need for a new kind of trust and rules. [Spinellis 2016; Cusumano 2010].

Figure 2 explains the elements of a business model, how they interrelate, and how the model is affected by the underlying business environment. Business models may not only emerge as a consequence of technological innovation, but can also be shaped by it. We apply the concept of business models in the way proposed by [Zotta et al. 2011] that is typical in the domain of innovation and technology management: technology is seen as an enabler of the business model, rather than as a part of the business model itself. We use the framework introduced by Rajala et al. [2003] as a guideline for our analysis.

The four elements of a business model – product, revenue logic, distribution, and services – are split into the components we consider to be in a close relationship with the industry evolution. Furthermore, in this study, the elements of the “business model” represent the general way of doing business within the industry, and are not strictly limited to the business of one firm.
4. VIEWPOINTS ON TRANSITION - EFFECTS OF THE BUSINESS MODEL ON THE SOFTWARE INDUSTRY

4.1 Conditions

The aspects of the software product are “what, for whom, and how” – what is the product, who are the end users, and what kind of technologies and tools are needed. In this chapter we take a look at the elements behind software products: technologies, tools, quality issues, the base characteristics of the commercial product, and the end users of the products.

4.1.1 Innovations. In Section 2 we listed a variety of enablers that indicated the beginning of a new era in the software business: hardware, open interfaces, the Internet, and virtualization. In innovation theory, changes are classified into four categories: incremental innovations, radical innovations, changes in technological systems, and changes in techno-economic paradigms [Dosi et al. 1988]. Incremental changes appear continuously in existing products and services (continuing the existing trend). Radical changes appear when new research findings are applied in products to transfer their properties or performance to a new step or cycle (movement to a new trend). Changes in technological systems are caused by combinations of several incremental and radical innovations in societal and organizational systems. Changes in paradigms are revolutionary and lead to pervasive effects throughout the target system under discussion. From the innovation theory point of view the enablers listed above have their primary source in technologies, but are also partially connected to societal changes, attitudes, and people’s expectations. We can also find features of all four innovation categories in the background of changes in software business models. In the following we examine the innovations that have played an important role in the evolution of the software business over the decades.

Hardware related innovations The invention of microprocessors and VLSI in late 1960s started the rapid growth of computing power. In this context it is worth referring to Moore’s law [Moore 1965], which forecast that the packing density of microprocessors and VLSI circuits would double at intervals of 18 months. This growth is still continuing and is accelerated by solutions based on parallel architectures and
multi-core processors. It was the beginning of the new era; it was cheaper to adapt the hardware for the
demands of software needs than the opposite. In the 1980s the rapid spread of networking (technologies)
accelerated this progress further and moved towards network-based (SOA-type) solutions.

**Innovations in tools and paradigms for software development** Until the 1960s, assembly (machine)
languages were commonly used to reach effective use of processing capacity. The first commonly used
“high level” programming languages were reasonably straight derivatives of computer machine language.
FORTRAN - The IBM Mathematical Formula Translating System - was developed in the middle of 1950s
for scientific and numeric computing [Wikipedia 2019]. In spite of having uniformity with the machine
code of IBM 701 it started the transition towards high-level, machine-independent programming tools.
Another language worth mentioning in this context is C-language. It was designed to support
development of the Unix operating system for PDP-11 and indicates its architecture [Wikipedia 2019a].

The transfer of computers to the commercial and administrative sector caused pressure for new
programming languages to take into account the needs of this new application area. Based on the
sponsorship of the DoD (Department of Defense), an initiative to develop a COmmon Business-Oriented
Language was originated [Wikipedia 2019b]. The initiative was based on a study according to which the
costs of the software work exceeded the costs incurred by the equipment (late 1950s) and especially
platform independence and portability of software were regarded as important properties. This can be
considered as an observation of the first software crisis. The first specification of COBOL was published in
1960. The observations on the background of COBOL can also be considered a starting point to the quality
driven approach of software work: portability, the dominance of software development costs instead of
hardware, application oriented language structures, applicability in a variety of use contexts, and
platform independence.

4.1.2 Focus on Software Quality. A transition started towards quality driven software development, first
following the concept of structural programming and further towards an object oriented approach. The
current approach of the main aspects related to software quality is documented by ISO/IEC in a series of
standards [ISO/IEC 2011; 2001]. The first programming language supporting structural programming
was Algol [Wikipedia 2019 c]. It was originally proposed in 1958 but it is best known as the Algol 60
specification from 1960. This language can be seen as an initiator of the “structural programming”
paradigm. Algol never became a language used in the software industry, but its derivatives, C, C++,
Pascal, Modula, Ada and many others, have been guides to the modern practice of software work. Algol
can also be seen as a starting point of the object oriented programming paradigm. In the early 1960s, the
Algol derivative SIMULA included the concept of classes and it has been the root of currently used
languages such as Java and C++.

4.1.3 The Evolution of Commercial software Products. Software project business, which initially was the
main model in developing software, appeared to be in many situations too slow and too expensive both to
customer and to supplier. However, it is still valid in some cases, e.g., in highly specialized Business-to-
Business systems. However, software prices have fallen so that you can buy “almost anything” for very
little money or on a "pay as you go" basis. The model also has its challenges in the pricing of packaged
SW: distributors and retailers get a big slice of the list price and eat away at the margins of the
developing company. [Gewirtz 2015]

The main benefit of selling productized software is the effective reuse of code and other materials. This
means generating new revenues without additional development cost. The software product business can
be extended to a product line approach, which is a technology to support the derivation of a wide range of
applications from a common core. Software product lines (SPL) are an effective approach for modular,
large-scale reuse of software. In addition to software product lines, mass customization enables the
building of products that are seemingly adapted to a particular use from the end-user perspective
[Verdouw et al. 2014]. This enables customers to get features similar to customized software at an
affordable price. Similar results can be achieved by allowing the end user to fine-tune the product by enabling end-user programming in applications. Examples of such an approach include spreadsheets or accounting applications, shifting a part of the customization effort to the user.

While the two models, the project and product based business, are seemingly at the different ends of the spectrum of software business, they surprisingly share many similarities in the actual software development itself. Today, both are done in an agile, customer-driven development, where either a direct customer or an imaginary customer played by, e.g., a product manager, feeds user requirements into the development process. Thus, there is a clear feedback loop from end users to the development, no matter what the business model is – the difference is often only in the monetization model. Another common characteristic is that it is becoming next to impossible to draw a borderline between development and maintenance in terms of actual technical contributions [Mikkonen & Systa 2014]; again, the business needs to mark the difference in these approaches. Finally, as everything is becoming upgradable increasingly often online, the differences between the business models are becoming unclear since anything can be so easily modified that it does not really matter what the fundamental business rationale is. This new model is what is actually driving the next era of the software business we have entered, the time of diversification, enabled by the Internet (Internet as an enabler) and increasingly rapid development cycles.

4.2 Revenue Logic

Revenue logic can be split into two components: who receives the revenues and on what kind of basis. The transition from work amount based projects to a use-based cloud solution is described in this chapter. This transition has also opened the field up for new kinds of players.

4.2.1 Revenue Models. Revenue logic and pricing models have varied over time. New business models have given software companies new kinds of choices for pricing. However, the biggest winners have been the customers, who can buy software for almost any purpose at a very low price. As enterprise software has become available as SaaS versions, small companies in particular can now acquire software that they could not afford before.

Project pricing is based on the amount of work. It is typically expensive and contains a potential cost risk for the customer in the case of exceeding work estimates. These potential risks often lead to maximum cost and sanctions defined in the delivery agreement.

Due to the nature of information intensive products, producing software is expensive but reproducing it is very cheap [Sainio & Marjakoski 2009]. Standardizing and productizing software and pricing optimize the margins, which may be as high as 90 percent in the software product business. Also, the customer benefits of the product pricing, i.e., standard software and fixed price, are safe, because the cost is predictable.

New SaaS and cloud models are good from the customer’s point of view: Customer expectations for services also pose a significant challenge for revenue logic. SaaS customers expect to pay according to time and usage and do not want to buy with a lump sum. At their best SaaS models enable the customer to "pay as you go" - pay according to usage. For the software vendor, this means differences in the revenue stream and needs to be considered in the pricing. The total revenue has to be gathered from small streams. It may also be a strength: if the customer commits to a certain cloud software and stays on the platform for a long time, he/she generates constant revenue every month. If the cloud strategy is based on mass customization, the pricing reflects it and collects certain features together in one pricing level. In any case, implementing and managing the desired pricing models is possible only if those needs are considered when designing the system architecture. Selling of licensing for a lump sum and typical service deals are priced differently. After developing a software product, the margins for selling it may be very high, even over 90 percent. As for services, they are typically very human-intensive, and may produce a
4.2.2 *Actors and Roles.* In the first phase of the software business, software development was done mainly inside the walls of hardware manufacturers. Companies themselves were both hardware and software suppliers. Hardware manufacturers could have the software done by subcontractors, but they were in any case tightly bound to a certain hardware ecosystem and certain technologies. When personal computers came on the market and private customers started to have software needs, open programming interfaces also started to appear. This enabled the rise of new business: software development could be done independently, without being restricted to a single hardware manufacturer. Later, this kind of outsourcing crossed borders between countries and was given new forms and names, for example offshoring and nearsourcing. Global outsourcing was popular in countries of high labor costs (like Finland). Afterwards, callback decisions for global outsourcing were seen in those countries, due to raised prices and unpredictable add-on costs. In the early days of software engineering, software was a monolithic product, produced by an independent vendor.

When standardizing and productizing of software became common, component manufactures appeared on the market. Modern software strongly relies on infrastructure and components from third-party vendors: service operators, open source suppliers, etc. The relationships between software development companies and service firms turned software production into software ecosystems, where different companies collaborate to create value. Selling components is typically a mass business; they are cheap but easy to buy and their re-producing cost is almost zero. The increasing amount of different enterprise software caused a need for integrations between single systems. Depending on the field, integrations may be either very common and simple or very specialized and complex. Integration needs led some companies to focus on producing integrations. Overall, the opening of interfaces and ecosystems and the following division of business into different individual parts caused the birth of system integrators - companies who deliver solutions by selling a stack of hardware, software, and services as one product. Up until the 1980s, vertically integrated companies delivered complete system stacks: hardware, operating system, and applications. In any case, software supply chains are transforming more and more into agile networks in response to the increasing volatility of business environments. [Cusumano 2008; Jensen & Cusumano 2013; Verdouw et al. 2014]

The opening and liberation of the software markets also caused the transfer of knowledge and competence from single, monolithic manufacturers to the network. The more complex the business becomes and the more the software also belongs to any other products, the more common it is for companies to subcontract the ICT competence from specialized companies. This transition has been an opportunity for ICT consultancy companies, whose business size, revenues, and competences have been growing continuously.

The invasion of SaaS and Cloud models has opened the field to a new kind of player. Customers expect cost-effective, efficient, and flexible delivery of IT services, with a maximum of monetary flexibility, which leads to the evolution from outsourcing to cloud. As a consequence, evolution from traditional IT outsourcing towards buying services from the Cloud is a certain significant trend that is changing the industry and actors within it. According to Leimester et al. [2010], the typical roles in the Cloud business are service providers, infrastructure providers, and service brokers [Riehle 2007]. *Service providers* - or content providers - develop applications that are offered on a cloud platform and use the hardware and infrastructure of an *infrastructure provider*. *Service brokers* - or aggregate service providers - offer new solutions by combining existing services into a new form of service. Also related to the Cloud environment, consultants serve as a support for the selection and implementation of relevant Cloud solutions. [Järvinen et al. 2014]
The term Open Source has almost as long a history as the history of computing. Since the majority of software was developed in the 1950s and 1960s in academic and research organizations by developers working in collaboration, the results were commonly shared inside the collaboration network, and the IPRs (Intellectual Property Rights) were not tightly controlled by the developers. The software artifacts were distributed under the principles of openness and co-operation. Until the late 1990s, open software covered a wide variety of informal practices and artifacts, such as system software (operating systems, compilers, editors, development tools, etc.). In the early 1980s, Richard Stallman launched the GNU Project and some years later the Free Software Foundation was established to promote the concept of free software. Until the late 1990s, the free software concept mainly covered tools provided for common use for free – Linux, Netscape, Mozilla, Java, MySQL are examples of this era; a collaborative society of volunteer developers (crowdsourcing) is typical of (most of) these. Simultaneously with the spread of open software, licensing rules and practices were developed to guarantee the use of these products in a way that respects the original principles of openness (e.g. Creative Commons). Because of the long history of “openness” it is reasonably difficult to specify the moment when open source software had a real effect on the software business. It can be claimed that this began roughly in the early 2000s. Without going into details, discussed elsewhere in this paper, open source has accelerated the transition from license-based business towards business based on customization, services, and maintenance support of client products. In addition, the use of open data nowadays has a fast growing business value as well. Open innovation has also shown its power as a part of software engineering: fast transfer towards non-structured (non-SQL) data and the growth of applications based on “block chain” technology are examples of this. Openness has had important consequences in the software business field: old, large, traditional companies have lost their competitiveness and new companies based on lean and agile practices are the winners of the game.

4.3 Distribution

When the software business was liberated and open interfaces appeared, organized delivery chains also started to form. Thus the product itself was delivered as a concrete media; the distribution was also quite similar to distributing any other physical product: with retailers, sales partners, etc. The longer the delivery chain, the more challenges it caused to software pricing. Resellers and other intermediaries received a big share of the revenues, which decreased the supplier profits.

Affordable devices, like mobile devices, and the Internet as a delivery mechanism introduced new distribution channels. For example, “app stores,” with related business consequences like different revenue-sharing models, are changing the software business. On the other hand, the browser has become the dominating platform for PC applications.

Another trend is extensive customization due to affordable devices and the pervasiveness of the Internet. Access to applications is no longer controlled by a particular device, but by a user ID. Especially for consumer markets, the owning of a user ID means the collected user data has become a valuable asset since it can be used in targeted advertising. This has opened a totally new business line, in which the biggest business value lays on the data collected from users and their activities.

The app store concept introduced by the mobile industry and aimed at owning the user data together with the appearance of hosting services has added additional stakeholders to the business ecosystems. This is typically a third player that is not just a distributor in addition to the producer and consumer competing for the revenue.

4.4 Services

When interfaces became open and independent software development was formed into a separate business, the need for additional services also began to arise. On the one hand, the software suppliers added different installation, maintenance and implementation services to their offerings. Further, the offerings were extended to cover user training, consultancy (either system dependent or independent),
designing and delivering of extensive “whole solutions,” and finally application provisioning, which relieved the customer from acquiring and maintaining the hardware environment.

The transition to SaaS especially affected the vendors of enterprise software, but has affected other software businesses, too. Game company sales are no longer based on products only; instead there are several online gaming services. In addition, platform companies, like Microsoft, had for a long time reported almost 100 percent revenue from products, but today’s offering has also moved online. [Mikkonen & Systa 2014] Due to carry-on devices and SaaS-based offering, we are facing a fundamental systemic transformation towards a world where digital resources are constantly available online, and available for all to use [Jansen & Cusumano 2013]. SaaS started an era of licensing and delivering software on-demand based on a centralized hosting solution. Customers do not have to invest in their own hardware or pay extra for maintenance, which makes the cost very predictable. For instance, for SME (small and medium-sized enterprise) clients this gives access to software that many of them could not otherwise afford [Resceanu et al. 2014]. In fact, the potential cost advantages of SaaS are the strongest driver for SaaS adoption [Sultan 2014].

Before SaaS, software products were charged for either as a lump sum or “rented,” i.e., charged per use or time. Customized projects were typically developed for a customer, and development was funded by charging a lump sum after delivery of the software. In both cases, a separate maintenance fee is often agreed. In the SaaS model, the maintenance is assumed to be included in the fee, and typically a separate maintenance business does not exist. On the other hand, the need for maintenance does not disappear and has to be included in the fees.

SaaS often reduces the possibilities for customization, unless the vendor can extend the versioning to hosting, too. However, versioning in hosting reduces the cost effectiveness of SaaS due to the multiple instances of HW and SW components. Thus, vendors typically try to satisfy the needs of multiple customers with a single product. The problem to be solved comprises multiple criteria and also the offering of SaaS products is wide, which is why different, systematic decision-making is needed to find the right product. [Godse & Mulik 2009]

The SaaS model also has organizational implications in companies. First, the role of IT departments is changing since development and IT need to collaborate more tightly. The term DevOps [Debois 2011] is often used to describe the required changes in the mindset. In addition, the SaaS mode outsources responsibility and work from customer companies. In SaaS, SW engineering and the customer interface also collaborate in new ways to constantly bring new value to customers. Instead of major releases managed by a separate business function, today’s software development is about continuous maintenance” [Mikkonen & Systa 2014]. For example, software engineering at Facebook is about experimenting with what adds business value [Feitelson et al. 2013]. Also, feedback collection becomes systematic and automated. In an extreme form, this changes R&D to an experimentation system as in the Stairway to heaven model [Fayad et al. 2000; Olsson & Bosch 2012].

5. TRENDS FOR THE FUTURE

Despite today’s divergence in the software business, we believe that we are just about to start seeing the wide opportunities of the future software business.

The future is a continuum of the past In this context we refer to the words of Larry Page (CEO of Google): “The main reason why companies fail is that they missed the future.” This is true, but we have to remember that the future is built of the components of today, and today is the future of yesterday. As a result, we have to look backwards and recognize how we have come to this point, and evaluate the opportunities it gives for the future. The use culture of computers reflects strongly the opportunities enabled by the computer technology of each era. It has also had a significant effect on software work, which has always been done with the best tools available and following the practices that are best for their use. The changes in the tools and practices have been naturally carried out as a result of the needs of
different interest groups; progress in technology has been the enabler of the step-by-step changes that have taken place over the decades. These changes have been dealt with from different aspects of software and systems engineering in the book [Endres & Rombach 2003]. The phenomena reported in the book cover Moore’s Law (discussed earlier), Hoagland’s law (the capacity of magnetic devices increases by a factor of ten every decade) as a source of current masses of stored information, Cooper’s law (the doubling of wireless bandwidth every 2.5 years) as an enabler of the fast growth of wireless solutions, and Fred’s law (indicating the unknown source: the transmission capacity of wired networks for a fixed price doubles annually). These are examples of the technology-related trends we can expect to continue. The book is structured around the software development lifecycle, covering a variety of laws that it is relevant to understand in each life cycle. [Endres & Rombach 2003]

A comprehensive review of the general trends that can be discovered today is given in the paper [Jaakkola et al. 2014]. The analysis is based on a wide variety of sources and trends are classified into the innovation categories discussed earlier in this paper. The most important segment is naturally paradigms. This covers the transition towards openness (from several points of view) and the growing role of (big) data analytics as part of applications. Application intelligence, including machine learning based technologies, is an important aspect of modern information systems. Autonomous - independent, (more or less) intelligent devices play an important role as applications themselves but also as a source of data and as a collaborative partner of applications. The radical changes category covers for instance the growing importance of the context sensitivity and built-in intelligence of applications, new technical solutions like block-chain, and the changing characteristics of data (enriched data formats). The incremental change factors category covers aspects like transfer towards mobility, consumerization as a phenomenon, and gamification - creating pressure to improve user interfaces and the push towards gamified real life processes. Renewal of distribution channels also belongs to this category; combined with consumerization progress, the role of software product has changed radically, as discussed elsewhere in this paper.

**Every business is a software business** As indicated by the rapid rise of service providers such as Uber and AirBnB, software is enabling new ways to deliver services we so far have expected only an established service provider to provide. In addition to transportation and lodging, the same idea can be applied to virtually any line of business, covering also fields such as electricity networking and banking, where barriers to entry have been almost impenetrable and few attempts have been made. Partly this is due to legal regulation, but as such an approach can often provide a lower-cost alternative to fully supported infrastructure based services, to at least some extent it will only be a matter of time before the majority of them will be opened up to new competition.

Ultimately, almost all organizations need the ability to maintain their own IT operations, either directly or with a dedicated partner, which has been consciously selected. Managing such a software business is tough because software’s ethereal nature offers infinite lucrative or catastrophic choices – the things to manage include the business model, market/customers, the execution strategy, the product or service, and the development process [Spinellis 2016].

**Every product is a software product** As an example, let us consider the challenges of the automobile market. Teslas are first and foremost computers that run on four wheels and are able to transport people from one point to another. Google and Apple are pushing the envelope even further with their plans to transport people automatically, without a driver other than a computer. In fact, even with a traditional car, millions of lines of code are needed to run it [Zax 2012]. The same trend is commonplace in numerous settings that have been considered unrelated to computing, such as ports or manufacturing plants. Much of this software is unlike the software that has been built before — it can never be shut down, as the systems that are controlled and, more generally, powered by software never sleep. The emerging Internet of Things (IoT) is paving the road to a society as well as humanity that is more dependent on software than any individual technology before — software is taking over the world [Andreessen 2011].

Clearly, systems of the scale mentioned above cannot be created just in time and for just one product or service. Instead, they are systems of systems created out of components that either happen to be readily
available [Hartman et al. 2018] or are engineered to the perfection that a group of developers deem critical enough. We expect that this will lead to the introduction of a collection of domain-specific “designs of dominance”, where one vendor becomes irreplaceable in a certain field, much like Google is today in the field of maps and web searches. These islands will then be control points in the future software business, where smaller companies, with less important positions and control points, perform fractal-like development to create services by combining and reusing existing systems.

This development model will introduce the best and the worst parts of today’s software business. On one hand, services are based on reusing existing systems that provide a robust starting point for development. On the other hand, services are constantly being refined towards a form where we are expected to consume them the most, disregarding at least some of the quality issues.

**Variance in business models** Even now, IT is fueling new business models, such as offering hardware and software for free, but making the customers pay as they go with zero initial cost. So far predominantly used only in the telecom industry, this business model requires the additional ability to fund underlying software development via IPO, partnering with funding organizations, or simply the company’s own capital. Opportunities such as crowdfunding via services such as Kickstarter (http://www.kickstarter.com) are making this a viable option even for smaller companies, not only global giants.

6. **DISCUSSION**

The modern business expects ability to execute sophisticated routines and to perform analytics that require business and technology competences that traditional industries do not have (e.g. complex data analytics). So far, the typical solution for companies that lack ICT skills has been to subcontract ICT work from specialized companies, while keeping the leadership regarding the services within the companies themselves.

As part of this process, ICT consultancy companies are learning more and more. Moreover, the latest trends, like DevOps, require even more skills for successful ICT operations, and therefore give even more responsibility to ICT-related operations. In the long run, this will lead to slow but major change – size, revenues, and control of the ICT companies will increase, and the traditional companies have only the underlying hardware – if they still want to host it; in any case hardware can be replaced – and the brand – which can be diluted by the ICT companies who have an opportunity to introduce their own brand through continuous maintenance of the service [Mikkonen & Systa 2014].

Due to the possibilities provided by ICT, ways of doing business and related business needs are changing faster and faster. Approaches such as the Lean Startup provide an iterative, almost scientifically justified approach to creating new enterprises. In general, applying these approaches in the domain of software development, software and associated business – or business and associated software – must co-evolve. One of the caveats is an accidental, perpetual vendor lock-in, or a situation where there is no way to switch the vendor, as everything takes place in real time within business operations. In an extreme case, this can lead to hindering someone else’s operations with hostile intent.

How the business related changes are seen in software quality, or actually, what has happened to the concept of software quality during the technological progress of decades. The paper [Jaakkola et al. 2017] handles reincarnation cycles in the certain areas of ICT. According to the paper, the main trigger of the changes is the progress in VLSI technology. It reflects directly to the processing capacity and memory size (both RAM and mass memory) of computers. Further, because data transmission is also based on complex calculations (in addition to the advanced transmission channels), even this part of ICT is tightly connected to the progress in VLSI technology. In all these fields changes have been exponential – already over decades. The progress creates platform for new applications, new kinds to use ICT infrastructure, also for new software development practices; this progress is handled in [Jaakkola et al. 2019].
Simultaneously to the technological progress the concept of “software quality” has lived its own life. The paper [Jaakkola et al. 2017] describes this progress in the following way: “In the 1950s, the period of a shortage of resources, quality software was based on minimal usage of the main memory (small size) and effective processing – the first wave. In the 1980s the focus was on the logical structure and maintainability of software – the second wave. The next wave (third) focused on quality (process) management. It was based on the idea that quality software is a product that is produced by a high capability and maturity processes.” At this moment, in the fourth wave, we have new components in software quality: interoperability both in software and in development process level, openly available components and data, ability for collaborative development of software, ability to monitor the development process and further to use the monitoring data in improving the software process, etc. New aspect in software quality is also the growing role of data quality, which indicates that focus must also be placed on the quality of the data that is handled; this would be also an essential part of the fourth wave.

7. CONCLUSIONS

Our paper covers a variety of aspects relevant to changes in the software business. Figure 3 summarizes our findings and discussion in this paper. We will not reiterate the detailed discussion related to the topics summarized in the figure. The line title indicates the factor under discussion and the columns in each line list the changes over time. The time scale between lines is not comparable: the listed items are in chronological order, but not comparable between lines. The changes indicated in the table are overlapping (no clear borders between the eras) and incremental (features of earlier eras remain and can also be seen in the latter periods).

The software business is currently in the final phases of a transition from shrink-wrapped software products (commercial, off-the-shelf software sold in retail) to cloud-based services, delivered either in the form of mobile apps - which often in essence are just an access method to a cloud service - or as full-blown web applications [Gewirtx 2015; Mikkonen & Taivalsaari 2013]. Indeed, software, together with associated data collection and data analysis, accounts for a greater and greater part of the value of products and services. In fact, companies such as Facebook have reported that it does not have a direction in its engineering - instead, it is up to the users to act so that a business-satisfying direction emerges based on usage data [Feitelson et al. 2013].

The development of the software industry has been guided by technological breakthroughs. This trend is still continuing and has also led to changes in paradigms. For example, accessibility and the low price of network capacity has had the effect on moving away from server solutions towards the cloud, or the heterogeneity of devices, and diversification of platforms has led to digital convergence.

Many of the strong trends of the industry have been simultaneously both opportunities and threats. Consumerization has extended the market and created opportunities for new businesses. In recent years, consumerization, including for example decreasing prices of mobile devices, has also led to a crash in software prices. It is possible to buy almost any software functionality for both private and enterprise purposes. Borders between these two segments have also almost disappeared; nowadays the biggest difference is in service capacity.

The Internet as a delivery channel has supported the general globalization trend. In contrast to many other industries, the Internet solves many logistics problems and eases entry into international markets. Naturally, some other new skills and competences are needed for successful internationalization. These skills cover, in addition to the widening variety of technical competencies, also a variety of “soft” skills, like understanding the differences between cultures in business, leadership and management, ability to benefit on the technical infrastructure in communication and interaction, etc.

A service culture and market have been established in the software industry. Customers want to have additional services and total solutions. However, this is both an enabler and a challenge; resourcing, work organizing, offering etc. have to be rethought and observed from new viewpoints. Margins in the service
business tend to be much lower than in the software product business, which requires new kinds of productizing and servicizing skills.

As the industry becomes more mature and diversified, it also requires wider competences and successful utilization of networks to create a value creation that satisfies customers. Optimal competence profiles change fast, which causes challenges in the educational sector. How to educate an expert with good basic skills, the ability to refresh and learn fast, who masters both technical issues and architectural, integrational, marketing issues?

A big trend among customership is that the customer is becoming closer to products and product development. Customers are joining in the value creation at an earlier and earlier phase. Customer actions and feedback define and guide the development work in quite a short time span. The value of the business is being formed more and more by data gathered from customers. All this requires companies to be truly open to customer expectations and feedback. In addition, fast reactions from suppliers are needed.

REFERENCES


Helena Holmström Olsson, Hiva Alahyari, Jan Bosch. 2012. Climbing the "stairway to heaven" – a multiple-case study exploring barriers in the transition from agile development towards continuous deployment of software. 38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA 2012). 392–399. DOI: 10.1109/SEAA.2012.54.


10.1109/MC.2007.147.